TOKYO INSTITUTE OF TECHNOLOGY

DOCTORAL THESIS

Memory-Efficient Object-Oriented Programming on GPUs

Author:
Matthias SPRINGER

Supervisor:
Prof. Dr. Hidehiko MASUHARA

Student Number:
15D*****

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Programming Research Group
Department of Mathematical and Computing Sciences

August 19, 2019

Declaration of Authorship

I, Matthias SPRINGER, declare that this thesis titled, "Memory-Efficient Object-Oriented Programming on GPUs" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

Graduate School of Information Science and Engineering
Department of Mathematical and Computing Sciences

Doctor of Philosophy

Memory-Efficient Object-Oriented Programming on GPUs

by Matthias SPRINGER

Object-oriented programming (OOP) is often regarded as too inefficient for high-performance computing (HPC), even though many important HPC problems have an inherent object structure. To make HPC available to engineers and researchers in other domains, our goal is to bring efficient, object-oriented programming to massively parallel Single-Instruction Multiple-Data (SIMD) architectures, especially GPUs.

In this thesis, we develop techniques and prototypes for optimizing the memory access of object-oriented GPU code. Our first prototype IKRA-RUBY explores modular array-based GPU computing in Ruby, a high-level programming language. Our main prototype IKRA-CPP explores object-oriented programming within parallel GPU code in CUDA/C++.

We propose a new object-oriented programming model called *Single-Method Multiple-Objects* (SMMO) that can express many important HPC problems and that can be implemented efficiently on GPUs. Our main optimization is the well-known Structure of Arrays (SOA) data layout, which improves vectorized access and cache performance. The main contributions of this thesis are threefold: First, we develop an embedded C++/CUDA data layout for IKRA-CPP that allows programmers to experience the performance benefit of SOA without breaking OOP abstractions. Second, we design DYNASOAR, a lock-free, dynamic GPU memory allocator for IKRA-CPP that is based on hierarchical bitmaps and optimizes the usage of allocated memory with an SOA data layout. In contrast to other state-of-the-art GPU allocators, DYNASOAR trades raw (de)allocation performance for better memory access performance, resulting in an up to 3 times speedup of SMMO application code. Finally, we extend DYNASOAR with a memory defragmentation system called COMPACTGPU, which further increases the performance benefits of SOA and lowers the overall memory usage through less fragmented allocations.

We evaluated IKRA-CPP with nine SMMO applications. Our experiments show that the SMMO programming model is powerful enough to express many important HPC problems from various domains. They also demonstrate that programmers can have the benefits of object-oriented programming and good runtime performance at the same time.

Acknowledgements

First and foremost, I would like to thank my academic advisor Prof. Hidehiko Masuhara for accepting me as a doctoral student in the Programming Research Group, for giving me the opportunity to freely pursue my research interests, and for supervising, guiding and supporting my work throughout these last four years. I would also like to thank Prof. Robert Hirschfeld, my academic advisor during my Bachelor's and Master's studies for the continuous support and collaboration even after my Master's studies.

In 2016 and 2018, I had to pleasure to work with Peter Wauligmann and Yaozhu Sun, who joined the Programming Research Group as part of a university exchange program. Chapters 3 and 4 contain material from two ARRAY workshop papers, which were co-authored by Peter Wauligmann and Yaozhu Sun. My co-authors have approved the inclusion of these papers in my thesis. I am grateful for their contributions to my research artifacts and wish them all the best with their own careers in industry and academia.

The financial support of MEXT (Japanese Ministry of Education, Culture, Science and Technology) and JSPS (Japan Society for the Promotion of Science) through their scholarship and fellowship programs allowed me focus entirely on my research. I am grateful for their support and their decision to fund my research proposal. I would also like to thank NVIDIA for donating a TITAN Xp GPU to our lab, which allowed me to run my experiments on the latest hardware architecture.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

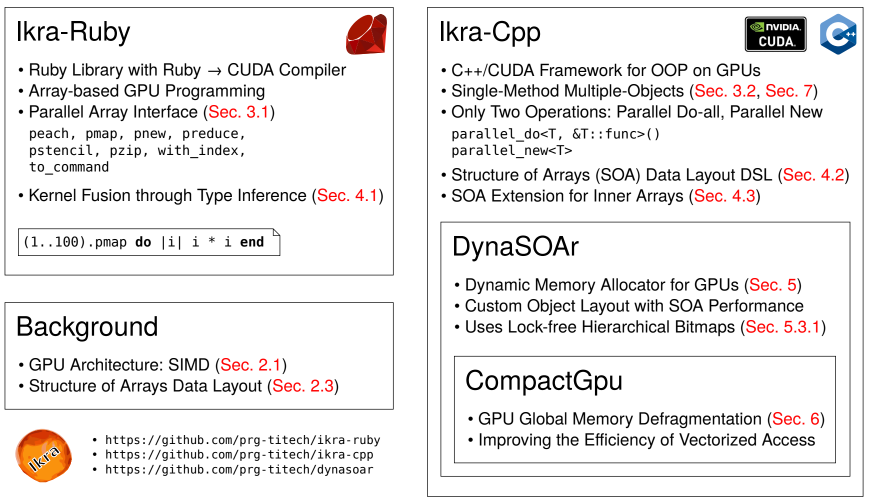
Introduction

High-performance, general-purpose GPU computing has long been a tedious job, requiring programmers to write hand-optimized, low-level programs. Such programs are hard to develop, debug and maintain. Even though object-oriented programming is well established in other domains and appreciated for its good abstraction, expressiveness, modularity and developer productivity, it is regarded as too inefficient for high-performance computing (HPC). This is despite the fact that many important HPC applications exhibit an inherent object structure. Our goal is to bring fast object-oriented programming to SIMD architectures, especially GPUs.

Thesis Statement. *Efficient object-oriented programming is feasible on GPUs.*

In the course of this thesis, we present the design and implementation of libraries/prototypes that allow programmers to utilize object-oriented programming on GPUs (Figure 1.1). We show that an object-oriented programming style, with most of its benefits, is feasible on GPUs without sacrificing performance. We investigate (a) how GPU parallelism can be expressed with object orientation and (b) how object-oriented code that runs on GPUs can be optimized.

Expressing Parallelism: Parallel Array Interface Our first prototype, IKRA-RUBY, is a Ruby library for high-level, array-based GPU programming. Programmers express parallelism over an array with small, functional, customizable operations such as *parallel map* or *parallel stencil*. Programmers can utilize the full range of Ruby's object-oriented capabilities to compose a larger GPU program from multiple parallel


Ikra-Ruby
• Ruby Library with Ruby → CUDA Compiler
• Array-based GPU Programming
• Parallel Array Interface (Sec. 3.1)
• Kernel Fusion through Type Inference (Sec. 4.1)
Background
• GPU Architecture: SIMD (Sec. 2.1)
• Structure of Arrays Data Layout (Sec. 2.3)
Ikra
Ikra-Cpp
NVIDIA CUDA
• C++/CUDA Framework for OOP on GPUs
• Single-Method Multiple-Objects (Sec. 3.2, Sec. 7)
• Only Two Operations: Parallel Do-all, Parallel New
• Structure of Arrays (SOA) Data Layout DSL (Sec. 4.2)
• SOA Extension for Inner Arrays (Sec. 4.3)
DynaSOAr
• Dynamic Memory Allocator for GPUs (Sec. 5)
• Custom Object Layout with SOA Performance
• Uses Lock-free Hierarchical Bitmaps (Sec. 5.3.1)
CompactGpu
• GPU Global Memory Defragmentation (Sec. 6)
• Improving the Efficiency of Vectorized Access

```
peach, pmap, pnew, preduce,
pstencil, pzip, with_index,
to_command
```

```
(1..100).pmap do |i| i * i end
```

```
parallel_do<T, &T::func>()
parallel_new<T>
```

```
• https://github.com/prg-titech/ikra-ruby
• https://github.com/prg-titech/ikra-cpp
• https://github.com/prg-titech/dynasoar
```

FIGURE 1.1: Overview of this thesis

operations in a modular way. By combining multiple parallel operations, programmers build a computation graph. IKRA-RUBY optimizes this computation graph by fusing multiple parallel operations into a small number of CUDA kernels.

Expressing Parallelism: Single-Method Multiple-Objects Our second prototype, IKRA-CPP, explores object-oriented programming within parallel GPU code in C++/CUDA. While IKRA-RUBY provides various parallel operations, we found that a simpler model with only one type of operation, *parallel do-all*, is sufficient for a many applications. We discovered a broad programming model based on parallel do-all with many object-oriented applications that can be implemented efficiently on massively parallel SIMD accelerators. We call this model *Single-Method Multiple-Objects* (SMMO), because parallelism is expressed by running a method on all objects of a type. SMMO fits well with the data-parallel execution pattern of GPUs and has many important real-world applications such as simulations for population dynamics, evacuations, wildfire spreading, finite element methods or particle systems. SMMO can also express breadth-first graph traversals and dynamic tree updates/constructions.

Optimizing Memory Access Object-oriented programming is slow on SIMD architectures mainly because of inefficient memory access. Getting data into and out of vector registers is often the biggest bottleneck and peak memory bandwidth utilization can be achieved only with efficient vector transactions. To optimize the memory access of IKRA-CPP applications, we developed an embedded C++ DSL that stores objects of the same type in the well-known Structure of Arrays (SOA) data layout. SOA is a form of structure splitting that stores all values of a field together. SOA is a standard optimization and best practice for GPU programmers but without proper language support, it leads to less readable code and breaks language abstractions. IKRA-CPP allows programmers to experience the performance benefit of SOA without sacrificing code readability or object-oriented abstractions in C++.

Dynamic Memory Allocation Dynamic memory management and the ability/flexibility of creating/deleting objects at any time is one of the corner stones of object-oriented programming. Unfortunately, existing GPU memory allocators are either notoriously slow in serving allocations or miss key optimizations for structured data, leading to poor data locality and low memory bandwidth utilization when accessing data. For this reason, many GPU programmers still avoid dynamic memory management entirely and try to statically allocate all memory.

As the main research contribution of this thesis, we extended IKRA-CPP with DYNASOAR, a fully-parallel, lock-free, dynamic memory allocator. DYNASOAR improves the usage of allocated memory with an SOA data layout and achieves low memory fragmentation through efficient management of free and allocated memory blocks with lock-free, hierarchical bitmaps. Contrary to other state-of-the-art allocators, our design is heavily based on atomic operations, trading raw (de)allocation performance for better overall application performance. In our benchmarks, DYNASOAR achieves a speedup of SMMO application code of up to 3x over state-of-the-art allocators. Moreover, DYNASOAR manages heap memory more efficiently than other allocators, allowing programmers to run up to 2x larger problem sizes with the same amount of memory.

Memory Defragmentation In a system with dynamic memory allocation, the efficiency of an SOA data layout depends heavily on low memory fragmentation. If data is fragmented, more vector accesses are needed for the same number of bytes, reducing the benefit of SOA. To further improve memory fragmentation and vectorized access, we extended DynaSOAr with CompactGpu, an incremental, fully-parallel, in-place memory defragmentation system. CompactGpu defragments the heap by merging partly occupied memory blocks. We developed several implementation techniques for memory defragmentation that are efficient on SIMD/GPU architectures, such as finding defragmentation block candidates and fast pointer rewriting based on bitmaps.

Summary In this thesis, we show that, contrary to common belief, object-oriented programming is feasible on GPU without sacrificing performance, if the programming system provides well-designed abstractions and programming models that can be implemented efficiently on GPUs. SMMO is the main programming model throughout this thesis and we show through various examples that it is sufficiently expressive and that it can run efficiently on GPUs.

While this thesis focuses mostly on memory access performance, future work could focus more on control flow divergence or advanced features of object-oriented programming such as virtual function calls or advanced modularity constructs such as multiple inheritance. We also plan to integrate Ikra-Cpp into Ikra-Ruby in the future, so that programmers can develop SMMO applications in a high-level programming language.

Outline This thesis is organized as follows. Chapter 2 gives an overview of the execution model and memory hierarchy of GPUs, as well as a brief summary of the object-oriented programming model and its challenges on GPUs. Chapter 3 investigates how object orientation can be used to express GPU parallelism. In particular, we describe the APIs of Ikra-Ruby and Ikra-Cpp. Chapter 4 shows how to optimize the memory access of Ikra-Ruby and Ikra-Cpp applications with kernel fusion and with a Structure of Arrays (SOA) data layout. Chapter 5 presents the design and implementation of the DynaSOAr dynamic memory allocator. Chapter 6 describes how memory that was dynamically allocated with DynaSOAr can be defragmented with CompactGpu to improve the efficiency of the SOA layout and to lower the overall memory consumption. Chapter 7 illustrates the design and implementation of a various SMMO applications from different domains, which underlines the importance of the SMMO programming model. Finally, Chapter 8 concludes this thesis.

Chapter 2

Background

This chapter gives an overview of the architecture and execution model of recent NVIDIA GPUs (Section 2.1), as well as a brief background on object-oriented programming (Section 2.2) and the Structure of Arrays (SOA) data layout (Section 2.3).

Contents

2.1 GPU Execution Model

Graphics processing units (GPUs) have become more and more popular in the last decade. While in the early years, GPUs were mainly used for producing visual output on a computer monitor and to accelerate graphical computations of computer games, they are now well established in a variety of other areas such as high-performance computing or machine learning.

The three main GPU manufacturers at present are AMD, Intel and NVIDIA [160]. We are focusing on NVIDIA architectures and the NVIDIA CUDA programming model in this work, as NVIDIA GPUs are most widely used in high-performance computing according to the *TOP500* list of supercomputers [38]. Since most GPU architectures follow similar design principles, we expect that our findings also apply to other architectures.

NVIDIA GPUs can be programmed with CUDA and OpenCL. Both are C++ dialects. While OpenCL works with a variety of GPU architectures, CUDA is specific to NVIDIA GPUs and provides access to more fine grained control flow, memory and synchronization primitives that may not be available on other architectures [163]. On

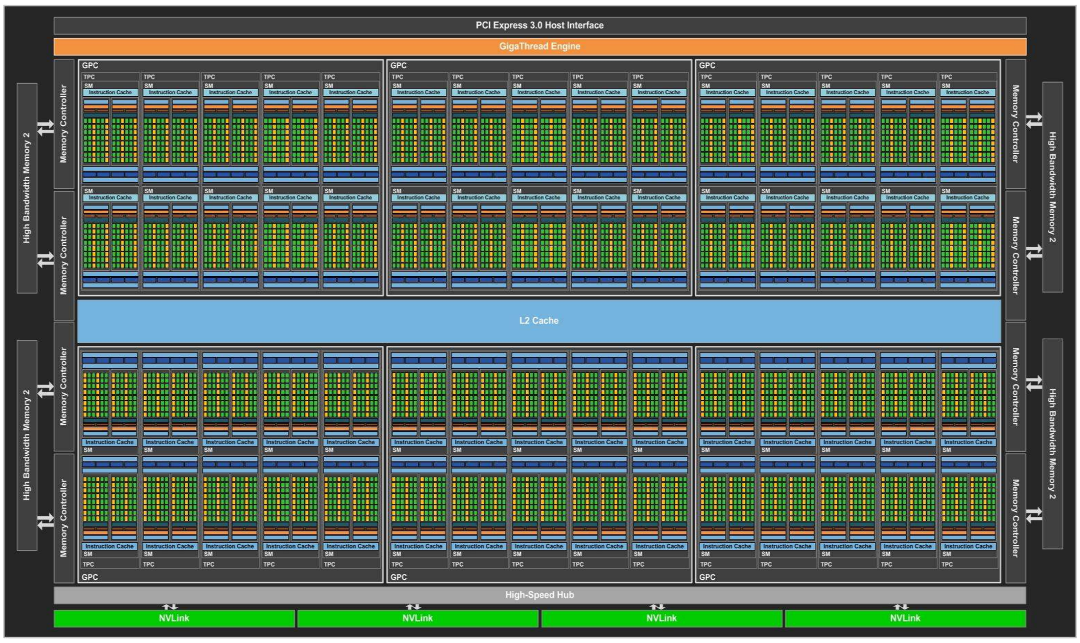


FIGURE 2.1: Architecture of NVIDIA GP100 (Source: NVIDIA Tesla P100 Whitepaper [35])

NVIDIA architectures, CUDA code has also been shown to be more performant than comparable OpenCL code [99, 61, 127].

2.1.1 Parallel Execution

Figures 2.1 and 2.2 show the hardware architecture (*SM block diagram*) of an NVIDIA GP100 GPU (Pascal architecture)[1]. This GPU consists of 60 *streaming multiprocessors* (SMs). Each SM has 64 *CUDA cores* (green boxes), amounting to a total of 3840 CUDA cores. Every group of 32 CUDA cores (*warp*) has a *warp scheduler* that schedules processor instructions for all 32 cores. Each GP100 SM can also be seen as a dual-core processor, with each core operating on vector registers that hold 32 scalars (128 bytes).

GPUs are commonly referred to as *massively parallel SIMD architectures*. However, they actually have three different ways of achieving parallelism.

- **SIMD:** All 32 CUDA cores of a warp execute the same processor instruction as determined by their warp scheduler. Therefore, GPUs are *Single-Instruction Multiple-Data* (SIMD) architectures. They achieve parallelism by executing the same instruction on a vector register (*multiple data*).

- **MIMD:** Each warp has its own warp scheduler, so different warps can execute different instructions. Therefore, GPUs also have *Multiple-Instruction Multiple-Data* (MIMD) parallelism. (Most CPU systems are MIMD architectures.)

- **Instruction-Level Parallelism (ILP):** Warp schedulers can issue two instructions per processor cycle (*dual-issue*), e.g., an arithmetic instruction and a memory access instruction. Details vary from architecture to architecture. On older NVIDIA archiectures, ILP was necessary to achieve peak performance [192, 193], but on Pascal *single-issue* can fully utilize all CUDA cores [42].

[1]We used a GP102-450-A1 GPU (TITAN Xp; Pascal architecture) for most of our benchmarks. NVIDIA has not published a whitepaper for this GPU, so there is no published SM block diagram for this GPU. However, its architecture is similar to a GP100 GPU.

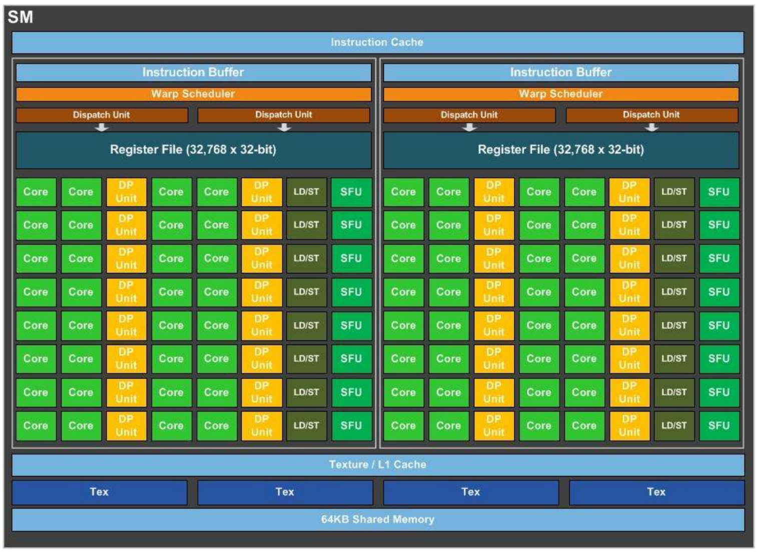


FIGURE 2.2: Architecture of a single NVIDIA GP100 SM (Source: NVIDIA Tesla P100 Whitepaper [35])

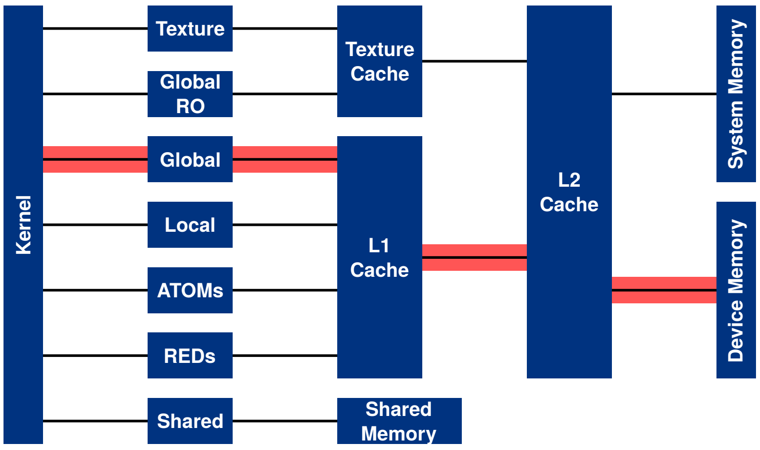


FIGURE 2.3: Memory access paths (Source: NVIDIA Visual Profiler)

SIMD and MIMD parallelism can be exploited with thread-level parallelism (TLP) in CUDA[2]. There are no dedicated abstractions for instruction-level parallelism (ILP) in CUDA. The scheduling of instructions is up to the compiler and the hardware. However, programmers can sometimes achieve better ILP by manually unrolling loops [192].

2.1.2 Memory Hierarchy

Recent NVIDIA GPUs have five main types of memory. The size of each memory type can differ among GPU architectures and models. In this work, we are focusing mainly on *device memory* (red access path in Figure 2.3).

In the following list, we briefly describe each memory type and report memory sizes for an NVIDIA TITAN Xp GPU, which we used for most of our benchmarks. Memory types are sorted from slowest to fastest.

[2]This is a key difference from C++, where SIMD parallelism is not expressed via threads but via automatic vectorization and SIMD intrinsics (manual vectorization).

- **Device Memory:** This is the largest type of memory. It resides in the GPU's DRAM. Device memory accesses are always routed through the L1/L2 caches [34] (Figure 2.3). The L1 cache acts as a *coalescing buffer*: Data is accumulated in the L1 cache before it is delivered to a warp [43]. A memory request of a warp is potentially broken down into multiple requests, one per cache line (i.e., a memory instruction may be executed multiple times) [39]. Our TITAN Xp GPU has 12 GB of device memory. The latency of device memory access can be as high as 1,000 cycles in case of an L1/L2 cache miss [95]. In CUDA, device memory is referred to as *global memory*.

- **Level 2 Cache:** All device memory accesses are cached in the L2 cache, which is 3 MB large on our TITAN Xp GPU. The L2 cache is shared by all SMs and has a cache line size of 128 bytes. Each cache line consists of four 32 byte segments, which can be read/written independently. The latency of an L2 hit is around 220 cycles [95].

- **Level 1 Cache:** By default, all device memory accesses are cached in the L1 cache. This behavior can be changed with compiler flags [43]. Our TITAN Xp GPU has 48 KB (or 24 KB[3]) of L1 cache per SM. The L1 cache line size is 128 bytes. Data is loaded from the L1 cache in 32 byte transactions, but written in 128 byte transactions [95]. L1 caches of different SMs are incoherent. The latency of an L1 hit is around 80 cycles [95].

- **Shared Memory:** Until the NVIDIA Kepler architecture, L1 and shared memory used the same on-chip memory. In recent architectures, the shared memory is separate. Shared memory is the fastest kind of GPU memory that can be explicitly programmed/accessed in CUDA code. Our TITAN Xp GPU has 96 KB of shared memory per SM.

- **Registers:** Processor registers are the fastest kind of memory. On Pascal, each SM has 65,536 32-bit registers (256 KB). Since each SM has 64 CUDA cores, an SM can execute exactly two warps at a time. However, more than two warps may be able to *reside* in its register file, depending on the number of registers that each warp uses. To *hide* the latency of (mostly memory) instructions, the SM can then issue instructions from another warp that is not blocked by a dependent instruction (*latency hiding* [193]).

The CUDA Toolkit Documentations mentions two additional memory types. These types of memory are not relevant for our work and will not be mentioned outside of this section.

- **Local Memory:** If an SM does not have enough resources, it uses (thread-)local memory. For example, registers may be spilled to local memory. On Pascal, the number of registers is limited to 255 per thread, but with a large number of threads per block, an SM may run out of registers earlier. Local memory actually resides in device memory and is not a separate address space.

- **Constant Memory:** Memory that does not change throughout a kernel can be stored in constant memory, which is a small part of the device memory but cached by a separate *constant cache*. The size of the constant cache is not published, but believed to be 2 KB per SM on Pascal [95, 37]. Constant memory accesses cannot be coalesced. However, they can be broadcasted to other threads

[3]There is no officially published number and other work reports contradicting numbers.

in a warp, should those threads access the same constant memory address. Our TITAN Xp GPU has 64 KB of constant memory.

Details of memory access, the memory hierarchy and memory coalescing (Section 2.1.4) are constantly changing with new GPU architectures. With every new architecture, NVIDIA publishes a *Tuning Guide* for achieving good performance on that architecture. However, many architectural/hardware details remain undisclosed, so that often the only way of understanding *why* certain programming/memory access patterns achieve good performance is through exhaustive microbenchmarking, profiling and reverse engineering [206, 95, 136, 213]. Many architecture/hardware characteristics mentioned in this chapter were originally obtained in this way.

2.1.3 CUDA Programming Model

CUDA is an extension of C++. Programmers express thread-level parallelism with CUDA *kernels*. A kernel is a C++ function, annotated with the `__global__` keyword. When programmers *launch* a CUDA kernel, they have to specify the number of threads that should execute the kernel (C++ function). Inside the kernel, programmers can access special variables to determine a thread's ID, so that different threads run the same code but with different data.

In many cases, a thread ID is used as an index into an array of *jobs*. If the number of threads equals the number of jobs, threads can be directly (one-to-one) mapped to jobs: A thread t_i processes job i. However, CUDA limits the number of threads per kernel and, furthermore, using too many threads can cause runtime inefficiencies. If the number of jobs exceeds the number of threads, then each thread may have to process more than one job, typically with a *grid-stride loop* [80].

Besides the `__global__` keyword, functions and top-level variable declarations can be annotated with the `__device__` and/or `__host__` keyword. The former keyword instructs the compiler to generate GPU code for a function or to statically allocate the variable on *device* (GPU) in global memory. The latter keyword (default if no keyword specified), instructs the compiler to generate CPU code for a function or to statically allocate the variable on the *host* (CPU). In essence, `__global__` functions are like `__device__` functions but only the former ones can be invoked as kernels from CPU code. It is not possible to call `__device__` functions from `__host__` functions or vice versa.

Thread Hierarchy CUDA threads are organized in a hierarchy. Every thread belongs to one CUDA *block*. The size of a block (*block dimension*) can be between 1 and 1024 threads[4]. The number of blocks (*grid dimension*) can be between 1 and $2^{31} - 1$, so a kernel can have up to billions of CUDA threads. Programmers have to specify block/grid dimensions as kernel launch parameters. Inside a CUDA kernel, programmers can access four special variables.

- `threadIdx.x`: The ID of the current thread within the block.
- `blockDim.x`: The size/dimension of each block.
- `blockIdx.x`: The ID of the block of the current thread.
- `gridDim.x`: The number of blocks (dimension of the *grid*).

[4]We are focusing on 1D blocks and grids.

The ID of a thread and the total number of threads is then calculated as follows.

$$tid = blockIdx.x \cdot blockDim.x + threadIdx.x \qquad (\textit{thread ID})$$

$$gridDim.x \cdot blockDim.x \qquad (\textit{number of threads})$$

Besides blocks, there is a second thread hierarchy level that cannot be directly controlled programmers. Every consecutive group of 32 threads of a block (thread IDs $[32 \cdot i; 32 \cdot (i+1))$ is called a *warp*.

$$warp\_id = \left\lfloor \frac{threadIdx.x}{32} \right\rfloor \qquad (\textit{warp ID of a thread within a block})$$

$$lane\_id = threadIdx.x \ \% \ 32 \qquad (\textit{lane ID of a thread})$$

CUDA warps map to physical warps, so threads of a warp execute the same instructions at the same time (SIMD parallelism). The threads within a warp are mapped to *lanes* with IDs between 0 and 31. CUDA provides warp-level primitives for exchanging (shuffling) values between different lanes of a warp [123]. Such primitives are not easy to use and require a certain amount of CUDA programming experience, but they are often necessary to achieve the best performance [47].

Block/Warp Scheduling Every CUDA block runs on a single SM and cannot be split to run on multiple SMs. If SM resources allow (as outlined below), multiple blocks may reside on a single SM. If the number of CUDA cores of an SM is less than the number of resident threads (i.e., block dimension times number of resident blocks per SM), then not all threads can run simultaneously and the SM has to context-switch between warps. This is usually the case so that memory latency can be hidden [193]. Among all resident warps (i.e., warps of all resident blocks), the warp scheduler selects a warp that is ready to execute (e.g., not blocked by a memory operation), until eventually the entire block finished executing. The *block scheduler* then selects a new CUDA block for execution (if there are more blocks). Context-switching between warps is extremely fast on GPUs because the state of each warp is kept in the register file, which is much larger on GPUs compared to CPUs, so that registers do not have to be swapped to device memory.

The maximum number of resident blocks per SM depends on the computate capability version, the resource requirements of each block and the available resources of an SM, as described in the CUDA Toolkit Documentation. On our TITAN Xp with compute capability version 6.1, those limitations are as follows.

- The total number of registers of all resident blocks must fit into the register file. I.e., all resident blocks together cannot use more than 256 KB of registers.
- The total amount of requested shared memory of all resident blocks must not exceed the SM's shared memory, which is 96 KB.
- No more than 32 blocks can reside on one SM.
- No more than 64 warps can reside on one SM. This limits the maximum number of resident threads per SM to 2048.

The NVIDIA Visual Profiler can be used to analyze such constraints. In the past, NVIDIA also provided an *Occupancy Calculator* Excel sheet.

Grid-Stride Loops CUDA programs with fewer threads than jobs typically process jobs with a grid-stride loop [80, 102]. In a grid-stride loop, a thread t_{tid} processes job *tid* and increments of n, where n is the total number of threads.

Listing 2.1: Example of a grid-stride loop

```
1 #define N 1000000
2 __device__ float data[N];
3
4 __global__ void kernel_increment_data() {
5   for (unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
6        i < N; i += blockDim.x * gridDim.x) { data[i] += 10.0f; }
7 }
8
9 int main() { kernel_increment_data<<</*gridDim.x=*/ 128, /*blockDim.x=*/ 128>>>(); }
```

Listing 2.1 shows an example of a grid-stride loop. Grid-stride loops often have better performance than a different assignment of jobs to threads because of better memory coalescing (Section 2.1.4). This is because the threads of a warp process jobs that are spatially local in memory and can thus be serviced with more efficient vectorized memory loads/stores. This listing also illustrates CUDA's kernel launch notation, which allows programmers to specify the number of blocks/threads (Line 9).

Warp Divergence CUDA warps are mapped to physical warps of an SM. Since physical warps only have one warp scheduler, all threads of a warp execute the same instructions. If the control flow of the threads of a warp diverges (e.g., because they enter different *if* branches), then the different control flow paths are executed sequentially until the control flow converges again. This phenomenon is called *warp divergence* (also called *thread divergence* or *branch divergence*).

There are no guarantees about the order in which divergent branches are executed or when the control flow reconverges again [123]. Warp divergence can make GPU programs significantly harder to reason about and is the main reason why locking is discouraged on GPUs.

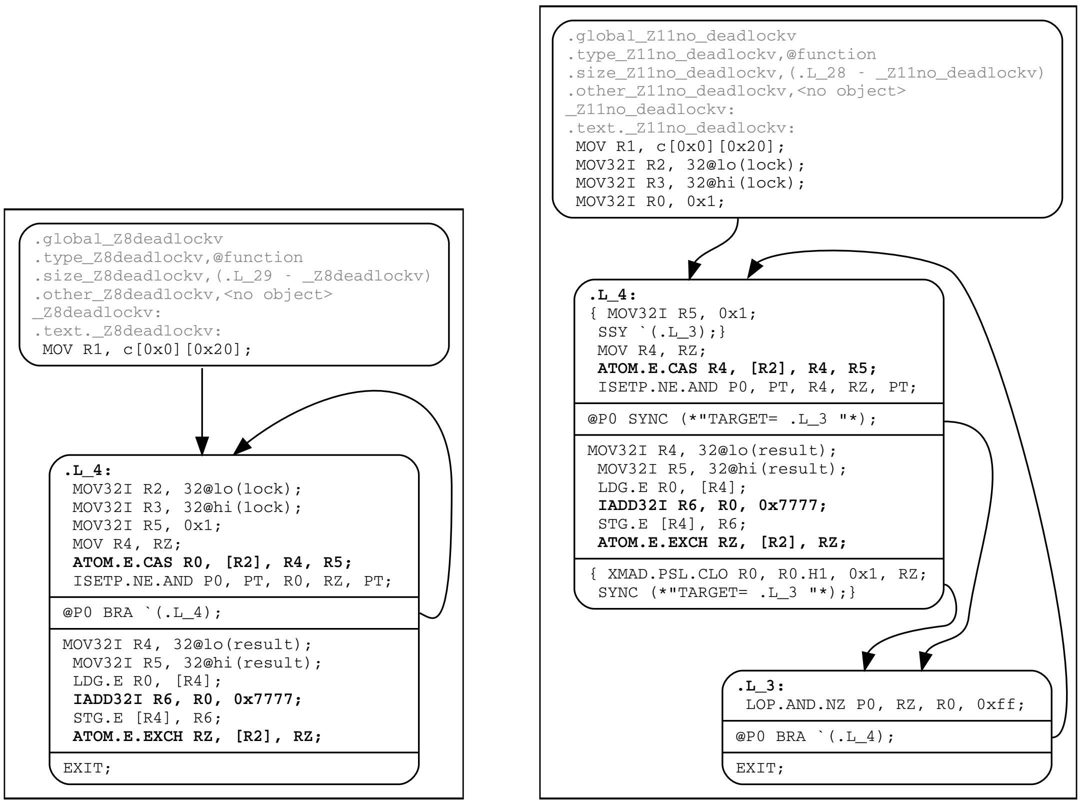
As an example, consider the two implementations of a critical section in Listing 2.2. Both implementations seem identical, but the first one deadlocks on our TITAN Xp GPU and the second one works [119][5]. In fact, whether either one of them deadlocks or not may vary among GPU architectures and compilers.

To understand why the first implementation deadlocks, we have to take a look at the basic block structure of the compiled PTX (Figure 2.4A). Each thread must aquire the lock in `L_4` to enter the critical section and then release the lock in the third segment of `L_4`, so that the next thread can enter the critical section. PTX instructions that start with an *at* character are *guarded* and conditionally executed (based on the result of the preceding instructions). The problem in (A) is that one thread acquires the lock, but then the other 31 unsuccessful threads jump back to the beginning of the loop. This is a control flow divergence and both control flows are executed sequentially. Unfortunately, our TITAN Xp happens to always schedule the jump of the unsuccessful threads before executing the critical section with the successful thread, so the lock is never released.

[5]See also: `https://stackoverflow.com/questions/31194291/cuda-mutex-why-deadlock`.

LISTING 2.2: Implementation of a critical section

```
__device__ int lock = 0;
__device__ int result = 0; // Modified in critical section

__global__ void deadlock() {
  while (true) {
    if (atomicCAS(&lock, 0, 1) == 0) { // lock
      result += 0x7777;
      atomicExch(&lock, 0); // unlock
      break;
    }
  }
}

__global__ void no_deadlock() {
  bool continue_loop = true;
  while (continue_loop) {
    if (atomicCAS(&lock, 0, 1) == 0) { // lock
      result += 0x7777;
      atomicExch(&lock, 0); // unlock
      continue_loop = false;
    }
  }
}

int main() {
  deadlock<<<1, 32>>>(); // Run with 32 threads, 1 warp
}
```



```
.global_Z8deadlockv
.type_Z8deadlockv,@function
.size_Z8deadlockv,(.L_29 - _Z8deadlockv)
.other_Z8deadlockv,<no object>
_Z8deadlockv:
.text._Z8deadlockv:
 MOV R1, c[0x0][0x20];

.L_4:
 MOV32I R2, 32@lo(lock);
 MOV32I R3, 32@hi(lock);
 MOV32I R5, 0x1;
 MOV R4, RZ;
 ATOM.E.CAS R0, [R2], R4, R5;
 ISETP.NE.AND P0, PT, R0, RZ, PT;
@P0 BRA `(.L_4);
MOV32I R4, 32@lo(result);
 MOV32I R5, 32@hi(result);
 LDG.E R0, [R4];
 IADD32I R6, R0, 0x7777;
 STG.E [R4], R6;
 ATOM.E.EXCH RZ, [R2], RZ;
EXIT;
```

(a) With deadlock

```
.global_Z11no_deadlockv
.type_Z11no_deadlockv,@function
.size_Z11no_deadlockv,(.L_28 - _Z11no_deadlockv)
.other_Z11no_deadlockv,<no object>
_Z11no_deadlockv:
.text._Z11no_deadlockv:
 MOV R1, c[0x0][0x20];
 MOV32I R2, 32@lo(lock);
 MOV32I R3, 32@hi(lock);
 MOV32I R0, 0x1;

.L_4:
{ MOV32I R5, 0x1;
 SSY `(.L_3);}
 MOV R4, RZ;
 ATOM.E.CAS R4, [R2], R4, R5;
 ISETP.NE.AND P0, PT, R4, RZ, PT;
@P0 SYNC (*"TARGET= .L_3 "*);
MOV32I R4, 32@lo(result);
 MOV32I R5, 32@hi(result);
 LDG.E R0, [R4];
 IADD32I R6, R0, 0x7777;
 STG.E [R4], R6;
 ATOM.E.EXCH RZ, [R2], RZ;
{ XMAD.PSL.CLO R0, R0.H1, 0x1, RZ;
 SYNC (*"TARGET= .L_3 "*);}

.L_3:
 LOP.AND.NZ P0, RZ, R0, 0xff;
@P0 BRA `(.L_4);
EXIT;
```

(b) Without deadlock

FIGURE 2.4: Basic block diagram for critical sections

Now consider (B). Before jumping back to the beginning of the loop, the compiler places a `SYNC` instruction, which converges threads after a conditional branch. If control flow is divergent, the GPU schedules the other not yet executed branches before continuing with the following instructions. Therefore, the successful thread now has a chance to execute the critical section and to release the lock.

Unfortunately, we do not know *why* the compiler generates a `SYNC` instruction in (B) but not in (A). This is unspecified and may even change with different CUDA Toolkit versions [57]. Interestingly, if we invert the boolean flag `continue_loop`[6], then our CUDA compiler generates the same deadlock-prone PTX as in (A). This example illustrates why locking within a warp is problematic on GPUs[7]. We avoid locking in our work as much as possible and resort to lock-free algorithms.

Consistency Model CUDA has a weak memory consistency model. In particular, global (device) memory accesses are not necessarily sequentially consistent. E.g., if a thread t_1 writes two variables a and b, then another thread t_2 is not guaranteed to see the changes of a and b in the same order. Moreover, there are no guarantees that t_2 will see the changes of a and b at all. Such weak behaviors are in part due to incoherent L1 caches. For example, the old value of a may still be in t_2's L1 cache while b is a cache miss. In such a case, t_2 would read the old value of a but the new value of b. Unfortunately, weak GPU behavior is not well documented by GPU manufacturers, so that researchers have to resort to reverse engineering or litmus testing to fully understand GPU concurrency [10, 173].

Weak semantics on GPUs and how to avoid them has been discussed in previous work [171]. There are four main ways of addressing weak behavior in CUDA.

- A **thread fence** (CUDA `__threadfence()`) ensures that all global memory changes of the current thread before the thread fence become visible to other threads before global memory changes after the thread fence become visible. After executing the thread fence, the L1 cache of the executing thread is guaranteed to be consistent with the L2 cache.

- **Atomic operations** bypass the L1 cache and go straight to the L2 cache. Atomics can ensure sequential consistency and visibility of changes if data is both read/written with atomics. Atomic instructions have a higher latency than comparable non-atomic instructions, but they became considerably faster with recent GPU architectures [69, 95].

- The CUDA/C++ `volatile` **qualifier** indicates that a memory address may be concurrently read/written by another thread. It disables certain optimizations (such as keeping a value in a register) and always causes a memory read/write upon data access, bypassing the L1 cache.

- The **L1 cache** can also be entirely **turned off** with a compiler flag: `-Xptxas -dlcm=cg`. Note that this is not equivalent to making every memory access a `volatile` access, since the `volatile` qualifier also deactivates other optimizations (such as caching data in registers).

In this thesis, we develop various lock-free algorithms. Their implementation depends heavily on atomic operations, which are sequentially consistent. Atomic operations and retry loops are common patterns in lock-free algorithms [26].

[6] Initialize to `false`, check for `!continue_loop` in the loop condition, set to `true` in the critical section.
[7] If only one thread per warp acquires a lock, locking is unproblematic.

2.1.4 Memory Coalescing

Many GPU applications are memory bound and for such applications, accessing device memory is the biggest bottleneck. *Memory coalescing* is a fundamental optimization of the GPU memory controller that combines multiple memory requests into a smaller number of physical memory transactions, if certain conditions are met.

- This optimization is specific to **device memory** (i.e., CUDA global memory).
- Only memory accesses (reads/writes) from the **same warp** can be combined.
- Memory addresses must be properly aligned and fall into an L1/L2 **cache line**.

The rules of memory coalescing differ among different NVIDIA compute capability versions. With compute capabilities between 3.x and 5.x, the NVIDIA CUDA C Programming Guide states that hits in the L1 cache can be coalesced into 128-byte transactions [39]. I.e., if the threads of a warp access memory locations on the same L1 cache line (L1 cache lines are 128 bytes long and 128-byte aligned), then these accesses are serviced by one physical 128-byte transaction. Starting from compute capability 6.x (Pascal), the NVIDIA Pascal Tuning Guide states that the L1 cache services loads (but not stores) at a granularity of 32 bytes [43], so memory loads can only be coalesced into smaller 32-byte transactions[8].

In the worst case, if each thread of a warp loads a 32-bit word and those memory accesses cannot be coalesced, then each thread's access generates a 32-byte transaction, which increases the amount of memory transfer by 8 [40], compared to one perfectly coalesced 128-byte transaction (CC 3.x-5.x) or four perfectly coalesced 32-byte transactions (CC 6.x).

Memory accesses of a warp that hit in the L2 cache and belong to the same L2 cache line, are coalesced into 32-byte transactions [39]. Whether these memory addresses must correspond to the same 32-byte L2 cache line segment is not specified, but this is likely the case.

All device memory is accessed through the L1/L2 caches. If a memory load does not hit in the L1/L2 caches, then the data must first be loaded from the GPU's DRAM into the L2 cache (and maybe L1 cache). The DRAM is accessed with 32-byte transactions. In the worst case, if the uncached 32-bit word accesses of each thread in a warp cannot be coalesced, then each 32-bit word access triggers a 32-byte DRAM transaction, which increases the amount of memory transfer by 8.

Memory coalescing is one of the most fundamental optimizations. The CUDA C Best Practices Guide puts a *high priority note* on coalesced access [36] and programmers should try to achieve good coalescing before trying any other optimizations.

The details of memory access and memory coalescing are changing with every new GPU architecture. While, until now, the number of generated memory transactions served as a good mental model to explain why memory coalescing increases performance, this no longer seems to be the case with the most recent architectures[9]. In fact, NVIDIA removed certain metrics and performance counters from their profiling tools, so that those values are no longer exposed to programmers. While we can no longer clearly explain *how* the hardware optimizes certain memory access patterns, the basic CUDA programming rules for achieving good memory coalescing and good memory bandwidth utilization fortunately remain the same.

[8]This is presumably to avoid *overfetching*, which required disabling the L1 cache for global memory access before Pascal.

[9]See also: `https://stackoverflow.com/questions/56142674/memory-coalescing-and-nvprof-results-on-nvidia-pascal`.

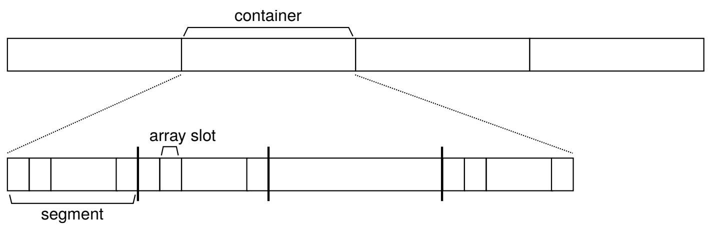


FIGURE 2.5: Array layout of memory coalescing experiment

Memory Coalescing vs. Vectorized Access Memory coalescing on GPUs is similar to vectorized memory access on CPUs but differs in one crucial aspect: Memory coalescing is a run-time optimization of the memory controller, whereas vectorization is a compile-time optimization[10].

On CPU systems, there are two main techniques for utilizing SIMD parallelism: Either the programmer manually vectorizes the code with SIMD intrinsics, which is tedious. Or the compiler auto-vectorizes code as part of an optimization pass. Both techniques require the programmer/compiler to fully understand the memory access pattern. Otherwise, the code cannot be vectorized. The resulting vectorized assembly code contains vector load/stores such as `movaps`, which, given a memory address, loads four packed (consecutive) floats (128 bits) into an SSE vector register.

On GPU systems, the memory controller coalesces memory accesses by analyzing the requested memory addresses at runtime. There is nothing to do for the compiler. The resulting PTX assembly code contains scalar loads/stores such as `ld.global`, which, given one memory address per thread, performs a memory load for each thread in the warp. The memory controller analyzes the memory addresses and determines the number of times that this instruction has to be repeatedly executed such that all memory load requests in the warp can be fulfilled with 128-byte transactions.

2.1.5 Memory Coalescing Experiment

To analyze the exact benefit of memory coalescing on a TITAN Xp GPU, we implemented a simple benchmark that increments every value of a large 32-bit floating point numbers array. The size of the array is 8 GB. We measured the runtime performance with different assignments of array slots (jobs) to CUDA threads.

Experiment Setup The array of size (number of elements) N is divided into $num_C \in \{1, 2, 4, 8, 16, 32\}$ containers of equal size, where num_C is a configurable parameter. Each container is divided into segments of equal size (number of elements) $size_S$, where $size_S$ depends on the parameter num_C (Figure 2.5).

$$size_S = \frac{32}{num_C} \qquad \textit{(segment size)}$$

For the moment, let us assume that the number of array slots N is equal to the number of GPU threads. We assign CUDA threads to array slots in such a way that every warp is processing one segment per container[11]. For example, for $num_C = 4$, the threads of a warp are assigned to one segment (size 8) per container. These

[10]See also: `https://stackoverflow.com/questions/56966466/memory-coalescing-vs-vectorized-memory-access`

[11]Recall that the warp size is 32. Moreover, $size_S \cdot num_C = 32$.

segments are in totally different memory areas, so at least four memory transactions are required for a warp to load all assigned array values. By increasing num_C, we can increase the degree of *scattering* and thus the number of required memory transactions. In the best case ($num_C = 1$), each warp is accessing only consecutive array slots, which should result in perfect memory coalescing and best performance.

Access Pattern Details Let *pos* be the array slot index that a given thread t_{tid} is processing. This index is the sum of three parts: The offset into the array at which the respective container begins, the offset into the container at which the respective segment begins and the offset of the respective array slot within the segment.

$$pos(tid) = array\_offset(tid) + container\_offset(tid) + segment\_offset(tid) \qquad (array\ slot)$$

The size of a container $size_C$ depends on the number of containers. All containers have the same number of array elements.

$$size_C = \frac{N}{num_C} \qquad (container\ size)$$

Each warp consists of 32 threads. Within a warp, threads are numbered from 0 to 31 (*lanes*). Containers are divided equally among lanes. $size_S$ is the number of lanes per container. For example, container 0 is processed by lanes 0 through $size_S - 1$ of all warps. The offset of the container for thread t_{tid} is then computed as follows.

$$\begin{aligned} array\_offset(tid) &= \left\lfloor \frac{lane\_id(tid)}{size_S} \right\rfloor \cdot size_C \qquad (array\ offset) \\ &= \left\lfloor \frac{tid \ \% \ 32}{32 \ / \ num_C} \right\rfloor \cdot \frac{N}{num_C} \end{aligned}$$

Within a container, each segment is processed by the $size_S$ threads of each warp. Segment i is processed by warp i. The offset of the segment for thread t_{tid} is computed as follows.

$$\begin{aligned} container\_offset(tid) &= warp\_id(tid) \cdot size_S \qquad (container\ offset) \\ &= \left\lfloor \frac{tid}{32} \right\rfloor \cdot \frac{32}{num_C} \end{aligned}$$

Finally, among the $size_S$ threads per segment, we assign one thread to each array slot. The offset of the array slot within the segment for t_{tid} is computed as follows.

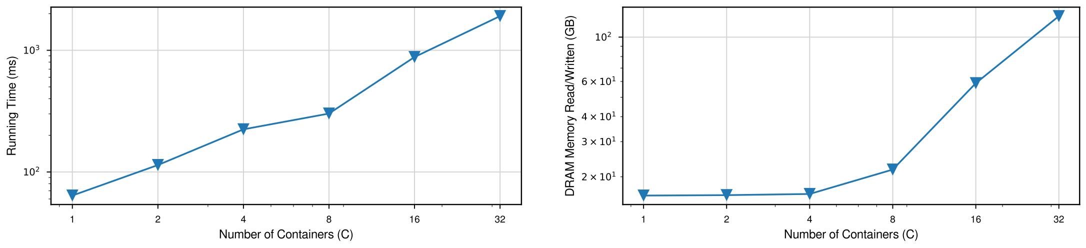
$$\begin{aligned} segment\_offset(tid) &= tid \ \% \ size_s \qquad (segment\ offset) \\ &= tid \ \% \ \frac{32}{num_c} \end{aligned}$$

Listing 2.3 shows the source code of the benchmark. Threads process array elements with a grid-stride loop, because spawning one thread per array slot would be too much overhead. This does not affect memory coalescing.

Results Figure 2.6 shows the results of the benchmark. We ran the benchmark with six different parameters $num_C \in \{1, 2, 4, 8, 16, 32\}$ and with a disabled L1 cache.

LISTING 2.3: Implementation of memory coalescing experiment

```
__global__ void kernel(float* volatile data, unsigned int N, unsigned int C) {
  for (unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
       tid < N; tid += blockDim.x * gridDim.x) {
    unsigned int index = ((tid % 32) / (32 / C)) * (N / C) // array offset
                       + (tid / 32) * (32 / C) // container offset
                       + tid % (32 / C); // segment offset
    data[index] += 8.0f; // 1 global memory read, 1 global memory write.
  }
}

int main() {
  float* data;
  cudaMalloc(&data, sizeof(float) * 2ULL * 1024 * 1024 * 1024);

  kernel<<<512, 512>>>(data, /*N=*/ 2ULL * 1024 * 1024 * 1024, /*C=*/ 4);
  cudaDeviceSynchronize();
}
```



(A) Running time (B) DRAM bytes read/written

FIGURE 2.6: Memory coalescing experiment results (lower is better)

Higher values of num_C increase the degree of scattering, which reduces memory coalescing and increases the running time (Subfigure A). The worst-case running time of $num_C = 32$ is 29.8 times slower than the best-case running time of $num_C = 1$.

Subfigure B shows the number of bytes read/written from the GPU's DRAM. For $num_C \in \{1, 2, 4\}$, this value is almost the same because the DRAM is always accessed in 32-byte transactions. Only for higher values of num_C does the number of bytes increase. In the worst case ($num_C = 32$), 127.4 GB are read/written from the DRAM. In this case, every 4-byte read/write access was serviced by a 32-byte transaction.

$$2 \cdot \frac{8 \cdot 1024 \cdot 1024 \cdot 1024 \text{ byte}}{4 \text{ byte}} \cdot 32 \text{ byte} = 128 \text{ GB} \qquad (\textit{exp. \#bytes accessed in DRAM})$$

The measured number of DRAM bytes is a bit less than 128 GB because a few transactions hit in the in the L2 cache and profiler metrics are not 100% accurate.

The results of this experiment highlight the importance of coalesced memory access and show that significant running time speedups can be achieved by optimizing memory access. Our work focuses largely on achieving good memory coalescing with object-oriented programming on GPUs.

2.2 Object-oriented Programming

Object-oriented programming (OOP) is a widely used programming paradigm. As of May 2019, 16 of the 20 most widely used programming languages (according to the TIOBE index[12]) support object-oriented programming.

Objects In well designed programs, objects represent abstract or real-world entities. This can help programmers in building a good mental model of the data structures and code interactions in a program. An object in OOP consists of three fundamental parts.

- **State:** The state of an object defines its properties. In pure OOP, the state of an object is *private* and cannot be directly accessed by other objects.
- **Identity:** Even if two different objects have the same state, they have different identity. Identity can be seen as a special property that is different for each object.
- **Behavior:** In pure OOP, objects communicate by exchanging messages. Upon receipt, an object may process the message by executing code, which may in turn read/modify the object's state. Mainstream languages follow the notion of *method calls* instead of *message sends*.

Object-oriented programming is widely used in academia and industry. Some of its main advantages are good abstraction, encapsulation, modularity, good code understandability and high programmer productivity [148].

2.2.1 Class-based Object-oriented Programming

The most prominent variant of object-oriented programming is *class-based object-oriented programming*. Objects are instances of classes, which define the state (fields) and behavior (methods) of objects. CUDA and OpenCL are extensions of C++, a class-based, object-oriented programming language.

Class-based, object-oriented programming languages can increase code reuse and abstraction through subclassing/inheritance. In typed languages, a subclass relationship usually creates a subtype relationship. This means that, with respect to type safety, an object of a superclass can be substituted with an object of a subclass (*Liskov Substitution* [124]). If a subclass *overrides* a method of a superclass, the programming system should dispatch to the correct method based on an object's runtime type as opposed to its static type (*virtual function call*).

C++ Object Layout The layout of objects in memory depends on the programming language/compiler and the platform's application binary interface (ABI) [150]. In C++, field values of an object are arranged in memory in the same order in which they are declared in the struct/class. If a struct/class inherits from another struct/class, then inherited fields come first[13]. Furthermore, if the struct/class has at least one virtual member function, then the object starts with a pointer to the virtual method table (*vtable*), which contains pointers to all virtual function implementations. The ABI defines the size and alignment of primitive types. For example, on recent NVIDIA

[12]`https://www.tiobe.com/tiobe-index/`

[13]We are focusing on single inheritance. There are more complex rules for multiple inheritance.

```
class A {
  int f1; double f2; char f3;
  void foo();
  virtual void bar();
};

class B : public A {
  int f4;
  virtual void bar() override;
  virtual void qux();
};
```

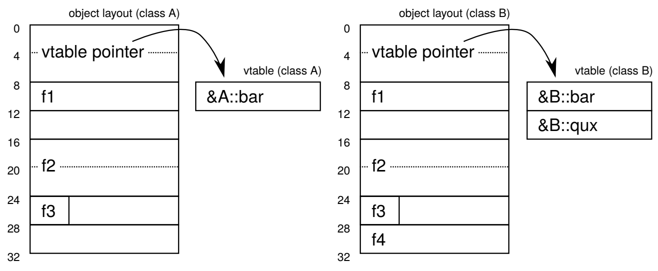


FIGURE 2.7: Example: C++ object layout on NVIDIA GPUs

GPUs, `char` is 1 byte, `int` is 4 bytes and `double` is 8 bytes. In addition, values of these types must be aligned to their respective byte size [41].

Figure 2.7 shows an example with two classes, where class `B` inherits from class `A`. Due to alignment, objects of both classes are 32 bytes in size, even though class `B` has an extra field. In particular, class `A` has a size of 32 bytes instead of 25 bytes. This is to ensure that if multiple `A` objects are allocated in an array, field values of each object are properly aligned.

2.2.2 Problems of Object-oriented Programming on GPUs

Even though object-oriented programming has a wide range of applications in high-performance computing [15, 98, 11, 46, 29] it is often avoided due to bad performance [151]. In C++, there are four main performance problems with object-oriented programming on GPUs.

- **Data Layout:** While the object-oriented programming paradigm does not dictate any particular layout of objects in memory, most systems/compilers store each object in one contiguous block of memory. Such a layout may not be ideal for execution on GPUs. To achieve good memory access performance, GPU programmers have to tune data layouts to maximize memory coalescing and L1/L2 cache performance [169], but current GPU languages such as CUDA and OpenCL do not allow programmers to change the layout of objects.

- **Dynamic Memory Allocation:** The ability/flexibility of creating and deleting objects at any time is one of the corner stones of object-oriented programming. CUDA provides a dynamic memory allocator (`malloc`/`free` interface) in GPU code, but this allocator is notoriously slow and unreliable [175]. Due to the massive number of threads and expensive inter-thread communication/synchronization, it is difficult to design efficient, dynamic memory allocators for GPUs.

- **Virtual Function Calls:** GPU compilers aggressively inline functions, because jumps and function calls are generally more expensive on GPUs [209, 212]. In C++, virtual functions are typically compiled into a jump to an address in the virtual method table (*vtable*). However, such jumps cannot be inlined by the compiler and are by a factor of 10x slower than regular function calls [110]. Moreover, virtual function calls can lead to warp divergence.

- **64-Bit Pointers:** Memory pointers on recent GPU architectures are 64 bits long. Therefore, if an object stores a pointer to another object, an 8-byte field is required. Without object-oriented abstractions (i.e., object pointers), a 4-byte integer ID may be sufficient, so object-oriented programming can indirectly increase the memory usage. Similar overheads have been observed in Java applications when switching from a 32-bit address space to a 64-bit address space. Related work proposed *pointer compression* to refer to objects with 32-bit values [190]. Similar techniques could be adopted to optimize GPU programs.

Related work describes techniques for optimizing the last two problems. In the course of this work, we are addressing the first two problems, which are largely unsolved and the main source of slowdowns: Data layout and dynamic memory allocation.

Optimizing Virtual Function Calls Virtual function calls are expensive because they are usually not inlined and compiled into jump/call instructions. Such jumps are particular expensive on GPUs, which were originally designed for highly regular control flow and neither have a branch target predictor nor execute code speculatively. We briefly review two techniques for inlining virtual function calls in C++:

- **Switch-Case Statements:** Instead of generating a vtable pointer lookup and a jump, generate an exhaustive switch-case statement that dispatches to the correct method implementation based on the receiver's type (example in Listing 7.13). This is possible because C++ is a statically-typed language and GPU programs are usually not separately compiled (i.e., one compilation unit). Therefore, the compiler knows all types that a receiver can have at runtime. Related work describes an instrumentation-based variant of this technique that works even in the absence of full compile-time type information [8].
- **Expression Templates [188]:** This advanced C++ template metaprogramming technique encodes the structure of a computation in its type. It is heavily utilized in linear algebra libraries [89].

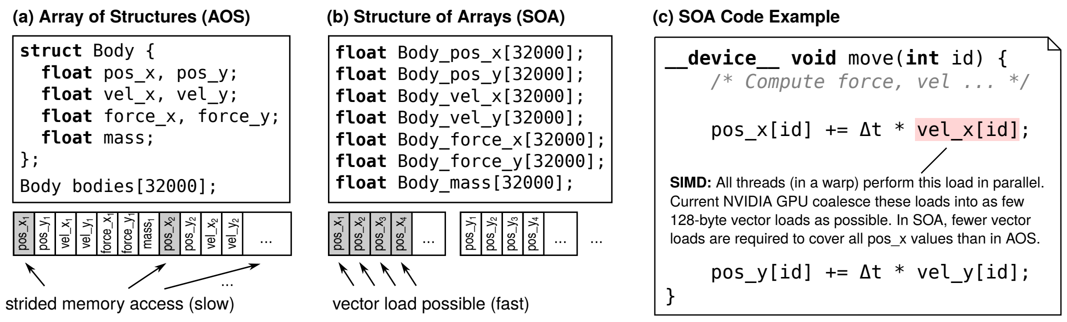
As our focus is on efficient memory access, we do not attempt to optimize virtual function calls in this thesis. Instead, we hand-write the respective switch-case statements whenever we encounter a virtual function call in our examples.

2.3 Array of Structure (AOS) vs. Structure of Arrays (SOA)

Structure of Arrays (SOA) and Array of Structures (AOS) describe memory layouts for a fixed-size set of objects [20]. In AOS, the standard layout of most systems, objects are stored as contiguous blocks of memory. In SOA, all values of a field are stored together (Figure 2.8).

Running Example As an example, we consider a simplified n-body simulation (full example in Section 7.1). Every body in the simulation is an object with fields for 2D position, 2D velocity, mass, etc. Listing 2.4 shows an excerpt of the source code in AOS layout. All bodies are stored in an array of `Body` base type, where `Body` is a C++ class/struct; thus the name *Array of Structures*. Objects are constructed with C++'s placement-new syntax, which runs the constructor on a given memory address.

Alternatively, instead of allocating objects in an array, objects could be dynamically allocated on the heap. While objects were still stored as contiguous blocks of memory, there would be no guarantee that the dynamic memory allocator places them next to


(a) Array of Structures (AOS)
(b) Structure of Arrays (SOA)
(c) SOA Code Example
pos_x1 pos_y1 vel_x1 vel_y1 force_x1 force_y1 mass1 pos_x2 pos_y2 vel_x2 vel_y2 ...
pos_x1 pos_x2 pos_x3 pos_x4 ... pos_y1 pos_y2 pos_y3 pos_y4 ... ...
strided memory access (slow)
vector load possible (fast)

```
struct Body {
  float pos_x, pos_y;
  float vel_x, vel_y;
  float force_x, force_y;
  float mass;
};
Body bodies[32000];
```

```
float Body_pos_x[32000];
float Body_pos_y[32000];
float Body_vel_x[32000];
float Body_vel_y[32000];
float Body_force_x[32000];
float Body_force_y[32000];
float Body_mass[32000];
```

```
__device__ void move(int id) {
    /* Compute force, vel ... */

    pos_x[id] += Δt * vel_x[id];

    pos_y[id] += Δt * vel_y[id];
}
```

SIMD: All threads (in a warp) perform this load in parallel. Current NVIDIA GPU coalesce these loads into as few 128-byte vector loads as possible. In SOA, fewer vector loads are required to cover all pos_x values than in AOS.

FIGURE 2.8: N-body simulation in AOS and SOA data layout

each other in an array-like form. Therefore, such a layout is no longer AOS and has likely performance characteristics different from AOS.

Listing 2.5 shows the same program in SOA layout. The array of base type `Body` was replaced by seven arrays, one for every field; thus the name *Structure of Arrays*. We call these arrays *SOA arrays*.

2.3.1 Abstractions for Object-oriented Programming

At first sight, the SOA code is harder to read/understand than the AOS code. This is mainly because most programmers are familiar with object-oriented programming and its syntax/notation. However, even objectively, the AOS code does a better job at expressing the object-oriented design:

- **Abstraction and Encapsulation:** Values that belong together are defined together. In AOS, `pos_x`, `pos_y`, etc. are defined and encapsulated inside the scope of class `Body`. Furthermore, `Body`'s fields could be made private, such that they can only be accessed from within the class.

- **Type System:** In AOS, objects can be referred to with class pointers. In SOA, they are referred to with integer IDs. In AOS, the type system can catch programming mistakes early on, while, in SOA, many programming mistakes can remain unnoticed until runtime.

- **C++ OOP Abstractions:** AOS allows C++ programmers to use C++ abstractions for object-oriented programming, such as constructor syntax, the `new` keyword, inheritance, member visibility, member function (method) calls and virtual functions. This is not possible with SOA code.

There are programming languages that allow programmers to specify custom memory layouts without breaking abstractions. *Shapes* [65] is one example for such a language. However, we are focusing on C++/CUDA in this work because GPUs are predominantly programmed in a C++ dialect. Unfortunately, standard C++/CUDA does not allow programmers to specify custom memory layouts. This is possible only with custom C/C++ dialects such as *ispc* [152] or custom preprocessors such as *ROSE* [157]. Some compilers automatically perform data layout optimizations [215, 32, 131], but they often fail at complex programs.

2.3.2 Performance Characteristics of Structure of Arrays

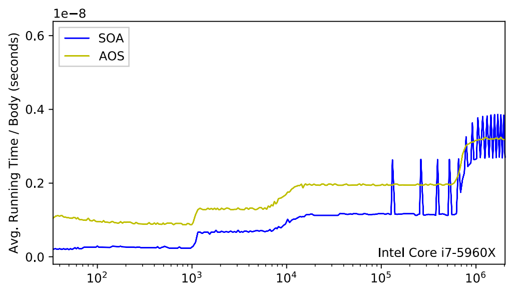
SOA is a well-studied best practice on SIMD architectures. Such architectures achieve parallelism by executing the same processor instruction on a *vector* register. Getting

LISTING 2.4: N-body simulation in AOS layout

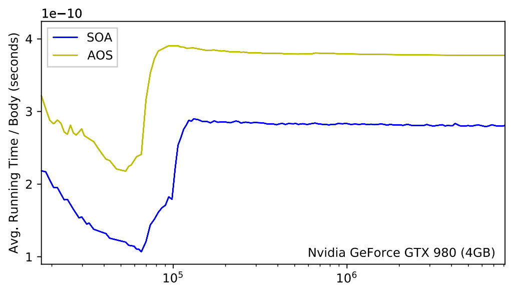
```cpp
class Body {
 public:
  float pos_x = 0.0; float pos_y = 0.0;
  float vel_x = 1.0; float vel_y = 1.0;
  float force_x; float force_y;
  float mass;

  // Class constructor
  Body(float mass, float x, float y) : mass_(mass), pos_x(x), pos_y(y) {}

  void move(float dt) {
    pos_x = pos_x + vel_x * dt;
    pos_y = pos_y + vel_y * dt;
  }
};

Body bodies[50];
// Counter for number of objects (instances).
int Body_inst = 0;

void create_and_move() {
  Body* b = bodies + bodies_inst++;
  // Call constructor with C++ placement new.
  new(b) Body(150.0, 1.0, 2.0);

  b->move(0.5);
  assert(b->pos_x == 1.5);
}
```

LISTING 2.5: N-body simulation in hand-written SOA layout

```cpp
float Body_pos_x[50]; float Body_pos_y[50];
float Body_vel_x[50]; float Body_vel_y[50];
float Body_force_x[50]; float Body_force_y[50];
float Body_mass[50];

// Counter for number of objects (instances).
int Body_inst = 0;

int new_Body(float mass, float x, float y) {
  int id = Body_inst++;
  Body_vel_x[id] = 1.0;
  Body_vel_y[id] = 1.0;
  Body_mass[id] = mass;
  Body_pos_x[id] = x;
  Body_pos_y[id] = y;
  return id;
}

void Body_move(int id, float dt) {
  Body_pos_x[id] += Body_vel_x[id] * dt;
  Body_pos_y[id] += Body_vel_y[id] * dt;
}

void create_and_move() {
  int b = new_Body(150.0, 1.0, 2.0);
  Body_move(b, 0.5);
  assert(Body_pos_x[b] == 1.5);
}
```



(A) Host running time



(B) Device running time

FIGURE 2.9: Running time of n-body simulation in AOS and SOA data layout

data into and out of vector registers is often the biggest bottleneck and peak memory bandwidth utilization can be achieved only with memory coalescing. SOA is one of the most basic optimizations that experienced GPU/CPU-SIMD programmers apply to optimize global memory access [207]. Previous work has reported speedups of SOA over AOS by multiple factors (e.g., [85]; see also Section 2.1.5).

Due to the data-parallel execution model, all threads of a SIMD work group (*warp* in CUDA) execute the same processor instruction in parallel on a vector of scalars. Therefore, if one thread is reading/writing a field such as `Body::pos_x`, then all other threads in the SIMD work group are likely reading/writing the same field (but usually of a different object). If neighboring threads process neighboring objects, then these reads/writes can be combined into a small number of physical memory transactions when using an SOA data layout (Figure 2.8). On GPUs, this is implemented in hardware: The memory controller coalesces memory reads/writes at run-time. On CPUs, the instruction set provides vectorized (packed) versions of many commonly used instructions (including vector loads/stores) and the compiler generates those instructions instead of their scalar versions (see Section 2.1.4). In AOS, the memory access follows a much more inefficient strided pattern, which does not allow for vector loads/stores because the memory addresses are not contiguous[14].

Besides better memory coalescing, SOA can also improve cache utilization if not all fields are used in a computation. Such fields do not occupy cache lines in SOA. For example, the implementation of `Body::move()` does not access the fields `force_x`, `force_y` and `mass`. However, since the L1 cache line size is 128 bytes on NVIDIA GPUs, in AOS, these fields will occupy the same cache lines as the actually accessed `pos_x`, `pos_y`, `vel_x`, `vel_y` fields, leaving less space in the cache for those fields.

Figure 2.9 shows the running time of `Body::move` with AOS and SOA data layout for a varying number of `Body` objects (x-axis). We report the average running time per `Body` object (y-axis). In host (CPU, Subfigure A) code, SOA is much faster at the beginning. This is because the compiler generates SSE vector instructions for SOA code. Furthermore, the cache is utilized more efficiently in SOA. This benefit starts fading away with higher body numbers. The spikes in the SOA graph are due to cache associativity issues. This could be resolved with a hybrid layout (Section 2.3.4). In device (GPU, Subfigure B) code, SOA is always faster than AOS, mostly due to better memory coalescing. For low body numbers, kernel invocation dominates the running time, thus the poor performance at the beginning.

[14]Recent CPU SIMD extensions such as AVX2 provide gather/scatter vector loads/stores [13, 83], but those instructions are not as fast as regular vector loads/stores.

2.3.3 Object vs. SOA Array Alignment

An object set stored in SOA layout can sometimes require less memory than the same object set in AOS[15]. In AOS, every field and every object must be properly aligned, whereas in SOA, only the SOA arrays themselves must be aligned.

LISTING 2.6: Example: Memory consumption of an object set in AOS/SOA

```
class DummyClass {
  double field_0;         // sizeof(double)       = 8
  char   field_1;         // sizeof(char)         = 1
};                        // sizeof(DummyClass)   = 16
DummyClass objects[50];   // sizeof(objects)      = 800

struct SoaContainer {
  double soa_field_0[50]; // sizeof(double[50])   = 400
  char   soa_field_1[50]; // sizeof(char[50])     = 50
};                        // sizeof(SoaContainer) = 456
```

As an example, consider the C++ source code in Listing 2.6. 50 objects in AOS layout require 800 bytes because every `double` field value (and `DummyClass` object) must be aligned to 8 bytes. However, 50 objects in SOA layout require only 456 bytes.

As a side note, the size of a C++ struct/class can sometimes be reduced by rearranging its fields (*structure packing*). In the example of Listing 2.6, structure packing cannot reduce the size of `DummyClass`.

2.3.4 Choosing a Data Layout

Choosing the best data layout for an application is challenging and depends on hardware characteristics and on the data access patterns of the application. Previous work has shown that, on different platforms, different layouts can sometimes achieve the best performance. For example, on platforms with low cache associativity, SOA is known to cause more cache evictions than AOS, which can increase the number of cache misses and slow down an application [131, 116].

Previous work has shown that a mixture of AOS and SOA such as AoSoA can sometimes achieve the best performance [65, 106, 199, 113, 134]. How to find good data layouts has been studied before [106, 4, 76] and is out of the scope of our work. When talking about custom data layouts, we are focusing on SOA in the remainder of this theis. However, our work could be extended to other layouts in the future.

[15]For reasonably large object sets, SOA never requires more space than AOS.

Chapter 3

Expressing Parallelism in Object-oriented Programs

This chapter investigates how object orientation can be utilized in GPU programs. We are particularly interested in how object orientation can be used to express parallelism.

Contents

Overview and Outline We developed two techniques/prototypes for expressing GPU parallelism in object-oriented programs.

- IKRA-RUBY[1] (Section 3.1): A data-parallel array class for Ruby with `each`/`map`/`reduce`/`stencil`/... operations that run on the GPU. Object orientation allows programmers to compose GPU programs of small parallel sections in a modular way. IKRA-RUBY is suitable mainly for mathematical computations.
- IKRA-CPP[2] (Section 3.2): To allow for efficient object-oriented programming within GPU code, we developed a CUDA framework for a subclass of object-oriented applications that we call *Single-Method Multiple-Objects* (SMMO). In SMMO, parallelism is expressed by running a method on all existing objects of a type. Many interesting applications in high-performance computing can be expressed with SMMO (Section 7).

[1] `https://prg-titech.github.io/ikra-ruby/index.html`
[2] `https://github.com/prg-titech/ikra-cpp`

SMMO applications can also be implemented in IKRA-RUBY by running a parallel `each` operation on an array of objects. Besides being implemented in a different programming language, IKRA-CPP is, in essence, a more restricted version of IKRA-RUBY with only one kind of operation: A parallel method call. This allows us to develop better optimizations for object-oriented code that runs on the GPU.

Our vision is that IKRA-CPP will eventually become a part of IKRA-RUBY, so that programmers can develop SMMO applications in a high-level language. However, this is out of the scope of this thesis and up to future work.

Publications This chapter is in part based on the following papers.

- Matthias Springer, Hidehiko Masuhara. **"Object Support in an Array-based GPGPU Extension for Ruby."** In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming.* ARRAY 2016. ACM, 2016, pp. 25–31. doi:10.1145/2935323.2935327
- Matthias Springer, Peter Wauligmann, Hidehiko Masuhara. **"Modular Array-Based GPU Computing in a Dynamically-Typed Language."** In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming.* ARRAY 2017. ACM, 2017, pp. 48–55. doi:10.1145/3091966.3091974
- Matthias Springer. **"DynaSOAr: Accelerating Single-Method Multiple-Objects Applications on GPUs."** Extended Abstract. In: *ACM Student Research Competition, Grand Finals, Graduate Category.* Originally submitted to SPLASH 2018. arXiv:1809.07444 (reviewed and published on ACM SRC website, no formal proceedings)

3.1 IKRA-RUBY: A Parallel Array Interface

Most mainstream, object-oriented programming languages, such as Java, Python or Ruby, provide a rich collection API for arrays, linked lists, hash maps, etc. These containers often have functional operations that execute a scalar computation on each (sometimes multiple) element(s) of the collection. The most prominent operations are `map` and `reduce`, popularized by the Map-Reduce programming model [48].

In this section, we investigate how collections can be used to express GPU parallelism. Arrays are particularly interesting because they allow for efficient random element access. Since the (functional) scalar computations are independent of each other, they can run in parallel on the GPU.

Ruby Array Interface As an example, consider the array interface (class `Array`) of the Ruby programming language. This interface provides many operations, but the following four ones are among the most used ones.

- `Array.new(n, &block)`: Creates a new array of size n. Each array slot is initialized with the return value of the block.
- `Array.map(&block)`: Applies a block (lambda function) to every element of an array and returns a new array of results. The original array remains unchanged.
- `Array.each(&block)`: Applies a block (lambda function) to every element of an array. In contrast to all other operations, this operation does not return a new

array. The original array remains unchanged. However, the block execution usually has a side effect.

- `Array.reduce(&block)`: Combines all elements of an array by applying a binary operation over and over. Returns a scalar value. The original array remains unchanged.

A Ruby *block* is an anonymous/lambda function. Programmers can customize the above operations with a block that specifies the computation of each scalar value. Blocks are ordinary Ruby objects. However, when defining/calling functions, Ruby distinguishes between ordinary parameters and block parameters. Ruby provides a special syntax for passing a block to a function (Listing 3.1).

LISTING 3.1: Example: Ruby `Array::map`

```
increment = Proc.new do |x| x + 1 end # Define a computation
[1, 2, 3].map(&increment) # Must pass blocks with ampersand ("block argument")
# => [2, 3, 4]

# More compact: Inline block definition
[1, 2, 3].map do |x| x + 1 end
# => [2, 3, 4]
```

Parallel Array Operations To explore array-based GPU computing in a high-level, object-oriented programming language, we developed IKRA-RUBY, a language extension for data-parallel and scientific computations on NVIDIA GPUs in Ruby. IKRA-RUBY uses arrays as an abstraction for expressing parallelism: We extended Ruby's `Array` class with parallel versions of the above operations and with a few extra operations that we describe later. The notation and API for these operations is similar to Ruby's sequential counterpart operations but method names are prefixed with `p` for *parallel*. In the simplest case, one CUDA thread is assigned to each array element, such that the array can be processed in parallel.

Programming Style When using IKRA-RUBY, we encourage a *dynamic* programming style that is governed by the following two concepts.

Integration of Dynamic Language Features Code in (blocks passed to) parallel array operations is limited to a restricted set of types and operations (dynamic typing and object-oriented programming is allowed). This allows us to generate efficient GPU code for parallel operations. All Ruby features (incl. metaprogramming) may still be used in other parts of a Ruby program. Therefore, programmers can still use external libraries (e.g., I/O or GUI libraries).

Modularity [140] While optimized low-level programs typically consist of a small number of kernels performing a variety of tasks, IKRA-RUBY allows programmers to compose a GPU computation from multiple reusable, smaller parallel array operations. IKRA-RUBY generates efficient CUDA kernels from these operations.

Compilation Process IKRA-RUBY is essentially a source-to-source compiler that compiles Ruby code to CUDA code. Ruby is a dynamically-typed programming

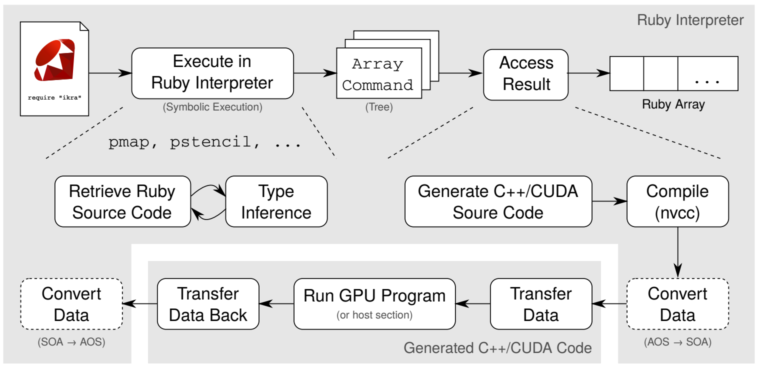


FIGURE 3.1: High-level overview of the compilation process

language, while CUDA is a statically-typed language. Due to Ruby's dynamic language features, whole-program static (ahead of time) analysis is difficult. Therefore, IKRA-RUBY generates CUDA programs at runtime (just in time) when all type information is known.

Figure 3.1 gives a high-level overview of IKRA-RUBY's compilation process. IKRA-RUBY executes parallel operations *symbolically* [103] in the Ruby interpreter. The result is an *array command* object. Such an object contains all information required for CUDA code generation and execution. An array command can be used like a normal Ruby array. However, only when its contents are accessed for the first time, IKRA-RUBY generates CUDA/C++ source code, compiles it using the CUDA compiler and runs the generated GPU program. The generated program copies data to the GPU, executes the parallel operation(s) on the GPU, copies the result back to the host memory and returns the result[3]. If a parallel operation modifies instance variables of objects as a side effect, then these changes are also copied back to Ruby.

Array commands can have other array commands as inputs, forming a *computation graph* of array commands. Such a graph can be optimized as a whole before emitting CUDA code. Recently, many machine learning framework follow such a design [2, 184]. The main optimization of IKRA-RUBY is *kernel fusion*: Compiling the computation graph into a small number of CUDA kernels. We describe this optimization in detail in Section 4.1.

Symbolic Execution During symbolic execution, IKRA-RUBY retrieves the source code of the block that is passed to a parallel operation and generates an abstract syntax tree[4]. The result of symbolic execution is an array command. Within a parallel operation, IKRA-RUBY currently supports expressions of primitive types, user-defined classes and polymorphic types. Advanced Ruby features such as metaprogramming, features that cannot be easily compiled to C++/CUDA (e.g., system calls, file I/O, GUI or FFI), and nested parallel operations are not supported. Parallel sections typically perform mathematical computations, which can be implemented with most basic Ruby functionality.

Array Commands A command object in the Command Design Pattern [70] is an object that contains all information that is necessary to perform an action at a later point of time. An array command in IKRA-RUBY is an object that contains all

[3]If object-oriented programming is used inside a kernel, data is first converted to a memory coalescing-friendly *Structure of Arrays layout* (Section 2.3).
[4]We utilize the Ruby *parser* library. See also: `https://rubygems.org/gems/parser`.

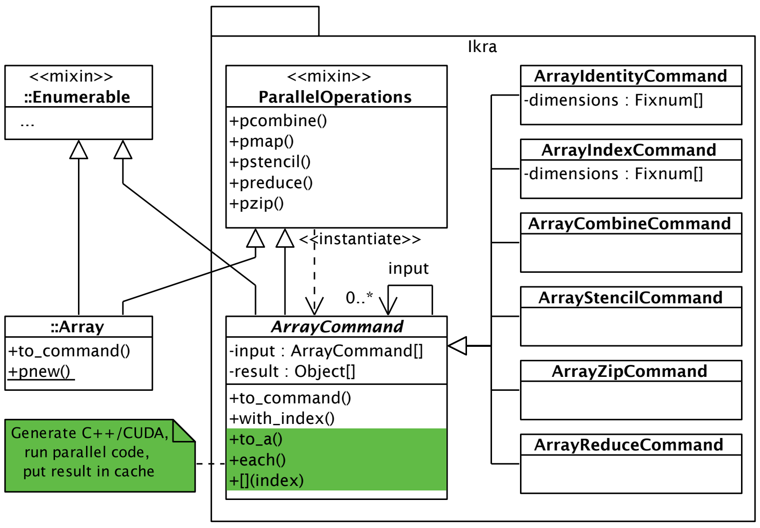


FIGURE 3.2: Integration of IKRA-RUBY in Ruby

information required for code generation and running a parallel operation. For example, a stencil array command contains the AST of the computation (block), a reference to the input array command, an array of neighborhood indices, and an out of bounds value (explained in Section 3.1.1).

An array command can be seen as a special *Ikra array*. It has all methods that an ordinary Ruby array has, but its contents are computed once it is accessed for the first time. The result of the computation is cached in Ruby, so that multiple accesses do not trigger recomputation. If the input of an array command is changed, the cached result is *not* invalidated (Listing 3.2). This is a deliberate decision and similar to how standard Ruby behaves with normal (non-parallel) array operations. However, in contrast to standard Ruby, results are computed upon access, so modifications of the input of an array command after symbolic execution can affect its result.

LISTING 3.2: Example: Caching the result of parallel array operations

```ruby
arr = [10, 20, 30, 40, 50]
squared = arr.pmap do |x| x * x end # No computation yet
arr[0] = 5 # Modifying "arr" before computation affects result of "squared"
squared[0] # First access triggers compilation and execution on GPU
# => 25

squared[1] # Result of "squared" is already in the cache
# => 400

arr[3] = 1000 # No recomputation of "squared"
squared[3]
# => 1600 (not 1000000)
```

Figure 3.2 gives an overview of IKRA-RUBY's integration in Ruby and the design of array commands. There are a variety of subclasses of `ArrayCommand` corresponding to the parallel operations that are supported in IKRA-RUBY (Section 3.1.1). The standard Ruby `Array` class and IKRA-RUBY's `ArrayCommand` class include two mixins [24]: `Enumerable` and `ParallelOperations`. The first mixin provides standard collection API functionality and requires an implementation of the `each` method in the class that it is mixed into. The second mixin provides IKRA-RUBY's parallel operations which are executed on the GPU.

3.1.1 Parallel Operations

This section gives an overview of the parallel operations that are provided by IKRA-RUBY. All operations can handle multidimensional IKRA-RUBY arrays, making code more readable if data is inherently multidimensional (e.g., images). For presentation reasons, we use only one dimension for most operations in this section.

If an operation performs a computation, then the size of the receiver (*base array*) determines the number of CUDA threads that are used. By default, IKRA-RUBY uses one CUDA thread per array element, but this can be changed.

Array Identity This operation wraps a Ruby array A in an IKRA-RUBY array (command), denoted by $id(A)$. Since an IKRA-RUBY computation graph consists of only array commands, this operation is necessary to make an external Ruby array A (which is not computed on the GPU) available in IKRA-RUBY.

Array identity is applied implicitly where required. For example, when a *parallel map* operation is applied to a Ruby array, IKRA-RUBY first wraps the Ruby array in an array identity command.

Array identity can also be used to *reshape* a Ruby array. E.g., this is useful if programmers want to convert a one dimensional Ruby array into a multidimensional IKRA-RUBY array. Reshaping does not change the actual data layout and is "for free". Array identity is exposed to Ruby programmers as `to_command`, taking an optional parameter for dimensions.

```
A.to_command()
A.to_command(dimensions: [15, 20])
```

IKRA-RUBY arrays can be converted back to Ruby arrays with `to_a`. This executes the computation graph on the GPU, unless the graph was already executed and the result is already cached.

Combine This operation is used to map over one or more arrays A_i of same size m and dimensions. It takes as input n arrays and a block (anonymous function) f taking n scalar values. It applies f to every element of the input and retains the original shape of the input (all dimensions).

$$combine(A_1, \ldots, A_n, f) = \begin{bmatrix} f(A_1[0], \ldots, A_n[0]) \\ \vdots \\ f(A_1[m-1], \ldots, A_n[m-1]) \end{bmatrix}$$

By default, IKRA-RUBY allocates m CUDA threads, i.e., every thread processes one tuple. This operation is exposed to Ruby programmers as `pcombine`:

```
A1.pcombine(A2, ..., An, &f)
```

Map This operation is a special case of *combine* with only one input array. It corresponds to an ordinary map operation but is executed in parallel.

$$map(A_1, f) = combine(A_1, f) = [f(A_1[0]), \ldots, f(A_1[m-1])]$$

This operation is exposed to Ruby programmers as `pmap`:

```
A1.pmap(&f)
```

For-Each This operation is similar to *map*: It runs a given Ruby block for every element in an array A_1. However, in contrast to all other operations, *for-each* is not a *functional* operation and does not have a return value. The Ruby block may as a side effect, just as in all other operations, modify the elements of A_1 (if they are objects), as well as other objects that are in scope, such as lexical variables. These changes are written back to Ruby after running the operation. This operation is exposed to Ruby programmers as `peach`:

```
A1.peach(&f)
```

Since *for-each* does not have a return value, it is not symbolically executed but immediately executed. Furthermore, certain optimizations that are described later, such as kernel fusion, are not applied to this operation.

Index This operation generates an array of size m of consecutive indices starting from 0 and ending with $m-1$.

$$index(m) = [0, 1, 2, \ldots, m-1]$$

In a multidimensional case, *index* takes d arguments m_i (d is the number of dimensions), where m_i is the size of the i-th dimension. Every value in the resulting array is then an array of size d containing the indices for every dimension.

$$index(m_1, m_2) = \begin{bmatrix} [0,0], \ldots, [0, m_1 - 1], \ldots, \\ [m_2 - 1, 0], \ldots, [m_2 - 1, m_1 - 1] \end{bmatrix}$$

This operation is not directly exposed to programmers. Similar to standard Ruby, programmers must invoke the method `with_index` after a parallel operation (the parameter m is provided implicitly) or use `Array.pnew` (see below).

```
A1.pmap.with_index(&f)

# API Example:
[10, 20, 30, 40, 50].map.with_index do |x, y| x + y end
# => [10, 21, 32, 43, 54]
```

New This operation is a combination of *index* and *map*. It creates a new array of size m and initializes it using the block (anonymous function) f. It is a parallel version of Ruby's `Array.new`.

$$new(m, f) = map(index(m), f) = [f(0), \ldots, f(m-1)]$$

Similiar to *index*, this operation takes multiple arguments in a multidimensional case. This operation is exposed to Ruby programmers as `pnew`:

```
Array.pnew(m, &f)
```

Stencil (Convolution) This operation takes as arguments an input array A, an array of relative indices I (neighborhood) of size k, a block f, and an out-of-bounds value o. It creates an array of same size and dimensionality where position i is initialized using f, passing the values in the neighborhood of $A[i]$ as arguments to f. Simple examples of stencil computations are image filtering kernels.

The following formula is used to calculate the value in the resulting array at position i. If all indices are within bounds (case 1), i.e., $0 \le i + I[j] < m$ for all $0 \le j < k$ (where m is the size of A), the value of the stencil computation is used. Otherwise (case 2), the fallback value o is used.

$$v(A, I, f, o, i) = \begin{cases} f([A[i + I[0]], \ldots, A[i + I[k-1]]]), & \text{(case 1)} \\ o, & \text{(case 2)} \end{cases}$$

Using this helper function v, a stencil computation is defined as follows.

$$\mathit{stencil}(A, I, f, o) = [v(A, I, f, o, 0), \ldots, v(A, I, f, o, m-1)]$$

This operation is exposed to Ruby programmers as `pstencil`. The value of the parameter I is inferred from the code of the block.

```
A.pstencil(o, &f)

# API Example (Gaussian blur 3x3 image kernel):
img = load_pixels("pic.png").to_command(dimensions: [640, 480])
filtered = img.pstencil(0) do |v|
  ( 1 * v[-1][-1]  +  2 * v[0][-1]  +  1 * v[1][-1] +
    2 * v[-1][ 0]  +  4 * v[0][ 0]  +  2 * v[1][ 0] +
    1 * v[-1][ 1]  +  2 * v[0][ 1]  +  1 * v[1][ 1] ) / 16.0
end
```

Zip This operation does not perform a computation but groups values of two or more arrays of same size and dimensionality.

$$\mathit{zip}(A_1, \ldots, A_n) = \begin{bmatrix} [[A_1[0], \ldots, A_n[0]] \\ \vdots \\ [A_1[m-1], \ldots, A_n[m-1]]] \end{bmatrix}$$

The result of this operation is an array of arrays. This operation is exposed to Ruby programmers as `pzip`:

```
A1.pzip(A2, ..., An)
```

Reduce This operation takes as arguments an input array A and a block f, whose function must be associative. Every block application reduces two elements into a single one. The block is applied until only one element is left (dimensions are ignored[5]). This operation is similar to Ruby's `Array.reduce`, but the return value is an array with one element instead of a scalar value.

$$\mathit{reduce}(A, f) = [f(\ldots f(f(A[0], A[1]), f(A[2], A[3]), \ldots) \ldots)]$$

There are no guarantees about the order in which elements are reduced, because reduction is done in parallel. This operation is exposed to Ruby programmers as `preduce`:

[5]Machine learning libraries provide more powerful versions that reduce values along one or multiple specified dimensions, e.g. `reduce_sum` in TensorFlow. This is not currently supported in IKRA-RUBY.

```
# Passing a block (function)
A.preduce(&f)
# Symbols are also possible as shortcuts
# E.g.: A.preduce do |a, b| a + b; end
A.preduce(:+)
```

3.1.2 Mapping Ruby Types to C++ Types

Ruby is a dynamically typed programming language. During type inference, IKRA-RUBY infers which types each expression in a parallel operation can have, based on the runtime types of the input to the parallel operation. If an expression is monomorphic (has a single type), a Ruby type is directly mapped to the corresponding type in the C++/CUDA source code.

TABLE 3.1: Mapping Ruby types to C++ types

Ruby Type	C++ / CUDA Type
`Fixnum`	`int`
`Float`	`float`
`TrueClass`	`bool`
`FalseClass`	`bool`
`NilClass`	`int`
`Array`	`array_t`, generated struct type
`ArrayCommand`	`array_command_t *` *(only in host sections, Section 4.1.3)*
(other)	`object_id_t` (`int`)
(polymorphic)	`union_t`

Table 3.1 shows the mapping of Ruby data types to CUDA/C++ data types. Numeric values are currently represented by `int` or `float`. `nil` is represented by `int` value 0. Arrays are either represented by `array_t` (a pointer-size pair) or a generated struct type for zip types. Other objects are represented by `int` object IDs generated by IKRA-RUBY's object tracer (Section 3.1.3).

```
union union_v_t {
  int int_;
  float float_;
  bool bool_;
  void *pointer;
  array_t array;
  array_command_t array_command; // later...
};
```

```
struct union_t {
  int class_id;
  union_v_t value;
};
```

FIGURE 3.3: Union type struct definition

For polymorphic expressions (e.g., if `Fixnum`s and `nil` are assigned to a variable), union type [3] structs (Figure 3.3) are used. Values are stored in `union_v_t` which can hold values (or pointers to values) for all C++ types of Table 3.1. The class ID field contains a number that identifies the runtime type[6]. If a method is called on a polymorphic expression, IKRA-RUBY generates a switch-case statement with all types that the expression can have at runtime. Such an implementation is much

[6]This is different from standard C++, where a vtable pointer is stored at the allocation site of an object. Such an implementation would impose a large overhead in IKRA-RUBY because we would then have to heap-allocate primitive types and refer to them with pointers.

more efficient in CUDA than a virtual function call because all call targets can be inlined by the compiler. If a monomorphic value is assigned to a polymorphic lvalue, IKRA-RUBY wraps the value in a union type struct. Arrays of union type structs are used to represent polymorphic arrays.

In contrast to other compilers that just-in-time generate GPU code from dynamic language code [67], IKRA-RUBY performs a *conservative* type inference pass, such that no unexpected types can appear at GPU program runtime. Therefore, IKRA-RUBY does not need to insert runtime type checking guards and code is never invalidated due to a failing runtime type check. This design decision is based on our assumption that most GPU code is monomorphic or exhibits a very small set of runtime types. In fact, we could not find any real GPU programs for which our conservative type inference system would infer an excessively large set of types of which only a few actually appear at GPU program runtime.

3.1.3 Object Tracer

If object-oriented programming is used within a parallel operation, we have to transfer all objects to the GPU that the GPU program may be accessing during its runtime. Object allocation and deallocation is not supported within parallel operations; only existing objects can be modified. Before CUDA code is generated, the *object tracer* identifies all objects that may potentially be accessed in a parallel operation. Since the exact runtime behavior of a program cannot be predicted, this analysis is conservative and may identify objects that are not actually accessed at runtime. The object tracer performs three main tasks.

- Decide **which Ruby objects may be accessed** in the parallel operation and must, therefore, be transferred to the GPU.
- Assign a **unique ID** to each such object.
- For each Ruby class that is used within a parallel operation, determine **which instance variables are read/written**. Only those instance variables are transitively traced and copied to the GPU (and copied back to Ruby after the kernel finished executing).

The object tracer starts tracing with a set of root objects: All elements of the base array and lexical variables that are accessed. Then, it traverses the object graph by following all instance variables that are read or written inside the parallel section. IKRA-RUBY's object tracer is somewhat similar to what a *system tracer* [73, 108] does in Smalltalk-80 systems.

Object tracing can result in additional type inference passes. The reason for that is Ruby's dynamic typing. During object tracing, IKRA-RUBY might find that the set of possible types of an instance variable is larger than previously assumed. If a method is called on such an instance variable, IKRA-RUBY has to translate also this new method to CUDA code. Furthermore, this method may, for the first time, access another instance variable of objects of a class that we have already begun tracing. The instance variable values of those already traced objects must now also be traced.

IKRA-RUBY's object tracer is currently not optimized for performance. The current implementation allows us to explore object-oriented programming within parallel operations, but object tracing and data transfers to the GPU can take a considerable amount of time. Assuming that most computation-intensive code runs on the GPU, future versions of IKRA-RUBY should optimize this by allocating most objects on the GPU and transferring them to the Ruby interpreter only upon access.

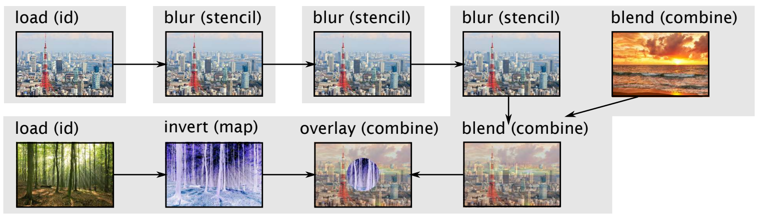
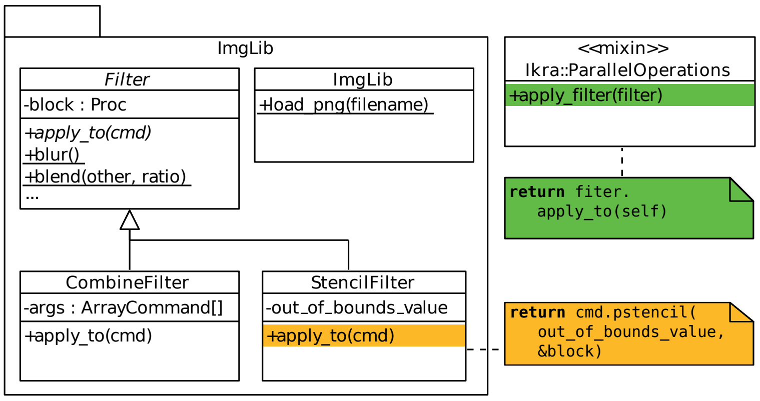


FIGURE 3.4: Example: Gray boxes indicate CUDA kernels (*kernel fusion*, Section 4.1).



```
return fiter.
  apply_to(self)
```

```
return cmd.pstencil(
  out_of_bounds_value,
  &block)
```

FIGURE 3.5: Architecture of image manipulation library example

3.1.4 Example: Image Manipulation Library

To illustrate IKRA-RUBY's API, we design a simple image manipulation library. This library provides methods for loading images from the file system and a few filters (*image kernels*) and effects.

As an example, Figure 3.4 shows which filters we are using and in which order we apply them: First, we load a picture of the Tokyo Tower. Then, we apply a blur filter multiple times. Next, we load a picture of a sunset and merge (blend) both pictures. Finally, we load a picture of a forest, invert it, and overlay it with the previously merged picture. Listing 3.4 shows the source code for this example. Notice how the code is modular with respect to composability, reusability, understandability: Image filters are provided by the library and can be arbitrarily combined.

The filters are implemented in the library using parallel map or stencil operations (Figure 3.5, Listing 3.3). Images are represented as 2D array command objects. The library defines an extension method `apply_filter` for applying a filter to an image with double dispatch [90]. Multiple filters can be applied in a sequence by chaining `apply_filter` method calls. Only when the final result is accessed, does IKRA-RUBY generate an optimized CUDA program, copy the images to device memory, run the CUDA kernels and copy back the result.

3.1.5 Summary

In this section, we presented the design and implementation of IKRA-RUBY, a Ruby library for data-parallel computations. Programmers express parallelism by running parallel operations on an array. IKRA-RUBY allows programmers to write modular code with respect to reusability and composability of parallel operations. Section 4.1 describes how IKRA-RUBY optimizes such GPU programs with kernel fusion.

LISTING 3.3: Example: Definition of image manipulation filters

```
module ImageLibrary::Filters
  def self.load_png(filename)
    # Pixels are just integers.
    image = read_png(filename)
    return image.pixels.to_command(dimensions: [image.height, image.width])
  end

  def self.blend(other, ratio)
    return CombineFilter.new(other) do |p1, p2|
        # pixel_add, pixel_scale: Helper functions for dealing with RGB values
        pixel_add(pixel_scale(p1, 1.0 - ratio), pixel_scale(p2, ratio))
    end
  end

  def self.pixel_add(p1, p2) # Add values for each channel, cap at 255.
    return min(((p1 & 0xff0000) >> 16 + (p2 & 0xff0000) >> 16), 255) << 16 # red
         + min(((p1 & 0x00ff00) >> 8 + (p2 & 0x00ff00) >> 8), 255) << 8 # green
         + min((p1 & 0x0000ff) + (p2 & 0x0000ff), 255) # blue
  end

  def self.pixel_scale(p1, r) # Ratio r must be between 0.0 and 1.0
    return (r * ((p1 & 0xff0000) >> 16)) << 16 # red
         + (r * ((p1 & 0x00ff00) >> 8)) << 8 # green
         + r * (p1 & 0x0000ff) # blue
  end

  def self.blur
    return StencilFilter.new(0) do |v|
           r = v[-1][-1][0] + ... + v[1][1][0]
           g = v[-1][-1][1] + ... + v[1][1][1]
           b = v[-1][-1][2] + ... + v[1][1][2]
           [r / 9.0, g / 9.0, b / 9.0]
    end
  end

  # Source code of other filters omitted.
end
```

LISTING 3.4: Example: Image manipulation library usage

```
require "image_library"

tt = ImgLib.load_png("tokyo_tower.png")
for i in 0...3
  tt = tt.apply_filter(ImgLib::Filters.blur)
end

sun = ImgLib.load_png("sunset.png")
combined = tt.apply_filter(ImgLib::Filters.blend(sun, 0.3))

forest = ImgLib.load_png("forest.png")
forest = forest.apply_filter(ImgLib::Filters.invert)
combined = combined.apply_filter(
    ImgLib::Filters.overlay(forest, ImgLib::Masks.circle(tt.height/4)))

ImgLib::Output.render(combined) # Draw pixels
```

3.2 IKRA-CPP: A C++/CUDA Library for Single-Method Multiple-Objects Applications

Based on our experiments, we found that IKRA-RUBY is suitable for mathematical and purely functional applications, but less suitable for applications that utilize object-oriented programming inside of parallel operations. Such applications do not take advantage of the full range of IKRA-RUBY's operations. Most of them use only *for-each* (`Array::peach`) and modify object fields as a side effect, as opposed to expressing state transitions in a functional way through a series of *map* operations.

In fact, in object-oriented programming languages, many programmers express computation as objects imperatively changing their internal state upon receipt of a message (method call). This is in contrast to functional programming, which favors immutability of state. If the state of an object is modifed in a purely functional system, the result is a brand new object [104]. Such a functional programming style certainly has its benefits [62] and can make programs easier to reason about [50]. However, such *functional objects* [200]/*value objects* [161] lack object identity, which many programmers see as a fundamental concept in object-oriented programming.

3.2.1 Single-Method Multiple-Objects

We identified a broad object-oriented programming model that can be implemented efficiently on SIMD architectures such as GPUs and has many real-world applications in the area of high-performance computing. We call this model *Single-Method Multiple-Objects* (SMMO). We can think of SMMO as OOP-speech for SIMD (*Single-Instruction Multiple-Data*). The most fundamental operation of SMMO is parallel *do-all* (*for-each*): Running one method in parallel on all existing objects of a type (*object set*). Parallel do-all operations typically perform some kind of computation by imperatively modifying the state of objects.

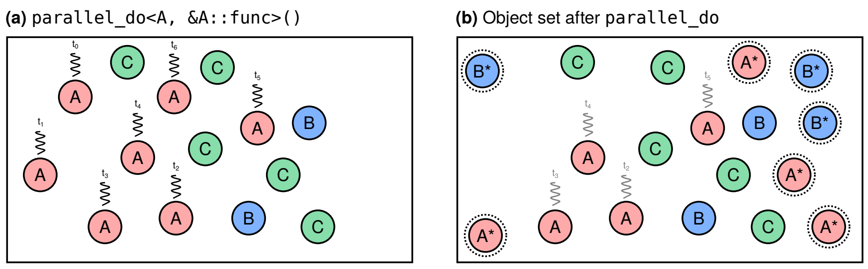
SMMO fits well with the data-parallel SIMD execution model of GPUs and can be implemented very efficiently. Since we only process objects (jobs) of the same type in a parallel do-all operation, we expect those computations to be mostly uniform with little warp divergence.

Definition and Runtime Semantics The SMMO programming model provides one main abstraction/operation for expressing parallelism: *Parallel do-all*. This operation runs a given method `T::func` for all objects of a given type S (and subtypes) that exist at the beginning of the parallel do-all operation. T must be a supertype of S or be equal to S, i.e., $S <: T$. If `T::func` is virtual, then the operation runs the most specific (overridden) method for each object, similar to a virtual method call.

A parallel do-all operation typically runs in multiple threads. The number of threads and the assignment of objects to threads is implementation-specific, but on GPUs, certain assignments may result in better performance than others (Section 5.2.7). Since objects are typically[7] processed in parallel, there is no defined order in which objects are processed. A parallel do-all operation returns when all threads finished executing, i.e., when all objects were processed.

New objects of any type may be created within a parallel do-all operation. However, these objects are not being processed by the same parallel do-all operation. Furthermore, existing objects may be deleted within a parallel do-all operation with

[7] A parallel do-all implementation that runs sequentially would also satisfy our definition of SMMO.


(a) parallel_do<A, &A::func>()
(b) Object set after parallel_do
A
B
C
A*
B*

```
void A::func() {
    if (rand() < 0.5) { new B(); delete this; }
    else { new A(); }
}
```

FIGURE 3.6: Example: Object set before (during) and after a parallel do-all operation

two limitations. Let S be the type of objects that are processed in a parallel do-all operation and `T::func` be the method that is executed for all objects of type S.

- No object may be deleted more than once. This is forbidden even in ordinary object-oriented programming.
- An object `obj` of type S may be deleted only at the end of `T::func` (last statement) and only by the method execution that is bound to `obj`. In other words, a method `T::func` can delete the object that it is bound to (`delete this`) but no other objects of type S. This is to avoid that an object is deleted while it is being processed by another thread.

Figure 3.6 illustrates a parallel do-all operation of a method `A::func` with an example. In this example, one thread is spawned for each object of type A (Subfigure A). The method `A::func` instantiates new objects of various types (annotated with a star), including objects of type A. However, these new objects are not processed by the same parallel do-all operation.

Example Applications SMMO is applicable to a broad class of problems[8] with many real-world applications, such as simulations for population dynamics, (e.g., Sugarscape [59]), evacuations [122], wildfire spreading [172], finite element methods or particle systems, to name just a few. SMMO can also express breadth-first search graph traversals and dynamic tree updates/constructions, e.g., in Barnes-Hut [28]. We show the design and implementation of a few SMMO applications in Chapter 7, highlighting their SMMO structure and their parallel do-all operations.

3.2.2 Programming Interface and Notation

We developed IKRA-CPP, a C++/CUDA framework that implements the SMMO programming model. IKRA-CPP facilitates the development of SMMO applications by providing abstractions for parallel do-all operations. In the course of this thesis, we will extend IKRA-CPP with a data layout DSL (Section 4.2), a dynamic memory allocator (Chapter 5) and a memory defragmentation system (Chapter 6). We plan to provide a Ruby frontend of IKRA-CPP in the future, so that programmers can write SMMO applications in a high-level language.

[8]We implemented a few SMMO applications from different domains: https://github.com/prg-titech/dynasoar/wiki/Benchmark-Applications

Class Definition IKRA-CPP applications must follow a certain notation. Listing 3.5 shows IKRA-CPP's notation with an n-body simulation (details in Section 7.1). User-defined classes/structs must inherit from a special template class `IkraBase`[9]. This template has two arguments: The class/struct itself (*curiously recurring template pattern* [33]) and the number of objects n of the class/struct that can exist at runtime. The class/struct must be initialized with a helper macro `IKRA_INITIALIZE_CLASS`. Finally, the programer must decide whether objects should reside on the host (CPU) or on the device (GPU): `IKRA_HOST_STORAGE` or `IKRA_DEVICE_STORAGE`. These macros statically allocate a *storage array* of size n, either on the host or on the device.

Data Layout and Object Creation All objects of a type are stored in their respective statically-allocated storage array. Such a layout is called an *Array of Structures* (AOS; Section 2.3). IKRA-CPP classes are not designed for heap allocation. This is in part because the default CUDA dynamic memory allocator is so slow that runtime heap allocation could severely reduce the performance of a GPU program, unless a custom memory allocator is used.

There are two ways of creating new objects in IKRA-CPP (Listing 3.6): With the C++ `new` keyword or with a *parallel new* operation (next paragraph). In either case, new objects are stored inside the AOS storage buffer and not on the heap. Existing objects cannot be deleted, so we can create at most 50 objects in the example code before running out of memory. We will extend IKRA-CPP with an efficient and fully fledged dynamic memory allocator in Chapter 5.

Expressing Parallelism IKRA-CPP provides a simple API for expressing parallelism. This API allows programmers to run a member function for all objects of a type, which is the foundation of SMMO. Our API abstracts from CUDA implementation details, resulting in more compact and readable code.

Programmers do not have to define kernels or write kernel invocation statements. A boolean flag `CUDA` controls whether an operation is executed on the device (GPU) or on the host (CPU). This flag is optional and omitted in all following examples. By default, parallel operations run where the data is located, as indicated by `IKRA_DEVICE_STORAGE` or `IKRA_HOST_STORAGE`.

- `parallel_do<CUDA, S, &T::func>(args...)`: Runs a member function `T::func` for all objects of type S that exist at invocation time, where $S <: T$. If `CUDA` is `true`, this operation spawns a GPU kernel and runs the the member function inside the kernel in parallel. Otherwise, the code executes on the host[10].

- `parallel_do_and_reduce<CUDA, S, &T::func, &T::reducer>(args...)`: Same as `parallel_do`, but the return values of `T::func` are reduced into a single value by applying a binary, static function `T::reducer` over and over. Parallel reductions can be implemented efficiently in CUDA with shared memory [79]. This operation is useful for termination detection of iterative algorithms where the termination criteria depends on a property of multiple objects.

- `parallel_new<CUDA, T>(n, args...)`: Instantiates n objects of type T. This operation calls the constructor of T in parallel with an object index (between $[0; n)$), followed by `args...`. T must have a suitable constructor.

[9]User-defined classes/structs cannot inherit from other user-defined classes/structs. We will remove this limitation later in Chapter 5.

[10]Currently sequentially, but we could use OS threads in the future.

LISTING 3.5: API example of IKRA-CPP

```
class Body : public IkraBase<Body, 50> {
 public: IKRA_INITIALIZE_CLASS
  float pos_x = 0.0f;
  float pos_y = 0.0f;
  float vel_x = 0.0f;
  float vel_y = 0.0f;
  float force_x = 0.0f;
  float force_y = 0.0f;
  float mass = 0.0f;

  __device__ __host__ Body(float m, float x, float y)
      : mass(m), pos_x(x), pos_y(y) {}

  __device__ __host__ Body(int index) { /* ... */ } // parallel_new constructor

  __device__ void apply_force(Body* other) {
    if (other != this) {
      // To avoid race conditions: Update other instead of this.
      float dx = pos_x - other->pos_x;
      float dy = pos_y - other->pos_y;
      float dist = sqrt(dx*dx + dy*dy);
      float F = kGravityConstant * mass * other->mass / (dist * dist);
      other->force_x += F*dx / dist;
      other->force_y += F*dy / dist;
    }
  }

  __device__ void compute_force() {
    force_x = force_y = 0.0f;
    device_do<Body>(&Body::apply_force, this);
  }

  __device__ void update() {
    pos_x = pos_x + vel_x * kDt;
    pos_y = pos_y + vel_y * kDt;
  }
};

// Preallocate memory for 50 Body objects on the GPU.
IKRA_DEVICE_STORAGE(Body);

int main() {
  // Create a few random Body objects on the host. Alternative: parallel_new.
  for (int i = 0; i < kNumBodies; ++i) {
    new Body(/*m=*/ rand_float(100.0f, 1000.0f),
             /*x=*/ rand_float(-1.0f, 1.0f), /*y=*/ rand_float(-1.0f, 1.0f));
  }

  for (int i = 0; i < kNumIterations; ++i) {
    parallel_do<Body, &Body::update>();
    parallel_do<Body, &Body::move>();
  }
}
```

LISTING 3.6: Object creation in IKRA-CPP

```
parallel_new<Body>(49); // Create 49 objects from host code
Body* b = new Body(100.0f, 0.0f, 0.0f); // Create a new object from device/host code
delete b; // Not supported yet. Later in this thesis...
Body* b2 = new Body(50.0f, 0.0f, 0.0f); // Out of memory
```

- `device_do<S, &T::func>(args...)`: Runs a member function `T::func` for all objects of type S in the current CPU/GPU thread, where $S <: T$. This is a sequential *for-each* loop. It is typically used inside of a parallel do-all for processing all pairs of objects (e.g., in n-body simulations). Whether objects created within the enclosing parallel do-all operation are enumerated is unspecified.

Note that a parallel operation does not necessarily have to run where the data is located, as indicated by `IKRA_*_STORAGE`, as long as the function/constructor has the correct `__device__` and/or `__host__` qualifiers. IKRA-CPP will take care of the necessary data transfers. The following four configurations are supported.

- **Code on GPU, Data on GPU:** If the program is suitable for GPU execution, then this configuration achieves the best performance.
- **Code on CPU, Data on CPU:** This configuration is useful if no GPU is available and for debugging purposes.
- **Code on CPU, Data on GPU:** During our experiments, we found that it was often very convenient to run setup code (e.g., loading and parsing data from an external source and creating the necessary objects) on the host.
- **Code on GPU, Data on CPU:** This configuration may be useful for highly compute-bound applications that access almost no memory.

To reduce the amount of data transfers and to achieve good runtime performance, performance-critical code should in general run where the data is located.

3.2.3 Implementation Details

IKRA-CPP is implemented entirely in C++. It does not need a separate compiler or preprocessor/code generator. IKRA-CPP consists mainly of two preprocessor macros (`IKRA_INITIALIZE_CLASS` and `IKRA_*_STORAGE`) and API functions for running a constructor or running a member functions on all objects of a type.

Storage and Allocation The purpose of the two preprocessor macros is to declare an array that contains all objects of the class. In particular, `IKRA_DEVICE_STORAGE` in Listing 3.5 generates a variable that contain the array of structures and an object counter variable.

```
__device__ Body objects_Body[50];
__device__ int counter_Body = 0;
```

The purpose of `IKRA_INITIALIZE_CLASS` is mainly to overload the C++ `new` operator, such that newly instantiated objects are always stored in the previously generated array. Note that the counter variable must be incremented with an atomic operation because multiple threads may be simultaneously instantiating objects.

```
__device__ void* Body::operator new() {
  assert(counter_Body < 50);
  return &objects_Body[atomicAdd(&counter_Body)];
}
```

In C++, object instantiation with the `new` keyword involves three steps.

1. Allocate heap memory for the new object. This request is delegated to the system-wide heap allocator (`malloc`) or to an overloaded `operator new`.
2. Zero-initialize the allocated memory. CUDA seems to omit this step.
3. Run the constructor. This includes field initializers.

Note that C++ operators can be overloaded for specific types. For example, the above source code listing overloads the `new` operator only for class `Body`, so other classes/structs are still heap-allocated with the system-wide memory allocator.

API Functions In the case of GPU execution, `parallel_do` and `parallel_new` variants launch a CUDA kernel with a configurable number of threads. By default, IKRA-CPP launches 256 blocks with 256 threads per block. Inside the template-generated CUDA kernel, IKRA-CPP processes objects with a grid-stride loop.

`parallel_do` and `parallel_new` are always called from host code. The functions `T::func` and `T::reducer` are device functions and passed to `parallel_do` as function pointers. Taking the address of a device function in host code is not allowed. This is because the host code and device code are essentially two different programs. They are compiled separately. However, the address of a device function can be taken in host code if it is used to instantiate a template that resides in device code. When calling such a template function pointer, the (CUDA) compiler can fully inline the function [183] (instead of generating a jump), because the template instantiation is specific to that function.

Memory Transfers If objects are accessed on a device that differs from the allocation device (e.g., accessed on CPU, allocated on GPU), IKRA-CPP automatically performs the required memory transfers. Programmers have to set a flag to enable support for this. Instead of statically allocating a storage array, IKRA-CPP then allocates the array at runtime in CUDA unified memory [164]. This CUDA feature automatically performs the required memory transfers and was introduced with CUDA 6.

3.2.4 Conclusion

In this section, we presented a second, more restricted way of expressing GPU parallelism: By running a method in parallel on all objects of a type. This simple programming model (*Single-Method Multiple-Objects*) is expressive enough for many important applications in high-performance computing (Chapter 7) and the full range of IKRA-RUBY operations is often not necessary.

We designed and implemented a small C++/CUDA framework IKRA-CPP for SMMO applications. At this point, IKRA-CPP is not much more than a wrapper around kernel invocation statements. The purpose of this section is to make the reader familiar with the SMMO programming model. We will extend IKRA-CPP with a data layout DSL (Section 4.2) and a full dynamic memory allocator (Chapter 5) later in this thesis.

3.3 Related Work

This section gives an overview of how GPU parallelism is expressed in other systems. Most systems fall into one of two categories: Parallel array interfaces or *for* loop parallelization.

3.3.1 Parallel Array/Tensor Interface

There are many libraries for various programming languages that provide parallel array functions that execute on the GPU, similar to IKRA-RUBY. This section is not an exhaustive description of all of them, but gives an overview of a few major ones.

Ishizaki et al. developed a JIT compiler than translates lambda expressions of Java 8 bytecode to NVVM IR, an LLVM IR-based intermediate representation that can be compiled to NVIDIA PTX code [91]. Their compiler targets array-based computations that are expressed with the Java Streams API. Object-oriented programming is supported within lambda expressions and virtual method calls are devirtualized: Either directly, if the receiver type is found to be monomorphic at JIT compilation time, or based on runtime profiling with a guard, otherwise. If the guard fails, the lambda expression is executed on the host. Their compiler follows the same assumption as IKRA-RUBY, namely that GPU code is mostly monomorpic, so guards are expected to fail rarely.

Fumero et. al. also developed an array-based GPU library for Java 8 [68]. Instead of targeting the Java Streams API, their JIT compiler comes with a custom Java array interface that provides functions that are similar to IKRA-RUBY. Their JIT compiler does not support object-oriented programming in GPU code, but parallel array operations can be chained, similar to IKRA-RUBY.

Fumero et al. designed also a GPU extension for the R programming language, a dynamically-typed language [67]. Their implementation is built on top of the Truffle AST interpreter framework [210] and the Graal JIT compiler. They use partial evaluation to generate optimized OpenCL code for hot code sections. In contrast to IKRA-RUBY, the resulting OpenCL code can handle only monomorphic types, whereas IKRA-RUBY generates a single CUDA program with union types that can handle all types which could theoretically show up during runtime. Consequently, their generated OpenCL code is more efficient, but requires recompilation if the runtime types are changing.

Lime is an object-oriented programming language for task-based and array-based parallelism [52]. The Lime compiler decides automatically which code should be executed on the device and which code should be executed on the host. In the former case, the compiler generates OpenCL code and in the latter case, the compiler generates a mixture of Java bytecodes.

Delite is a Scala framework for building and executing implicitly parallel DSLs [30]. Parallel operations form a computation graph of *Delite ops*, which are similar to array commands in IKRA-RUBY. Delite schedules the execution of the computation graph. Independent parts of the computation graph can be executed in parallel. Delite also provides abstractions for defining data-parallel computations.

Firepile is a Scala library for array-based GPU programming [147]. Firepile provides a parallel array class whose functional operations are executed in parallel on the GPU. Similar to IKRA-RUBY, the Firepile compiler is usually invoked right before a kernel run (just-in-time). Object-oriented programming is supported in parallel GPU code. Similar to IKRA-RUBY, Firepile uses union types to represent polymorphic types and devirtualizes virtual method calls with switch-case statements.

TensorFlow is a machine learning framework developed by Google [2]. Programmers build a computation graph of linear algebra nodes with a Python or C++ frontend. This graph is then scheduled to execute on one or multiple devices such as CPUs, GPUs or TPUs (tensor processing units). In contrast to IKRA-CPP, TensorFlow operations cannot be customized with code. There are a variety of machine learning frameworks that follow a similar design.

3.3.2 For-Loop Parallelization

There are a variety of systems that accelerate programs with GPUs by finding and offloading loops to the GPU. MegaGuards is a Python framework that transparently compiles and offloads compute-intensive loop to the GPU [158]. It detects parallelizable loops with polyhedral optimization techniques. MegaGuards avoids runtime type checks (guards) inside GPU code with a type stability analysis. If positive, all type checks within GPU code are replaced by a single, larger guard before the loop.

There are also parallelization frameworks that let programmers manually annotate *for* loops for acceleration on GPUs. For example, OpenACC allows programmers to annotate C/C++ loops with pragmas that trigger GPU code generation [181].

JaMP is a Java extension with OpenMP-style directives for parallel *for* loops [205]. It supports object-oriented programming within parallel *for* loops and generates a C struct type for every Java class. Instead of referring to referenced objects with pointers, objects are fully inlined into the C struct. JaMP distinguishes between shared objects, which can be accessed by any thread and are replicated on all devices, and managed arrays, which are partitioned among all devices.

Concord is a heterogeneous C++ programming framework for running irregular application code on integrated GPUs [16]. It provides two abstractions for expressing parallelism: A parallel *for* loop and a parallel reduction loop. Concord is implemented with Clang and LLVM and generates OpenCL code for parallel loop bodies. Most C++ features, including virtual function calls and multiple inheritance, are supported in parallel GPU code. Virtual function calls are implemented with vtable pointers, as usual in C++. Concord also provides a software-based shared virtual memory, so that pointers can be shared between GPU code and CPU code. Concord's main selling point is improved energy efficiency of application code over a CPU-only implementation.

Chapter 4

Optimizing Memory Access

Memory access is the one of the biggest bottlenecks in many GPU applications. In this chapter, we describe two memory access optimizations for IKRA-RUBY and IKRA-CPP: *Kernel fusion* and a data layout DSL for the *Structure of Arrays* (SOA) layout. These optimizations can improve memory bandwidth utilization and cache performance, and thus increase the overall application performance.

Contents

Outline This chapter is organized as follows. Section 4.1 describes how we extended IKRA-RUBY with kernel fusion. Section 4.2 describes the design and implementation of an embedded SOA data layout DSL for IKRA-CPP. This DSL makes SOA easier to use for CUDA/C++ programmers. Section 4.3 presents an extension of SOA to inner array types and its implementation in IKRA-CPP. Finally, Section 4.4 concludes this chapter.

Publications This chapter is in part based on the following papers.

- Matthias Springer, Peter Wauligmann, Hidehiko Masuhara. **"Modular Array-Based GPU Computing in a Dynamically-Typed Language."** In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming.* ARRAY 2017. ACM, 2017, pp. 48–45. doi:10.1145/3091966.3091974
- Matthias Springer, Hidehiko Masuhara. **"Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming with Structure-of-Arrays Layout."** In: *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing.* WPMVP 2018. ACM, 2018, pp. 1–9. doi:10.1145/3178433.3178439
- Matthias Springer, Yaozhu Sun, Hidehiko Masuhara. **"Inner Array Inlining for Structure of Arrays Layout."** In: *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming.* ARRAY 2018. ACM, 2018, pp. 50–58. doi:10.1145/3219753.3219760

4.1 Kernel Fusion in IKRA-RUBY

Unoptimized IKRA-RUBY programs suffer from two types of slowdowns. IKRA-RUBY optimizes GPU programs using two techniques that are well-known in statically-typed languages but not in dynamically-typed languages such as Ruby.

Global Memory Access IKRA-RUBY encourages a programming style where a GPU application is composed of many small parallel operations. The image manipulation library of Section 3.1.4 illustrates this programming style. A naive compilation strategy compiles each parallel operation into a separate CUDA kernel. This is problematic because the intermediate results of each GPU kernel must be loaded from/stored in global memory. To eliminate this slowdown, IKRA-RUBY *fuses* multiple parallel operations into one large CUDA kernels, such that intermediate results can remain in GPU registers. Kernel fusion is a common and well-studied GPU optimization [194, 64, 165].

Ruby Interpreter Loops Many GPU applications are iterative applications of parallel operations. For example, the *Himeno* benchmark [153] is an iterative application of a stencil operation. If the loops of iterative applications are executed in the Ruby interpreter, the overall program performance can suffer for two reasons. First, the Ruby interpreter is much slower than C++ code. Second, every time a parallel operation executed in Ruby, it is symbolically executed, which incurs some overheads even if the same parallel operation was seen before. To eliminate such slowdowns, we introduce the concept of a *host section*. A host section is a block of Ruby code that contains a more complex program (typically with a loop) with multiple parallel operations. In such a case, the entire block is translated to C++ code, avoiding switching from the Ruby interpreter to external C++ programs multiple times.

Microbenchmarks show that both techniques together achieve performance that is comparable to hand-written CUDA code. As the main contribution of this section, we show how kernel fusion can be implemented as part of the type inference process of host section code.

4.1.1 Kernel Fusion

All array commands except for *index* and *array identity* have at least one input array command (`input` in Figure 3.2). E.g., the inputs of a *combine* operation are the array commands that are being mapped over. During code generation, IKRA-RUBY traverses the tree of dependent (input) commands (*computation graph*). Depending on the access pattern of the dependent commands, IKRA-RUBY may fuse multiple array commands into a single CUDA kernel. This can increase the performance of the generated CUDA code, because intermediate results can remain in GPU registers and do not have to be written back to global memory.

Consider the IKRA-CPP code in Listing 4.1 as an example. This listing contains two chained parallel map operations. Without kernel fusion (Listing 4.2), the first kernel stores the temporary result of `r1` in global memory and the second kernel loads the temporary result of `r1` from global memory. These two loads/stores can be eliminated with kernel fusion (Listing 4.3).

TABLE 4.1: Input access patterns of array commands

Command	Input Access Pattern
combine	same location (for all inputs)
stencil	multiple (fixed pattern)
reduce	multiple
zip	same location (for all inputs)
(`with_index`)	no input, but always accessed as "same location"

Table 4.1 lists access patterns for dependent computations for all array commands. "Same location" means that for the computation of the element at position *i* only the element at the same position in the dependent command(s) is required. For example, to calculate element 12 in *map*, only element 12 from the input is required. "Multiple" means that an array command needs multiple elements from the dependent command(s). For example, a stencil computation requires an entire neighborhood of values from the input.

IKRA-RUBY can currently merge dependent computations only if the access pattern is "same location". In that case, one thread can first compute the dependent operation and then directly proceed with the following computation without any synchronization (Listing 4.3).

In this section, we focus on the process of identifying *which* parallel operations should be merged into a CUDA kernel. We omit implementation details of the compilation process, e.g., *how exactly* the code of one parallel operation is inlined into another parallel operation and how C++/CUDA code is generated from an AST.

Example Figure 4.1 shows an example. White boxes indicate parallel operations and gray boxes indicate fused CUDA kernels. The leftmost gray box corresponds to Lines 2–3. Those operations are fused together because the input access pattern for *combine*/*map* is "same location"[1]. The first input of the stencil computation cannot be fused because its access pattern is "multiple". The *index* input can be fused because input generated by `with_index` is always accessed as "same location".

[1] *id* is added implicitly when programmers use Ruby arrays.

LISTING 4.1: Example program for kernel fusion

```ruby
r0 = (0...1000).to_a
r1 = r0.pmap do |x| 2 * x end
r2 = r1.pmap do |x| x + 1 end
r2.to_a
```

LISTING 4.2: Generated CUDA code without kernel fusion

```cuda
__global__ void kernel_1(float* input, float* output) {
  for (unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
       i < 1000; i += blockDim.x * gridDim.x) { output[i] = 2 * input[i]; }
}

__global__ void kernel_2(float* input, float* output) {
  for (unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
       i < 1000; i += blockDim.x * gridDim.x) { output[i] = input[i] + 1; }
}

void run_gpu_program(float* input, float* output) {
  float *d_data1, *d_data2;
  cudaMalloc(&d_data1, sizeof(float) * 1000);
  cudaMalloc(&d_data2, sizeof(float) * 1000);
  cudaMemcpy(d_data1, input, sizeof(float) * 1000, cudaMemcpyHostToDevice);

  // Read input from d_data1, store temp. result in d_data2.
  kernel_1<<<256, 256>>>(d_data1, d_data2);
  cudaDeviceSynchronize();
  // Read temp. result from d_data2, store final result in d_data1.
  kernel_2<<<256, 256>>>(d_data2, d_data1);
  cudaDeviceSynchronize();

  cudaMemcpy(output, d_data1, sizeof(float) * 1000, cudaMemcpyHostToDevice);
}
```

LISTING 4.3: Generated CUDA code with kernel fusion

```cuda
__global__ void kernel_fused(float* input, float* output) {
  for (unsigned int i = threadIdx.x + blockIdx.x * blockDim.x;
       i < 1000; i += blockDim.x * gridDim.x) {
    float tmp_result_1 = 2 * input[i];
    output[i] = tmp_result_1 + 1;
  }
}

void run_gpu_program(float* input, float* output) {
  float *d_data1, *d_data2;
  cudaMalloc(&d_data1, sizeof(float) * 1000);
  cudaMalloc(&d_data2, sizeof(float) * 1000);
  cudaMemcpy(d_data1, input, sizeof(float) * 1000, cudaMemcpyHostToDevice);

  // Read input from d_data1, store final result in d_data2.
  kernel_fused<<<256, 256>>>(d_data1, d_data2);
  cudaDeviceSynchronize();

  cudaMemcpy(output, d_data2, sizeof(float) * 1000, cudaMemcpyHostToDevice);
}
```

```
A1 = [1, 2, 3]; A2 = [10, 20, 30]                              # Ruby arrays
a = A1.pmap.with_index do |e, idx| ... end                        # combine
b = a.pcombine(A2) do |e1, e2| ... end                            # combine
c = b.pstencil([-1, 0, 1], 0).with_index do |values, idx| ... end # stencil
d = c.preduce do |r1, r2| ... end                                  # reduce
```

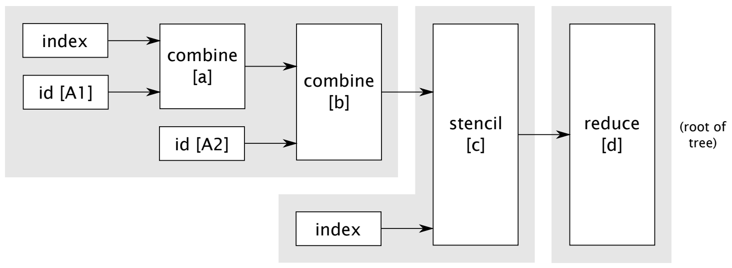


FIGURE 4.1: Example: Kernel fusion result. Gray boxes are kernels.

4.1.2 Host Sections

Many scientific computations (e.g., numerical partial differential equations) exhibit an iterative structure where an array or matrix is updated for a fixed number of times or until convergence. The example code in Listing 4.4 does not compute anything meaningful but illustrates how to write such computations in IKRA-RUBY. The loop is enclosed in a host section, a code block that is compiled to C++ and executed on the host, as opposed to *parallel operations* which are executed on the device. The value of the last statement of a host section is the return value of the host section. Inside host sections, only *simple* Ruby code may be written: Everything that is allowed inside a parallel operation plus parallel operations themselves. More advanced Ruby features (such as metaprogramming) are forbidden.

LISTING 4.4: Example: Iterative computation in host section

```
 1 input = [10, 20, 30, 40, 50, 60]
 2 result = Ikra.host_section do
 3   arr = input.to_command(dimensions: [2, 3])
 4   for i in 0...10
 5     if arr.preduce(:+)[0] % 2 == 0
 6       arr = arr.pmap do |i| i + 1; end                     # map_A
 7     else
 8       arr = arr.pmap do |i| i + 2; end                     # map_B
 9     end
10     arr = arr.pmap do |i| i + 3; end                       # map_C
11   end
12   arr
13 end
```

One C++/CUDA program is generated for the code in Listing 4.4. That program contains a C++ function for the host section and multiple CUDA kernels. As part of code generation, control flow statements inside host sections are executed symbolically as opposed to control flow statements outside of host sections, which are executed by the Ruby interpreter.

Host sections are translated with a conservative kind of ahead-of-time compilation and kernel fusion technique: In the above example, it is not clear until runtime if the parallel operation in Line 10 will be executed together with the one in Line 6 (map_A +

map$_C$) or the one in Line 8 (*map*$_B$ + *map*$_C$), or both in different iterations. Therefore, IKRA-RUBY generates both fused kernel variants and launches the appropriate one at runtime.

4.1.3 Symbolic Execution in Host Sections

Host sections are pieces of Ruby code that are entirely translated to C++ code. They may contain one or more parallel operations but no advanced language features like metaprogramming. The compilation process of host sections is identical to the one of parallel operations, but there are additional steps to handle parallel operations within them. Since no code generation can be done once a host section (C++ code) is executing, fused kernels must be generated up front.

IKRA-RUBY statically analyzes all code paths through a host section with parallel operations and generates a number of fused kernels, even some that might never be used at runtime. At runtime, IKRA-RUBY keeps track of which parallel operations were executed symbolically and eventually launches a fused kernel, which may contain multiple parallel operations, when the result is accessed.

Kernel Fusion via Type Inference Within host sections, there are additional type inference rules to handle parallel operations. IKRA-RUBY performs kernel fusion through type inference: The type of a parallel operation is the array command object that it evaluates (symbolically) to in the Ruby interpreter.

LISTING 4.5: Example: IKRA-RUBY type inference

```
 1 a = 10 # type(a) = Int
 2
 3 b = Array.pnew(a) do |i| 2.5 + i; end
 4 # Eval Ruby: Array.pnew(CodeRef.new(:a)) do ... end
 5 # => ArrayCombineCommand instance
 6 # type(b) = ArrayCombineCommand$_3$[Float, (ArrayIndexCommand[Float, ∅])]
 7
 8 c = b.pmap do |i| 0.5 + i; end
 9 # Eval Ruby: type(b).pmap do ... end
10 # => (another) ArrayCombineCommand instance
11 # type(c) = ArrayCombineCommand$_8$[Float, (
12 #     ArrayCombineCommand$_3$[Float, (ArrayIndexCommand[Float, ∅])])]
13
14 d = Array.pnew(a) do |i| 2.5 + i; end
15 # Eval Ruby: Array.pnew(CodeRef.new(:a)) do ... end
16 # => (yet another) ArrayCombineCommand instance
17 # type(d) = ArrayCombineCommand$_{14}$[Float, (ArrayIndexCommand[Float, ∅])]
```

Listing 4.5 illustrates the type inference rules with an example. The type of variable `a` is `Int`. The type of variable `b` is an `ArrayCombineCommand` object (Figure 3.2). This is because a parallel *new* operation is implemented as a *map/combine* operation wrapped around an *index* operation (see Section 3.1.1). The type of variable `c` is another `ArrayCombineCommand` object.

All array commands with a code block have a subscript that identifies the block. In the above example, code blocks are referred to with line numbers. Therefore, even though the types of `b` and `d` look similar, they are actually different types. For presentation reasons, we only account for array command base types (`Float` in all types in the above example), code blocks and computation structure (i.e., the nesting of array commands) in this section and omit other properties of array command types such as array dimensions or out-of-bounds values of stencil operations.

Notice that `ArrayCommand` objects are not only used for code generation but also serve as type representations/objects in our type inference engine. After type inference, IKRA-RUBY generates a CUDA kernel for each array command type that is *executed*. An expression of array command type is executed when its result is accessed, via either Ruby method `to_a` or array subscript notation (`[]`).

Compilation Overview A host section is translated to C++/CUDA as shown below. This process is similar to symbolic execution of parallel operations in the Ruby interpreter (see Section 3.1). However, since host sections are entirely translated to C++ code, parallel operations are now symbolically executed in C++ instead of in the Ruby interpreter.

1. Retrieve the Ruby source code of the host section.
2. Generate an AST (abstract syntax tree) of the source code.
3. Convert the AST to SSA (static single assignment) form.
4. Insert a `to_a` method call on last expression.
5. Eliminate expressions with circular types by inserting `to_a` method calls.
6. Perform type inference. (This step also performs kernel fusion.)
7. Generate C++ source code for the host section and CUDA source code for all array commands on which `to_a` or `[]` is called.

The SSA form simplifies type inference: If a variable is written a second time with a value of different type, a new C++ variable is allocated and a union type can sometimes be avoided. The return value (last expression) of a host section must be an array command or an array. In the former case, to ensure that an array command is executed, IKRA-RUBY wraps the last expression in a `to_a` method call. That method does not have any effect for arrays but triggers array command execution. The next two steps, eliminating circular types and type inference, are described in detail in the following paragraphs.

4.1.4 Type Inference

Ruby is a dynamically typed language. Since C++ and CUDA are statically typed languages, we represent polymorphic expressions with union types (also known as *sum types* [154]).

For simplicity, we assume that all expressions are monomorphic in this section. Therefore, our types and typing rules do not account for union types. Furthermore, we only describe the typing rules that involve array commands. In addition, we assume that arrays are explicitly wrapped (*boxed*) in array commands before a parallel operation is invoked on them.

Types Figure 4.2 gives a simplified overview of the types in our system that we need for fusion of array commands. An expression can be of primitive type (`TPrim`), of array type with a certain primitive base type (`TArray`) or of array command type

(`TArrayCommand`). There are multiple subtypes of `ArrayCommand`, depending on the type of the operation (Figure 3.2).

Array command types have two parameters. The first parameter is the base type of the array command, similar to the base type of an array. The second parameter is a list of all dependent (input) array command types (`AcInput`). Similar to C++ expression templates [188], an array command type in Ikra-Ruby encodes the structure of a computation in its type. The code generator can use this information to apply additional code optimizations, kernel fusion in the case of Ikra-Ruby.

Typing Rules To perform kernel fusion, our type inference engine captures method calls whose receiver types are a subclass of `ArrayCommand`. The type of such a method call is the result of the evaluation of that method call in the Ruby interpreter, where the receiver and all arguments are replaced with their respective types (T-ParOp). The rule T-ParOp can be broken down into one rule per operation type: T-ParMap, T-ParCombine and rules for other operation types that we omit here.

As an example, consider T-ParCombine. This typing rule captures `pcombine` method calls on receivers with `ArrayCommand` subtype. If this method call has n arguments (excluding the code block), then the code block `b` must be a function taking $n + 1$ arguments: The type of the first argument is the base type of the receiver array command. The types of the following n arguments are the base types of the argument array command types. The resulting type is an `ArrayCombineCommand` with a list of the receiver type and all argument types as the second type parameter, encoding the structure of the computation.

Before programmers can run a parallel operation on an array, they must wrap it in an array command, as described above (T-Box). Similarly, an array command can be converted into an array, which triggers execution on the GPU (T-UnBox).

Breaking Circular Types The type of an array command encodes the structure of its computation, so an array command type effectively encodes a data flow path through the program. If there are multiple possible paths (one of which is chosen at runtime), our type inference system uses a union type that contains all paths. This is problematic with loops or recursion. Since the number of loop iterations is not generally known ahead of time, the union type would grow infinitely, because every additional loop iteration adds another path to the union type.

Listing 4.6: Circular union type

```
1 arr = [1, 2, 3].to_command
2
3 for i in 1...100 do
4   arr = arr.pmap do |x|
5     x + 1
6   end
7 end
8
9 result = arr.to_a
```

Consider Listing 4.6 as an example. After Line 1, the type of variable `arr` is `ArrayIdentityCommand[Int, ∅]`. After the first loop iteration, the type of `arr` is `ArrayCombineCommand`$_4$`[Int, ArrayIdentityCommand[Int, ∅]]`. Every additional loop iteration wraps the type in another `ArrayCombineCommand`$_4$.

$$\begin{aligned}
\texttt{TPrim} &::= \texttt{Bool} \mid \texttt{Float} \mid \texttt{Int} \\
\texttt{TArray} &::= \texttt{Array}[\texttt{TPrim}] \\
\texttt{TArrayCommand} &::= \texttt{ArrayCommand}_{loc}[\texttt{TPrim}, \texttt{AcInput}] \\
&\quad \mid \texttt{ArrayIdentityCommand}[\texttt{TPrim}, \emptyset] \\
&\quad \mid \texttt{ArrayIndexCommand}[\texttt{Int}, \emptyset] \\
&\quad \mid \texttt{ArrayCombineCommand}_{loc}[\texttt{TPrim}, \texttt{AcInput}] \\
&\quad \mid \texttt{ArrayStencilCommand}_{loc}[\texttt{TPrim}, (\texttt{TArrayCommand})] \\
&\quad \mid \texttt{ArrayReduceCommand}_{loc}[\texttt{TPrim}, (\texttt{TArrayCommand})] \\
\texttt{AcInput} &::= \emptyset \mid \texttt{TArrayCommand} \circ \texttt{AcInput} \\
\texttt{T} &::= \texttt{TPrim} \mid \texttt{TArray} \mid \texttt{TArrayCommand}
\end{aligned}$$

(types omitted for classes, zip types, nil and arrays with non-primitive base types)

FIGURE 4.2: List of types

$$\frac{\Gamma \vdash e_i : T_i \qquad T_i <: \texttt{ArrayCommand}_*[*, *]}{\Gamma \vdash e_0\texttt{.m(}e_1\texttt{, ..., }e_n\texttt{, \&b)} : eval(T_0\texttt{.m(}T_1\texttt{, ..., }T_n\texttt{, \&b))}} \quad \text{(T-PAROP)}$$

$$\frac{\Gamma \vdash e_0 : T_0 \qquad T_0 <: \texttt{ArrayCommand}_*[B_0, *] \qquad \Gamma \vdash \texttt{b} : B_0 \to R}{\Gamma \vdash e_0\texttt{.pmap(\&b)} : \texttt{ArrayCombineCommand}_{loc(\texttt{b})}[R, (T_0)]} \quad \text{(T-PMAP)}$$

$$\frac{\Gamma \vdash e_i : T_i \qquad T_i <: \texttt{ArrayCommand}_*[B_i, *] \qquad \Gamma \vdash \texttt{b} : B_0 \to B_1 \to \ldots \to B_n \to R}{\Gamma \vdash e_0\texttt{.pcombine(}e_1\texttt{, ..., }e_n\texttt{, \&b)} : \texttt{ArrayCombineCommand}_{loc(\texttt{b})}[R, (T_0, T_1, \ldots, T_n)]} \quad \text{(T-PCOMBINE)}$$

(rules omitted for `pnew`, `preduce`, `pstencil` and `pzip`)

$$\frac{\Gamma \vdash e_0 : \texttt{Array}[B]}{\Gamma \vdash e_0\texttt{.to\_command()} : \texttt{ArrayIdentityCommand}[B, \emptyset]} \quad \text{(T-BOX)}$$

$$\frac{\Gamma \vdash e_0 : T_0 \qquad T_0 <: \texttt{ArrayCommand}_*[B, *]}{\Gamma \vdash e_0\texttt{.to\_a()} : \texttt{Array}[B]} \quad \text{(T-UNBOX)}$$

FIGURE 4.3: Typing rules

Since the number of loop iterations is generally unknown until runtime, the type of `arr` after the loop is an infinite union type with one type for each possible number of loop iterations.

$$\begin{aligned} \mathit{type}(\texttt{arr}) = \{\ & \mathit{id}, \\ & \mathit{map}_4[\mathit{id}], \\ & \mathit{map}_4[\mathit{map}_4(\mathit{id}]], \\ & \ldots, \\ & \mathit{map}_4[\mathit{map}_4[\ldots \mathit{map}_4[\mathit{id}]\ldots]]\ \} \end{aligned}$$

We abbreviate `ArrayCombineCommand` with *map* and `ArrayIdentityCommand` with *id* in the above formula for presentation reasons. Furthermore, we omit base types.

The infinitely large type of `arr` is a problem. Line 9 of the source code accesses the result of `arr` and will trigger CUDA kernel execution. Since `arr` is a union type, this will translate to a switch-case statement with one case per type. Our type inference engine avoids such circular types by changing the source code of the program. The type T of an expression is *circular* if T is included as a dependent array command of T. Not necessarily as a directly dependent array command, but maybe somewhere nested deep inside T. For example, given the type $T = \mathit{map}_4[\mathit{id}]$, another iteration of the loop yields the type $T' = \mathit{map}_4[\mathit{map}_4[\mathit{id}]] = \mathit{map}_4[T]$, so `arr` has a circular type.

Whenever a circular type is detected, we insert two chained method calls `to_a()` and `to_command()`, which execute the CUDA kernel and reset the type (Listing 4.7).

LISTING 4.7: Eliminated circular union type

```
arr = [1, 2, 3].to_command

for i in 1...100 do
  arr = arr.pmap do |x|
    x + 1
  end .to_a.to_command
end

result = arr.to_a
```

The elimination of circular types is baked into our type inference engine and we do not describe it in more detail in this section. There may be multiple ways of eliminating a circular type. Where exactly we insert the two chained method calls is an implementation details and not important. What is important is that the type of `arr` after the loop of Listing 4.7 is now no longer circular.

$$\mathit{type}(\texttt{arr}) = \{\ \mathit{id}, \mathit{map}_4[\mathit{id}]\ \}$$

Type Inference by Example To illustrate type inference in a larger example, consider the host section source code of Listing 4.8, which is identical to Listing 4.4, but in SSA form and with a `to_a` method call at the end.

In the following paragraphs, we take a look at the inferred types of all $\texttt{arr}_i$ variables. $\texttt{arr}_1$ is an identity command for the Ruby array `input` and $\texttt{arr}_2$ cannot be fully inferred yet because $\texttt{arr}_6$ is still unknown.

LISTING 4.8: Example: Iterative IKRA-RUBY computation in host section in SSA form

```
1  result = Ikra.host_section do
2      arr1 = input.to_command(dimensions: [2, 3])
3      for i in 0...10
4          arr2 = φ(arr1, arr6)
5          if arr2.preduce(:+)[0] % 2 == 0
6              arr3 = arr2.pmap do |i| i+1; end                    # mapA
7          else
8              arr4 = arr2.pmap do |i| i+2; end                    # mapB
9          end
10         arr5 = φ(arr3, arr4)
11         arr6 = arr5.pmap do |i| i+3; end                        # mapC
12     end
13     arr7 = φ(arr1, arr6)
14     arr7.to_a
15 end
```

$$\begin{aligned} type(\texttt{arr}_1) &= id \\ type(\texttt{arr}_2) &= \{\ type(\texttt{arr}_1), type(\texttt{arr}_6)\ \} = \{\ id, type(\texttt{arr}_6)\ \} \end{aligned}$$

Next, we infer the types for the first two *map* operations. Different subscripts of *map* operations indicate that the operations are different array commands.

$$\begin{aligned} type(\texttt{arr}_3) &= map_A[type(\texttt{arr}_2)] = \{\ map_A[id], map_A[type(\texttt{arr}_6)]\ \} \\ type(\texttt{arr}_4) &= map_B[type(\texttt{arr}_2)] = \{\ map_B[id], map_B[type(\texttt{arr}_6)]\ \} \\ type(\texttt{arr}_5) &= \{\ type(\texttt{arr}_3), type(\texttt{arr}_4)\ \} \\ &= \{\ map_A[id], map_A[type(\texttt{arr}_6)], map_B[id], map_B[type(\texttt{arr}_6)]\ \} \end{aligned}$$

Next, we infer the type of the last *map* operation.

$$\begin{aligned} type(\texttt{arr}_6) &= map_C[type(\texttt{arr}_5)] \\ &= \{\ map_C[map_A[id]], map_C[map_A[type(\texttt{arr}_6)]], \\ &\qquad map_C[map_B[id]], map_C[map_B[type(\texttt{arr}_6)]]\ \} \end{aligned}$$

As can be seen from the definitions above, the type of $\texttt{arr}_6$ is circular. If we try to fully expand its definition, it will have an infinite number of elements.

IKRA-RUBY breaks this circular type by inserting a `to_a.to_command` method call, which will launch the kernel, return its result as an array and then box it in another array command. Consequently, this method call will stop the kernel fusion process. IKRA-RUBY currently inserts the method call in Line 11, but it could also be inserted in Lines 4 or 10.

```
11 arr6 = arr5.to_a.to_command.pmap do |i| i + 3; end              # mapC
```

We can now complete the type inference process and fill in the $\texttt{arr}_2$ placeholders in the other definitions.

$$
\begin{aligned}
type(\texttt{arr}_6) &= map_C[id] \\
type(\texttt{arr}_2) &= \{\ id, map_C[id]\ \} \\
type(\texttt{arr}_5) &= \{\ type(\texttt{arr}_3), type(\texttt{arr}_4)\ \} \\
&= \{\ map_A[id], map_A[map_C[id]], map_B[id], map_B[map_C[id]]\ \} \\
type(\texttt{arr}_7) &= \{\ type(\texttt{arr}_1), type(\texttt{arr}_6)\ \} = \{\ id, map_C[id]\ \}
\end{aligned}
$$

For code generation, only $\texttt{arr}_2$, $\texttt{arr}_5$ and $\texttt{arr}_7$ are of interest, because their result is accessed. IKRA-RUBY generates kernels and invocations for them: The static type of a variable (or union type class ID field if polymorphic) determines the kernel to be launched. In total, IKRA-RUBY generates the following kernels in this example. Some of them might never be launched at runtime.

(5.1) $map_A[id]$	(7.1) id
(5.2) $map_A[map_C[id]]$	(7.2) $map_C[id]$
(5.3) $map_B[id]$	(r.1) $reduce[id]$
(5.4) $map_B[map_C[id]]$	(r.2) $reduce[map_C[id]]$

4.1.5 Code Generation

After type inference, IKRA-RUBY generates C++ source code for the host section. Expressions of array command type, have a generated `array_command_t` type in the C++ code. This struct contains a pointer to the cached result of the array command. If an expression's polymorphic type can be one of multiple array commands, then IKRA-RUBY uses a union type in the generated C++ code and the class ID indicates the exact array command at runtime.

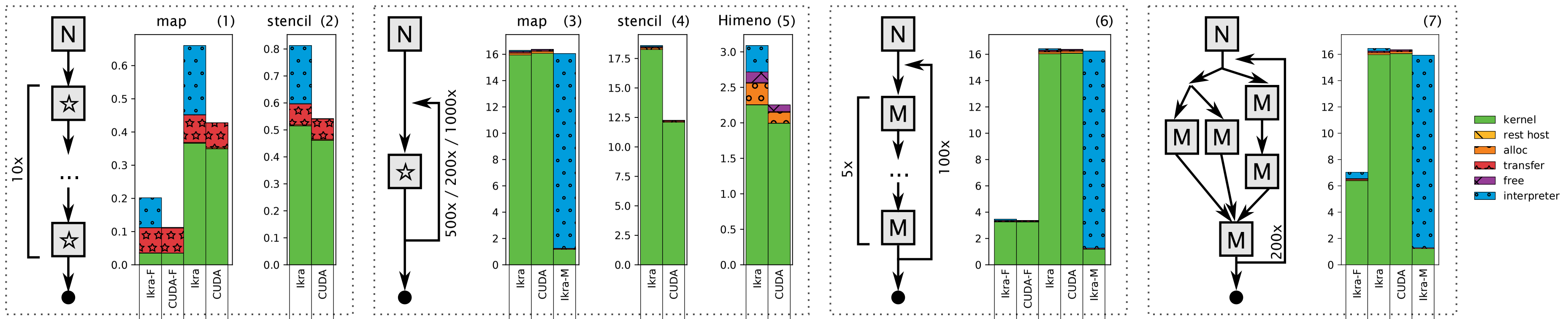
In the above example, all variables except for $\texttt{arr}_1$ are polymorphic and will have type `union_t` in the generated C++ code. The class ID field is used to determine which kernel should be launched. For example, either kernel 7.1 or kernel 7.2 should be launched in Line 14 depending on whether there was at least one loop iteration. This information is implicitly encoded in the class ID field and IKRA-RUBY generates a switch-case statement, similar to polymorphic method calls. In the generated code, Lines 2, 4, 6, 8, 10 and 13 do not launch a kernel but merely return a new `array_command_t` object, possibly wrapped inside a union type struct containing the class ID for the command.

Whenever an array command access is detected, IKRA-RUBY generates a CUDA kernel for the array command and a kernel invocation snippet which checks if a cached result is available and otherwise transfers data (if necessary) and launches the kernel. Since array commands may have dependent commands, generated kernels may consist of multiple fused parallel operations.

4.1.6 Benchmarks

Figure 4.4 shows the runtime performance of a number of microbenchmarks[2] (small parallel operations, only *for* loops) of the current IKRA-RUBY implementation in various configurations. Benchmarks were run on a computer with an Intel Core i7-6820HQ CPU (2.70 GHz), 32 GB RAM, an NVIDIA GeForce 940MX GPU, Ubuntu

[2]Source code: `https://github.com/prg-titech/ikra-ruby`, branch `array17`



	Ikra-F (1)	CUDA-F (1)	Ikra (1)	CUDA (1)	Ikra (2)	CUDA (2)	Ikra (3)	CUDA (3)	Ikra-M (3)	Ikra (4)	CUDA (4)	Ikra (5)	CUDA (5)	Ikra-F (6)	CUDA-F (6)	Ikra (6)	CUDA (6)	Ikra-M (6)	Ikra-F (7)	Ikra (7)	CUDA (7)	Ikra-M (7)
style	no loop						simple loop							simple loop					complex loop			
kernel operation	map				stencil		map			stencil				map					map			
#loop iterations	n/a				n/a		500			200		1000		100					200			
#kernels in loop	n/a				n/a		1			1		1		5					5			
with fusion	✓	✓							✓					✓	✓			✓	✓			✓
with host section	✓		✓		✓		✓			✓		✓		✓		✓			✓			
#kernels after fusion	1	1							1					4	2			1	8			1
#runtime kernel invocations	1	1	11	11	11	11	501	501	1	201	201	1001	1001	101	101	501	501	1	201	var.	var.	1

FIGURE 4.4: Microbenchmark runtime in seconds. *Ikra-F* is IKRA-RUBY with all code in a single host section and kernel fusion. *Ikra* is without kernel fusion. *Ikra-M* is a lower bound where all code is in a single kernel (even among iterations). *CUDA-F* and *CUDA* are hand-written baseline implementations with/without (manual) kernel fusion. Compilation time (not shown here) is around 2 seconds for IKRA-RUBY-generated code.

16.04.1 (kernel version `4.4.0-43-generic`), Ruby 2.3.1 and the CUDA Toolkit V8.0.44. Program 5 is a 3D stencil computation on a matrix of size $129 \times 65 \times 65$. All other programs operate on matrices of size 6×10^7 (228 MB) with a one-dimensional CUDA block size of 1024.

The benchmarks show the performance speedup due to kernel fusion and how generated IKRA-RUBY code performs in comparison to hand-written CUDA code. We compare 7 different programs with various compilation strategies. For every program, we show the structure of parallel operations (control/data flow graph). The square boxes indicate parallel operations and the arrows indicate data flow. The letter *M* indicates a *map* operation, the letter *N* a *new* operation, and the star a can be either *map* or *stencil*.

Programs 1, 6 and 7 show the benefit of kernel fusion. In Program 1, a single kernel is launched for all 11 parallel operations, giving a 10x speedup compared to the version without kernel fusion. In Program 6, 101 kernels are launched for 501 parallel operations. All 5 kernels within the loop are fused together, giving a 5x speedup compared to the version without kernel fusion. If all 501 *map*/*new* operations are fused together by executing the loop in the Ruby interpreter (no host section; *Ikra-M*), another 2x speedup is possible. However, in the general case, the number of loop iterations is unknown. Moreover, the resulting kernel code becomes large; increasing the number of iterations too much even results in an `nvcc` compilation error. As can be seen from the *interpreter* time, IKRA-RUBY also spends a very long time in the Ruby interpreter if a tree of 501 array commands is analyzed and fused together. This shows that there is still potential for optimization of the part of IKRA-RUBY that runs in the Ruby interpreter.

Program 7 shows the benefit of kernel fusion in a program with more complex control flow, giving a speedup of 2x. This program generates source code with union types to keep track of which kernel to launch at the end of an iteration. This leads to additional runtime overheads. However, this overhead (*rest host*) is much smaller than the kernel runtime and could not even be measured with confidence in our experiments. The reason that Program 7 does not achieve the same speedup from kernel fusion as Program 6 is that the number of parallel operations inside the loop is smaller and the number of loop iterations is larger (200 vs. 100).

Programs 3–5 consist of a loop with a single *map* or *stencil* operation. In such cases, IKRA-RUBY cannot perform kernel fusion inside the loop, which is why we only report the performance for *Ikra*. *Himeno* is a benchmark with a memory-bound stencil computation. It has a high *alloc* time because IKRA-RUBY (and the CUDA baseline) do currently not reuse memory but allocate a new piece of memory every time an array is updated within the loop.

All benchmarks have a low *transfer* time, i.e., time spent for transferring data between the device and the host. This is because data is transferred to the host only when the result of a parallel operation is accessed. The loops in all benchmarks are *for* loops with a fixed number of iterations. Only after the last iteration, the result is accessed and transferred back to the host. In a more realistic case, where the control flow (e.g., number of iterations) depends on the data, more data transfer will occur.

When comparing IKRA-RUBY's performance with hand-written CUDA code, we can see that the kernel running times are almost identical for *map* operations. The generated code of *stencil* operations is not yet fully optimized (or fused). In addition, IKRA-RUBY spends some time in the Ruby interpreter for performing type inference and generating Ruby code. Overall, it spends very little time in the remaining host section code (allocating/comparing union types/array commands, loop overhead).

Number of Generated Kernels Due to the kernel fusion process described in Section 4.1.3, IKRA-RUBY generates one kernel per data flow path (excluding loops). This can lead to a combinatorial explosion of the number of generated kernels. Based on an analysis of the kernel structure of a large number of parallel programs by Shen et al. [168], we believe that the number of kernels remains manageable in real applications. Their work showed that the kernel structure of all analyzed programs is similar to the ones in our benchmarks[3] and never more complex than the structure in Program 7.

4.1.7 Future Work

Besides improving the overall efficiency of our implementation, we plan extend IKRA-RUBY with kernel fusion of stencil operations and with better memory management in future. Furthermore, there are cases when kernel fusion can lead to a slowdown of programs, e.g., if the first kernel contains highly divergent control flow. We will analyze such programs in the future.

Kernel Fusion of Stencil Operations IKRA-RUBY can currently only fuse operations whose input pattern is "same location". We plan to extend kernel fusion to certain stencil computations that exhibit a *simple* neighborhood. Stencil computations can be fused with shared memory or with redundant computations. Consider, for example, the following two stencil operations.

$$\begin{aligned} A_1 &= \mathit{stencil}(A_0, [-1,0], f, 10) \\ A_2 &= \mathit{stencil}(A_1, [-1,0], g, 10) \end{aligned}$$

The resulting arrays after each iteration are defined as follows.

$$A_1 = \begin{bmatrix} 10 \\ f(A_0[0], A_0[1]) \\ f(A_0[1], A_0[2]) \\ \dots \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 10 \\ g(10, f(A_0[0], A_0[1])) \\ g(f(A_0[0], A_0[1]), f(A_0[1], A_0[2])) \\ \dots \end{bmatrix}$$

The definition of A_2 represents the computation of a fused stencil operation. Many arguments of function g are computed twice (redundantly). To reduce this overhead, IKRA-RUBY could split A_0 into multiple subarrays, assign each subarray to a CUDA block and store intermediate results in shared memory. Inter-block synchronization or redundant computation is then only necessary at the subarray borders (*ghost region* [105]). An optimized version of this technique is known as *temporal blocking* [133].

[3] IKRA-RUBY's programming style could increase the number of parallel sections.

Reusing Memory Many programs that update a vector or matrix iteratively only need access to the data of the preceding iteration. To save memory and for performance reasons (e.g., caching), CUDA programmers allocate only one (for *combine* operations) or two (for *stencil* operations) arrays in hand-written code and keep writing to these arrays. However, all parallel operations in IKRA-RUBY (except `peach`) are *functional* and return a new array instead of modifying the existing array in-place. Furthermore, IKRA-RUBY does not have a garbage collector, so the memory is released only after the execution of the host section or if the programmer frees memory explicitly in the Ruby code.

To decide whether it is safe to reuse a previously allocated array, IKRA-RUBY must perform an escape analysis. This is to ensure that data from a previous iteration is not accessed at a later point of time. Such advanced memory management issues are subject to future work.

4.1.8 Related Work

Kernel fusion is an optimization that is supported in many other GPGPU frameworks and languages, but their focus is on different aspects. For example, Harlan [84] supports nested kernels, Futhark [82] has support for nested parallelism and a powerful fusion engine for map/reduce combinations, and Kernel Weaver focuses on database queries [208]. Furthermore, all of these tools focus on statically-typed programming languages, making translation within a kernel easier compared to IKRA-RUBY, because no union types are required and making kernel fusion itself easier, because it is known ahead of execution time which kernels are executed together.

4.2 A Data Layout DSL for IKRA-CPP

Maintaining an SOA layout in C++/CUDA manually is troublesome. SOA code is less readable and less expressive than AOS code and native C++ language constructs for object-oriented programming cannot be used in SOA code.

To give programmers the performance benefit of SOA and the expressiveness of AOS-style object-oriented programming at the same time, we extended IKRA-CPP with a lightweight, embedded DSL[4] (Listing 4.9). This DSL is implemented entirely in C++ with template metaprogramming, operator overloading, helper classes and preprocessor macros. It can be used on CPUs and GPUs and should work with every modern C++14 compiler and the NVIDIA CUDA Toolkit 9.0 or higher (in GPU mode).

Contribution The main contribution of IKRA-CPP is twofold. First, to the best of our knowledge, IKRA-CPP is the first C++ data/memory layout DSL that supports a wide range of OOP abstractions, most notably member function calls and constructors. Second, there exist other solutions that allow for a limited set of OOP abstractions (e.g., referencing objects with class pointers instead of IDs) with a custom data/memory layouts in C++ (Section 4.2.6). However, these solutions rely on external tools such as custom preprocessors or compiler/language extensions. In contrast, IKRA-CPP is implemented entirely in C++ and requires only a modern C++ compiler.

[4] IKRA-CPP's data layout DSL is a *performance-oriented DSL*, aiming to "make the compiler more productive (producing better code)" [162]. We call it a *DSL* because it defines a new syntax/notation for application code.

LISTING 4.9: N-body simulation in IKRA-CPP: SOA layout but AOS notation

```
 1 class Body : public IkraSoaBase<Body, 50> {
 2  public: IKRA_INITIALIZE_CLASS
 3   float_ pos_x = 0.0;
 4   float_ pos_y = 0.0;
 5   float_ vel_x = 1.0;
 6   float_ vel_y = 1.0;
 7   float_ force_x;
 8   float_ force_y;
 9   float_ mass;
10
11   Body(float m, float x, float y) : mass(m), pos_x(x), pos_y(y) {}
12
13   void move(float dt) {
14     pos_x = pos_x + vel_x * dt;
15     pos_y = pos_y + vel_y * dt;
16   }
17 };
18
19 // This macro defines SOA arrays and an object (instances) counter.
20 IKRA_HOST_STORAGE(Body);
21
22 void create_and_move() {
23   Body* b = new Body(150, 1.0, 2.0);
24   b->move(0.5);
25   assert(b->pos_x == 1.5);
26 }
```

4.2.1 Language Overview

In this section, we describe the basic functionality of IKRA-CPP's data layout DSL, focusing on host (CPU) code.

Notation and API A class whose objects are stored as SOA is called a *SOA class* and its instances are called *SOA objects*. Recall that classes in IKRA-CPP must inherit from `IkraBase`. This class template is for AOS layouts. If we would like to store objects in SOA layout, classes must inherit from `IkraSoaBase`. This class template provides useful helper methods and type aliases. Same as with `IkraBase`, the maximum number of instances of an SOA class is a compile-time constant and template parameter of `IkraSoaBase` (Listing 4.9, Line 1).

The main programming restriction of `IkraSoaBase` compared to `IkraBase` is that SOA objects can only be created with the `new` keyword and must be referred to with pointers or references (Listing 4.10). Stack/static allocation is not allowed, because the fields of an object are not stored as a consecutive chunk of data as in a traditional AOS layout. They can only reside within a structure of arrays, i.e., within the storage buffer generated by `IKRA_*_STORAGE`.

There is one main change in notation compared to our original IKRA-CPP implementation from Section 3.2. Regardless of AOS or SOA layout, programmers must now use special *proxy field types* for field declarations. These proxy types are available for all primitive C++ types and end with an underscore, e.g., `float_` instead of `float` (Listing 4.9, Lines 3–9). While the old notation (plain `float`) is technically still supported for AOS layouts, we recommend using proxy types because the layout of a class can then be switched from AOS to SOA or vice versa with minimal effort by changing the superclass from `IkraBase` to `IkraSoaBase` or vice versa.

LISTING 4.10: Allocation restrictions of `IkraSoaBase`

```
class A : public IkraBase<A, 100> { /* ... */ };
class B : public IkraSoaBase<B, 100> { /* ... */ };

IKRA_HOST_STORAGE(A);
IKRA_HOST_STORAGE(B);

int main() {
  A obj1; // OK, but stored outside of the array of structures.
  B obj2; // Error
  A* ptr1 = new A(); // OK. Allocated as AOS within storage buffer.
  B* ptr2 = new B(); // OK. Allocated as SOA within storage buffer.
  B& obj3 = *obj2; // OK
}

void function1(B obj); // Error
void function2(B* ptr); // OK
void function3(B& ptr); // OK
```

Supported C++ OOP Features In SOA mode, IKRA-CPP supports many but not all OOP features/abstractions of C++. This paragraph gives an overview of the main supported features and the main restrictions.

- Same as in AOS mode, classes are defined with **standard C++ notation** (`class` keyword). Classes cannot be templatized and class inheritance is not possible. Programmers must use the two proprocessor macros `IKRA_INITIALIZE_CLASS` and `IKRA_*_STORAGE`.
- Objects must be referred to with **class pointers** or object references.
- Member fields must be declared with **proxy types**. IKRA-CPP provides proxy types for all **primitive types**. Furthermore, there is a notation for defining proxy types for other base types.
- Member functions must be **non-virtual** but can be **templatized**.
- **Class constructors**, including field initializers, are supported.
- Class constructors and member functions can be **overloaded**.
- Instance creation is supported **only with the `new` keyword**. Same as in AOS mode, existing objects cannot be deleted (allocation only, no deallocation).
- Given an object pointer, members can be accessed with the C++ **member of pointer** (arrow) operator. Given an object reference, members can be accessed with the C++ **member of object** (dot) operator. This is the main feature that gives IKRA-CPP code an object-oriented *look and feel*, even in SOA mode.

We will lift some of these restrictions with DYNASOAR, which is an extension of IKRA-CPP with a dynamic memory allocator (Section 5).

4.2.2 Implementation Details

IKRA-CPP is implemented entirely in C++. It does not need a separate compiler or preprocessor/code generator. It is based on four ideas:

- All data is stored in a large, statically-allocated **storage buffer** (`char` array, generated by `IKRA_*_STORAGE`).
- Upon allocation, all objects are assigned unique **integer IDs** (starting from 1).
- Objects are referenced with **fake pointers** that encode their object ID. This is a form of *type punning*, i.e., "a programming technique that subverts or circumvents the type system of a programming language in order to achieve an effect that would be difficult or impossible to achieve within the bounds of the formal language" [202]. Type punning can lead to fragile code and a future, more mature implementation of IKRA-CPP could rely on a more powerful preprocessor such as ROSE [157].
- Several **operators** of proxy field types are **overridden** to decode IDs from fake pointers and to calculate physical memory addresses within the storage buffer.

In this section, we explain those concepts in more detail, using a more verbose notation. The code in Listing 4.11 is identical to the one in Listing 4.9 (without constructor), but with simplified, expanded preprocessor macros. In particular, note that tokens like `float_` are preprocessor macros that expand to different proxy field type instantiations like `float__<...>`[5], from now on simply called *proxy types*.

LISTING 4.11: Macro-expanded `Body` class from a Listing 4.9 (simplified)

```
class Body : public IkraSoaBase<Body> {
  const static int kMaxInst = 50;
  const static int kObjSize = 7 * 4;                    7 floats = 28 bytes

  static char storage[kMaxInst * kObjSize];
  static int size = 0;

  float__<1, 0> pos_x = 0.0;                  field index = 1, offset = 0
  float__<2, 4> pos_y = 0.0;                  field index = 2, offset = 4
  float__<3, 8> vel_x = 1.0;                  field index = 3, offset = 8
  float__<4, 12> vel_y = 1.0;                field index = 4, offset = 12
  float__<5, 16> force_x;                    field index = 5, offset = 16
  float__<6, 20> force_y;                    field index = 6, offset = 20
  float__<7, 24> mass;                       field index = 6, offset = 24

  static Body* get(int id);                             Calculate Address

  void* operator new() { return get(++size); }

  void move(float dt) {
    pos_x = pos_x + vel_x * dt;
    pos_y = pos_y + vel_y * dt;
  }
};
```

SOA Objects SOA objects can only be referred to with pointers or references. In particular, they cannot be stack-allocated. This is because the address of an SOA object is not a physical memory location but encodes an object ID (*fake pointer*). Based on an object ID and the memory locations of the SOA arrays within the storage

[5]Internally, this is implemented with the `__COUNTER__` preprocessor macro, which is supported by most compilers. See https://gcc.gnu.org/onlinedocs/cpp/Common-Predefined-Macros.html.

buffer, the actual, physical memory location of each field value of that object can be computed. This can be seen as a form of *address translation*. This translation is performed transparently and implemented in C++, as part of the data layout DSL.

All objects of an SOA class `C` have unique IDs between $[1; maxInst(\texttt{C})]$, where *maxInst*(`C`) is the user-specified maximum number of objects of `C`. ID 0 is reserved for null pointers. E.g., running `new C()` for the first time returns a `C*` pointer encoding ID `1`. This pointer does *not* point to an actual memory location. Accessing the address of this fake pointer would most likely result in a memory access violation (Listing 4.12).

LISTING 4.12: Dereferencing a fake pointer

```cpp
C* obj = new C();
char* ptr = reinterpret_cast<char*>(obj);
printf("%c\n", *ptr); // This usually works, but with IkraSoaBase it segfaults!
```

Proxy Types SOA fields are declared with *proxy types* (`Field_` template instantiations, Listing 4.13). These types behave like normal C++ types (base types) in most cases, but access data at a different physical location inside the storage buffer. Proxy type values always appear as *lvalues*, i.e., as values that have a memory location. This is because their implementation calculates the actual, physical data location based on their own lvalue address. Proxy types support the following operations, implemented via operator overloading.

- *Reading a Value:* A proxy type lvalue can be converted to its base type value without an explicit typecast (*implicit conversion operator*[6], Line 17).
- *Writing a Value:* A base type value and a proxy type lvalue can be assigned to a proxy type lvalue by overloading the *assignment operator* (Lines 20, 45).
- *Method Call:* For proxy type lvalues with a pointer base type, a method call is forwarded to the object at the physical data location by overloading the *member of pointer "arrow" operator*[7] (Line 23).
- *Address-of:* It is possible to take the address of a proxy type lvalue (*address-of operator*, Line 26).
- *Dereference:* It is possible to dereference a proxy type lvalue (*pointer dereference operator*, Line 29) if its base type is a pointer type.
- *Array acccess:* It is possible to access a proxy type lvalue of array base type with array syntax (*array subscript operator*, Line 34).
- *Initialization:* Proxy type lvalues can be initialized with base type values or proxy type lvalues; e.g., with a field initializer of a class constructor (constructor, Lines 14, 40).

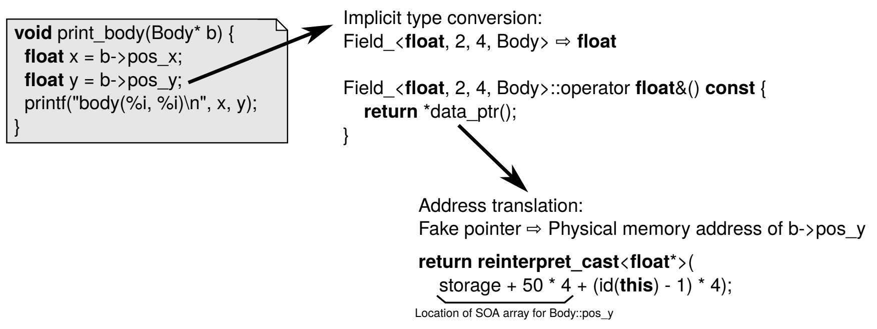
SOA field types are defined in `IkraSoaBase` as template instantiations of `Field_` (Listing 4.13). This class provides the necessary operator implementations and calculates the address inside the storage buffer at which the field value of a certain object can be found (Line 49).

[6]The `auto` keyword is not supported. E.g., a proxy type lvalue cannot be assigned to a variable declared as `auto` without an explicit type cast, unless the variable is of reference type.

[7]Note for experienced C++ programmers: This is similar to how `std::unique_ptr` is implemented.

LISTING 4.13: Implementation of proxy types

```
template<class Self>                                    Template Parameter "Self": CRTP [33]
class IkraSoaBase {
  template<int Index, int Offset>
  using float__ = Field_<float, Index, Offset, Self>;

  template<int Index, int Offset>
  using int__ = Field_<int, Index, Offset, Self>;
};

template<typename T, int Index, int Offset, class Owner>
class Field_ {
  // Constructor
  Field_() {}
  Field_(const T& value) { *data_ptr() = value; }

  // Implicit conversion (implicit type cast)
  operator T&() const { return *data_ptr(); }

  // Assignment
  void operator=(const T& value) { *data_ptr() = value; }

  // Method call
  T* operator->() const { return data_ptr(); }

  // Address-of
  T* operator&() { return data_ptr(); }

  // Pointer dereference
  typename std::remove_pointer<T>::type& operator*() {
    return **data_ptr();
  }

  // Array access (subscript operator) to std::array
  T::value_type& operator[](size_t pos) {
    return (*data_ptr())[pos];
  }

  // Support assignment of other proxy types.
  template<int Index2, int Offset2, class Owner2>
  Field_(const Field_<T, Index2, Offset2, Owner2>& value) {
    *data_ptr() = *value.data_ptr();
  }

  template<int Index2, int Offset2, class Owner2>
  void operator=(const Field_<T, Index2, Offset2, Owner2>& value) {
    *data_ptr() = *value.data_ptr();
  }

  T* data_ptr() const;                                  Calculate Address (details later)
};
```



```
void print_body(Body* b) {
  float x = b->pos_x;
  float y = b->pos_y;
  printf("body(%i, %i)\n", x, y);
}
```

Implicit type conversion:
Field_<float, 2, 4, Body> ⇨ float

```
Field_<float, 2, 4, Body>::operator float&() const {
   return *data_ptr();
}
```

Address translation:
Fake pointer ⇨ Physical memory address of b->pos_y

```
return reinterpret_cast<float*>(
   storage + 50 * 4 + (id(this) - 1) * 4);
```

Location of SOA array for Body::pos_y

FIGURE 4.5: Address translation of a fake pointer

Custom Proxy Types IKRA-CPP provides proxy type macros such as `float_` for all primitive C++ types. Proxy types of other types can be defined with the helper macro `field_` (Listing 4.14). This macro takes as argument the desired base type.

LISTING 4.14: Field proxy type notation

```
class DummyClass : IkraSoaBase<DummyClass, 50> {
 public: IKRA_INITIALIZE_CLASS
  float_ field_0; // Base type: float
  field_(void*) field_1; // Base type: void*
  field_(std::array<float, 10>) field_2; // Base type: std:array<float, 10>
  int_ field_3; // Base type: int
  /* ... */
}; IKRA_HOST_STORAGE(DummyClass);
```

Address Computation – Simplified Proxy types perform a form of address translation: They translate a fake pointer into a physical memory address (Figure 4.5). Proxy objects always appear as lvalues, i.e., values that have an address. This address does not point to allocated memory but is based on a fake pointer.

In the most basic case, given the address of a field proxy object *this* (i.e., an object of type `Field_<...>`), the address of a field value `C::f` can be computed with the formula below. *id* is a function that decodes the object ID from a proxy object address. Note that we always translate addresses from the perspective of a field proxy object and not from the perspective of an SOA object, because we overloaded the operators of `Field_` and not the operators of the SOA class.

$$\begin{aligned} \mathit{addr}(\mathit{this}, \texttt{C::f}) = &\ \mathit{storage} \\ &+ \mathit{maxInst}(\texttt{C}) \cdot \mathit{offset}(\texttt{C::f}) \\ &+ (\mathit{id}(\mathit{this}) - 1) \cdot \mathit{sizeof}(\texttt{C::f}) \end{aligned}$$

The first two lines in the equation compute the beginning of the *SOA array* storing all values of `C::f`. The third line computes the offset into that array. How exactly object IDs are encoded in fake pointers is determined by the *addressing mode* and described in the next few paragraphs. IKRA-CPP supports three different addressing modes, one of which must be chosen at compile time: *Zero Addressing* and two variants of *Valid Addressing*. The former one is more space-efficient but relies on non-standard C++ constructs, so it might not work with some compilers[8].

[8]We verified that it works with g++ 5.4.0, clang 3.8.0 and CUDA 9.0.

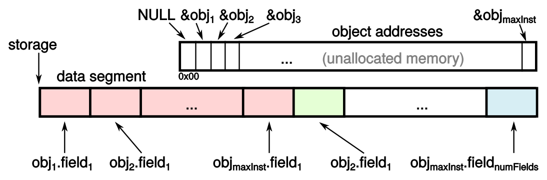


FIGURE 4.6: Storage buffer layout in zero addressing

4.2.3 Addressing Modes

This section describes of three addressing modes. Zero addressing and storage-relative zero addressing are implemented in IKRA-CPP. In accordance with the C++ zero overhead principle [177], zero addressing is the default mode.

Zero Addressing In zero addressing mode (Figure 4.6, Listing 4.15), an object of an SOA class `C` with ID i is referenced with a `C*` fake pointer pointing to address i (Listing 4.16).

Values are grouped by field within the storage buffer (SOA layout). No field values are stored for object 0 (null pointer). Given a `C*` fake pointer *obj*, the physical memory location within the storage buffer of the field value `C::f`, i.e., `&obj->f`, is calculated as follows. Compile-time constants are in blue.

LISTING 4.15: Address computation in zero addressing

```
Body* Body::get(int id) {
  return reinterpret_cast<Body*>(id)
}

template<typename T, int Index, int Offset, class Owner>
T* Field_<T, Index, Offset, Owner>::data_ptr() {
  Owner* obj = reinterpret_cast<Owner*>(this);
  return reinterpret_cast<T*>(Owner::storage
      + Offset * Owner::kMaxInst - sizeof(T)
      + sizeof(T) * reinterpret_cast<uintptr_t>(obj));
}
```

LISTING 4.16: Fake pointers in zero addressing mode

```
C* a = new C(); // First object. ID 1
C* b = new C(); // Second object. ID 2
printf("%p, %p\n", a, b); // Prints: 0x00000001, 0x00000002
```

$$\begin{aligned} addr_{zero}(obj, \texttt{C::f}) = {} & storage \\ & + maxInst(\texttt{C}) \cdot offset(\texttt{C::f}) \\ & - sizeof(\texttt{C::f}) \\ & + \underline{obj} \cdot sizeof(\texttt{C::f}) \end{aligned}$$

🟦 constant
🟥 variable

Intuitively, the address is computed as follows: Start with the address of the storage buffer. Add the offset at which the SOA array for `C::f` begins. Finally, add the offset at which the *(obj - 1)*th value begins. We have to subtract 1 to account for the fact that object IDs start at 1 but C++ arrays start indexing with 0.

Since the storage buffer is statically allocated, the first three lines of the address calculation are compile-time constants and the fourth line is a strided memory access. After constant folding, this is identical to a hand-written SOA layout with statically allocated SOA field arrays. In a hand-written SOA layout, the address of the field value `C::f` of an object with ID i, i.e., `&C_f[i]`, is computed as follows.

$$\begin{aligned} addr_{manual}(i, \texttt{C::f}) = {} & \texttt{\&C\_f[0]} \\ & + \underline{i} \cdot sizeof(\texttt{C::f}) \end{aligned}$$

Modern compilers are good at peephole optimizations. We compared the compiled assembly code of a single field access for both a hand-written SOA layout and for the equivalent IKRA-CPP code in C++ and CUDA. The resulting assembly code was identical (Section 4.2.4).

One crucial assumption of zero addressing is that the C++ object size of SOA classes and `Field_` instantiations is zero bytes (e.g., `sizeof(Body) = 0`). In that case, the fake address of an SOA object is equal to the addresses of all its proxy field objects[9]. Listing 4.15 shows the implementation of the function `data_ptr`, which calculates the physical memory address of an SOA field value within the storage buffer. If `Field_` instantiations had a byte size larger than zero, Line 7, which calculates the fake pointer of the SOA object, would have to be changed.

Zero addressing has one main advantage: It allows us to use the C++ constructor syntax without wasting memory. If `Field_` instantiations and SOA objects have a C++ object size of zero bytes[10], we can use the `new` keyword for instance creation. In host code (not GPU code), all memory is zero-initialized before running a constructor. Zero-initializing a memory segment of zero bytes is a *no operation*, even if the segment starts at a bogus memory address (*fake pointer*). On the contrary, zero-initializing at least one byte at a bogus memory address will likely result in a memory access violation and crash the program.

According to the C++ standard, the size of a class or struct should be greater than zero (even if it has no members) [92], but many compilers can be instructed to use a size of zero. If this is not supported by a compiler, either valid addressing or a different mechanism for instance creation must be used.

Valid Addressing Mode Since zero addressing does not conform to the C++ standard, IKRA-CPP provides an alternative addressing mode. In *valid addressing*, the C++ object size of every proxy field object is one byte (e.g., `sizeof(double_) = 1`). Consequently the C++ object size of every SOA object is *numFields* bytes. Note that in either addressing mode, the C++ `sizeof` keyword reports a number that is different from the actual memory consumption an SOA object within the storage buffer.

In order to support the `new` keyword, the address of an SOA object must then point to valid (allocated) memory; thus the name *valid* addressing. Otherwise, zero-initialization would cause a memory access violation. The challenge of valid addressing is to add an as small as possible amount of padding (*wasted* memory) such that no data is overwritten by zero initialization.

Programmers should use zero addressing if supported by their compiler, since it does not waste any padding memory. We expect the same runtime performance as in

[9] E.g., for a `Body* b`: b = `&b->pos_x` = `&b->pos_y` = ...

[10] This break pointer arithmetics with SOA object pointers. However, we provide a custom iterator type that programmers can use instead.

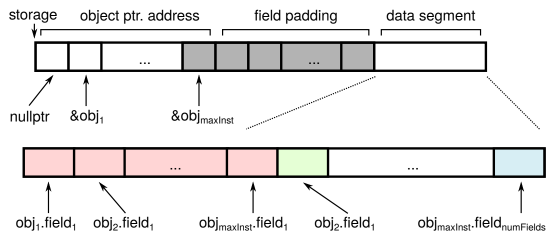


FIGURE 4.7: Storage buffer layout in storage-relative zero addressing

zero addressing, because address computation is in both cases reduced to a strided memory access after constant folding.

Storage-relative Zero Addressing This addressing mode is one of two variants of valid addressing. In this addressing mode, an object of SOA class `C` with ID *i* is referred to with a fake `C*` pointer pointing to the i^{th} byte of the storage buffer (Figure 4.7). E.g., if the storage buffer is allocated at address `0x4000`, then the address of $\texttt{obj}_3$ is `0x4003`.

The *data segment*, where field values are stored, is identical to the one in zero addressing. However, it starts at byte offset *padding*, i.e., *padding* many bytes are wasted in this addressing mode.

$$padding = maxInst(\texttt{C}) + 1 + numFields(\texttt{C})$$

Since all fake object pointers point to one of the first *maxInst*(`C`) bytes of the storage buffer (*object ptr. address* part in Figure 4.7), not a single byte of the data segment is overwritten by zero-initialization, which zeros out *numFields*(`C`) many bytes. Note that, similar to zero addressing, we have to add 1 in the above formula because object IDs start with 1 (0 is reserved for null pointers).

In general, the physical memory location of a field value `C::f` of an object with fake pointer *obj* is calculated as follows. This formula is identical to the one in zero addressing, except for the offset into the data segment and the object ID computation/decoding part.

$$\begin{aligned}
& addr_{valid}(obj, \texttt{C::f}) = storage \\
\text{data segment offset} \quad & \boxed{+maxInst(\texttt{C}) + 1 + numFields(\texttt{C})} \\
& + maxInst(\texttt{C}) \cdot offset(\texttt{C::f}) \\
& - sizeof(\texttt{C::f}) \\
\text{ID computation} \quad & + \boxed{(obj - storage)} \cdot sizeof(\texttt{C::f})
\end{aligned}$$

Since address computation is done inside field proxy types (i.e., `Field_` instantiantions, *not* `IkraSoaBase`), we have to express the above formula in terms of the address (*this* pointer) of the field proxy object instead of the fake object address *obj*. The fake object address *obj* of an SOA object can be computed based on the address of a field proxy object of `C::f` as follows, where *index*(`C::f`) is the field index of `C::f` (start counting from 1).

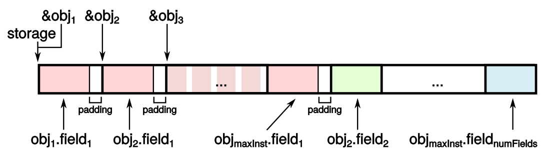


FIGURE 4.8: Storage buffer layout in first field addressing

$$obj = this - index(\texttt{C::f}) + 1$$

Since every `Field_` instantiation is 1 byte in size, we can retrieve the base address (fake pointer) *obj* of an object from the address *this* of the i^{th} field proxy object of *obj* by subtracting $i - 1$ from it. For example, let `0x4003` be the address of the third field proxy object (`vel_x`) of a `Body` object. Then, $obj = \texttt{0x4003} - 3 + 1 = \texttt{0x4001}$. This fake object pointer can be plugged into $addr_{valid}(obj, \texttt{Body::vel\_x})$. Putting both definitions together, the physical memory location of a field value `C::f` with respect to its proxy object's address *this* is then calculated as follows.

$$\begin{aligned} addr_{valid}(this, \texttt{C::f}) &= storage \\ &+ maxInst(\texttt{C}) + 1 + numFields(\texttt{C}) \\ &+ maxInst(\texttt{C}) \cdot offset(\texttt{C::f}) \\ &- sizeof(\texttt{C::f}) \cdot (index(\texttt{C::f}) + storage) \\ &+ \underline{this} \cdot sizeof(\texttt{C::f}) \end{aligned}$$

The formula above was rearranged to keep the number of terms small. After constant folding, the address of a field value can be calculated with the same instructions as in zero addressing mode.

First Field Addressing This addressing mode is the second variant of valid addressing and not currently implemented in IKRA-CPP. Its purpose is to reduce the amount of waste due to the padding area. It could potentially also be useful for virtual function support in the future.

An object of SOA class `C` with ID *i* is referred to with a fake `C*` pointer pointing to the physical memory location of the value of the first field of object *i* (Figure 4.8). If the SOA class has at least one virtual function, then the first field is the vtable pointer[11]. If the number of fields of the SOA class is larger than the size of the first field, then the memory of the first field must be padded with $sizeof(\texttt{C:}:first) - numFields(\texttt{C})$ bytes to avoid overwriting values of the first field of other objects (with larger object IDs) due to zero initialization. Since object deallocation is not yet supported in IKRA-CPP, this would currently not be problem because object slots with greater IDs are guaranteed to be empty. However, we will extend IKRA-CPP with dynamic object deallocation in the next chapter. Given a fake `C*` pointer *obj*, the memory location of a field `C::f` is calculated as follows.

[11] Most C++ compilers store the vtable pointer in the first 8 bytes of an object.

$$\begin{aligned} addr_{\mathit{first}}(obj, \texttt{C::f}) = {} & storage \\ & + maxInst(\texttt{C}) \cdot offset^*(\texttt{C::f}) \\ & + \left(\frac{obj - storage}{sizeof^*(\texttt{C::}\mathit{first})} - 1 \right) \cdot sizeof^*(\texttt{C::f}) \end{aligned}$$

$sizeof(\texttt{C::f})$ and $offset(\texttt{C::f})$ denote the C++ size of a field type and the offset of a field within the class (Listing 4.9). In addition, $sizeof^*(\texttt{C::f})$ and $offset^*(\texttt{C::f})$ also take into account padding that may be added to the first field. The physical memory location of a field value `C::f` with respect to its proxy object's address *this* is calculated as follows. We get this formula by plugging in the definition of *obj* from storage-relative zero addressing.

$$\begin{aligned} addr_{\mathit{first}}(this, \texttt{C::f}) = {} & storage - sizeof(\texttt{C::f}) \\ & + maxInst(\texttt{C}) \cdot offset^*(\texttt{C::f}) \\ & - (index(\texttt{C::f}) + storage - 1) \cdot R \\ & + \underline{this} \cdot R \\ \text{where } R = {} & \frac{sizeof^*(\texttt{C::f})}{sizeof^*(\texttt{C::}\mathit{first})} \end{aligned}$$

Even though the definition of $addr_{\mathit{first}}$ contains a fraction, its value is always an integer. However, its calculation is not straightforward. Similar to the previous addressing modes, we rearranged the terms in the above formula such that it can be computed with one addition and one multiplication after constant folding (strided memory access). While the formula always gives us integer values, single parts such as $this \cdot R$ may be fractions. Floating point operations as part of the address computation are highly inefficient and must be avoided.

Therefore, there are two options for implementing first field addressing in IKRA-CPP. Either we compute the formula differently (with more arithmetic operations) or we enforce R to be an integer, i.e., the size of every field must be a multiple of the size of the first field.

This addressing mode is superior to storage-relative zero addressing only if the field padding size is zero or one byte. Otherwise, there would be no space savings, because field padding is incurred for every object, i.e., *maxInst* many times.

4.2.4 Code Generation Experiment

IKRA-CPP's data layout DSL is *embedded* into C++/CUDA and does not rely on an external preprocessor or code generator. It is relies heavily on template metaprogramming, operator overloading and type punning. Therefore, the code that the C++ compiler is seeing is quite complex. In this section, we analyze how well C++ compilers can optimize such code.

Listing 4.17 shows the setup of our experiment. We would like to analyze the assembly code that a C++ compiler generates for writing an integer value to a field of a given object. We consider three cases.

1. An ordinary C++ class, where the object is referred to with an object pointer (AOS style).

LISTING 4.17: Example: Writing a field of an object in AOS/SOA/IKRA-CPP

```
// Case 1: AOS style
class DummyClassAos {
 public:
  int field0;
  int field1;
};

// Case 2: Handwritten SOA
int soa_field0[100];
int soa_field1[100];

// Case 3: Ikra-Cpp (SOA)
class DummyClassIkraCpp : public IkraSoaBase<DummyClassIkraCpp, 100> {
 public: IKRA_INITIALIZE_CLASS
  int_ field0;
  int_ field1;
}; IKRA_HOST_STORAGE(DummyClassIkraCpp)

void write_field0(DummyClassIkraCpp* obj) { obj->field0 = 0x7777; }
void write_field0_handwritten_soa(uintptr_t id) { soa_field0[id] = 0x7777; }
void write_field0_aos(DymmyClassAos* obj) { obj->field0 = 0x7777; }
```

LISTING 4.18: Generated assembly code for functions in Listing 4.17

```
00000000004005e0 <_Z12write_field0P9DummyClassIkraCpp>:
  4005e0: c7 04 bd b0 a3 60 00  movl   $0x7777,0x60a3b0(,%rdi,4)
  4005e7: 77 77 00 00
  4005eb: c3                    retq
  4005ec: 0f 1f 40 00           nopl   0x0(%rax)

0000000000400620 <_Z21write_field0_handwritten_soam>:
  400620: c7 04 bd 60 10 60 00  movl   $0x7777,0x601060(,%rdi,4)
  400627: 77 77 00 00
  40062b: c3                    retq
  40062c: 0f 1f 40 00           nopl   0x0(%rax)

0000000000400640 <_Z25write_field0_aosP16DummyClassAos>:
  400640: c7 07 77 77 00 00     movl   $0x7777,(%rdi)
  400646: c3                    retq
  400647: 66 0f 1f 84 00 00 00  nopw   0x0(%rax,%rax,1)
  40064e: 00 00
```

2. A hand-written SOA layout, where the object is referred to with an integer index into SOA arrays.
3. An IKRA-CPP class in SOA layout, where the object is referred to with a fake pointer in zero addressing mode.

Listing 4.18 shows the generated assembly code of gcc 5.4.0 with `-O3` optimization. Lower optimization levels result in considerably less efficient code because the overloaded operators and the `data_ptr` function (Listing 4.13) are not inlined.

We can see that the generated assembly code of the IKRA-CPP version is nearly identical to the hand-written SOA version. The assembly code of both functions differs only in the address of the (conceptual) SOA array. This shows that, at least in this small experiment, a modern compiler is able to optimize IKRA-CPP code through peephole optimizations such as constant folding.

Automatic Vectorization Unfortunately, this is not always the case for other compiler optimizations. One example is automatic loop vectorization. We analyzed the generated assembly code of a benchmark that runs `Body::move` (Listing 4.9) on the host (CPU) in a loop with many iterations. In the hand-written SOA code, gcc and clang vectorize the method call `Body::move` of multiple loop iterations with SSE (*Streaming SIMD Extensions*) processor instructions. However, only gcc performs the equivalent loop vectorization with IKRA-CPP code. Clang is able to apply optimizations like loop unrolling but considers the memory reads/writes[12] as potentially *dependent* memory operations and thus unsafe for vectorization.

There are three approaches to solve this problem. First, we can try rewriting the address computation part of IKRA-CPP, in an attempt to give the compiler additional hints that trigger optimizations. Due to our type punning-based implementation, this approach is fragile and could break at any time. Second, code can be vectorized manually, either with C++ SSE intrinsics or with a vectorization framework like *Sierra* [114, 115]. Considering that real applications, which exhibit code that is more complex than our example here, cannot be automatically vectorized (yet) with today's compilers, even if written in SOA style, this approach seems feasible to us. Third, IKRA-CPP could be implemented as a compiler extension or with a custom preprocessor/code generator, which is the cleanest and most stable solution.

Note that automatic vectorization is a minor issue for GPU code. CUDA follows the SIMT (Single-Instruction Multiple-Threads) model. Since SIMD parallelism is exposed to programmers as threads, programmers effectively vectorize their code manually. Similarly, programs written for the Intel SPMD Prgram Compiler (ispc) [152] code are implicitly vectorized.

4.2.5 Preliminary Performance Evaluation

We evaluated IKRA-CPP on a computer with an Intel Core i7-5960X CPU (4x 3.00 GHz), 32 GB RAM and an NVIDIA GeForce GTX 980 GPU, a 64-bit Ubuntu 16.04.1, gcc 5.4.0 and the NVIDIA CUDA Toolkit 9.0.176 in zero addressing mode.

We benchmarked an iterative application of `Body::move` (Listing 4.9) in a parallel do-all operation. This benchmark is quite simple, but it clearly isolates the overheads of IKRA-CPP, specifically address computation. Since the generated assembly code

[12]The fundamental problem is pointer casting. In the simplest case, an expression like `array[reinterpret_cast<uintptr_t>(id)]`, where `id` is a pointer encoding an integer array offset is already considered unsafe. This could potentially be solved with the C/C++ `restrict` keyword.

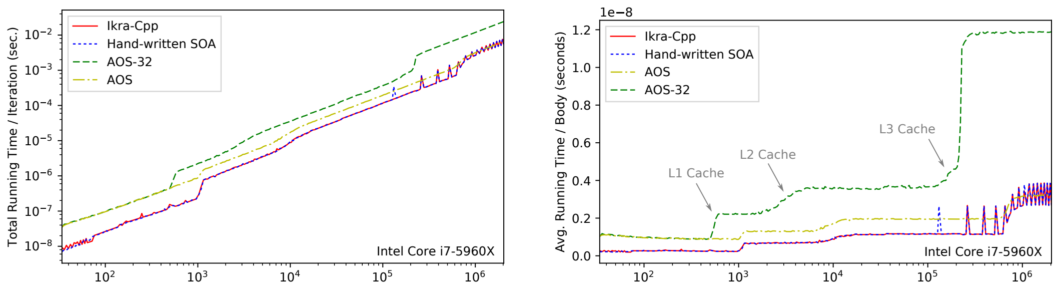


FIGURE 4.9: Host mode running time

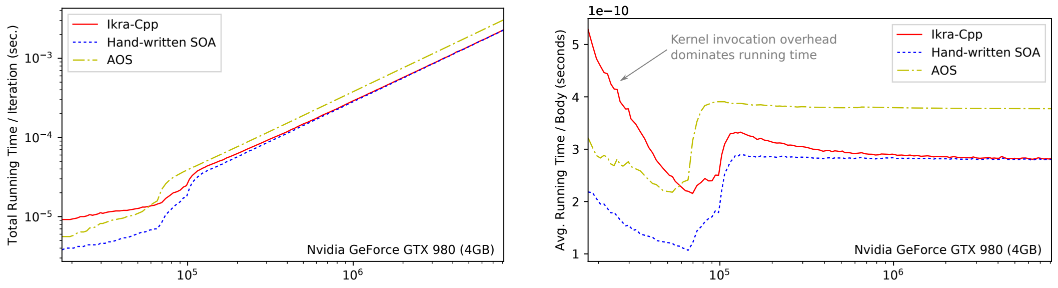


FIGURE 4.10: Device mode running time

for IKRA-CPP is nearly identical to a hand-written SOA layout, we expect minimal overheads. The number of iterations was chosen such that every program ran for at least 5 seconds. We calculated the average running time per iteration and report the minimum time out of 12 program runs.

Figures 4.9 and 4.10 show the running time on CPU and GPU. The x-axis denotes the number of objects and the y-axis denotes the running time in seconds. The left subfigure shows the average running time of one entire iteration and the right subfigure shows the average running time for a single `Body` object.

In host mode, IKRA-CPP's performance is almost identical to the hand-written SOA code. AOS-32 is a variant of AOS where 16 supplemental `double` fields were added to the `Body` class, similarly to the SoAx benchmark section [85]. We can think of such fields as additional properties of a body (e.g., mass or radius) that are not utilized in this particular computation. The AOS-32 line of the right subfigure clearly shows the effect of the L1, L2, L3 caches (32 KB, 256 KB, 20 MB) in host mode.

The performance difference between IKRA-CPP and the hand-written SOA layout in device mode is due to a higher kernel invocation overhead of IKRA-CPP. More than 10,000 iterations (kernel invocations) are performed for small problem sizes. With a larger number of bodies, we get closer to hand-written SOA code, because each kernels performs more work.

4.2.6 Related Work

The AOS-SOA tradeoff is a well-known problem and has been studied in previous work in the context of C structs. To the best of our knowledge, there is no system that provides an AOS-like programming style for object-oriented programming with SOA performance characteristics.

SoAx [85] is a C++ library for AOS-style C/C++ programming with an implicit SOA layout. It is based on preprocessor macros and template metaprogramming.

SoAx does not support OOP concepts like classes or methods. SOA struct types are defined with `std::tuple` instantiations and a helper macro that defines every SOA array separately. Object field values can only be accessed through a getter method of an SOA container object, which takes an object ID as argument, and not through SOA pointers. While such code is less expressive, it has two benefits: First, such code is easier to optimize for compilers than IKRA-CPP code because it does not require decoding an object ID from a pointer. Second, it allows programmers to create multiple containers (structures of arrays), each of which has its own object ID range.

Array of Structures eXtended (ASX) [179] is a library similar to SoAx. Objects in ASX can be allocated in ASX containers as well as on the stack (as single objects). ASX containers support both SOA and AOS data layouts, one of which must be chosen as a template parameter. ASX allows fields to be accessed with C++ member syntax (dot/arrow operators), but there are certain restrictions with respect to the size of field types.

Columnar Objects [135] is a Python extension (implemented in PyPy) that stores objects as SOA. Columnar Objects can deliver significant speedups of object-oriented analytical applications. In contrast to IKRA-CPP and other libraries, subclassing is possible. However, newly introduced SOA arrays of subclasses store *null* values for objects of superclasses, which can waste memory. Similar to IKRA-CPP, objects are accessed through proxy objects which delegate field accesses to the actual, physical memory locations.

The *Intel SPMD Program Compiler (ispc)* [152, 25] is an experimental C compiler with language features for better SIMD support. Among other features, it can store an array of C structs in a hybrid SOA layout (also called *Array of Structures of Arrays* (AoSoA) [180, 198] or *Tiled AOS* [106]). If a struct type is annotated with the `soa<`N`>` keyword and used to declare an array (where N should be the SIMD width), then the array is stored as hybrid SOA with an SOA length of N. Array elements can be accessed with the usual C syntax. Furthermore, it is possible to take the address of an SOA object and fields can be accessed using an SOA object pointer. From that perspective, ispc's functionality is very similar to IKRA-CPP. It would interesting to see how easily ispc can be extended to support OOP concepts like methods.

Shapes is a high-level programming language that allows programmers to specify custom data layouts for better memory cache performance [65]. Objects are stored in *pools* and every pool can have a different object layout, e.g., AOS, SOA or a mixed layout. The developers of Shapes are recently working on compiling Shapes applications for SIMD architectures [182].

IKRA-CPP could be implemented as a compiler extension. To the best of our knowledge, no such extension exists for a widely used language. We believe that this is due to the high engineering effort of writing a new compiler or such an invasive compiler extension [137].

4.2.7 Summary

We presented a first implementation of IKRA-CPP's data layout DSL for object-oriented programming with SOA performance characteristics. IKRA-CPP allows programmers to write object-oriented code in AOS notation, while data is stored as SOA for better performance. SOA object members are always accessed through fake pointers or object references. How exactly an object ID is encoded in a pointer is determined by the addressing mode. Our main insights are that (a) object ID decoding and field address computations can be done efficiently after constant folding and that (b) an AOS-style notation can be achieved transparently in C++ with operator overloading, template metaprogramming, and preprocessor macros. Preliminary benchmarks show that simple examples written with IKRA-CPP and compiled with gcc are on par with hand-written SOA code.

IKRA-CPP is the basis of DYNASOAR (Chapter 5). The data layout DSL of DYNASOAR is heavily based on IKRA-CPP's DSL.

4.3 Inner Arrays in a Structure of Arrays

SOA works well with simple data structures, but cannot be easily applied to structs that contain inner arrays (i.e., fields of array type), potentially of non-constant size. Such structures appear frequently in graph-based applications and in object-oriented designs with associations of high multiplicity.

Such arrays are typically allocated separately on the heap, requiring an additional pointer indirection. In this section, we analyze different data layout techniques for inner arrays. We extended IKRA-CPP with additional proxy field types that implement those layouts.

Applications that iterate over arrays one-by-one or in a fashion that is *uniform* among all objects are particularly interesting, because their memory access can be optimized. While a standard SOA layout does not affect the layout of inner arrays, a different, more SIMD-friendly layout can group elements by array index and increase memory coalescing on GPUs for such applications.

Examples To experiment with various inner array layouts, we implemented two important real-world SMMO applications in IKRA-CPP: Breadth-first search (BFS) and an agent-based, object-oriented traffic flow simulation. In graphs, vertices often have a varying number of neighbors and adjacency lists (arrays) are the preferred representation for BFS on GPUs [77]. The traffic flow simulation exhibits graph-based features for representing street networks and utilizes array-based data structures within the simulation logic.

4.3.1 Data Layout Strategies for Inner Arrays

This section gives an overview of seven inner array layout strategies. We focus on C++-style arrays with fixed size after allocation. Figures 4.11–4.14 illustrate these strategies visually, using the `Vertex` class for a breath-first algorithm as an example (Listing 4.19). This class has two fields: A distance field storing the computed distance of the vertex from the source vertex and an array of `Vertex` pointers (adjacency list). Note that the adjacency lists of different vertices may have different lengths. Therefore, we also have to store the size of each list/array (`num_neighbors`).

In the following paragraphs, we describe multiple layout strategies of the adjacency list array. We implemented these strategies in IKRA-CPP with additional proxy field types.

We consider three categories of layout strategies: *AOS*, *SOA* and *SOA with Array as Object* (i.e., SOA without handling inner arrays specially). In every category, inner arrays can either be *fully inlined*, *partially inlined* or not inlined at all (*without inlining*). Whether objects of a class are stored in AOS or SOA depends on the chosen superclass (`IkraBase` or `IkraSoaBase`).

AOS without Inlining (Figure 4.11 (left), Listing 4.20) This is the default data layout that many programmers choose intuitively. Objects are stored as AOS (`IkraBase`), LISTING 4.19: Data structure of frontier-based BFS in IKRA-CPP

```
class Vertex : public IkraBase<Vertex, 100> { // or: IkraSoaBase<Vertex, 100>
 public: IKRA_INITIALIZE_CLASS
  int_ distance = std::numeric_limits<int>::max();
  ??? /* (some array type) */ neighbors;

  // Constructor can run on host and device.
  __device__ ___host__ Vertex(int num_neighbors) : neighbors(num_neighbors) {}

  __device__ int num_neighbors() { return neighbors.size(); }

  __device__ void visit(int frontier); // Implementation later...
};

IKRA_DEVICE_STORAGE(Vertex)
```

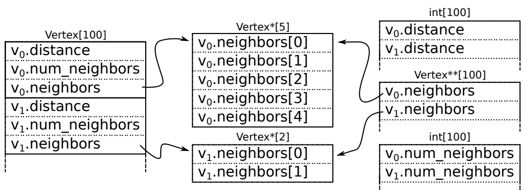


FIGURE 4.11: No inlining: AOS (left side) and SOA (right side)

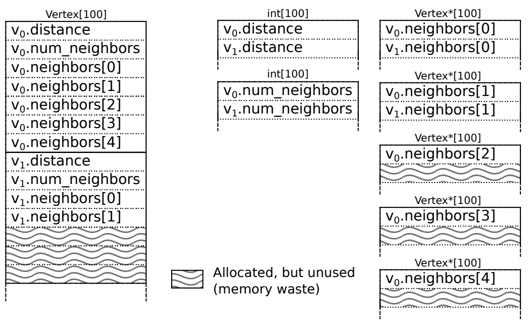


FIGURE 4.12: Full inlining: AOS (left side) and SOA (middle, right side)

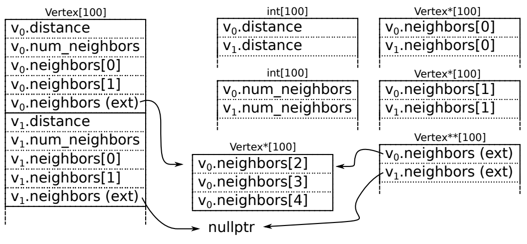


FIGURE 4.13: Partial inlining: AOS (left side) and SOA (middle, right side)

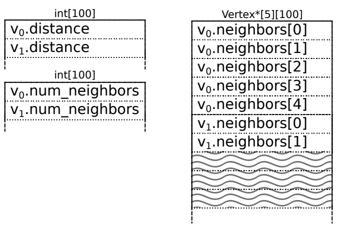


FIGURE 4.14: SOA, array as object

LISTING 4.20: Notation: AOS without inlining

```
// Preferred notation:
inlined_array_(Vertex*, 0)
    neighbors;
```

```
// Alternative 1:
int_ num_neighbors;
field_(Vertex**) neighbors;
```

```
// Alternative 2:
field_(std::vector<
    Vertex*>) neighbors;
```

LISTING 4.21: Notation: AOS with full inlining

```
// Preferred notation:
fully_inlined_array_(Vertex*, N)
    neighbors;
```

```
// Alternative:
int_ num_neighbors;
field_(std::array<Vertex*, N>) neighbors;
```

i.e., all field values of an object are stored together. Inner arrays are heap-allocated. This layout is useful if inner arrays have different sizes, because then no memory is wasted by array inlining. In standard C++, inner arrays can be allocated manually (using `malloc`/`new`) or with a helper class like `std::vector<T>`.

IKRA-CPP provides a proxy field type `inlined_array_(T, 0)` which internally stores a memory pointer to a heap-allocated array and the size of the array. Fields of this type must be initialized in the constructor with the desired array size (Listing 4.19, Line 7). IKRA-CPP will then allocate the array with the system/CUDA-wide dynamic memory allocator (`malloc`). Note that the CUDA allocator is slow, so this implementation is not suitable if the constructor is part of performance-critical code.

Depending on the class structure, compilers may have to add padding to ensure that all fields are properly aligned. E.g., the location that stores the 64-bit memory pointer to the inner array must be aligned to a multiple of 64-bit. This is a general disadvantage of AOS (Section 2.3.3). Note that this layout is identical to "AOS with Partial Inlining" with an inlining size of zero, but the conditional branch of that layout strategy is optimized away by the compiler.

AOS with Full Inlining (Figure 4.12 (left), Listing 4.21) This layout strategy still stores objects as AOS (`IkraBase`), but inlines inner arrays fully into objects, such that they can be accessed more efficiently without a pointer indirection. Very small inner arrays may be able to share cache lines with other fields, and thus benefit cache utilization.

The downside of this layout is that it potentially wastes memory; all inner arrays must have the same size, i.e., the largest size among all inner arrays, to be able to hold all elements. The amount of wasted memory depends on the variance among inner array sizes.

In standard C++, fully-inlined inner arrays of size N can be declared with `std::array<T, N>`. IKRA-CPP provides a proxy field type `fully_inlined_array_`.

AOS with Partial Inlining (Figure 4.13 (left), Listing 4.22) This layout is a mixture of the previous two strategies. Up to N inner array elements are inlined into objects, where N is a compile-time constant. Elements with an index $\geq N$ are stored externally on the heap. The benefit of this approach is efficient access to the first N elements. Access to elements on the external storage is as expensive as with "Without Inlining", i.e., it requires a pointer indirection. This strategy requires an additional conditional branch to determine whether an element is stored in the inline storage or on the external storage. This imposes little overhead on GPUs, which do generally not execute instructions speculatively.

LISTING 4.22: Notation: AOS with partial inlining

```
inlined_array_(Vertex*, N) neighbors; // Preferred notation
field_(absl::InlinedVector<Vertex*, N>) neighbors; // Alternative

// Data layout is equivalent to, however, much harder to use:
int_ num_neighbors;
field_(std::array<Vertex*, N>) neighbors;
field_(Vertex**) neighbors_other;
```

LISTING 4.23: Notation: SOA without inlining

```
// Preferred notation:
inlined_array_(Vertex*, 0)
    neighbors;
```

```
// Alternative 1:
int_ num_neighbors;
field_(Vertex**) neighbors;
```

```
// Alternative 2:
field_(std::vector<
    Vertex*>) neighbors;
```

In ordinary C++, inner arrays can be partially inlined with a helper class such as `absl::InlinedVector<T, N>`[13]. IKRA-CPP supports this layout with the previously mentioned proxy field type `inlined_array_(T, N)`. The second argument *N* is the number of inlined array slots. The total array size must be specified during field initialization.

SOA without Inlining (Figure 4.11 (right), Listing 4.23) This layout is identical to "AOS without Inlining", but stores objects as SOA (`IkraSoaBase`). It has the usual benefits of SOA-style allocation: First, if not all fields are used all the time, it can improve cache utilization because those fields will not occupy cache lines. Second, it allows for efficient loads/stores from/into vector registers if objects with consecutive IDs are simultaneously accessed (memory coalescing). Third, less memory is wasted for object padding compared to AOS, because only the SOA arrays themselves must be aligned to certain byte sizes, but not each object.

With respect to inner arrays, the same advantages and disadvantages as in the first strategy apply. IKRA-CPP provides a proxy field type `inlined_array_(T, 0)` which implements this layout.

SOA with Full Inlining (Figure 4.12 (right), Listing 4.24) This layout is identical to "AOS with Full Inlining", but stores objects as SOA (`IkraSoaBase`). In particular, inner arrays are also stored in SOA, as if every array slot were a separate field of the class. There is a separate SOA array for each inner array index.

This layout provides opportunities for vectorized operations and memory coalescing not only for primitive fields but also when accessing inner array elements. This is possible if objects with consecutive IDs are processed in the same warp and inner array elements with the same indices are accessed.

Unfortunately, many parallel graph algorithms [132] on GPUs do not benefit much from additional memory coalescing in this layout. Even though reading the pointers of an adjacency list can be coalesced, data reads/writes on neighboring vertices are still uncoalesced, because vertex IDs of neighbors are usually *random* and not consecutive (Section 4.3.2).

Similar to "AOS with Full Inlining", this layout provides more efficient array access without a pointer redirection but can waste memory. It is supported by IKRA-CPP as `fully_inlined_array_(T, N)`.

[13] `InlinedVector` is part of the Abseil library. See https://github.com/abseil/abseil-cpp

LISTING 4.24: Notation: SOA with full inlining

```
// Preferred notation:
fully_inlined_array_(Vertex*, N) neighbors;

// Data layout is equivalent to, however, much harder to use:
int_ num_neighbors;
field_(Vertex*) neighbors_1;
field_(Vertex*) neighbors_2;
/* ... */
field_(Vertex*) neighbors_N;
```

LISTING 4.25: Notation: SOA with partial inlining

```
// Preferred notation:
inlined_array_(Vertex*, N) neighbors;

// Data layout is equivalent to, however, much harder to use:
int_ num_neighbors;
field_(Vertex*) neighbors_1;
field_(Vertex*) neighbors_2;
/* ... */
field_(Vertex*) neighbors_N;
field_(Vertex**) neighbors_other;
```

`fully_ininlined_array_` is used in both this layout and "AOS with Full Inlining". Whether this proxy type stores the inner array as AOS or as SOA depends on the layout of the class (`IkraBase` or `IkraSoaBase`).

SOA with Partial Inlining (Figure 4.13 (right), Figure 4.25) This strategy is identical to "AOS with Partial Inlining", but stores objects as SOA (`IkraSoaBase`). The first N inner array elements are inlined and stored as SOA, as in the previous strategy. This layout is supported by IKRA-CPP as `inlined_array_(T, N)`.

`ininlined_array_` is used in both this layout and "AOS with Partial Inlining". Whether this proxy type stores the inlined inner array slots as AOS or as SOA depends on the layout of the class (`IkraBase` or `IkraSoaBase`).

SOA with Array as Object (Figure 4.14, Listing 4.26) This layout treats inner arrays as normal C++ objects and does not perform any data layout transformations on them. There is *one* SOA array for every field, including inner arrays. This layout is useful for GPU applications with nested parallelism. If threads with consecutive IDs simultaneously access consecutive inner array slots, then those accesses can be coalesced. E.g., this is the case when optimizing BFS with virtual warp-centric programming [86]. This layout is supported by IKRA-CPP as `field_(std::array<T, N>)`.

From an inlining perspective, this layout inlines the inner array fully. It could easily be adapted to *partial inlining* (`field_(absl::InlinedVector<T, N>)`) or *without inlining* (`field_(std::vector<T>)`), but we do not analyze such layouts any further in this work.

LISTING 4.26: Notation: SOA with array as object

```
field_(std::array<Vertex*, N>) neighbors;
```

Choosing a Layout Strategy Programmers have a variety of layout strategies to choose from. Which strategy is best depends on the hardware architecture, the data access patterns of the application and the characteristics of the dataset. Even for experienced programmers this process can to some degree be a trial and error.

With IKRA-CPP, programmers still have to take all these factors into account, but switching between strategies is now much easier. As a rule of thumb, we suggest to start experimenting with a partial inlining size that ensures that 80% of all inner array elements are inlined.

4.3.2 Performance Evaluation

We evaluated the previously described data layouts with three benchmarks: A synthetic benchmark, a frontier-based BFS implementation and a traffic flow simulation.

We ran all experiments on a machine with an Intel i7-5960X CPU (8x 3.00 GHz), 32 GB main memory, an NVIDIA GeForce GTX 980 GPU (4 GB memory), Ubuntu 16.04 and the `nvcc` compiler from the NVIDIA CUDA Toolkit version 9.1. This work focuses on GPU execution, but similar performance effects can be observed on CPUs with a compiler with good auto-vectorization.

Synthetic Benchmark To isolate the performance effects of array inlining, we created a synthetic benchmark (Listing 4.27) with a dummy class containing an `int` data field and an `int` array. The array has between 32 and 64 elements (chosen randomly). The benchmark adds the data field value to all array elements in a loop, i.e., all inner array elements are read and written.

Figure 4.15 shows the running time for 262,144 dummy objects. The "Array as Object" SOA version is more than 30% faster than the AOS version. The fully inlined SOA version is an order of magnitude faster than the AOS version.

The "SOA (partial)" line shows the running time with various partial inner array inlining sizes N (x-axis) in SOA layout. For $N = 0$ (no inlining), the performance is worse than "Array as Object". In both cases, the entire inner array is stored in one block of memory. However, in the former case, the array is located on the heap and can only be accessed with a pointer indirection, causing a slowdown. Note that, while "Array as Object" is faster, it wastes a considerable amount of memory.

The performance of partial inlining (SOA) increases with partial inlining sizes, because inner array accesses can be coalesced. Partial array inlining sizes larger than 32 do not improve the SOA-mode performance anymore, because, due to warp divergence caused by differing inner array sizes, those additional inner array accesses are unlikely to be coalesced.

The performance of partial inlining (AOS) decreases with partial inlining sizes after 32 because more and more cache entries are blocked by non-existing array slots.

Breadth-first Search BFS is an important and fundamental algorithm in graph processing. A variety of implementation strategies have been proposed for GPUs, some based on advanced techniques such as hierarchical queues [128] or virtual warp-centric programming [86]. The frontier-based BFS algorithm [139] is among the simplest ones and provides a reasonable speedup compared to CPU execution.

Frontier-based BFS (Listing 4.28) computes the distance of every vertex from a designated start vertex. At first, the start vertex has distance zero and all other vertices have distance infinity. The algorithm now proceeds iteratively. In iteration i, all vertices with distance i (i.e., the *frontier*) are processed in parallel: For every vertex in the frontier, all of its neighbors are updated with distance $i + 1$, unless they

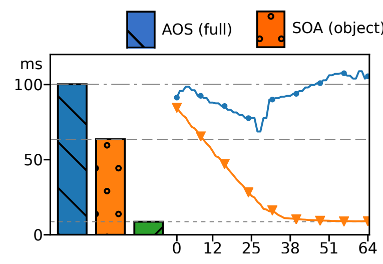


FIGURE 4.15: Synthetic benchmark (ms)

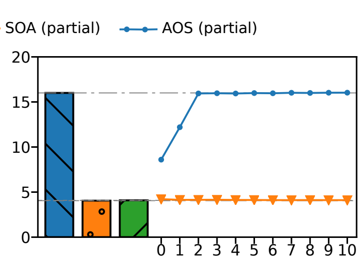


FIGURE 4.16: Frontier-based BFS (sec.)

LISTING 4.27: Source code of synthetic benchmark

```
__device__ int rand_int(int min, int max) { /* return random int */ }

class DummyClass : public ???<DummyClass, 262144> {
 public: IKRA_INITIALIZE_CLASS
  int_ data;
  ???(int, ?) arr;

  __device__ DummyClass() : data(rand_int(0, 100)), arr(rand_int(32, 65)) {}

  __device__ void benchmark() {
    // Can the accesses of arr[i] be coalesced?
    for (int i = 0; i < arr.size(); ++i) { arr[i] += data; }
  }
}; IKRA_DEVICE_STORAGE(DummyClass);

void init_benchmark() { parallel_new<DummyClass>(262144); }
void run_benchmark() { parallel_do<DummyClass, &DummyClass::benchmark>(); }
```

LISTING 4.28: Source code of frontier-based BFS

```
__device__ still_running = false;

__device__ void Vertex::visit(int frontier) {
  if (distance == frontier) {
    for (int i = 0; i < num_neighbors(); ++i) {
      if (neighbors[i]->distance > frontier + 1) {
        still_running = true;
        neighbors[i]->distance = frontier + 1;
      } // else: Vertex was already visited.
    }
  }
}

void run_bfs(Vertex* start_vertex) {
  // Ikra-Cpp objects can be accessed from host and device.
  start_vertex->distance = 0;

  // Read/write still_running with cudaMemcpyTo/FromSymbol.
  for (int iteration = 0; still_running; iteration++, still_running = false) {
    // Notation: Parameter types (int) must be specified before function pointer.
    parallel_do<Vertex, int, &Vertex::visit>(iteration);
  }
}
```

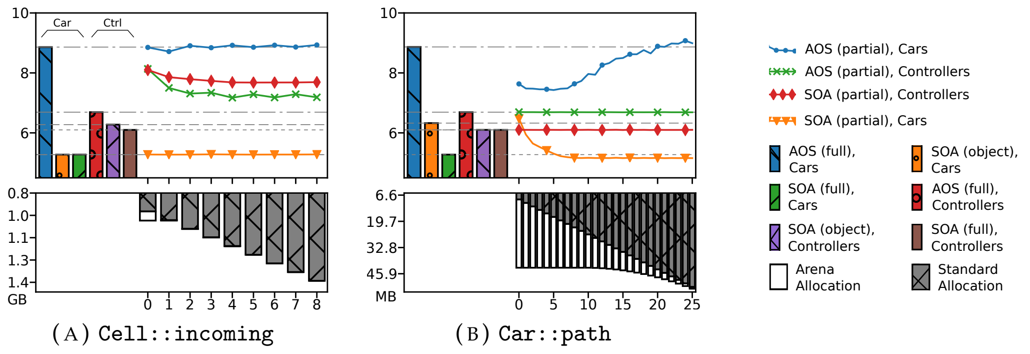


FIGURE 4.17: Running time and memory requirement of traffic simulation

already have a smaller/equal distance value. The algorithm terminates if no updates are performed anymore. BFS is an interesting example for IKRA-CPP because the adjacency lists are arrays of different sizes.

Figure 4.16 shows the running time of the frontier-based BFS algorithm with different layout strategies on the Pennsylvania road network (1,088,092 vertices, 3,083,796 directed edges, avg. degree 2.83) [117]. The graph clearly shows the benefit of SOA over AOS. The performance of AOS degrades with a growing inlining size, because more cache entries for non-existing `neighbors` array slots are wasted. BFS does not benefit from any additional memory coalescing when inlining inner arrays in SOA mode (compare "SOA (object)" and "SOA (full)"), because the neighbors accessed in the inner *for* loop (Listing 4.28, Line 5) have *random* (as opposed to consecutive) IDs. While array reads `neighbors[i]` can be coalesced, accesses to fields of `neighbors[i]` cannot be coalesced. Different optimization techniques can be applied to speed up such algorithms, most notably virtual warp-centric programming [86], which is a form of nested parallelism. That optimization would see a speedup from an *Array as Object* layout.

Traffic Flow Simulation In Section 7.5, we are developing a complex traffic flow simulation that implements the Nagel-Schreckenberg model [145]. This implementation has six array fields in five different classes. These arrays are accessed in most parallel do-all operations of traffic and we can improve the overall runtime performance by optimizing their access[14]. We consider two inner arrays in this paragraph.

- `Cell::outgoing`: Read sequentially in a parallel do-all operation of a method of class `Car`.
- `Car::path`: Cleared at the beginning of a Nagel-Schreckenberg iteration, then filled sequentially and read sequentially in parallel do-all operations of methods of class `Car`.

The benefit of an SOA layout over an AOS layout is clearly visible in this benchmark (Figure 4.17). Regardless of which inner array inlining strategy is chosen, the running time spent on processing cars is always significantly lower in SOA (e.g., first bar vs. second bar). We can observe a similar behavior for traffic controllers.

The benefit of array inlining can be observed best in Subfigure (B) when comparing the running time of a fully inlined `Car::path` (third bar) with the one that is stored

[14]Refer to Section 7.5 for application implementation details.

"as object" (second bar). The fully inlined version is 15% faster. This is because our implementation of the Nagel-Schreckenberg algorithm iterates over the path array in every thread. Because all cars are processed in order (i.e., thread *i* processes car with ID *i*), memory accesses are coalesced. Furthermore, a slightly better speedup can be achieved with partial array inlining (orange line), starting from an inlining size of 6. Accesses to `Car::path` at indices higher than 6 are unlikely to be coalesced because very few threads actually access these array slots. We believe that the additional speedup is due to better cache utilization and prefetching: With partial inlining, a cache line can contain multiple elements of the same inner array from different objects.

The lower part of every subgraph shows the memory usage of inner arrays. The gray part represents inlined allocation (regular fields and inlined slots of inner arrays). The white part represents external storage (heap allocation). Too much inlining wastes memory because not all inner arrays are equally large. Recall that the number of inlined slots of an inner array is the same for every object. On the contrary, inner arrays of different objects may have different sizes on the external storage.

4.3.3 Conclusion and Related Work

In this section, we presented an overview of various data layout strategies for inner arrays in a Structure of Arrays layout. Depending on the data access pattern, such arrays can be split and regrouped by array index to take advantage of memory coalescing when accessing inner array elements. Since writing and maintaining such low-level code is tedious, we extended IKRA-CPP with additional proxy types that store inner arrays in the discussed layouts.

Object inlining has been proposed for Java-like object-oriented languages for better cache performance and reducing overheads due to allocation and pointer indirections [51]. Later work applied the idea of data inlining to arrays, which can simplify address arithmetics of array accesses and eliminate load instructions in the assembly code [203]. IKRA-CPP applies the same idea to arrays in an SOA layout and we are seeing similar speedups. However, we inline arrays under simplified assumptions: Arrays are of fixed size and inlining is controlled manually by the programmer. Future work could attempt to automate this process.

4.4 Summary

We discussed two global memory access optimizations in this chapter: (a) Kernel fusion (IKRA-RUBY) and (b) the Structure of Arrays (SOA) data layout (IKRA-CPP). These two optimizations improve memory access by (a) reducing the amount of memory transfer between global memory and GPU registers and (b) improving memory coalescing and cache performance.

In the remainder of this thesis, our focus will be on IKRA-CPP. In the next two chapters, we will extend IKRA-CPP with a dynamic memory allocator and a memory defragmentation system.

Chapter 5

Dynamic Memory Allocation with SOA Performance Characteristics

Dynamic memory management and the ability/flexibility of creating/deleting objects at any time is one of the corner stones of object-oriented programming. We believe that poor support for dynamic memory allocation is one of the reasons why many GPU programmers avoid object-oriented programming entirely.

In this chapter, we present DynaSOAr, a CUDA framework for SMMO applications. DynaSOAr is a parallel, lock-free, dynamic memory allocator, combined with an efficient parallel do-all operation and an embedded C++/CUDA DSL to enable object-oriented abstractions in an SOA layout. The DSL built on top of Ikra-Cpp and DynaSOAr can be seen as Ikra-Cpp extended with a dynamic memory allocator, although with a slightly different notation and syntax.

Contents

Outline This chapter is organized as follows. Section 5.1 describes DYNASOAR's design goals and its programming interface, an extension of SMMO. Section 5.2 presents DYNASOAR's high-level design and architecture. Section 5.3 describes optimizations that improve the runtime performance of (de)allocate operations. Section 5.4 describes how DYNASOAR handles concurrency and argues for its correctness. Section 5.5 compares DYNASOAR with other dynamic GPU memory allocators. Section 5.6 presents a performance evaluation with synthetic benchmarks and a variety of SMMO applications, which are described in more detail in Chapter 7. Finally, Section 5.7 concludes this chapter.

Overview Dynamic memory management on GPUs is a hard problem. Due to the massive parallelism and data-parallel execution of GPUs, the number of simultaneous (de)allocations is significantly higher than on other parallel hardware architectures. Massively parallel SIMD allocations also follow patterns different from CPU/MIMD allocations: Most memory requests are small in size[1] and due to mostly regular control flow, many allocations have the same byte size. In recent years, fast, dynamic memory allocators have been developed for GPUs [175, 88, 201, 191, 12, 174, 55, 71] and demanded by application developers [216, 176, 122, 155, 166, 120, 121], showing a growing interest in better programming models and abstractions that have long been available on other platforms. However, while these allocators often provide good (de)allocation performance, they miss key optimizations for structured data (such as SOA), leading to poor data locality and memory bandwidth utilization when accessing allocated memory.

In contrast to state-of-the-art allocators, DYNASOAR does not only control the data placement/layout through its memory allocator, but also data access through its do-all operation. In SMMO applications, DYNASOAR achieves superior performance compared to state-of-the-art allocators due to three main optimizations.

- Objects are stored in a **Structure of Arrays (SOA)** data layout, a best practice for structured data in SIMD programs, making usage of allocated memory more efficient when used in conjunction with DYNASOAR's do-all operation.
- Memory fragmentation caused by dynamic object allocation/deallocation is minimized with **hierarchical bitmaps**. This is important because fragmentation diminishes the benefit of the SOA layout (Section 6) and adversely affects cache performance [75].

[1] If thousands of threads were to request large memory allocations, a GPU would run out of memory immediately.

- Object allocation and deallocation performance is optimized with a number of **low-level techniques**. For example, DynaSOAr combines allocation requests within SIMD thread groups (*warps*) to reduce the number of memory accesses during allocations [88] and takes advantage of efficient bitwise operations/integer intrinsics.

Contributions This chapter makes the following contributions.

- The design and implementation of DynaSOAr, a dynamic object allocator for CUDA; with fast (de)allocation and a parallel do-all operation. To the best of our knowledge, DynaSOAr is the first dynamic allocator that stores objects in an SOA data layout.
- An extension of the SOA data layout to dynamic object sets and subclassing. While in Ikra-Cpp, the maximum number of objects of each class had to be specified as a compile-time constant, objects can be freely allocated in DynaSOAr without such restrictions.
- A concurrent, lock-free, hierarchical bitmap data structure, based on atomic operations and retry loops.
- A comparison and evaluation of existing state-of-the-art GPU memory allocators on SMMO applications.

Publications This chapter is in part based on the following papers.

- Matthias Springer, Hidehiko Masuhara. **"DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access."** In: *Proceedings of the 33rd European Conference on Object-oriented Programming*. ECOOP 2019. Leibniz-Zentrum für Informatik, Dagstuhl Publishing, 2019, LIPIcs, Vol. 134, pp. 17:1–17:37. `doi`:10.4230/LIPIcs.ECOOP.2019.17.

5.1 Design Goals

DynaSOAr is a CUDA framework for SMMO applications and consists of three parts.

Memory Allocator We developed a dynamic memory allocator that provides `new`/`delete` operations in GPU code and stores objects in an SOA data layout. The main task of the allocator is to decide where to store each field value of each object on the heap.

Data Layout DSL We developed an embedded C++ DSL to support OOP abstractions while storing objects in a custom layout. We could alternatively implement DynaSOAr in a language that allows programmers to specify custom data layouts (e.g., Shapes [65] or ispc [152]), but such languages have limited GPU support.

Parallel Do-All We developed an object enumeration strategy for SMMO applications that achieves efficient access of allocated memory on SIMD architectures. By controlling memory allocation and memory access, applications can achive better performance with DynaSOAr than with other state-of-the-art allocators, which are only concerned with memory allocation.

DynaSOAr's DSL builds on top of Ikra-Cpp's embedded C++ DSL for object-oriented programming with SOA layout (Section 4.2). Its purpose is to make DynaSOAr easier to use for programmers. This chapter is mainly about the memory allocator and the parallel do-all operation.

5.1.1 Programming Interface

In contrast to general memory allocators, DynaSOAr is an *object allocator*. The types (classes/structs) that can be allocated must be specified at compile time. DynaSOAr provides five basic operations. The operations for object enumeration follow the Ikra-Cpp API (Section 3.2.2), but are implemented differently and do not support host-side execution. All operations except for `parallel_do` and `parallel_new` are *device* functions that can only be called from GPU code.

- `new(d_allocator) T(args...)`: Allocates a new object of type T and returns a pointer to the object. The *placement new* notation [18] is a common C++ pattern for arena allocation and `d_allocator` is the allocator/arena in which the object is allocated.
- `destroy(d_allocator, ptr)`: Deletes an object with pointer `ptr`, assuming that the object was allocated with `d_allocator`[2].
- `HAllocatorHandle::parallel_do<S, &T::func>(args...)`: Launches a GPU kernel that runs a member function `T::func` for all objects of type S and subtypes[3] existing at launch time (*parallel do-all*), where $S <: T$. `T::func` may allocate new objects, but they are not enumerated by this parallel do-all. `T::func` may deallocate any object of different type $U \neq S$, but `this` is the only object of type S it may deallocate (delete itself). This is to avoid race conditions.
- `HAllocatorHandle::parallel_new<T>(n, args...)`: Launches a GPU kernel that instantiates n objects of type T. This operation calls the constructor of T in parallel with an object index (between 0 and n) as first argument, followed by `args...`. T must have a suitable constructor.
- `DAllocatorHandle::device_do<S, &T::func>(args...)`: Runs a member function `T::func` for all objects of type S in the current GPU thread, where $S <: T$. Can only be used inside of a `parallel_do` or a manually launched GPU kernel. This is a sequential *for-each* loop. It is typically used for processing all pairs of objects (e.g., in n-body simulations).

Listing 5.1 shows parts of an implementation of an n-body simulation with collisions to illustrate DynaSOAr's API and DSL (full example in Section 7.2).

Data Layout DSL Similar to Ikra-Cpp, programmers must define their classes with our data layout DSL. Through this DSL, DynaSOAr can programmatically *reflect* on the classes/fields that are defined in an application, somewhat similar to the Java Reflection API or metaobject protocols [31].

Before the actual application code begins, programmers must pre-declare all classes (Line 3) and define an allocator type (Line 4). Every allocator type is a template instantiation of `SoaAllocator`. The first template argument is the maximum number

[2]There is no *placement delete* syntax, so it is a common pattern to use a separate `destroy` function [178].
[3]To avoid branch divergence, we launch a separate kernel for every type.

LISTING 5.1: DYNASOAR API Example: n-body

```
#include "dynasoar.h"

class Body; // Pre-declare all classes. This simple example has only one class.
using AllocatorT = SoaAllocator</*max_num_obj=*/ 16777216, /*T...=*/ Body>;
__device__ DAllocatorHandle<AllocatorT> d_allocator;

class Body : public AllocatorT::Base { // Can subclass other user-defined class.
 public:
  // Pre-declare all field types. DynaSOAr uses these to compute the size of blocks.
  declare_field_types(Body, float /*pos_x_*/, float /*pos_y_*/,
                             /* ... */, bool /*was_merged_*/)
 private:
  // Declare fields with proxy types but use like normal C++ fields (as in Ikra-Cpp).
  Field<Body, 0> pos_x_; // Position X
  Field<Body, 1> pos_y_; // Position Y
  /* other fields omitted... */
  Field<Body, 9> was_merged_; // Was this body merged into another one?

 public:
  __device__ Body(float pos_x, float pos_y, float vel_x, float vel_y, float mass)
    : pos_x_(pos_x), pos_y_(pos_y), vel_x_(vel_x), vel_y_(vel_y), mass_(mass) {}

  // This constructor is invoked by parallel_new.
  __device__ Body(int id) : Body(/*pos_x=*/ random_float(0, 1), /*...*/) {}

  __device__ void apply_force(Body* other) {
    if (other != this) {
      float dx = pos_x_ - other->pos_x_; float dy = pos_y_ - other->pos_y_;
      float dist = sqrt(dx*dx + dy*dy);
      float F = kGravityConstant * mass_ * other->mass_ / (dist * dist);
      other->force_x_ += F * dx / dist; other->force_y_ += F * dy / dist;
    }
  }

  __device__ void step_1_compute_force() {
    force_x_ = force_y_ = 0.0f;
    d_allocator->device_do<Body, &Body::apply_force>(this);
  }

  __device__ void step_2_move(float dt) {
    vel_x_ += force_x_ * dt / mass_; vel_y_ += force_y_ * dt / mass_;
    pos_x_ += dt * vel_x_; pos_y_ += dt * vel_y_;
  }

  __device__ void step_6_delete_merged() {
    if (was_merged_) { destroy(d_allocator, this); }
  }
};

int main() {
  // Create new allocator. This will allocate a large buffer ("heap") on the GPU.
  auto* h_allocator = new HAllocatorHandle<AllocatorT>();
  // Allocate 65536 new bodies, randomly initialized.
  h_allocator->parallel_new<Body>(65536);
  for (int i = 0; i < kIterations; ++i) {
    h_allocator->parallel_do<Body, &Body::step_1_compute_force>();
    h_allocator->parallel_do<Body, &Body::step_2_move>(/*dt=*/ 0.5);
    /* some steps omitted... */
    h_allocator->parallel_do<Body, &Body::step_6_delete_merged>();
  }
  delete h_allocator; // Deallocate buffer and all allocations within.
}
```

of objects of the smallest type, which indirectly defines the heap size, as described in detail later. The other template arguments list all types that are under control of the allocator (variadic template). This design imposes some restrictions with respect to separate compilation: While different compilation units can be developed/compiled separately and then linked together in C++, the compilation unit containing the DynaSOAr memory allocator requires definitions of all classes/structs that should be dynamically allocated with DynaSOAr. However, this is a minor issue at the moment, since GPU applications rarely use separate compilation.

All user-defined classes/structs must in some way inherit from a special class `AllocatorT::Base`; either as a direct subclass or by inheriting from a class that already inherits from that class. Fields types must be pre-declared (Line 10) and then declared with proxy types (Lines 14–17).

Class Inheritance In contrast to Ikra-Cpp, a user-defined class can inherit from another user-defined class that is managed by the allocator. Multiple inheritance is not supported at the moment. A class can be marked as *abstract* by defining a static field `static const bool kIsAbstract = true;` after the field pre-declarations.

Programmers can downcast object pointers with the `cast<T>()` member function defined by `AllocatorT::Base`. This function is similar to C++ `dynamic_cast<T*>(ptr)`; it performs a dynamic type check and returns `nullptr` if the object is not of runtime type `T`.

The n-body simulation example in this chapter does not utilize class inheritance, but Chapter 7 contains a number of source code examples with class inheritance.

5.1.2 Memory Access Performance

The main insight of our work is that optimizing only for fast (de)allocations is not enough. Optimizing the access of allocated memory can result in much higher speedups, because device (*global*) memory access is the biggest bottleneck of memory-bound GPU applications:

Latency Global memory access instructions have a very high latency at around 400–800 clock cycles, compared to arithmetic instructions at around 6–24 cycles. Programmers can hide latency with *high occupancy* [193] (i.e., running many threads).

Memory Bandwidth The global memory bandwidth is a limiting factor. Peak memory transfer rates can be achieved only with *memory coalescing*: When the threads in a GPU application simultaneously access different memory addresses, the GPU coalesces accesses from the same SIMD thread group (*warp* in CUDA, every 32 consecutive threads) into one physical transaction if the addresses are on the same 128-byte cache line [94]. However, if threads access data on multiple cache lines (e.g., non-contiguous, spread-out addresses), more transactions are needed[4], which reduces transfer rates significantly. The CUDA Best Practices Guide puts a *high priority* note on coalesced memory accesses [36].

Caches Hits in the L1/L2 cache are served much faster (less latency, memory bandwidth pressure) than global memory loads. Field reordering and structure splitting are common techniques for increasing the number of hot fields in cache [32].

[4]This is similar to vectorized loads, but coalescing is performed at runtime by the hardware.

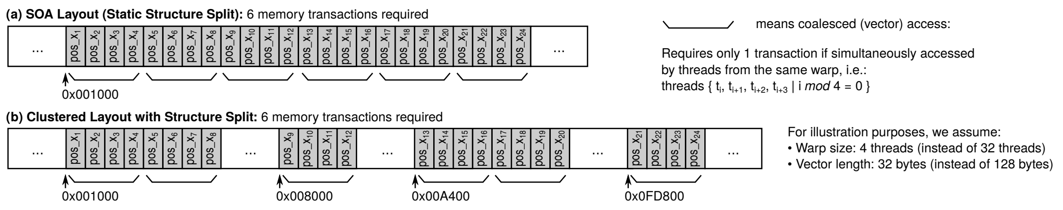

(a) SOA Layout (Static Structure Split): 6 memory transactions required
0x001000
(b) Clustered Layout with Structure Split: 6 memory transactions required
0x001000
0x008000
0x00A400
0x0FD800
means coalesced (vector) access:
Requires only 1 transaction if simultaneously accessed by threads from the same warp, i.e.: threads { t_i, t_i+1, t_i+2, t_i+3 | i mod 4 = 0 }
For illustration purposes, we assume:
• Warp size: 4 threads (instead of 32 threads)
• Vector length: 32 bytes (instead of 128 bytes)

FIGURE 5.1: Data layouts: SOA layout and DynaSOAr's SOA-style layout

DynaSOAr achieves good memory access performance with an SOA-style data layout: First, SOA increases memory coalescing because values of the same field, which are accessed simultaneously in SIMD, are stored together. Second, SOA is an extreme form of structure splitting and can improve cache utilization because fields that are not accessed do not occupy cache lines.

5.1.3 High Density Memory Allocation

An SOA data layout (Figure 5.1a) achieves good memory performance but is not suitable for dynamic allocation: The size of SOA arrays is fixed and new allocations cannot be accommodated once all array slots are occupied.

DynaSOAr's design is based on the insight that a *clustered layout* with SOA-style structure splitting (Figure 5.1b) has the same cache/vector performance characteristics as an SOA layout, if scalar values are stored in dense clusters of at least 128 bytes (vector and cache line size) and clusters are aligned to 128 bytes, regardless of where the clusters themselves are located in memory.

Figure 5.1 illustrates the number of required memory transactions for reading 24 floats simultaneously. For illustration purposes, we assume a warp size (vector length) of 4 instead of 32. Both layouts require the same number memory transactions because accesses can be equally well coalesced/vectorized in both layouts. This shows that a perfect SOA layout is not required for perfect memory coalescing. This insight is exploited by DynaSOAr's allocation policy and gives DynaSOAr more freedom in the placement of allocations.

5.1.4 Parallel Object Enumeration Strategy

Current GPUs follow the Single-Instruction Multiple-Threads (SIMT) execution model. Intuitively, every SIMD lane corresponds to a thread and every group of consecutive 32 threads forms a *warp* which executes an instruction on a vector register.

To benefit from memory coalescing, the threads of a warp must access addresses on the same 128-byte L1 cache line. In an SOA data layout, this is achieved when the threads of a warp read/write the same fields of objects with *contiguous indices* at the same time. Intuitively, threads in a warp should process *neighboring* objects.

In DynaSOAr, programmers invoke GPU kernels with parallel do-all operations. These operations must (a) spawn enough GPU threads to hide latency, but not too many to avoid inefficiencies, and (b) assign objects to threads in such a way that memory access is coalesced.

5.1.5 Scalability

Memory allocations require some sort of synchronization between threads to prevent *collisions*, i.e., two threads allocating the same memory location. To avoid collisions, some allocators such as Cilk [22] and Hoard [19] utilize private heaps, but such designs can lead to high memory consumption (*blowup*) [19] and are infeasible on massively parallel architectures with thousands of threads.

State-of-the-art GPU allocators such as ScatterAlloc [175] and Halloc [7] reduce collisions with hashing, which scatters allocations almost randomly in the heap. This would render an SOA layout useless and defeat one of DYNASOAR's main optimizations.

With such design restrictions, DYNASOAR is bound to have less efficient allocations than other allocators. However, as we show throughout this chapter, DYNASOAR can more than make up for slow allocations with more efficient access of allocated memory.

False Sharing Previous CPU memory allocator designs emphasize mechanisms for reducing false sharing, which can degrade performance [19]. A memory segment A is falsely shared if it is brought into a processor cache because it happens to be located on the same cache line as another memory segment B that is actually accessed. If caches are *coherent*, the entire cache line is invalidated if another processor modifies A, even though A is never read or written by the original processor.

False sharing is not an issue on GPUs, because L1 caches are not coherent. Programmers must use the `volatile` keyword or atomic operations to enfore a read or write to the shared L2 cache or global memory.

5.2 Architecture Overview

DYNASOAR manages a single, large heap in global memory on device. The heap is divided into M blocks and every block has the same number of bytes. M and the size of each block are compile-time constants that are calculated based on the set of classes that are under control of the allocator (template arguments to `SoaAllocator`), as described below.

A block contains only objects of the same type (class/struct), stored in SOA data layout (Figure 5.2). Once a block is initialized (*allocated*) for a certain type, only objects of that type can be stored in that block until the block is deallocated.

The maximum number of objects in a block (*block capacity*) depends on its type, because structs/classes may have different sizes. To improve clustering, DYNASOAR allocates new objects in already existing, non-full blocks (*fast path*). We call such blocks *active*, because they participate in allocations (Figure 5.3). If no active block could be found, a new block is allocated and becomes active (*slow path*).

5.2.1 Block Structure

Every block has two 64-bit object bitmaps: An *object allocation bitmap* and an *object iteration bitmap*. The allocation bitmap tracks allocated slots in the block. The iteration bitmap is used for object enumeration and overwritten with the allocation bitmap before every parallel do-all operation. Its purpose is to ensure that objects that were created during a parallel do-all operation are not enumerated by the same parallel do-all operation; that would a race condition.

The *type identifier* is a unique ID for the type T of a block. The remainder of the block is occupied by padding and the *data segment*, storing $1 \leq N_T \leq 64$ objects in SOA layout. The data segment begins with SOA arrays for inherited fields and ends with SOA arrays of newly introduced fields.

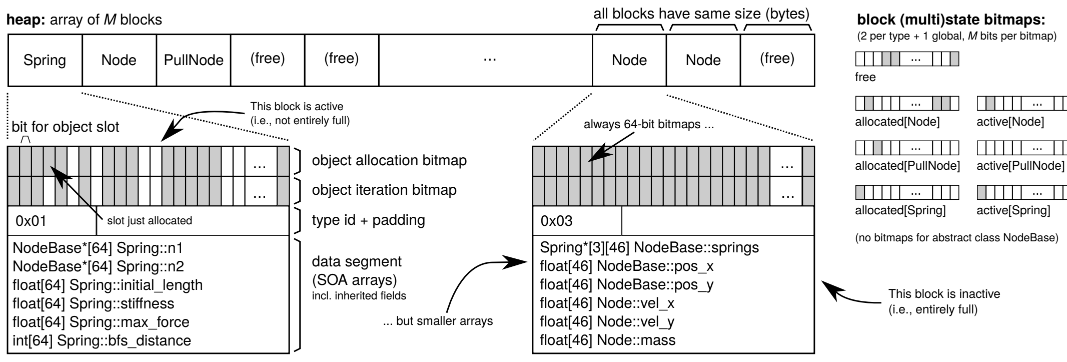


FIGURE 5.2: Example: Heap layout for an FEM simulation (Section 7.4) of a crack in a composite material. The heap is divided into M blocks of equal size. Every block has the same structure: an allocation bitmap, an iteration bitmap, and a type identifier, followed by a data segment storing objects in SOA layout.

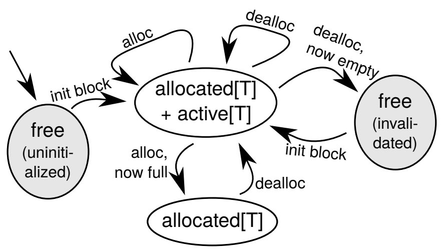


FIGURE 5.3: Block state transitions. At first, blocks are in an uninitialized state. As part of allocation, new active blocks may be initialized (*allocated*). Active blocks become inactive when they are full. Inactive blocks become active again an object is deallocated. Active blocks are invalidated when their last object is deallocated. Invalidated blocks can be reinitialized (to any type) and are handled similar to uninitialized blocks.

Slots are marked as (de)allocated with atomic AND/OR operations that change a single bit of the object allocation bitmap. Based on their return value[5], we know ...

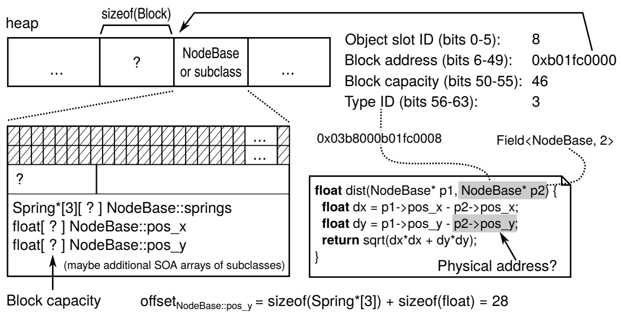
- ... if an allocation was successful or another thread was faster allocating the same slot.
- ... if a particular allocation filled up a block (i.e., allocated the last slot).
- ... if a particular deallocation emptied a block (i.e., deallocated the last slot).

If a thread filled up a block or emptied a block, it is that thread's *responsibility* to update the other internal data structures. This is a common pattern in lock-free designs [143]. Note that every block has the same byte size and structure; e.g., the bitmaps are always at the same offset. This is an important property for the correctness of our (de)allocation algorithms.

5.2.2 Block Capacity

The capacity of a block (maximum number of objects) depends on the size (bytes) of the type of objects in the block. If DYNASOAR manages objects of types T_1, T_2, ..., T_n and $s = \text{argmin}_{i \in 1 \ldots n}\, size(T_i)$ is the index of the smallest type, then the capacity N_T of a block of type T is determined as follows.

[5]An atomic operation returns the value in memory before modification.


heap
sizeof(Block)
…
?
NodeBase or subclass
…
Object slot ID (bits 0-5): 8
Block address (bits 6-49): 0xb01fc0000
Block capacity (bits 50-55): 46
Type ID (bits 56-63): 3
0x03b8000b01fc0008
Field<NodeBase, 2>
…
…
?
Spring*[3][?] NodeBase::springs
float[?] NodeBase::pos_x
float[?] NodeBase::pos_y
(maybe additional SOA arrays of subclasses)
Physical address?
Block capacity
offset_NodeBase::pos_y = sizeof(Spring*[3]) + sizeof(float) = 28

```
float dist(NodeBase* p1, NodeBase* p2) {
  float dx = p1->pos_x - p2->pos_x;
  float dy = p1->pos_y - p2->pos_y;
  return sqrt(dx*dx + dy*dy);
}
```

FIGURE 5.4: Object pointer example. The static type of `p2` is `NodeBase*`. The corresponding block has SOA arrays for `NodeBase` fields and for the additional fields of the runtime type of `p2`. The size of those arrays is not statically known and depends on the runtime type of `p2`.

$$N_T = \left\lfloor \frac{64 \cdot size(T_s)}{size(T)} \right\rfloor \qquad (\textit{block capacity})$$

According to this formula, a block of the smallest type T_s has capacity 64. This is why the bitmaps within a block have 64 bits. Given a fixed heap size, the size of T_s determines the block size in bytes and thus the number of blocks M.

$$M = \frac{\text{heap size (bytes)}}{\text{size of block of type } T_s \text{ (bytes)}} \qquad (\textit{block capacity})$$

In our current implementation, programmers specify the size of the heap not in bytes but in terms of the number of objects of the smallest type T_s (first template argument of `SoaAllocator`). This number must be a multiple of 64. This is merely an implementation detail and will change in future versions of DYNASOAR.

Very Small/Large Object Sizes As soon as a type T is more than twice as big as T_s, the benefit of the SOA layout is starting to fade away for T, because the number of objects in such a block N_T will be smaller than 32. However, assuming 32-bit scalar types, the maximum amount of memory coalescing can only be achieved with vector loads (cluster sizes) of 32 values (Section 2.1.5). Moreover, DYNASOAR cannot handle cases in which a type is more than 64 times bigger than the smallest type. In reality, these limitations proved to be insignificant. None of our benchmarks experienced a slowdown due to unfavorable block sizes.

5.2.3 C++ Data Layout DSL and Object Pointers

Field access is simple in most object-oriented systems: Given an object pointer, which is a memory location, a field value is stored at a fixed offset from the memory location.

In DYNASOAR, an object pointer is not a memory location, but a combination of various components (*fake pointer*), similar to *global references* in *Shapes* [65]. Upon field access, the DYNASOAR DSL transparently converts object pointers to memory locations, without breaking C++'s OOP abstractions. We follow the implementation strategy of IKRA-CPP, where fields are declared with proxy types `Field<B, N>`, which can be implicitly converted to `T&` values [81], where `T` is the N-th predeclared field type of `B`. This conversion is defined by our DSL and computes the actual, physical memory location within a data segment.

Fake pointers in DYNASOAR are different from fake pointers in IKRA-CPP. They do not just encode an object ID, but four different components. A DYNASOAR object pointer (Figure 5.4) is based on the address of the block in which the object is located. All blocks are aligned to 64 bytes, so we can store the object slot ID in the 6 least significant bits[6]. Since recent GPU architectures have at most 24 GB of memory and no virtual memory, only the 35 least significant bits are used in memory addresses and the remaining 29 bits are always zero[7]. We store additional information in these bits: The 8 most significant bits store the type identifier of the class type for fast instance-of checks. The next 6 bits store the capacity of the block. The remaining 15 bits are effectively not utilized at the moment. Note that, while C++ stores runtime types with a vtable pointer at the beginning of an object, we store runtime type information in unused pointer bits.

Field Access While in most object-oriented systems, runtime type information is only required for virtual function calls, DynaSOAr requires the block capacity (a property of the runtime type!) also for field accesses, because SOA array offsets within the data segment depend on it.

For example, `p2` in Figure 5.4 is statically known to be of type `NodeBase*`, but the block capacity (size of SOA arrays) depends on the runtime type, which can be any subclass of `NodeBase`. Those subclasses can have different block capacities. The size of SOA arrays and the object slot ID are required to compute the physical location of `p2->pos_y` based on the block address, so we store both inside object pointers.

LISTING 5.2: Field address computation

```
// Impl. conv. operator: E.g., convert Field<NodeBase, 2> to float& in Figure 5.4.
// BaseType: N-th predeclared type in B (within declare_field_types).
template<typename B, int N>
Field<B, N>::operator BaseType&() {
  int offset = ...; // Computed with templ. metaprog. offset_{B::fieldname} in Figure 5.4.
  auto obj_ptr = reinterpret_cast<uint64_t>(this) - N;
  // Bits 0-49 and clear 6 least significant bits.
  auto* block_address = reinterpret_cast<char*>(obj_ptr & 0x3FFFFFFFFFFC0);
  int obj_slot_id = obj_ptr & 0x3F; // Bits 0-5
  int block_capacity = (obj_ptr & 0xFC000000000000) >> 50; // Bits 50-55
  auto* soa_array = reinterpret_cast<BaseType*>(
      block_address + field_offset * block_capacity);
  return soa_array[obj_slot_id];
}
```

This computation (Listing 5.2), along with bit-shifting and bit-AND operations for extracting all components from a fake object pointer, is performed on every field read/write. Compilers may eliminate redundant computations of multiple accesses with peephole optimizations, but field accesses still require more complex arithmetics compared to an AOS layout. This overhead may seem large, but arithmetic operations are much faster than memory access, even in case of an L2 cache hit. Overall, the performance benefit of SOA is much larger than the address computation overhead.

5.2.4 Block Bitmaps

To find blocks or free memory quickly during object enumeration or object allocation, DynaSOAr maintains three bitmaps of size M, where M is the number of blocks on the heap.

[6]Recall that blocks never have more than 64 objects, so 6 bits are enough.
[7]We experimentally verified this on NVIDIA Maxwell and NVIDIA Pascal.

- The *free block bitmap* indexes block locations that are not yet allocated. This bitmap is used to determine where new blocks are allocated. Bit i is 1 iff block i is free (uninitialized or invalidated). Initially, every bit is 1.
- There is one *block allocation bitmap* for every type T. That bitmap indexes blocks of type T and is used for enumeration of all objects. Blocks of subclasses are not included in bitmaps of the superclass. Initially, every bit is 0.
- There is one *active block bitmap* for every type T, indexing allocated, non-full blocks. If a bit is 1, then the same bit in the block allocation bitmap must also be 1. This bitmap is used to find a block in which a new object can be allocated. Initially, every bit is 0.

Due to concurrent (de)allocations, block bitmaps cannot be kept consistent with the actual block states all the time, as indicated by object allocation bitmaps and type identifiers of blocks. However, we designed our algorithms in such a way that they can handle such inconsistencies and keep block states and block bitmaps *eventually consistent* (Section 5.4).

5.2.5 Object Slot Allocation

When a new object is created, DYNASOAR allocates memory and runs the constructor on the object pointer. Algorithm 1 shows how memory is allocated. This algorithm runs entirely on the GPU and is completely lock-free.

DYNASOAR tries to allocate memory in an already existing, active block. If no block could be found, it first initializes a new block at a location that is known to be free (*slow path*). The state of the new block is *allocated* and *active*, so that the new block can also be found by other threads.

Once a block was selected, an object slot is reserved by atomically finding and flipping a bit from 0 to 1 in the object allocation bitmap (details in Algorithm 7). Based on the return value of the atomic operation, we know if this operation just allocated the last slot. In that case, the block is marked as *inactive* (Line 12).

Since the allocator is used concurrently by many threads, we may select a block (Line 2) that is full or no longer exists when attempting to reserve an object slot (Line 8). If the block is full, object reservation fails and we retry by selecting a new active block. If the block no longer exists, we have to consider three cases[8].

1. There is currently no block at this location. In this case, object reserveration fails, because all slots are marked as allocated in the object allocation bitmap when a block is deleted. We call this process *block invalidation* (Section 5.4.2).
2. The block was deleted and a new block of the same type was allocated at the same location. Such ABA problems are harmless and allocation will succeed.
3. The block was deleted and there is now a block of different type at the same location. At this point, the constructor has not run yet, so no data in the data segment was corrupted. This is because all blocks have the same structure, i.e., the object allocation bitmap is always at the same location. We can safely rollback the allocation by running the deallocation routine.

[8]We give a more systematic correctness argument in Section 5.4.

Algorithm 1: DAllocatorHandle::allocate<T>() : T* GPU

```
1  repeat                                                ▷ Infinite loop if OOM
2      bid ← active[T].try_find_set();                   ▷ Find and return the position of any set bit.
3      if bid = FAIL then                                ▷ Slow path
4          bid ← free.clear();                           ▷ Find and clear a set bit atomically, return position.
5          initialize_block<T>(bid);                     ▷ Set type ID, init. obj. alloc. bitmap. See Alg. 10.
6          allocated[T].set(bid);
7          active[T].set(bid);
8      alloc ← heap[bid].reserve();                      ▷ Reserve an object slot. See Alg. 7.
9      if alloc ≠ FAIL then
10         ptr ← make_pointer(bid, alloc.slot);
11         t ← heap[bid].type;                           ▷ Volatile read
12         if alloc.state = FULL then active[t].clear(bid) ;
13         if t = T then return ptr ;
14         deallocate<t>(ptr);                           ▷ Type of block has changed. Rollback.
15 until false;
```

Algorithm 2: DAllocatorHandle::deallocate<T>(T* ptr) : void GPU

```
1  bid ← extract_block(ptr);
2  slot ← extract_slot(ptr);
3  state ← heap[bid].deallocate(slot);                   ▷ Opposite of slot reservation. See Alg. 8.
4  if state = FIRST then                                 ▷ Deallocated first object of full block.
5      active[T].set(bid);
6  else if state = EMPTY then                            ▷ Deallocated last object of block.
7      if invalidate(bid) then                           ▷ Try to invalidate block. See Alg. 11.
8          t ← heap[bid].type;
9          active[t].clear(bid);
10         allocated[t].clear(bid);
11         free.set(bid);
```

5.2.6 Object Deallocation

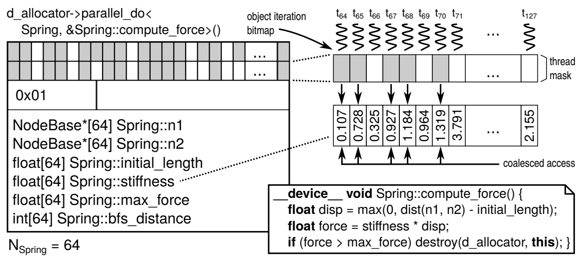
When an object is deallocated, DYNASOAR first determines its runtime type T from the fake object pointer. Then, DYNASOAR runs the C++ destructor and deallocates the memory as shown in Algorithm 2.

We first determine block and object slot IDs from the object pointer and deallocate the object slot by atomically flipping its bit in the object allocation bitmap from 1 to 0. Based on the return value of the atomic operation we know the fill level of the block right before the deallocation.

If this deallocation freed the first object slot (block previously full), we mark the block as active (Line 5), so that other threads can find it and allocate objects in it.

If this deallocation freed the last object slot (block now empty), we attempt to delete the block (Lines 7–11). Safe memory reclamation is known to be difficult in lock-free algorithms [141]. The main problem is that one or more contending threads, in the course of their lock-free operations, may have selected the block that we are about to delete for new allocations.

To avoid the block from being modified by other threads, we *invalidate* it (Section 5.4.2). Block invalidation attempts to atomically flip all bits in the object allocation bitmap from 0 to 1. If this atomic operation failed to flip at least one bit from 0 to 1 (because it was already 1), another thread must have reserved an object slot in the



```
d_allocator->parallel_do<
  Spring, &Spring::compute_force>()
```

```
NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance
```

```
__device__ void Spring::compute_force() {
  float disp = max(0, dist(n1, n2) - initial_length);
  float force = stiffness * disp;
  if (force > max_force) destroy(d_allocator, this); }
```

FIGURE 5.5: Thread assignment example. 64 threads with consecutive IDs are assigned to every allocated block of type `Spring`. Since not all object slots are in use, as indicated by the block iteration bitmap, some threads have no work to do. All other threads can benefit from memory coalescing when reading/writing fields of the object that they are assigned to.

meantime. In this case, we rollback the changes to the object allocation bitmap and abort block invalidation and deletion.

If invalidation was successful, the block is guaranteed to be empty and cannot be modified by other threads anymore because all bits in the object allocation bitmap are 1. The type of the block may have changed in the meantime (Line 8), but it is now safe to mark this block location as *free*, so that a new block can be initialized at this location.

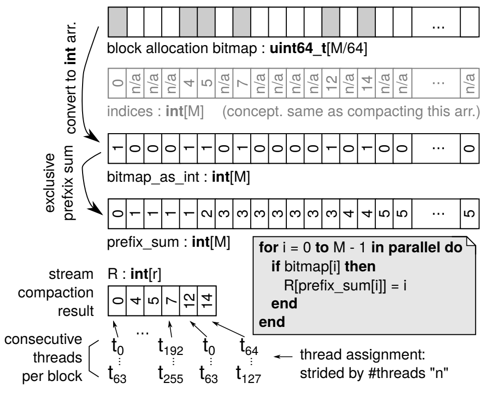
5.2.7 Parallel Object Enumeration: `parallel_do`

Parallel do-all is the foundation of SMMO applications. It launches a GPU kernel that runs a method `T::func` on all objects of a type T (and subtypes). To avoid branch divergence, we run a separate kernel for each subtype. `T::func` may read and write fields of the object that it is bound to (`this`). The goal of parallel do-all is to assign objects to GPU threads in such a way that memory coalescing is maximized for those field accesses.

Memory coalescing is maximized when all threads of a warp access consecutive memory addresses at the same time (and addresses are properly aligned). In this case, all those memory accesses can be serviced by vector loads/writes. In CUDA, threads are identified by thread IDs. Each warp consists of a consecutive range of threads. E.g., warp 0 consists of threads $t_0, t_1, \ldots t_{31}$. Assuming a block capacity of N_T, DYNASOAR assigns N_T consecutive threads to the objects in a block (Figure 5.5). This leads to good memory coalescing on average. Perfect memory coalescing can be achieved if the following two conditions apply.

- N_T is a multiple of the warp size 32. If this is not the case, then there are warps whose threads process elements in two or more different blocks.
- Objects have good clustering, i.e., every block except for at most one is entirely full. Due to the way objects are allocated (only in active blocks), we expect a high fill level on average.

DYNASOAR uses the block allocation bitmap to find blocks to which threads should be assigned. Assigning only one object to a thread is too inefficient if the number of objects is large. Therefore, a thread t_{tid} may have to process an object slot in multiple blocks. $num_B(tid)$ is the number of blocks that are assigned to a thread t_{tid}. Our scheduling strategy always assigns the same object slot position $id_O(tid)$ to a thread, but in multiple blocks $id_B(tid)$. In the following formulas, R is an array of indices of all allocated blocks of type T, i.e., all blocks containing objects of type T.


convert to **int** arr.
block allocation bitmap : **uint64_t**[M/64]
0 n/a n/a n/a 4 5 n/a 7 n/a n/a n/a n/a 12 n/a 14 n/a n/a ... n/a
indices : **int**[M] (concept. same as compacting this arr.)
1 0 0 0 1 1 0 1 0 0 0 0 1 0 1 0 0 ... 0
bitmap_as_int : **int**[M]
exclusive prefxix sum
0 1 1 1 1 2 3 3 3 3 3 3 3 4 4 5 5 ... 5
prefix_sum : **int**[M]
stream compaction result
R : **int**[r]
0 4 5 7 12 14
consecutive threads per block
t_0 t_{63} ... t_{192} t_{255} t_0 t_{63} t_{64} t_{127}
thread assignment: strided by #threads "n"

```
for i = 0 to M - 1 in parallel do
  if bitmap[i] then
    R[prefix_sum[i]] = i
  end
end
```

FIGURE 5.6: Example: Compacting block allocation bitmap indices and assigning $n = 256$ threads to 6 allocated blocks with $N_T = 64$. The prefix sum retains the order of indices (i.e., R is sorted), but this is not required for the correctness of our algorithms.

The total number of threads n can be hand-tuned by the programmer. With those formulas, every thread can by itself determine the objects that it should process.

$$id_O(tid) = tid \ \% \ N_T \qquad (\textit{assigned object slot index})$$

$$id_B(tid) = \left(R\left[\frac{tid + k \cdot n}{N_T}\right] \mid k \in [0; num_B(tid)) \right) \qquad (\textit{assigned block indices})$$

$$num_B(tid) = \left\lceil \frac{r \cdot N_T - tid}{n} \right\rceil \qquad (\textit{number of assigned blocks})$$

The formulas id_O and id_B effectively implement a grid-stride loop (Section 2.1.3). The array R is required because every thread should by itself find the $\frac{tid}{N_T}$-th, $\frac{tid+n}{N_T}$-th, etc. allocated block of type T quickly, without scanning the entire block allocation bitmap or communicating with other threads, which would be very slow.

DYNASOAR precomputes R before every parallel do-all operation (Figure 5.6). Conceptually, this is an application of stream compaction [14]: Given a bitmap of size M (size of heap in number of blocks), generate an *indices* array of size M containing i at position i if the i-th bit is set. Otherwise, store an *invalid marker*. Now filter/compact the array to retain only valid values, resulting in an array R of size r. Note that we do not care if the original ordering of indices is retained. This stream compaction could be implemented as follows.

1. Convert the bitmap into an integer array.
2. Compute the exclusive prefix sum [167, 21] of the integer array.
3. Compute the final result R: If a bit is set in the bitmap, store its index at the position that was computed by the exclusive prefix sum.

Every step of this stream compaction implementation can run in parallel. Section 5.3.1 describes how this algorithm is further optimized with hierarchical bitmaps to avoid scanning empty bitmap parts.

5.3 Optimizations

This section describes performance optimizations that DYNASOAR applies in addition to the SOA data layout to achieve good (de)allocation performance and to reduce memory fragmentation.

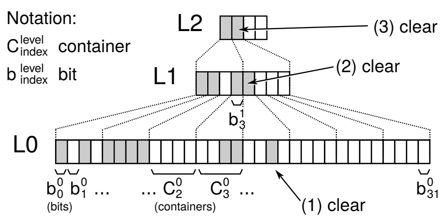


FIGURE 5.7: Example: Hierarchical bitmap of size 32 with container size 4 (instead of 64). This example illustrates how (1) a *clear*(18) operation triggers (2) a *clear*(4) operation in the nested bitmap, which triggers (3) a *clear*(1) operation in the next nested bitmap.

5.3.1 Hierarchical Bitmaps

DYNASOAR uses bitmaps for finding blocks or free space for blocks. Since, with growing heap sizes, bitmaps can reach several megabytes in size, we use a hierarchy of bitmaps, such that *set* bits (ones) can be found with a logarithmic order of memory accesses.

LISTING 5.3: Structurally recursive C++ bitmap implementation (simplified)

```
template<int N, bool HasNested>
struct Bitmap;

template<int N>
struct Bitmap<N, /*HasNested=*/ false> {
  static const int kNumContainers = (N + 64 - 1) / 64; // ceil(N / 64)
  uint64_t containers[kNumContainers];
};

template<int N>
struct Bitmap<N, /*HasNested=*/ true> {
  static const int kNumContainers = (N + 64 - 1) / 64; // ceil(N / 64)
  static const bool kContinueHierarchy = kNumContainers > 1;

  uint64_t containers[kNumContainers];
  Bitmap<kNumContainers, kContinueHierarchy> nested;
};
```

Our hierarchical bitmaps are structurally recursive (nested bitmaps; Listing 5.3) and hide their hierarchy as an implementation detail from their interface. Such bitmaps are used in database systems [144] and garbage collectors [185], but we do not know of any hierarchical bitmaps that support concurrent modifications.

Data Structure A hierarchical bitmap of size N bits consists of two parts: an array of size $\lceil N/64 \rceil$ of 64-bit *containers* (`uint64_t`), and a *nested bitmap* of size $\lceil N/64 \rceil$ if $N > 64$. A container C^l_i consists of bits $b^l_{64 \cdot i}$, ..., $b^l_{64 \cdot i+63}$ and is represented by one bit b^{l+1}_i in the nested (higher-level) bitmap (Figure 5.7). That bit is set if at least one bit is set in the container.

$$b^{l+1}_i = \bigvee_{k=0}^{63} b^l_{64 \cdot i+k} \qquad (\textit{container consistency})$$

We chose a container size of 64 bits because C++ has a 64-bit integer type and CUDA (and most other architectures) provide atomic operations for modifying 64-bit values. Bits in a container are changed with atomic operations. Higher-level bits (and thus bitmaps) are *eventually consistent* with their containers. Keeping both consistent all the time is impossible without locking, because two different memory

Algorithm 3: Bitmap::try_clear(pos) : void $\boxed{\text{GPU}}$

```
cid ← pos / 64;                                   ▷ Container ID
offset ← pos % 64;                ▷ Index of bit within the container
mask ← 1 << offset;
prev ← atomicAnd(&container[cid], ∼mask);
success ← (prev & mask) ≠ 0;
if success ∧ has_nested ∧ popc(prev) = 1 then
    nested.clear(cid);
return success;                   ▷ Was the bit switched from 1 to 0?
```

population count:
number of set bits

locations cannot be changed together atomically. However, due to the design of the bitmap operations, the bitmap is guaranteed to be in a consistent state when all bitmap operations (of all threads) are completed, at the end of a GPU kernel. Bitmap operations retry or give up (*FAIL*) to handle temporary inconsistencies. This is a key difference compared to other lock-free hierarchical data structures such as SNZI [56], which have stronger runtime consistency guarantees and require more complex algorithms.

Operations All bitmap operations except for *indices()* are device functions that run entirely on the GPU. All operations that modify memory are thread-safe and their semantics are atomic. Internally, they are all implemented with atomic memory operations.

- `try_clear(pos)` atomically sets the bit at position `pos` to 0. Returns *true* if the bit was 1 before and *false* otherwise.
- `clear(pos)` *switches* the bit at position `pos` from 1 to 0. Retries until the bit was actually changed by the current thread. This is identical to the following code snippet: `while (!try_clear(pos)) {}`
- `try_set(pos)` atomically sets the bit at position `pos` to 1. Returns *true* if the bit was 0 before and *false* otherwise.
- `set(pos)` *switches* the bit at position `pos` from 0 to 1. Retries until the bit was actually changed by the current thread. This is identical to the following code snippet: `while (!try_set(pos)) {}`
- `try_find_set()` returns the position of an arbitrary bit that is set to 1 or *FAIL* if none was found. This operation must be used with caution, because the returned bit position might already have changed when using the result.
- `clear()` atomically clears and returns the position of an arbitrary set bit. This is a combination of `try_find_set` and `try_clear` and identical to the following code snippet: `while ((i = try_find_set()) != FAIL && try_clear(i)) {}; return i;`
- `get(pos)` and `operator[](pos)` return the value of the bit at position `pos`.
- `indices()` returns an array of indices of all set bits. This is a host function and cannot be used in a GPU kernel.

Algorithm 4: Bitmap::try_find_set() : int $\boxed{\text{GPU}}$

```
1  if has_nested then
2      cid ← nested.try_find_set();          ▷ Select container based on L+1 bitmap.
3      if cid = FAIL then return FAIL ;
4  else
5      cid ← 0;                              ▷ This is the top-most bitmap level. Only 1 container.
6  offset ← ffs (container[cid]);            find first set: index of 1st set bit[9]
7  if offset = NONE then
8      return FAIL;
9  else
10     return 64*cid + offset;
```

Algorithm 5: Bitmap::indices() : int[N] $\boxed{\text{CPU}}$

```
1  if has_nested then
2      selected ← nested.indices();          ▷ Select containers based on L+1 bitmap.
3  else
4      selected ← [0];                       ▷ This is the top-most bitmap level. Only 1 container.
5  R ← array(N);
6  r ← 0;
7  for cid ∈ selected in parallel do         ▷ [GPU] (CUDA kernel)
8      c ← container[cid];
9      s ← atomicAdd(&r, popc(c));
10     for i ← 0 to popc(c)) do
11         R[s + i] ← 64*cid + nth_set_bit (c, i);     idx. of ith set bit in c
12 return R.subarray(0, r);
```

Set and Clear with Atomic Operations As many other lock-free algorithms, our hierarchical bitmaps are based on a combination of atomic operations and retries [44]. The return value of an atomic operation indicates if a bit was actually changed and if it is this thread's responsibility to update the higher-level bitmap (Figure 5.7).

As an example, Algorithm 3 shows how to clear the bit at position `pos`. In Line 4, the respective container is bit-ANDed with a mask containing ones everywhere except for that position. This will clear the bit at position `pos` but leave all other bits unchanged. The current thread actually changed the bit if it is set in `prev` (Line 5). If this operation cleared the last bit (Line 6), then the bit in the higher-level bitmap must be cleared.

Note that higher-level bits are always changed with *clear(pos)*/*set(pos)* and not with their respective *try_* versions, because other concurrently running bitmap operations may not have updated all higher-level bitmaps yet, leaving the data structure in a temporarily inconsistent state. If we were to use *try_* versions, a mandatory update of the higher-level bitmap could be accidentally dropped due to a bitmap inconsistency. However, *clear(pos)*/*set(pos)* ensure that the update is performed eventually by retrying (and spin-blocking the thread) until the update was successful.

Finding an Arbitrary Set Bit Instead of scanning the entire L0 bitmap, set bits can be found faster with a top-down traversal of the bitmap hierarchy, as shown in

[9]This is slightly different from CUDA, where `__ffs(b)` returns the position of the 1st set bit plus 1, or 0 if none exists. In this work, *ffs*(b) returns the position of the 1st set bit, or *NONE* if none exists.

Algorithm 4. A request is first delegated to the higher-level bitmap (Line 2) to select a container. When that call returns, a set bit is chosen in the selected container (Line 6).

This operation fails if there is no set bit in the entire bitmap data structure. However, even if the bitmap has set bits, this operation can fail if it reads an inconsistent combination of container values from different hierarchy levels. For example, consider the case where a container with exactly one set bit is chosen by the recursive call. However, before reaching Line 6, another thread clears that bit as part of a concurrent bitmap operation. In that case, *try_find_set* fails even though there may be set bits in other containers.

DynaSOAr is affected by such bitmap inconsistencies when searching for active blocks (Algorithm 1, Line 2) or free block positions (Line 4). While bitmap inconsistencies do not affect correctness, they can lead to higher fragmentation if DynaSOAr fails to find active blocks and instead initializes additional blocks, even though objects could have been accommodated in already existing blocks. We analyze the effect of bitmap inconsistencies in the benchmark section (Section 5.6.3).

Enumerating Set Bit Indices Before launching a parallel do-all kernel, DynaSOAr uses the *indices* operation to generate a compact array of allocated block indices (R in Figure 5.6). No GPU code is running at this time, so the bitmap is guaranteed to be in a consistent state. To ensure good scaling with increasing heap sizes, and thus increasing block bitmap sizes, DynaSOAr utilizes the bitmap hierarchy to quickly skip containers without any set bits (Algorithm 5).

First, an index array is generated for the higher-level bitmap (Line 2). This array is then processed in parallel; the *for* loop in Line 7 is a GPU kernel and every thread processes one or multiple containers selected by the recursive call. If a container C_i^l does not have any set bits, then its corresponding bit b_i^{l+1} is in a cleared state in the higher-level bitmap and not included in `selected`. Every thread reserves space in the result array R by increasing an atomic counter and fills its portion of the array with bit indices. This algorithm proved to be faster and requires less memory than a prefix sum algorithm, which needs multiple array copies/buffers per bitmap. Interestingly, related work has recently made the same observation with BFS algorithms on GPUs [69]. Note that, in contrast to the prefix sum-based implementation of Section 5.2.7, this algorithm does not necessarily retain the order of indices, i.e., the result array R is not sorted.

5.3.2 Reducing Thread Contention

In Algorithms 4 and 7, threads are competing with each other for bits: Only one thread can reserve any given object slot and only a limited number of threads can succeed with allocations in a block. To guarantee correctness, our design is heavily based on atomic operations. These operations became considerably faster with recent GPU architectures [47, 6], but performance can still suffer when too many threads choose the same bit, because threads have to retry if allocation fails. DynaSOAr employs two techniques to reduce such *thread contention*.

Allocation Request Coalescing Originally proposed by XMalloc [88], DynaSOAr combines memory allocation requests of the same type within a warp. One *leader thread* reserves all object slots in a single block on behalf of all participating threads. If the selected active block does not have enough free object slots, DynaSOAr reserves as many slots as possible and then chooses another active block for the

Algorithm 6: DAllocatorHandle::allocate<T>() : T* $\boxed{\text{GPU}}$

```
1  repeat                                                         ▷ Infinite loop if OOM
2      active ← __activemask();                                   ▷ Bitmap of active threads in warp
3      leader ← ffs(active);                                      ▷ Leader = active thread with lowest ID
4      rank ← __lane_id();                                        ▷ Rank of this thread
5      if leader = rank then                                      ▷ This thread is the leader.
6          bid ← active[T].try_find_set();
7          if bid = FAIL then                                     ▷ Slow path
8              bid ← free.clear();
9              initialize_block<T>(bid);
10             allocated[T].set(bid);
11             active[T].set(bid);
12         alloc_bitmap ← heap[bid].reserve_multiple( popc(active) );
13         if popc(alloc_bitmap) $> 0$ then
14             t ← heap[bid].type;
15             if alloc.state = FULL then  active[t].clear(bid) ;
16             if $t \neq T$ then  deallocate_multiple<t>(bid, alloc_bitmap) ;
17     alloc_bitmap ← __shfl_sync(active, alloc_bitmap, leader);
18     bid ← __shfl_sync(active, bid, leader);
19     id_in_active ← popc(__lanemask_lt() & active);
20     slot ← nth_set_bit(alloc_bitmap, id_in_active);
21     if slot $\neq$ FAIL then                                   ▷ else: Not enough slots reserved
22         return make_pointer(bid, slot);
23 until false;
```

remaining allocation requests. This reduces the number of atomic memory operations, because multiple bits in an object allocation bitmap are set in one operation. Furthermore, the constructor for newly allocated objects can run more efficiently, because field accesses are more likely coalesced.

Algorithm 6 shows how allocation request coalescing is implemented in DYNASOAR. This algorithm is an improved version of Algorithm 1. The following paragraph is intended for experienced CUDA programmers, as the algorithm takes advantage of CUDA warp-level primitives for intra-warp synchronization [123].

Out of all threads that are allocating an object of type T, the algorithm first select a leader thread. The leader attempts to reserve an object slot for every participating thread, similar to Algorithm 1. The slot reservation algorithm was extended such that it can allocate multiple slots in one go: n=*popc*(active) is the number of requested allocations and the algorithm attempts to reserve up to n slots. The result is an *allocation bitmap* with up to n set bits, which were reserved in the object allocation bitmap of the block. Block ID and allocation bitmap are lane-shuffled to the other participating threads in the warp. Every thread locates the position of the i-th set bit in the allocation bitmap, where i is the index of the thread in the warp among all participating threads (Line 20). If not enough bits were reserved for all threads (e.g., because the selected block was almost full), then these threads retry from the beginning, potentially with a new leader thread.

Bitmap Rotation Instead of a plain *find first set* (ffs) in Algorithms 4 and 7, bitmaps are first rotating-shifted by a value depending on the warp ID and a seed that is changed with every retry. This increases the probability of threads choosing different active blocks for allocation and reduces the probability of threads trying to reserve

```
Algorithm 7: Block::reserve() : (int, state)     ▷ Assuming block size 64.  GPU
1  repeat
2      pos ← ffs(∼bitmap);
3      if pos = NONE then return FAIL ;
4      mask ← 1 << pos;
5      before ← atomicOr(&bitmap, mask);
6      success ← (before & mask)) = 0;
7      block_full ← before = 0xFF...F;
8  until success ∨ block_full;
9  if success then
10     if popc(before) = 63 then
11         return (pos, FULL)
12     else
13         return (pos, REGULAR)
14 return FAIL;
```

```
Algorithm 8: Block::deallocate(pos) : state     ▷ Assuming block size 64.  GPU
1  mask ← 1 << pos;
2  before ← atomicAnd(&bitmap, ∼mask);
3  success ← (before & mask)) ≠ 0;
4  assert(success);                                  ▷ Precondition.
5  if popc(before) = 1 then
6      return EMPTY;
7  else if popc(before) = 64 then
8      return FIRST;
9  else
10     return REGULAR;
```

the same object slots in a block. This is a key optimization technique that improved performance by an order of magnitude.

Lookup Retry While bitmap traversals are relatively cheap, block initializations are expensive because in addition to initializing object bitmaps, bits in three different bitmaps (plus hierarachy) must be changed (slow path of Algorithm 1). To avoid unnecessary block initializations, it proved beneficial to retry the search for active blocks (Line 2) a constant number of times before entering the slow path. This optimization resulted in lower fragmentation and improved performance.

5.3.3 Efficient Bit Operations

DYNASOAR is taking advantage of efficient bitwise operations such as *ffs* ("find first set") and *popc* ("population count"). Modern CPU and GPU architectures have dedicated instructions for such operations. As an example, Algorithm 7 shows how a single object slot is reserved[10]. Instead of checking all bits in a loop, *ffs* in Line 2 is used to find a free slot (index of a cleared bit) in the object allocation bitmap and *popc* in Line 10 counts the number of previously allocated slots (number of set bits) to decide if this request filled up the block. Similarly, to decide if a deallocation freed the

[10]This algorithm was extended for allocation request coalescing (not shown here).

Algorithm 9: nth_set_bit(bitmap, n) : int $\boxed{\text{GPU}}$

```
1 for $i \leftarrow 0$ to $n-1$ do                    ▷ Clear first $n-1$ bits.
2     bitmap ← bitmap & (bitmap - 1);
3 return ffs(bitmap);
```

Algorithm 10: DAllocatorHandle::initialize_block<T>(int bid) : void $\boxed{\text{GPU}}$

```
1 heap[bid].type ← T;                                 ▷ Volatile write.
2 __threadfence();
3 heap[bid].bitmap ← 0;    ▷ Volatile write, assuming block capacity 64.
```

first or last slot, *popc* counts the number of bits before modifying the object allocation bitmap (Algorithm 8).

As another example, due to allocation request coalescing, every thread must now extract its reserved object slot from a set of allocations performed by a leader thread on behalf of the entire warp. This boils down to finding the i-th set bit in a 64-bit bitmap b of newly reserved object slots, where i is the rank of a thread among all allocating threads in the warp (Algorithm 6, Line 20). Instead of checking every bit in b one-by-one (loop with 64 iterations in the worst case) and keeping track of the number of set bits seen so far, we clear the first $i-1$ bits with bitwise AND and then calculate *ffs*(b) (Algorithm 9).

5.4 Concurrency and Correctness

CUDA has a weak consistency model for global memory access (Section 2.1.3). Writes to memory performed by one thread are *not* guaranteed to become visible to other threads in the same order. However, atomic writes *have* that property (*sequential consistency*). Furthermore, *thread fences* can be used between two memory writes to enforce sequential consistency, if necessary.

Moreover, global memory reads/writes may be buffered in registers/caches, without a global memory load/store. Thus, memory writes by one thread may not become visible to other threads until the next GPU kernel, unless reads/writes are `volatile` or performed with atomic operations.

All bitmap operations are sequentially consistent and do not suffer from load/store buffering because they are based on atomic memory operations.

5.4.1 Object Slot Reservation/Freeing

Inside a block, object allocations are tracked with an object allocation bitmap. Every object allocation bitmap has 64 bits, regardless of the block capacity. If a block's capacity is smaller than 64, then the last $64-N$ bits are set to 1 during block initialization to prevent threads from reserving these slots during object allocation.

Object slots are reserved/freed with atomic operations. These bypass the incoherent L1 caches and are thread-safe: E.g., based on their return value, we know if the current thread reserved a selected slot or if another contending thread was faster (Algorithm 7, Line 5). Based on their return value, we also know if the current thread reserved the last slot (Line 10), in which case the block should be marked as inactive by the allocation algorithm.

Slot Reservation `Block::reserve()` (Algorithm 7) reserves a single object slot in the block. Our actual implementation (`Block::reserve_multiple(n)`; not discussed here) may reserve multiple slots at once due to allocation request coalescing.

1. **Preconditions:** Block was initialized at least once. (Calling this method on invalidated blocks or full blocks is OK. This function will simply return FAIL.)
2. **Postconditions:** If the result is different from FAIL, then the resulting slot position is reserved for this thread (and no other thread).
3. **Return Value:** Success indicator, atomically reserved slot position, block state.
4. **Linearization Point:** Atomic OR operation (Line 5).

Slot Freeing `Block::deallocate(pos)` (Algorithm 8) frees a single object slot in the block. To support allocation request coalescing, we have a modified version of this algorithm that can rollback multiple slots at once (not discussed here).

1. **Preconditions:** Bit `pos` is set to 1 in the object allocation bitmap. (Deleting an object multiple times or trying to delete an arbitrary pointer is illegal.)
2. **Postconditions:** Bit `pos` is set to 0 in the object allocation bitmap.
3. **Return Value:** Block state.
4. **Linearization Point:** Atomic AND operation (Line 2).

5.4.2 Safe Memory Reclamation with Block Invalidation

Safe memory reclamation (SMR) in lock-free algorithms is notoriously difficult. An SMR problem arises in DynaSOAr when deleting blocks. A block should be deleted as soon as its last object has been deleted. This by itself is easy to detect with atomic operations (Algorithm 8, Line 6). However, a contending thread may already have selected the now empty block in the course of its own concurrent allocate operation, before the block is actually deleted. Now it is no longer safe to delete the block, but the deleting thread is not aware of that.

Elaborate techniques for SMR such as hazard pointers [142] and epoch-based reclamation [66] have been proposed in previous work [27, 141]. DynaSOAr is able to exploit a key characteristic of its data structure to solve this SMR problem in a simple way: Since all blocks have the same size and structure, object allocation bitmaps are always located at the same position. Therefore, we can optimistically proceed with bitmap modifications and rollback changes if necessary.

Our solution to SMR is *block invalidation*. Before deleting a block, a thread tries to *invalidate* (atomically set to 1) all bits in the object allocation bitmap. Bits that were already 1 are not considered invalidated because those object slots are in use. After successful invaldation, bits remain invalidated until a new block is initialized at the same location. Other threads may still be able to find the block in the active block bitmap for a while, but slot reservations can no longer succeed.

Allocating threads can detect changes in the block type. Before a previously invalidated block becomes available for allocations again (by initializing its object allocation bitmap), we update the block type. We put a thread fence between both writes to ensure that threads see the new block type before they see free slots in the bitmap (Algorithm 10). Threads allocate objects optimistically and rollback changes should they detect a different block type (Algorithm 1, Line 14; also see Section 5.4.3).

Algorithm 11: DAllocatorHandle::invalidate(int bid) : bool $\boxed{\text{GPU}}$

```
1  bitmap_ptr ← &heap[bid].bitmap;
2  before ← atomicOr(bitmap_ptr, 0xFF...F);      ▷ Invalidate (set) all obj. alloc. bitmap bits.
3  if before ≠ 0xFF...F then                    ▷ ≥ 1 bit was invalidated.
4      t ← heap[bid].type;
5      if before = 0 then                       ▷ All 64 bits invalidated by this atomicOr.
6          return true;
7      else                                     ▷ Not all bits invalidated. Rollback.
8          before_rollback ← atomicAnd(bitmap_ptr, before);
9          if before_rollback ≠ 0xFF...F then   ▷ Other thread cleared a bit.
10             active[t].clear(bid);            ▷ Other thread expects an inactive block.
11         if (before_rollback & before) = 0 then   ▷ Empty again. Retry invalidation.
12             return invalidate(bid);
13 return false;
```

Details Block invalidation[11] (Algorithm 11) fails if a thread is unable to invalidate at least one bit. In that case, if at least one bit was changed through invalidation, this change must be rolled back (Line 8): In `before` exactly those bits are zero that were invalidated by the thread.

While a thread is running an invalidation operation, other threads may continue to concurrently reserve/free object slots in the same block, unaware of the fact that a thread is trying to invalidate the block. Those threads will update block state bitmaps based on the object allocation bitmap state that they are seeing. Therefore, block invalidation must update block bitmaps, as every invalidated bit appears to be an allocated object slot to other threads.

Since block invalidation fills up a block, the block's *active[t]* state should be removed after Line 7, because, if we enter this *else* branch, the thread just *filled up* the block by reserving the remaining object slots (however, not all 64 slots, otherwise, we would be in the *then* branch of Line 5). However, we defer this step, as an invalidation rollback would likely have to mark the same block as *active[t]* again. Unless, another thread concurrently freed an object slot in-between invalidation and invalidation rollback. For such a thread it will seem as if its deallocation just freed the first slot, causing it to activate the block (Algorithm 2, Line 5). However, since we deferred block deactivation, this *set(bid)* operation will spin until we deactivate the block (Algorithm 11, Line 10). If invalidation rollback empties the block again, we try to invalidate the block one more time[12].

Note that block invalidation is independent of the type of a block. After invalidating at least one bit, the block type is fixed until invalidation rollback or block initialization, since other threads do not change invalidated bits. As such, the block cannot be deleted or reinitialized to another type by another thread. Other threads can, however, delete and initialize a block with different type after invalidation rollback. It is, nevertheless, safe to assume a block type of *t* in Line 10, since this is merely an execution of a deferred operation that should have happened earlier when the block type was known to be *t*.

[11]For presentation reasons, we assume a block capacity of 64 in this and other algorithms.
[12]Our actual implementation is iterative instead of recursive.

5.4.3 Object Allocation

The critical parts during allocations (Algorithm 1) are *block selection* (Line 2) and *object slot reservation* (Line 8). Both operations by themselves are atomic, but not together. Block selection returns the index of an active block of type T, so we expect that after Line 8, we reserved an object slot in a block of type T. However, due to concurrent operations of other threads, some of these assumptions may be violated.

Block Full An active block was selected by `try_find_set` but the block filled up before making an allocation (i.e., the block is no longer active). In this case, object slot reservation will fail. Whenever allocation fails, it will restart from the beginning.

Block Deallocated A block was selected by `try_find_set` but deallocated before reserving a slot. In this case, slot reservation will fail because the block is now in an invalidated state.

Block Replaced (ABA) A block was selected by `try_find_set` but deallocated and reinitialized to a block of same type T. This is harmless: We do not care about block identity.

Block Replaced (Different Type) A block was selected by `try_find_set` but deallocated and reinitialized to another type[13] $t \neq T$. In this case, the allocation must be rolled back (Line 14). All blocks have the same basic structure, so no object data can be overwritten accidentally during bitmap updates. Note that the rollback may trigger additional block bitmap updates.

Active Block Not Selected A block becomes active shortly after `try_find_set` fails. Or, due to bitmap hierarchy inconsistencies, `try_find_set` fails to find an active block even though active blocks exist. This is harmless: No assumption is violated. A new block will be initialized, which merely increases fragmentation.

Note that a block cannot be deallocated after an object slot was already reserved, because block invalidation would fail. Thus, the type of a block can also no longer change.

5.4.4 Object Deallocation

The critical part during deallocations (Algorithm 2) is consistency between *object slot deallocation* (Line 3) and *block state updates*. If the current thread deallocated the first object (i.e., the block was full), then the block's bit in the active block bitmap must be set. If the current thread deallocated the last object (i.e., the block is empty), then the block must be deleted. The problem is that object slot deallocation and the corresponding block state update together are not atomic.

Allocate After Delete-First A thread t_1 deleted the first object of a block. However, before marking the block as active (Line 6), another thread t_2 allocated this slot again; the block should be inactive. In this case, t_2 reserved the last slot, so it will mark the block as inactive (Algorithm 1, Line 12). This operation expects the bit to be in a set state and it will retry until t_1 sets the bit.

[13]Block initialization (Algorithm 10) has a thread fence between setting the block type and resetting the object allocation bitmap, so threads are guaranteed to read the correct type t after an allocation succeeded.

Block Deleted after Delete-First A thread t_1 deleted the first object of a block. However, before marking the block as active, other threads deallocated all other objects and a thread t_2 deleted the block. This is not possible because t_2 expects the block to be active (Line 9), i.e., bit set to 1, and blocks until then.

Block Replaced after Delete-First A thread t_1 deleted the first object of a block. However, before marking the block as active, the block was reinitialized to another type. This is not possible because only deleted blocks can be reinitialized (see previous point).

Allocate after Delete-Last A thread t_1 deleted the last object of a block. However, before deleting the block, another thread t_2 allocated an object again, so it is unsafe to delete the block now. This case in handled by block invalidation.

Block Deleted after Delete-Last A thread t_1 deleted the last object of a block. However, before deleting the block, another thread t_2 allocated an object and yet another thread t_3 deleted that object, rendering the block empty again and deleting it. Now the block is already deleted when t_1 is trying to delete the block. In this case, block invalidation of t_1 will fail because the block is still in an invalidated state and t_1 fails to invalidate all object slot bits.

Block Replaced after Delete-Last Same as before, but yet another thread t_4 reininitializes the block to a different type. Now t_1 will invalidate and delete a new block whose type is different. This is OK. Block invalidation will succeed only if the block is empty. Both block invalidation and block deletion are independent of and do not assume a certain block type.

Divergent Branch Scheduling The ordering of *if* statement branches in Algorithm 2 is crucial. Let us assume that a thread t_1 deletes the first object of a full block (Line 4) and another thread t_2 of the *same warp* deletes the last object of the same block and succeeds with block invalidation (Line 7). The *clear* operation in Line 9 will block until the respective bit was actually changed to 0. Therefore, to avoid a deadlock, it is crucial that the *set* operation of Line 5 is scheduled to execute before the *clear* operation. Since both t_1 and t_2 belong to the same warp, both *if* branches are executed sequentially (*thread divergence*). Only if the CUDA compiler schedules the branch of Line 4 before the branch of Line 6 in the compiled PTX, is this implementation free of deadlocks. While CUDA leaves the order of execution of divergent branches undefined (Section 2.1.3), to the best of our knowledge, all recent CUDA versions schedule branches in the order in which they appear in the source code.

Algorithm 12 is an alternative implementation of Algorithm 2 that does not depend on the ordering of *if* branches. Block bitmaps are not changed immediately. Instead, we maintain helper variables that indicate if and how a bit should be changed: -1 indicates *clear*, 0 indicates *no change*, 1 indicates *set*. At first, all participating threads in a warp determine the necessary changes, potentially causing thread divergence between Line 5 and Line 13. However, after Line 13, all threads converge again and modify their respective bits with *try_* bitmap operations. These operations return immediately, regardless of their success. Only if a thread performed all necessary bitmap changes can a thread exit the loop.

Note that block invalidation (Line 9) also modifies block bitmaps. Therefore, block invalidation must be incorporated into Algorithm 12 and changed in a similar way. Only for presentation reasons, we keep block invalidation in a separate algorithm.

Algorithm 12: DAllocatorHandle::deallocate<T>(T* ptr) : void $\boxed{\text{GPU}}$

```
bid ← extract_block(ptr);
slot ← extract_slot(ptr);
state ← heap[bid].deallocate(slot);
active_op ← allocated_op ← free_op ← 0;
if state = FIRST then
    t ← T;
    active_op ← 1;
else if state = EMPTY then
    if invalidate(bid) then        ▷ invalidate() must be inlined and changed similarly.
        t ← heap[bid].type;
        active_op ← -1;
        allocated_op ← -1;
        free_op ← 1;
while active_op + allocated_op + free_op ≠ 0 do
    if active_op = 1 then
        if active[t].try_set(bid) then active_op ← 0 ;
    if active_op = -1 then
        if active[t].try_clear(bid) then active_op ← 0 ;
    if allocated_op = 1 then
        if allocated[t].try_set(bid) then allocated_op ← 0 ;
    if allocated_op = -1 then
        if allocated[t].try_clear(bid) then allocated_op ← 0 ;
    if free_op = 1 then
        if free[t].try_set(bid) then free_op ← 0 ;
    if free_op = -1 then
        if free[t].try_clear(bid) then free_op ← 0 ;
```

5.4.5 Correctness of Hierarchical Bitmap Operations

A container C_i^l consists of bits $b_{64\cdot i}^l$, ..., $b_{64\cdot i+63}^l$ and is represented by one bit b_i^{l+1} in the nested (higher-level) bitmap. That bit is set if and only if at least one bit is set in the container.

Definition 1 (Consistency). *A bit in level b_i^{l+1} is **consistent** with its corresponding container C_i^l in the lower-level bitmap if and only if:*

$$b_i^{l+1} = \bigvee_{k=0}^{63} b_{64\cdot i+k}^l = \mathbb{1}\left(\sum C_{\lfloor i/64 \rfloor}^l > 0\right)$$

In Definition 1, $\mathbb{1}$ is the indicator function and $\sum C_i^l$ is the number of set bits in C_i^l, as computed by popc (population count) in the algorithms.

We say that the L_{l+1} bitmap is in a consistent state with the L_l bitmap if all bits b_i^{l+1} in the L_{l+1} bitmap satisfy the consistency criterion. The bitmap data structure as a whole is in a consistent state if all bitmap levels L_i satisfy the consistency criterion.

We do not make any consistency guarantees during the execution of a GPU program. Instead, we guarantee eventual consistency: If the bitmap data structure is in a consistent state before a GPU kernel launch, it is guaranteed to be in a consistent state after the GPU kernel finished running.

Definition 2 (Semantics of Bitmap Operations)**.** *Every bitmap L_l provides operations for setting and clearing bits (Section 5.3.1). These operations may update bits in the higher-level bitmap L_{l+1} if they **set** the **first** bit ($SF^l_{\lfloor i/64 \rfloor}$) or **clear** the **last** bit ($CL^l_{\lfloor i/64 \rfloor}$) of a container C^l_i, respectively:*

$$\underbrace{\mathit{set}(b^l_i) \text{ and } \mathbb{1}\left(\sum C^l_{\lfloor i/64 \rfloor} = 0\right)}_{\textit{set-first: } SF^l_{\lfloor i/64 \rfloor}} \text{ then } \mathit{set}(b^{l+1}_{\lfloor i/64 \rfloor}) \qquad \forall i \in [0; \mathit{num\_bits})$$

$$\underbrace{\mathit{clear}(b^l_i) \text{ and } \mathbb{1}\left(\sum C^l_{\lfloor i/64 \rfloor} = 1\right)}_{\textit{clear-last: } CL^l_{\lfloor i/64 \rfloor}} \text{ then } \mathit{clear}(b^{l+1}_{\lfloor i/64 \rfloor}) \qquad \forall i \in [0; \mathit{num\_bits})$$

We would like to show that, assuming that a bitmap data structure is initially in a consistent state and given a multiset of bitmap operations O_0 on the L_0 bitmap, the entire bitmap data structure is in a consistent state after executing all operations. In contrast to other concurrent data structures [56], we do not guarantee consistency during runtime.

Definition 3 (Legal Bitmap Operations)**.** *Let $\#\mathit{set}(b^l_i)$ and $\#\mathit{clear}(b^l_i)$ be the number of set and clear operations of b^l_i in a multiset of bitmap operations O_l. We call $\mathbb{S}(b^l_i) = \#\mathit{set}(b^l_i) - \#\mathit{clear}(b^l_i)$ the **set-surplus** of b^l_i. O_l is **legal** if it satifies the following conditions for every bit b^l_i in L_l.*

1. *Overall bit operation is clear, remain or set: $\mathbb{S}(b^l_i) \in \{-1, 0, 1\}$.*
2. *Bit is in a cleared or set state afterwards: $b^l_i + \mathbb{S}(b^l_i) \in \{0, 1\}$ (denoted by b'^l_i).*

Intuitively, an already cleared bit cannot be cleared again and an already set bit cannot be set again[14]. For example:

- $C^l_0 = \texttt{01001...}$, $O_l = \{\mathsf{set}(b^l_2), \mathsf{set}(b^l_2), \mathsf{clear}(b^l_2)\}$. $\mathbb{S}(b^l_2) = 2 - 1 = 1$. $b'^l_i = 0 + 1 = 1$. Therefore, O_l is legal.
- $C^l_0 = \texttt{01001...}$, $O_l = \{\mathsf{set}(b^l_2), \mathsf{set}(b^l_2), \mathsf{clear}(b^l_2), \mathsf{set}(b^l_2)\}$. O_l is illegal because $\mathbb{S}(b^l_2) = 3 - 1 = 2$.

Note that illegal bitmap operations deadlock in our implementation because set and clear spin-block and retry until they acutally changed the bit. If a legal bitmap operations multiset is executed fully concurrent (i.e., one thread per operation), then there is always a thread/operation that can make progress.

Induction Hypothesis 1. *Let us assume that a multiset of bitmap operations O_l on the L_l bitmap is legal according to Definition 3 for an arbitrary l and that L_l is initially consistent with L_{l+1}.*

Lemma 1. *Under the induction hypothesis, the bitmap operations multiset O_{l+1} that is generated by the operations in O_l according to Definition 2 is also legal. Furthermore, after executing O_l, L_l is still consistent with L_{l+1}.*

Proof. Let us first consider the bitmap operations of a single container C^l_i. Let $\#SF^l_i$ be the number of times a first bit is set in the container and $\#CL^l_i$ be the number of

[14]try_clear and try_set allow clearing/setting already cleared/set bits.

times a last bit is cleared in the container. Then, according to Definition 2, $b^{l+1}_{\lfloor i/64 \rfloor}$ is set $\#SF^l_i$ times and cleared $\#CL^l_i$ times. We have to prove that the set-surplus $\mathbb{S}(b^{l+1}_{\lfloor i/64 \rfloor}) = \#SF^l_i - \#CL^l_i$ satisfies the legality criteria of Definition 3.

Without loss of generality, let us assume that all set-first and clear-last operate on the same bit b^l_k. Then, $\mathbb{S}(b^{l+1}_{\lfloor i/64 \rfloor}) = \mathbb{S}(b^l_k) \in \{-1, 0, 1\}$. Hence, the generated bitmap operations O_{l+1} for any bit on the L_{l+1} bitmap satisfy the first legality condition of Definition 3.

Now we have to show that also the second legality condition holds and that $b^{l+1}_{\lfloor i/64 \rfloor}$ is consistent with C^l_i after executing O_l. We consider two cases.

1. The higher-level bit is initially in a cleared state: $b^{l+1}_{\lfloor i/64 \rfloor} = 0$. Therefore, due to initial consistency, $\sum C^l_{\lfloor i/64 \rfloor} = 0$. Therefore, $\#SF^l_i - \#CL^l_i \in \{0, 1\}$, otherwise, O_l would not be legal. Therefore, $b^{l+1}_{\lfloor i/64 \rfloor} + \mathbb{S}(b^{l+1}_{\lfloor i/64 \rfloor}) \in \{0, 1\}$.
   (a) If $\#SF^l_i - \#CL^l_i = 0$, then $\vee^{63}_{k=0} b^l_{64 \cdot i + k} = 0$ after O_l. At the same time, $\mathbb{S}(b^{l+1}_{\lfloor i/64 \rfloor}) = 0$, so $b^{l+1}_{\lfloor i/64 \rfloor} = 0$ after O_l, which is consistent with the state of C^l_i after O_l.
   (b) If $\#SF^l_i - \#CL^l_i = 1$, then $\vee^{63}_{k=0} b^l_{64 \cdot i + k} = 1$ after O_l. At the same time, $\mathbb{S}(b^{l+1}_{\lfloor i/64 \rfloor}) = 1$, so $b^{l+1}_{\lfloor i/64 \rfloor} = 1$ after O_l, which is consistent with the state of C^l_i after O_l.
2. The higher-level bit is initially in a set state: $b^{l+1}_{\lfloor i/64 \rfloor} = 1$. Therefore, due to initial consistency, $\sum C^l_{\lfloor i/64 \rfloor} > 0$. Therefore, $\#SF^l_i - \#CL^l_i \in \{-1, 0\}$, otherwise, O_l would not be legal. Therefore, $b^{l+1}_{\lfloor i/64 \rfloor} + \mathbb{S}(b^{l+1}_{\lfloor i/64 \rfloor}) \in \{0, 1\}$.
   (a) If $\#SF^l_i - \#CL^l_i = -1$, then $\vee^{63}_{k=0} b^l_{64 \cdot i + k} = 0$ after O_l. At the same time, $\mathbb{S}(b^{l+1}_{\lfloor i/64 \rfloor}) = -1$, so $b^{l+1}_{\lfloor i/64 \rfloor} = 0$ after O_l, which is consistent with the state of C^l_i after O_l.
   (b) If $\#SF^l_i - \#CL^l_i = 0$, then $\vee^{63}_{k=0} b^l_{64 \cdot i + k} = 1$ after O_l. At the same time, $\mathbb{S}(b^{l+1}_{\lfloor i/64 \rfloor}) = 0$, so $b^{l+1}_{\lfloor i/64 \rfloor} = 1$ after O_l, which is consistent with the state of C^l_i after O_l.

If all containers in L_l are consistent with their respective bits in L_{l+1}, then the entire L_l bitmap is consistent with the L_{l+1} bitmap. Futhermore, all generated bitmap operations O_{l+1} are legal because they satisfy both legality criteria. □

Base Case 1. *The bitmap data structure is initially in a consistent state. Furthermore, O_0 is legal. Otherwise, programmers use the bitmap data structure incorrectly.*

5.5 Related Work

CUDA provides an on-device dynamic memory allocator, but it is unoptimized and slow. To solve this issue, multiple custom allocators have been developed in the last years (Table 5.1). These allocators achieve good performance by exploiting an allocation pattern that many applications on massively parallel SIMD architectures exhibit: Most allocations are small in size and due to mostly regular control flow, many allocations have the same byte size.

Halloc [7] is one of these allocators. It is a slab allocator and can allocate only a few dozen predetermined byte sizes between 16 bytes and 3 KB. This is fast but can lead to internal fragmentation. DYNASOAR can avoid such internal fragmentation because allocation sizes are determined from compile-time type information of the application. A slab in Halloc contains same-size allocations and tracks allocations with a bitmap. To avoid scanning large bitmaps, a hash function determines which bits to check during allocations. Only one slab can be active per allocation size and if the active slab becomes too full, it is replaced with a new one. In contrast, more than one block per type can be active in DYNASOAR and blocks are filled up entirely.

XMalloc [88] was the first allocator with allocation request coalescing, which was adopted by many other allocators, including DYNASOAR. Coalesced requests are served from *basicblocks*, which are organized in one of multiple lock-free free lists depending on their size.

FDGMalloc maintains a private heap for every warp [201], similar to Hoard [19]. It does not have a general *free* operation and can only deallocate entire heaps, so it is not suitable for SMMO applications.

CircularMalloc (CMalloc) [191] allocates memory in a ring buffer. Every allocation has a pointer to the next allocation or free chunk, wrapping around at the end of the buffer. CMalloc traverses the linked list for free chunks during allocations. To reduce allocation contention, every multiprocessor starts its traversal at a different location. This is similar to DYNASOAR's *bitmap rotation* technique for reducing thread contention.

ScatterAlloc [175] hashes allocation requests to memory *pages* depending on their allocation size and the multiprocessor ID. Pages hold allocations of the same size, but slightly smaller requests can be accommodated, leading to internal fragmentation. While DYNASOAR uses hierarchical bitmaps, ScatterAlloc uses hashing with linear probing for finding pages during allocations. For benchmarks, we use mallocMC [54], a reimplementation of ScatterAlloc that is still maintained.

Both Halloc and ScatterAlloc maintain fill levels to quickly skip congested memory areas that are above a certain threshold, because the performance of any hashing technique degrades with an increasing number of collisions. In DYNASOAR, temporary inconsistencies in bitmap hierarchies increase with the number of concurrent allocations, but DYNASOAR can dynamically adapt to such cases by initializing additional blocks.

TABLE 5.1: Comparison of allocators. *Coal.* means *allocation request coalescing*.

Allocator	Coal.	SOA	Container	Finding Free Memory
DYNASOAR	✓	✓	Block	Hierarchical Bitmap
DYNASOAR-NoCoal	✗	✓	Block	Hierarchical Bitmap
BitmapAlloc	✗	✗	✗	Hierarchical Bitmap
CircularMalloc	✗	✗	✗	Linked List, Ring Buffer
Default CUDA Allocator	✗	✗	(Unknown)	(Unknown)
FDGMalloc	✓	✗	Priv. Heap, Superblock	Linked List
Halloc	✗	✗	Slab	Bitmap, Hashing
mallocMC (ScatterAlloc)	✓	✗	Superblock, Region, Page	Hashing
XMalloc	✓	✗	(4 block hierarchies)	Lock-free Free Lists

5.6 Benchmarks

We evaluated DynaSOAr with multiple real-world SMMO applications that exhibit different memory allocation patterns (Table 5.2). We describe the SMMO structure of these applications in detail in Chapter 7. All benchmarks were run on a computer with an Intel Core i7-5960X CPU, 32 GB main memory and an NVIDIA TITAN Xp GPU (12 GB device memory), and compiled with nvcc (-O3) from the CUDA Toolkit 9.1 on Ubuntu 16.04.4.

We compare the running time with different allocators. If possible, we also measured the running time of *baseline* implementations that do not use any dynamic memory management.

Benchmark Applications Our benchmarks are from different domains and fall into four categories.

1. Objects allocated up front, no deallocation: nbody
2. Objects allocated up front, then only deallocation: collision, structure
3. Cellular automaton (CA) with static cells network: sugarscape, traffic, wa-tor
4. Other: barnes-hut, game-of-life

Baselines (SOA/AOS) are application variants without any dynamic memory allocation. Baselines of category (1) are trivial to implement with static allocation. In category (2) applications, every object has a boolean `active` flag to prevent deleted objects from being enumerated in the future. In category (3) applications, classes are merged with the underlying static cell data structure, which wastes memory in case of empty cells (Section 5.6.2). Category (4) applications cannot be implemented with only static allocation, unless the application is changed fundamentally.

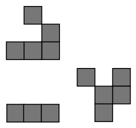
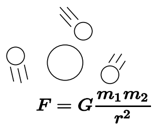
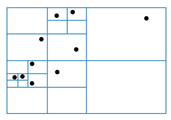
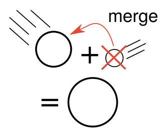
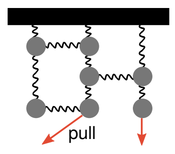
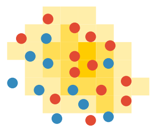
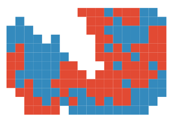
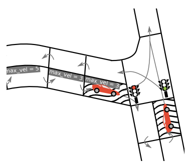
Parallel Do-All in Custom Allocators Other allocators do not provide do-all operations, which are required for SMMO applications. To compare DynaSOAr with other allocators, we developed standalone `parallel_do` and `device_do` implementations that can be used with any allocator.

These implementations maintain arrays for allocated and deleted object pointers of each type. Pointers are added to these arrays with atomic operations. At the end of a parallel do-all operation, deleted pointers are removed from the array of allocated pointers. This process is non-trivial because the same memory location/pointer could be allocated and deleted multiple times throughout a parallel do-all operation. After all deleted pointers were removed, the array of allocated pointers is compacted with a prefix sum operation (same as Figure 5.6).

Depending on the number of (de)allocations, this mechanism may take a long time. A better allocator-specific mechanism could likely be developed with some reverse engineering. For that reason, we break down running times into *enumeration* time and remaining time. Enumeration time should not be taken into account when comparing the performance of different allocators.

BitmapAlloc To analyze the performance of pure bitmap-based object allocation without SOA layout, blocks and fake object pointers, we developed a second allocator *BitmapAlloc*. This allocator treats the entire heap as one large object array, whose slots are managed by hierarchical bitmaps, similarly to DynaSOAr: one *allocation bitmap*

TABLE 5.2: Description of SMMO benchmark applications

	Benchmark Description	**#par. do-all**	**#classes**	**alloc/dealloc.**	**smallest class**	**largest class**
	**Game of Life:** A CA due to J. H. Conway. This version has a time complexity of O(#alive cells) instead of the standard O(#cells) algorithm. Cells can be dead, alive or alive-candidates. Alive-candidates are dead cells that may become alive in the next iteration. Only alive-candidates and alive cells are processed.	4 / iteration	4 (2 dyn.)	✓ / ✓	5B, 2 fields	8B, 1 field
	**N-Body:** Simulates the movement of particles according to gravitational forces. A `device_do` operation is required to calculate (and then sum up) the gravitational force between every pair of particles. This benchmark has no dynamic object (de)allocation.	2 / iteration	1 (0 dyn.)	✗ / ✗	28B, 7 fields	(same)
	**Barnes-Hut:** An extension of N-Body in which bodies are stored in a quad tree [28], to evaluate DYNASOAR with dynamic tree data structures. The running time is dominated by the construction and maintenance (i.e., frequent node inserts and removals) of the quad tree. Tree nodes are dynamically (de)allocated.	10 / iteration	3 (1 dyn.)	✓ / ✓	68B, 9 fields	102B, 12 fields
	**Particle Collisions:** Similar to N-Body, but particles are merged according to perfectly inelastic collision when they are getting too close. The number of particles decreases gradually. This benchmark has dynamic object deallocation but no dynamic object allocation.	6 / iteration	1 (1 dyn.)	✗ / ✓	38B, 10 fields	(same)
	**Structure:** Simulates a fracture in a composite material, modeled as an FEM. Intuitively, the mesh is a graph and edges between nodes are springs. When pulling the mesh on one side, the material starts to break eventually. Isolated nodes are detected with a BFS [78] and removed. Literature describes extensions that would benefit from dynamic allocation [125].	3 / iteration	5 (4 dyn.)	✗ / ✓	32B, 6 fields	46B, 7 fields
	**Sugarscape:** An agent-based social simulation [59]. Agents inhabit a 2D grid and can move to neighboring cells. Cells contain sugar which is consumed by agents. Sugarscape can simulate a variety social dynamics (e.g., trade, war, environmental pollution). We simulate resource consumption, ageing and mating.	12 / iteration	4 (2 dyn.)	✓ / ✓	52B, 7 fields	74B, 11 fields
	**Wa-Tor:** An agent-based predator-prey simulation [49]. Fish/sharks occupy a 2D grid of cells and can move to neighboring cells. Fish and sharks reproduce after some iterations. Fish die when they are eaten and sharks die when they run out of food.	8 / iteration	4 (2 dyn.)	✓ / ✓	60B, 4 fields	64B, 5 fields
	**Nagel-Schreckenberg:** A traffic flow simulation on a street network [145] that can reproduce traffic jams and other phenomena. Streets are modeled as a network of cells, with at most one vehicle per cell. New vehicles are continuously added to the simulation and existing vehicles are removed at their final destination.	3 / iteration	4 (1 dyn.)	✓ / ✓	97B, 10 fields	124B, 6 fields
	Linux Scalability: Not an SMMO application. This microbenchmark allocates, then deallocates a fixed number of same-size objects in each thread, without accessing the memory [118].	n/a	1 (1 dyn.)	✓ / ✓	4B, 1 field	(same)

per type and one *free slot bitmap*. Allocation bitmaps are also used in `parallel_do` and `device_do` implementations.

The main downside of BitmapAlloc is its inefficient memory usage. It supports only a single allocation size, potentially leading to high internal fragmentation.

5.6.1 Performance Overview

Figure 5.8 shows the running time of all benchmarked SMMO applications. We gave each allocator some extra memory to avoid memory scarcity slowdowns: The heap size is 8 GiB, at least 4 times bigger than the maximum amount of all allocated memory at any point throughout the program execution. DynaSOAr achieves superior performance over other allocators due to the SOA layout, a dense object allocation policy and an efficient parallel do-all operation.

All applications except for structure see a speedup by switching from AOS to SOA (compare baselines). In structure, most fields are used together, so SOA does not pay off for this benchmark.

Despite having no dynamic (de)allocation during the benchmark, nbody can see a slight speedup with dynamic memory allocation. This is likely due to fewer cache associativity collisions compared to a denser allocation within an array [116].

In collision, DynaSOAr/BitmapAlloc enumerate objects with a bitmap scan (`device_do`; 1 bit/object). This is more effcient than in other allocators, which must read object pointers from an array (8 bytes/object). The baseline versions must read an `active` flag (1 byte/object) from every object, including deleted ones.

game-of-life and wa-tor are applications that (de)allocate a large number of objects, so enumeration time dominates the running time with custom allocators. DynaSOAr and BitmapAlloc have much more efficient parallel do-all operations than other allocators.

sugarscape and wa-tor have a 2D grid structure of cells. Baseline versions take advantage of this geometric structure, leading to more coalesced memory accesses. In contrast, programmers have no control over where dynamic allocators place objects in memory. For this reason, the baseline versions are faster than the versions with dynamic memory management.

In general, in applications with dynamic memory management, objects are always referred to with 64-bit object identifiers/pointers, while all baseline versions use 32-bit integer indices. This especially penalizes benchmarks with small objects; they grow considerably just by switching from 32-bit integers indices to 64-bit pointers.

5.6.2 Space Efficiency

To evaluate how efficiently allocators manage memory, we gave them the same heap size and experimentally determined the maximum problem size before running out of memory (Figure 5.9).

For category (1) and category (2) applications that allocate all memory during startup (collision, nbody, structure), the baseline versions are more space-efficient. The exact number of objects per type is known ahead of time, so placing objects in memory is trivial. However, even though category (2) applications delete objects throughout their runtime, the memory consumption of the baseline versions does not decrease over time. This is a problem even for DynaSOAr because blocks can only be deleted when they are entirely empty, which can take some time. This problem can be solved with memory defragmentation (Chapter 6).

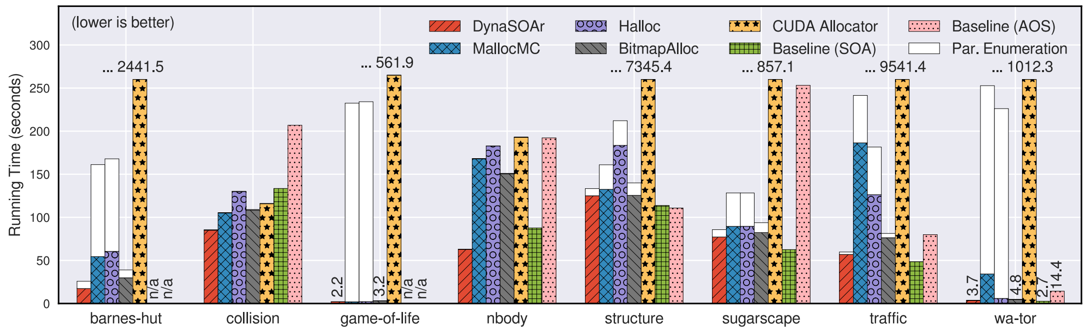


FIGURE 5.8: Running time of SMMO application benchmarks

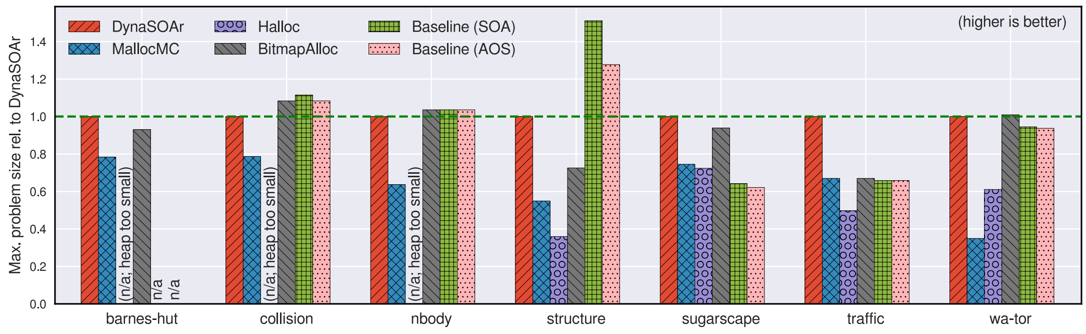
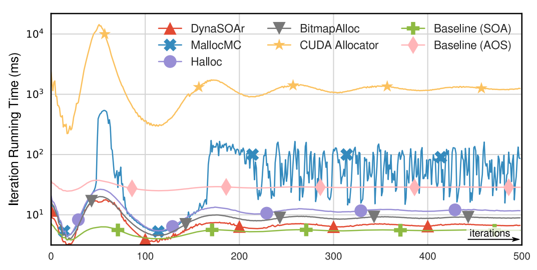
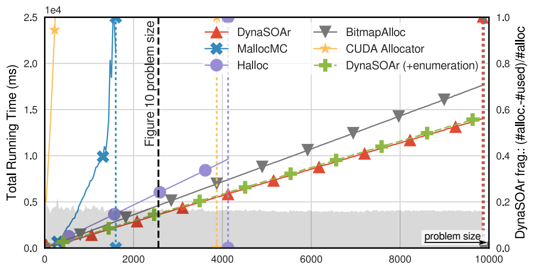


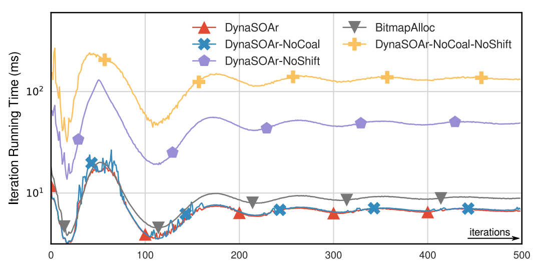
FIGURE 5.9: Space efficiency of SMMO application benchmarks (without object enumeration arrays, relative to DYNASOAR



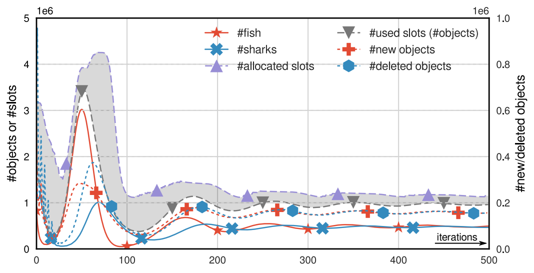
(A) Comparison with other allocators



(B) Fixed heap size, increasing problem size



(C) Isolating single DYNASOAR optimizations



(D) Number of (de)allocations and fragmentation

FIGURE 5.10: Detailed analysis of wa-tor (without enumeration time, unless indicated)

Category (3) applications (sugarscape, traffic, wa-tor) exhibit a fixed grid/network structure of cells, upon which a dynamic set of agents is moving. The baseline versions allocate the fields of agents directly inside cells. Classes for agents are combined with the respective cell class and some fields have `null` values (or garbage) if they are not used. This wastes memory because not all cells are occupied by agents all the time. In those applications, DynaSOAr is not as fast as optimized SOA baseline implementations, but it can handle significantly larger problem sizes.

Out of all dynamic memory allocators, DynaSOAr is most space-efficient. MallocMC and Halloc are based on a hashing approach. With rising heap fill levels, it becomes increasingly difficult to find free memory for allocations, so they fail to use the entire heap memory. DynaSOAr and BitmapAlloc can avoid this problem with bitmaps, which act as an index for free memory.

Albeit negligible in these benchmarks, DynaSOAr and Baseline (SOA) also benefit from slightly smaller object sizes: Only SOA arrays must be aligned/padded and not every single object.

5.6.3 Detailed Analysis of wa-tor

wa-tor (Section 7.6) is a particularly interesting benchmark. It exhibits a massive number of (de)allocations in waves, until an equilibrium between fish and sharks is reached. This allows us to measure the performance at a massive and at a lower number of concurrent (de)allocations. For a fair comparison of allocators, we do not include the time spent on enumeration in this section, unless indicated.

Figure 5.10a shows that DynaSOAr always provides superior performance compared to other allocators; during (de)allocation spikes (around iteration 50), as well as if fewer concurrent (de)allocations take place. The performance of mallocMC degrades after a few iterations and does not recover, possibly due to a highly fragmented heap.

In Figure 5.10b, all allocators were given a heap size of 1 GB and the problem size increases gradually on the x-axis. mallocMC performs well at first, but its performance drops rapidly as soon as the heap starts filling up. DynaSOAr can handle much larger problem sizes, given the same amount of heap memory. The running time grows linearly with the problem size, showing that recent GPU architectures can handle atomic operations quite well.

Fragmentation in DynaSOAr is different from other allocators: DynaSOAr does not have internal or external fragmentation by design, but memory within allocated blocks is only available for a certain type. This sort of fragmentation decreases with better clustering. In DynaSOAr, fragmentation F is the relative number of unused objects slots among all allocated blocks *Blocks* (gray area in Figure 5.10b, d).

$$F = \frac{\sum_{b \in Blocks} (N_{type(b)} - used(b))}{\sum_{b \in Blocks} N_{type(b)}} \approx \frac{1}{\#\text{blocks}} \sum_{b \in Blocks} \frac{\#\text{free slots}(b)}{\#\text{slots}(b)} \qquad (\textit{fragmentation})$$

At iterations 60–80 in Figure 5.10d, DynaSOAr has high fragmentation because many fish objects were deallocated. However, a block can only be deallocated when *all* of its objects were deallocated. The fragmentation level decreases gradually because more fish/shark objects are deallocated over time and new allocations are performed in existing (active) blocks. Therefore, new blocks are rarely allocated and there is a chance that an active block will eventually run empty. As can be seen in Figure 5.10b,

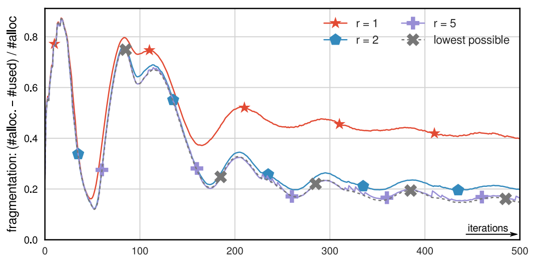


FIGURE 5.11: Memory fragmentation of wa-tor by the number of active block lookup attemps r (Algorithm 1, Line 2). The x-axis denotes iterations and the y-axis denotes the fragmentation rate. With only 1 retry ($r = 2$), fragmentation is reduced by 50%.

fragmentation is independent of the problem size and constant at around 18% (gray area) after 500 Wa-Tor iterations.

We implemented multiple DYNASOAR variants to pinpoint the source of DYNASOAR's speedup over other allocators (Figure 5.10C). The most important optimization is the rotation-shifting of bitmaps. Without shifting (*-NoShift variants), performance degrades severely due to high thread contention, because (1) all threads are trying to allocate objects in the same few active blocks and (2) all threads compete for the same few free block locations instead of choosing free block locations in different parts of the heap. Allocation request coalescing is another optimization that reduces thread contention significantly (compare DynaSOAr-NoCoal-NoShift and DynaSOAr-NoShift), but it cannot improve performance much further if we are already rotation-shifting bitmaps (compare DynaSOAr and DynaSOAr-NoCoal).

In Figure 5.11, we experiment with the number of active block lookup attempts before entering the slow path, which strongly affects fragmentation. By default, DYNASOAR attempts to locate an active block five times ($r = 5$) before initializing a new active block. This is close to the lowest achievable fragmentation level (i.e., without any thread contention). Due to unfortunate allocate-deallocate patterns, a fragmentation rate of 0% is not achievable without manually relocating objects through memory defragmentation or predicting future (de)allocations.

5.6.4 Raw Allocation Performance

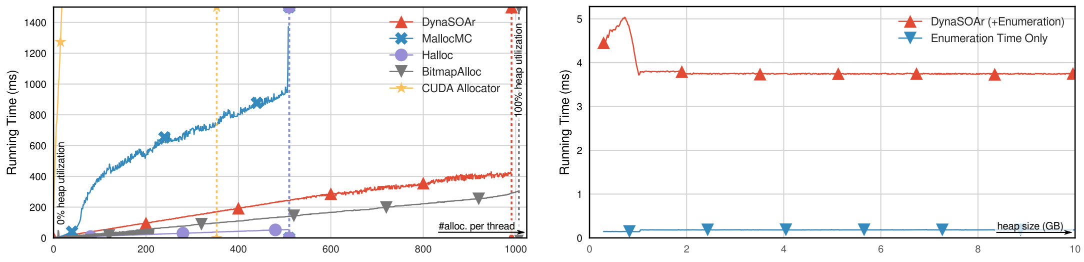
The *Linux Scalability* microbenchmark [118] measures the raw (de)allocation time of memory allocators. We set the heap size to 1 GiB and one CUDA kernel allocates n 64-byte objects in each of the 16,384 threads. A second CUDA kernel deallocates all objects. In Figure 5.12A, the x-axis denotes the number of allocations n per thread and the y-axis shows the total benchmark running time divided by n.

We chose the size of the heap such that it can hold exactly $16384 \times n$ objects with $n = 1024$ (100% heap utilization). No allocator can reach perfect utilization because some memory is used for internal data structures such as bitmaps.

Halloc is the fastest allocator. Both Halloc and mallocMC fail to allocate more than 510 objects (49.8% utilization). This is better than in some other benchmarks, probably because only objects of one size are allocated. DYNASOAR (96.9% utilization), BitmapAlloc (98.4% utilization) and Halloc scale almost perfectly with the number of allocations.

5.6.5 Parallel Object Enumeration

The overhead of object enumeration in DYNASOAR is negible in most benchmarks (Figure 5.8, Figure 5.10B). In Figure 5.12B, the problem size is fixed but the heap size increases on the x-axis. DYNASOAR's performance (and that of object enumeration) is independent of the size of the heap, if enough memory is available for the application. This shows that our hierarchical bitmaps work well with various heap sizes.



(A) Linux Scalability: Increasing #allocations (B) Scaling study: Heap size (wa-tor)

FIGURE 5.12: Scaling study: Number of allocations and heap size

5.7 Conclusion

We presented DYNASOAR, a new dynamic object allocator for SIMD architectures. The main insight of our work is that memory allocators should not only aim for good raw (de)allocation performance, but also optimize the usage of allocated memory. DYNASOAR was designed for GPUs, but its basic ideas are applicable to other architectures and systems with good or guaranteed vectorization such as the Intel SPMD compiler (ispc) [152].

DYNASOAR achieves good memory access performance by controlling (a) memory allocation and (b) memory access with a parallel do-all operation. DYNASOAR's main speedup over other allocators is due to an SOA-style object layout, which can benefit memory bandwidth utilization (through coalesced memory access) and cache utilization. To allow for dynamic (de)allocation of objects, DYNASOAR allocates objects in blocks instead of a plain SOA layout. DYNASOAR utilizes hierarchical bitmaps for fast and compact allocations with low fragmentation.

Our benchmarks show that DYNASOAR can achieve significant speedups over state-of-the-art allocators of more than 3x in SMMO application code due to better memory access performance. DYNASOAR also has a significantly lower memory footprint than other allocators, mainly because by design DYNASOAR has no internal fragmentation and is not based on hashing. Our work also shows how an SOA layout can support class inheritance without wasting memory: by allocating objects in blocks and encoding block sizes in object pointers.

Chapter 6

GPU Memory Defragmentation

Memory fragmentation is a challenging problem of dynamic memory allocators and has been widely studied on single-core and multi-core CPU systems (MIMD architectures). On such systems, dynamic memory allocators can achieve low memory fragmentation with good allocation policies [96] and compacting garbage collectors.

However, memory fragmentation has not been studied thoroughly on SIMD architectures, including GPUs. In this chapter, we present the design and implementation of CompactGpu, a memory defragmentation system for the DynaSOAr dynamic GPU memory allocator.

Contents

Outline This chapter is organized as follows. Section 6.1 describes the effect of memory fragmentation on GPUs. Section 6.2 gives a high-level overview of the DYNASOAR memory allocator (see Chapter 5 for details). Section 6.3 describes the design and implementation of COMPACTGPU. Section 6.4 discusses alternative designs that utilize CUDA shared memory. Section 6.5 evaluates the defragmentation quality and performance impact of COMPACTGPU with real and synthetic benchmarks. Finally, Sections 6.6 and 6.7 explore related work and conclude this chapter.

Overview Despite the recent popularity of massively parallel single-instruction multiple-data (SIMD) architectures, the memory fragmentation problem has not been studied thoroughly on such architectures. This is because dynamic memory allocators for SIMD architectures such as GPUs have just been developed recently [88, 175, 71] and not been around long enough yet.

We need to study memory (de)fragmentation on massively parallel SIMD architectures because allocations follow different patterns on such architectures. Most allocations are small in size[1] and due to mostly regular control flow, many allocations have the same byte size. Such patterns are reflected in the design of state-of-the-art GPU allocators. For example, Halloc [7], one of the fastest GPU allocators can allocate only a few dozen predetermined byte sizes between 16 bytes and 3 KB. Such specialties must be exploited by memory defragmentation systems to achieve good performance.

In this chapter, we present COMPACTGPU, an incremental, fully parallel, in-place memory defragmentation system for GPUs. COMPACTGPU is implemented as an extension to DYNASOAR, but it could also be implemented in other systems. GPUs/SIMD architectures are predominantly programmed in a C++ dialect (e.g., CUDA, OpenCL, ispc [152], Sierra [115]) and memory management in C++ is manual, so we cannot rely on a garbage collector to collect metainformation for us.

COMPACTGPU is an efficient GPU memory defragmentation system that is optimized for GPU-specific allocation patterns. It is fully parallel and in many cases the performance gain of defragmentation is much larger than the defragmentation overhead. This is due to careful design and engineering efforts: COMPACTGPU is based on parallel block merging, utilizes bitmaps to speed up pointer rewriting, exhibits mostly uniform control flow and requires no synchronization between GPU threads.

We evaluated COMPACTGPU with synthetic and real benchmarks. COMPACTGPU can improve application performance by up to 16% and reduce the overall memory consumption of an application, while incurring minimal runtime overheads.

Publications This chapter is in part based on the following papers.

- Matthias Springer, Hidehiko Masuhara. **"Massively Parallel GPU Memory Compaction."** In: *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*. ISMM 2019. ACM, 2019, pp. 14–26. doi:10.1145/3315573.3329979

- Matthias Springer. **"CompactGpu: Massively Parallel Memory Defragmentation on GPUs."** Extended Abstract. In: *ACM Student Research Competition at PLDI 2019*. 3 pages. (reviewed, no formal proceedings)

[1]If thousands of threads were to request large allocations, a GPU would run out of memory immediately.

6.1 Why GPU Memory Defragmentation?

Before introducing the design of our system, we review important performance characteristics of SIMD architectures.

Effects of Fragmentation Fragmentation measures the degree of scattering of allocations across the heap and is caused by unfortunate allocate-deallocate patterns. High fragmentation leads to three main disadvantages.

- **Premature Out-of-Memory:** Large allocations cannot be accommodated even if there is enough free memory overall (*external fragmentation*).
- **Low Cache Hit Rate:** Poor data locality causes poor cache performance [75], because fragmented data occupies more cache lines.
- **Low Vector Load/Store Efficiency:** SIMD vector load/store instructions are less efficient, because accessing fragmented data requires more vector transactions than accessing compact data.

While the first two points are well-established and apply to most architectures, the specific effects on SIMD architectures have received little attention.

Effect of Fragmentation on Vectorized Access SIMD architectures achieve parallelism by executing instructions on a vector register. However, vector load/store operations are less efficient with higher fragmentation. When threads in a GPU application simultaneously access different memory addresses, the GPU *coalesces* accesses from the same SIMD work group (*warp* in CUDA, every 32 consecutive threads) into one physical memory transaction if the addresses are on the same 128-byte cache line (Section 2.1.4). More fragmentation leads to more scattered memory addresses, resulting in poorer performance due to a higher number of memory transactions.

Memory fragmentation can greatly affect vectorized access, even if data is stored in a Structure of Arrays (SOA) data layout. Recent NVIDIA architectures coalesce simultaneous accesses of consecutive memory addresses into 128-byte vector transactions. Accessing fragmented data requires more vector transactions than accessing the same amount of dense data. This reduces the overall performance of memory-bound applications because memory bandwidth is limited [126].

Memory Defragmentation To optimize the memory access of global memory, we are developing a memory defragmentation system for GPUs in this chapter. In essence, every memory defragmentation system has to solve four basic problems.

1. Determine which parts of the heap are fragmented.
2. Based on that information, decide which objects[2] to move (relocate) and where to move them.
3. Physically relocate objects in memory.
4. Find and rewrite pointers to relocated objects. (Alternative: Ensure that objects can still be accessed through their old pointers.)

[2]We use the term *object* instead of *allocation* throughout this chapter because we are focusing on object-oriented systems.

Most memory defragmentation systems are part of a garbage collector. Since garbage collectors have to scan large parts of the heap anyway, they can gather additional metainformation almost for free. This information can be used to select memory areas for compaction [149, 101] or to determine which parts of the heap contains pointers that must be rewritten [187].

6.2 Heap Layout and Data Structures

A variety of dynamic memory allocators for GPUs have been developed in recent years. CompactGpu is implemented in DynaSOAr, but its basic ideas can be adapted to other dynamic memory allocators as long as they follow a few basic design requirements.

- The heap is divided into fixed-size **memory blocks**, in which objects are allocated. Every DynaSOAr block has a constant block capacity (based on its type), regardless of the number of allocated objects.
- A block contains only **objects of the same size**. This requirement is crucial. Same-size objects can be compacted much more easily than objects of different size. Every DynaSOAr block contains objects of only one type, so all objects of a block have the same size.
- Blocks of the same object size have the **same capacity**. All DynaSOAr blocks have the same size in bytes, so all blocks of the same type/object size have the same capacity. Section 5.2.2 describes how exactly block capacities are determined in DynaSOAr.
- The allocator maintains **fill levels** for each block. The fill level of every DynaSOAr block can be determined by counting the number of set bits in the object allocation bitmap.

DynaSOAr, Halloc [7] and UAlloc [71] are three examples of allocators that satisfy these requirements. DynaSOAr is the only memory allocator with an SOA data layout, which allows for efficient vectorized memory access of allocated memory. While all allocators would benefit from better cache performance and more space-efficient memory usage, memory defragmentation in DynaSOAr additionally leads to more efficient vectorized memory accesses and thus better memory bandwidth utilization.

6.2.1 Running Example

We use a simple fish-and-sharks simulation (wa-tor) as a running example to describe the data structures of CompactGpu. Fish and sharks inhabit a 2D grid of cells in a predator-prey relationship. This application has four classes: `Cell`, `Agent`, `Fish` and `Shark`. The last two classes are subclasses of the abstract class `Agent`. We describe wa-tor in more detail in Section 7.6.

wa-tor exhibits a large number of allocations and deallocations in GPU code, which leads to memory fragmentation. By reducing memory fragmentation, CompactGpu can reduce the overall memory consumption of wa-tor.

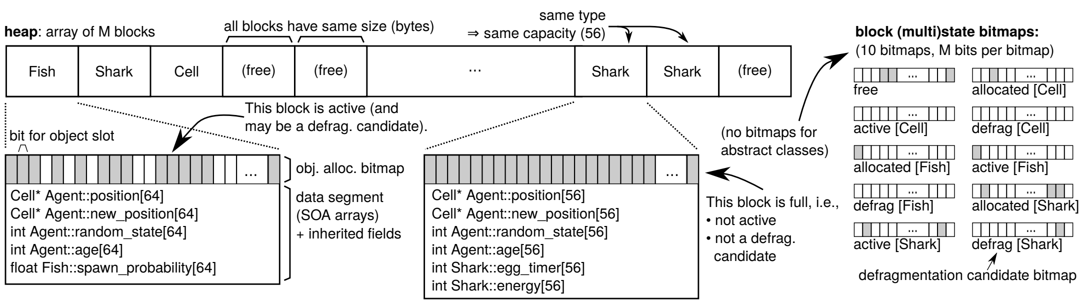

heap: array of M blocks
all blocks have same size (bytes)
same type ⇒ same capacity (56)
Fish
Shark
Cell
(free)
(free)
...
Shark
Shark
(free)
block (multi)state bitmaps:
(10 bitmaps, M bits per bitmap)
free
allocated [Cell]
active [Cell]
defrag [Cell]
allocated [Fish]
active [Fish]
defrag [Fish]
allocated [Shark]
active [Shark]
defrag [Shark]
defragmentation candidate bitmap
(no bitmaps for abstract classes)
This block is active (and may be a defrag. candidate).
bit for object slot
obj. alloc. bitmap
data segment (SOA arrays) + inherited fields
Cell* Agent::position[64]
Cell* Agent::new_position[64]
int Agent::random_state[64]
int Agent::age[64]
float Fish::spawn_probability[64]
Cell* Agent::position[56]
Cell* Agent::new_position[56]
int Agent::random_state[56]
int Agent::age[56]
int Shark::egg_timer[56]
int Shark::energy[56]
This block is full, i.e.,
• not active
• not a defrag. candidate

FIGURE 6.1: Example: Heap layout for wa-tor. The heap consists of equally sized blocks. Up to 64 objects can be stored in a block, as indicated by the *object allocation bitmap*. A block can be in one or multiple of 10 possible (multi)states, as indicated by the state bits shown for every block. There are *allocated*, *active* and *defrag* states for the three classes `Fish`, `Shark` and `Cell`, but not for class `Agent` because it is an abstract class.

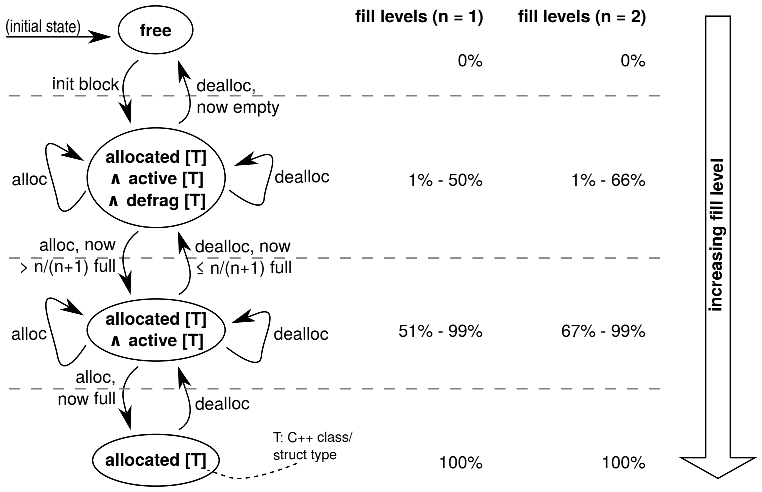


FIGURE 6.2: Block states. Initially, every block is *free*. New objects are allocated in *active* blocks of the corresponding type. We introduced a new state *defrag* to indicate defragmentation candidates. Only objects from such blocks are relocated during defragmentation.

6.2.2 Overview of the DYNASOAR Allocator

Chapter 5 described the DYNASOAR memory allocator in detail. In the following paragraphs, we give a simplified summary of DYNASOAR and describe which parts of the allocator had to be modified to implement COMPACTGPU.

DYNASOAR is a slab allocator [23]. It divides the heap into M blocks of equal byte size, each of which can contain up to 64 objects (*capacity*) of the same C++ class/struct type, depending on the size of the type (Figure 6.1). A position where an object can be stored is called an *object slot*. A 64-bit *object allocation bitmap* keeps track of allocations. Objects are stored in the *data segment* in an SOA data layout: one *SOA array* per field.

Block States A block can be in one or more *multistates*. There are $3 \times \#types + 1$ possible states: one global *free* state and three states for each type T in the system (Figure 6.2).

- **free:** The block is empty and does not contain any objects. No type is specified for this block.
- **allocated[T]:** The block may contain objects only of type T. No other objects can be stored in the block.

- **active[T]:** The block contains objects of type T. It is not full yet, i.e., it has space for at least one more object. *active[T]* $\Rightarrow$ *allocated[T]*.
- **defrag[T]:** The block is considered for defragmentation. We call such a block a *defragmentation candidate*. We introduced this state to support defragmentation in DynaSOAr and will describe its purpose in the next section.
  defrag[T] $\Rightarrow$ *allocated[T]* $\wedge$ *active[T]*.

Allocation, deallocation and defragmentation routines frequently lookup blocks by state. For that reason, block states are indexed by hierarchical bitmaps of size M, as described in Section 5.2.4: one bitmap per state. The bitmap hierarchy is currently not utilized by CompactGpu, so *defrag[T]* does not necessarily have to be hierarchical.

Fragmentation Our definition of fragmentation F differs from other systems. We define it as the fraction of allocated but unused memory. If a block is in an *allocated* state, we consider all of its object slots as *allocated*. However, only object slots that actually contain an object, as indicated by the object allocation bitmap, are *used*. Fragmentation is defined as the average *free level* among all allocated blocks.

$$F = \frac{1}{\#\text{blocks}} \sum_{b \in Blocks} \frac{\#\text{free slots}(b)}{\#\text{slots}(b)} \qquad (\textit{fragmentation})$$

Our goal is to reduce F as much as possible. Zero fragmentation means that all allocated blocks are 100% full and the other blocks are empty (*free*). In that case, vectorized memory access is most efficient. Conversely, a vector load on a block that is 60% full will on average read 40% garbage. Moreover, unused memory in an allocated block is not available for objects of other types. This leads to less space-efficient memory usage.

The blocks themselves may be widely *scattered* in the heap. For example, in Figure 6.1, three allocated blocks are stored at the beginning of the heap and two are stored towards the end. This does not affect cache utilization or vector load/store efficiency, because with up to 64 objects per blocks, most SOA arrays are much larger than a cache line or the size of a vector load/store (128 bytes on NVIDIA GPUs). Neither can it lead to external fragmentation and premature out-of-memory errors, because all blocks have the same byte size.

Object Allocation To reduce fragmentation, even without active memory defragmentation, DynaSOAr allocates new objects of type T always in *active[T]* blocks. These are blocks that have space for at least one more object. Only if no active block could be found, DynaSOAr locates a free block and turns it into an *allocated[T]* and *active[T]* block (*slow path*).

DynaSOAr then reserves an object slot inside the block by atomically flipping a bit in the object allocation bitmap from 0 to 1. If this operation was successful, the block state may have changed, so we may have to update the respective bits in the block state bitmaps.

If the number of objects of a type drops, fragmentation can increase, because a block is deallocated only if all of its objects are deallocated. This kind of fragmentation can be eliminated with CompactGpu.

Programming Interface DynaSOAr provides an embedded C++ DSL for defining classes/fields. Through this DSL, CompactGpu can programmatically *reflect* on the classes/fields that are defined in an application, somewhat similar to the Java Reflection API or metaobject protocols [31]. This functionality is used in the pointer rewriting step to restrict heap scans to a smaller part of the heap (Section 6.3.5) by excluding parts of the heap that are guaranteed to be free of pointers that must be rewritten.

6.3 Defragmentation with COMPACTGPU

COMPACTGPU is a memory defragmentation system for GPUs, implemented as a DYNASOAR extension. COMPACTGPU is:

- **Configurable:** The desired target fragmentation rate can be tuned with parameters. Better defragmentation can lead to more space savings and better memory access performance, but also has a higher defragmentation overhead.
- **Incremental:** A single defragmentation pass is very fast and compacts only a fraction of the heap. Compacting the entire heap requires multiple passes.
- **In-place:** No auxiliary storage is necessary and the entire heap remains usable.
- **A stop-the-world approach:** A defragmentation pass can run only when no other GPU code is running. This is because, in current GPU architectures, there is no efficient way of interrupting a kernel to run a defragmentation pass, should the allocator run out of memory during the kernel. Many GPU programs (including all SMMO examples in Section 7) are a sequence of GPU kernel invocations [168], so there are usually plenty of opportunities to run a defragmentation pass in-between.
- **Fully parallel:** Every step is implemented as a perfectly parallel CUDA kernel. No synchronization among threads is necessary for defragmentation.
- **Not order preserving:** After defragmentation, objects are likely arranged in a different order on the heap.

Programmers initiate defragmentation manually, typically after a parallel do-all operation, and specify the C++ type that should be defragmented. COMPACTGPU extends DYNASOAR with an additional host allocator handle function for initiating defragmentation.

- `HAllocatorHandle::parallel_defrag<T, k1, k2>()`: Initiate memory defragmentation for objects of type T. Internally, this function may run multiple defragmentation passes. k_1 and k_2 are parameters that control the number of defragmentation passes and are described later.

COMPACTGPU is based on three fundamental ideas.

- **Block Merging:** The heap is defragmented by moving/relocating objects from source blocks to target blocks.
- **Forwarding Pointers:** After relocating objects, pointers to the new object locations are stored in source blocks.
- **Bitmaps:** To speed up pointer rewriting, bitmaps are utilized to quickly decide whether a pointer must be rewritten.

We considered various alternative designs (Section 6.4), but the combination of forwarding pointers with bitmaps proved to be most performant on GPUs.

Block Merging COMPACTGPU compacts the heap by merging blocks of the same type. Blocks of the same type have the same capacity, so *a source block can be merged into a target block of the same type if both blocks are no more than 50% full*. This is to ensure that all objects of the source block fit into the target block. While it would be sufficient to require that both fill levels *together* are no more than 100%, this techique is easier to implement and more space-efficient, because the fill level of a block can then be encoded in a single bit (i.e., 1 means $\leq 50\%$ and 0 means $> 50\%$).

We can extend this idea to higher fill levels: A source block can be merged into two target blocks if none of the three blocks is more than 66% full. Or in general: A source block can be merged into n target blocks if none of the $n+1$ blocks is more than $\frac{n}{n+1}$ full. In each case, the source block is eliminated and the number of allocated blocks is reduced by one.

Defragmentation Factor We call n the *defragmentation factor*. This value is problem-specific and must be chosen by the programmer at compile time. Blocks that are no more than $\frac{n}{n+1}$ full are *defragmentation candidates*. Only those blocks are considered during defragmentation. Higher defragmentation factors increase the overhead of memory defragmentation but can lead to a lower final fragmentation rate. Whether a higher defragmentation factor pays off depends on the application.

During defragmentation, all objects from a source block are moved to one or multiple target blocks. The source block is deleted. Target blocks lose their defragmentation candidate state *defrag[T]* if they are now more than $\frac{n}{n+1}$ full. They also lose their active state *active[T]* if they are now entirely full.

Given a defragmentation factor of n, COMPACTGPU is guaranteed to bring down fragmentation to $1 - \frac{n}{n+1} = \frac{1}{n+1}$, if all defragmentation candidates are eliminated. This may require multiple defragmentation passes, as will be described later. For example, for $n = 2$, all blocks with $\leq 66\%$ fill level are eliminated during defragmentation[3]. Only blocks with a higher fill level are left over. Consequently, the final fragmentation level is guaranteed to be less than $1 - 66\% = 33\%$.

6.3.1 Defragmentation Candidate Bitmaps

A defragmentation pass must be able to quickly find all defragmentation candidates in order to choose source and target blocks efficiently. COMPACTGPU extends object allocation and deallocation routines of DYNASOAR to keep track of defragmentation candidates. COMPACTGPU maintains *defrag[T]* bitmaps (one per type) in which a bit is set if the corresponding block is a defragmentation candidate. Every defragmentation candidate is by definition also an active block, because it is not entirely full. Similar to other block state bitmaps, defragmentation candidate bitmaps may have to be updated whenever block fill levels change. An alternative implementation could scan the object allocation bitmaps of every block and generate *defrag[T]* on demand.

Algorithms 13, 14 and 15 are extended versions of Algorithms 1, 7 and 2. They show which parts of object (de)allocation[4] we changed to keep track of defragmentation candidates. Recall that the ordering of *if* branches is crucial to avoid deadlocks in Algorithm 15 (Section 5.4.4). Assuming that CUDA schedules branches in the order in which they appear in the source code, the *if* branches must be ordered from *high fill level* to *low fill level*.

[3]Since every source block must be matched with $n = 2$ target blocks, up to two defragmentation candidates may be left over.

[4]We are not taking into account allocation request coalescing or block invalidation here.

Algorithm 13: DAllocatorHandle::allocate<T>() : T* $\boxed{\text{GPU}}$

```
repeat                                                ▷ Infinite loop if OOM
    bid ← active[T].try_find_set();     ▷ Find and return the position of any set bit.
    if bid = FAIL then                                      ▷ Slow path
        bid ← free.clear();        ▷ Find and clear a set bit atomically, return position.
        initialize_block<T>(bid);
        allocated[T].set(bid);
        $\boxed{\text{defrag[T].set(bid);}}$
        active[T].set(bid);
    alloc ← heap[bid].reserve();          ▷ Reserve an object slot. See Alg. 14.
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;
        $\boxed{\text{if alloc.state = LEQ then defrag[t].clear(bid) ;}}$
        if alloc.state = FULL then active[t].clear(bid) ;
        if $t = T$ then return ptr ;
        deallocate<t>(ptr);               ▷ Type of block has changed. Rollback.
until false;
```

Algorithm 14: Block::reserve() : (int, state) ▷ Assuming block size 64. $\boxed{\text{GPU}}$

```
(same as Algorithm 7, Lines 1–8)
if success then
    if popc(before) = 63 then
        return (pos, FULL)
    else if $\boxed{popc(\text{before}) = 64 \cdot \frac{n}{n+1} + 1}$ then    ▷ E.g., $popc(\text{before}) = 33$ for $n = 2$
        $\boxed{\textbf{return } (\text{pos}, LEQ)}$
    else
        return (pos, REGULAR)
return FAIL
```

6.3.2 Defragmentation Pass

A single defragmentation pass consists of four main steps. Every step is implemented as a CUDA kernel and runs in parallel.

1. Copy objects from source to target locations.
2. Store forwarding pointers in source blocks.
3. Scan the heap and rewrite pointers to source locations.
4. Update block state bitmaps.

In the following sections, we describe each of these steps.

6.3.3 Copying Objects

COMPACTGPU copies objects from source to target blocks. Those blocks must be defragmentation candidates, to ensure that all objects of a source block fit into the corresponding target blocks.

Algorithm 15: DAllocatorHandle::deallocate<T>(T* ptr) : void $\boxed{\text{GPU}}$

```
1  bid ← extract_block(ptr);
2  slot ← extract_slot(ptr);
3  state ← heap[bid].deallocate(slot);
4  if state = FIRST then                          ▷ Deallocated first object of full block.
5      active[T].set(bid)
6  if $\boxed{\text{state} = LEQ}$ then                                          ▷ Now $\leq \frac{n}{n+1}$ full
7      $\boxed{\text{defrag[T].set(bid);}}$
8  else if state = EMPTY then                     ▷ Deallocated last object of block.
9      if invalidate(bid) then
10         t ← heap[bid].type;
11         active[t].clear(bid);
12         $\boxed{\text{defrag[t].clear(bid);}}$
13         allocated[t].clear(bid);
14         free.set(bid);
```

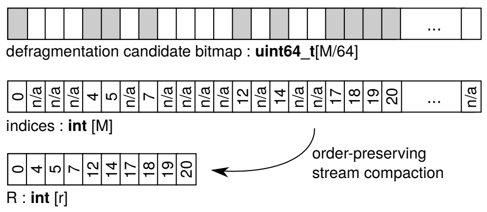

...
defragmentation candidate bitmap : uint64_t[M/64]
0 n/a n/a n/a 4 5 n/a 7 n/a n/a n/a n/a 12 n/a 14 n/a n/a 17 18 19 20 ... n/a
indices : int [M]
order-preserving stream compaction
0 4 5 7 12 14 17 18 19 20
R : int [r]

FIGURE 6.3: Example: Compacting a bitmap of defragmentation candidates (*defrag[T]*). There is such a bitmap for every type. See Figure 5.6 for full details. Must preserve order of indices.

Choosing Source/Target Blocks We utilize the defragmentation candidate bitmap to quickly find and assign target blocks to source blocks (Figure 6.3). We first generate an *indices* array of size M that contains i at position i if the i-th bit is set. Otherwise, we store an *invalid marker*. Now we filter/compact the array to retain only valid values, resulting in array R of size r. This *stream compaction* [14] is implemented with a parallel prefix sum operation (CUB library [138]).

Based on array R, each GPU thread can later by itself (without synchronization) efficiently determine its assigned source block and corresponding target blocks (Figure 6.4). Given a defragmentation factor n, the $B = \lfloor \frac{r}{n+1} \rfloor$ blocks with indices $R[0]$ through $R[B-1]$ are source blocks. Given a source block $R[s\_rid]$, its corresponding target blocks are:

$$\{\, R[s\_rid + i \cdot B] \;\mid\; i \in 1 \ldots n \,\} \qquad (\textit{target block IDs})$$

Note that the first B blocks in R are source blocks. This fact will be used later to optimize pointer rewriting.

Copying Objects Objects are copied in parallel. We assign 64 consecutive threads to every source block (Figure 6.4) and every thread copies at most one object. Some

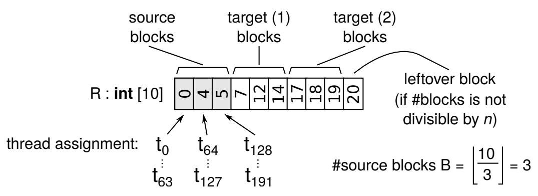


FIGURE 6.4: Example: Assigning source and target blocks ($n = 2$). E.g., objects from source block 5 are moved into blocks 14 and 19. One block is left over.

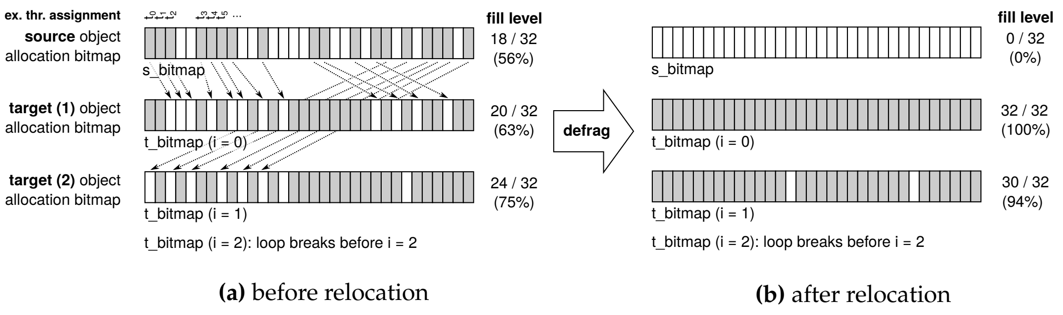


FIGURE 6.5: Example: Relocating objects ($n = 3$). Assuming block size 32 instead of 64. All 18 objects fit into the first two selected target blocks, so no third block is needed. The first 12 objects fit into the first target block. The remaining 6 objects fit into the second one.

threads will have no work to do, because the capacity of the block may be less than 64 and because not all slots in a source block are occupied. However, by assigning 64 threads, we are assigning exactly two full warps, which is reduces thread divergence.

Algorithm 16 shows how objects are copied (Figure 6.5). There are 64 threads for every source block. A thread t_{tid} copies the ($\mathsf{s\_loc} = tid \ \% \ 64$)-th object of the source block (ID s_bid). Let s_oid be the slot ID of this object. The target slot is the s_loc-th free slot among all target blocks. Let t_oid and t_bid be the slot ID and block ID of that slot. To determine the target slot, we may have to examine the object allocation bitmap of multiple or all n target blocks (Line 7). This causes some *thread divergence* because the number of *for* loop iterations differs among threads, but n is usually small. Furthermore, note that no synchronization is required among threads.

On GPUs, memory accesses have a much higher latency than arithmetic instructions [193]. Therefore, we have to keep the number of extra accesses in addition to the field copies low. Object copies cannot be avoided without changing the defragmentation strategy and these extra accesses represent the overhead of our copy phase implementation. Since all threads in a warp copy from/to the same blocks, we require at most $1 + n$ read transactions from the array R and the same number read transactions of object allocation bitmaps per warp. Since all threads in a warp have the same source/target blocks, they access the same array slot of R and the same object allocation bitmaps, so these accesses can be coalesced.

Better Source/Target Choices? The number of object copies (and pointer rewritings) could be reduced by selecting less full defragmentation candidates as source blocks. Such an optimization does not pay off for three reasons.

First, selecting source blocks becomes much more difficult. How would a GPU thread know which block is less full? We would either have to sort the array R with a comparator function that counts the set bits in each block's object allocation bitmap (a random memory access!). Or we would have to maintain additional *defrag[T]* bitmaps for various fill levels. Both variants would greatly reduce performance.

Second, since memory is accessed in 128-byte vector transactions, reading/writing a slightly lower number of scalar values (that are likely scattered within an SOA array) is unlikely to reduce the number of memory transactions.

And third, there would be more threads without work, but since all threads in a warp must execute the same instructions, these threads nevertheless have to *wait* for the copying threads in the warp (*warp divergence*).

```
Algorithm 16: move_objects<T>() : void                                    $\boxed{\text{CPU}}$
1  for $tid \leftarrow 0$ to $64 \cdot B$ in parallel do               ▷ $\boxed{\text{GPU}}$ CUDA kernel
2      s_rid ← tid / 64;  s_bid ← $R[s\_rid]$;                           ▷ Source block ID
3      s_loc ← tid % 64;                          ▷ This thread copies s_loc$^{th}$ obj.
4      s_bitmap ← heap[s_bid].bitmap;
5      if $s\_loc < popc(s\_bitmap)$ then
6          t_loc ← s_loc;                 ▷ This thread copies to t_loc$^{th}$ free slot.
7          for $i \leftarrow 0$ to $n$ do            ▷ Thr. divergence increases with $n$.
8              t_rid ← s_rid + $i \cdot B$;
9              t_bid ← $R[t\_rid]$;                                      ▷ Target block ID
10             t_bitmap ← ~heap[t_bid].bitmap;
11             t_slots ← $popc$(t_bitmap);
12             if $t\_loc < t\_slots$ then
13                 break;                            ▷ Target block t_bid determined.
14             else
15                 t_loc ← t_loc − t_slots;
16         s_oid ← $nth\_set\_bit$(s_bitmap, s_loc);
17         t_oid ← $nth\_set\_bit$(t_bitmap, t_loc);
18         s_ptr ← $make\_pointer$(s_bid, s_oid);
19         t_ptr ← $make\_pointer$(t_bid, t_oid);
20         *t_ptr ← *s_ptr;                                            ▷ Copy all fields.
21     end                                               ▷ else: No work for this thread.
```

```
Algorithm 17: place_forwarding_ptrs<T>() : void                           $\boxed{\text{CPU}}$
1  for $tid \leftarrow 0$ to $64 \cdot B$ in parallel do               ▷ $\boxed{\text{GPU}}$ CUDA kernel
       (same as in $move\_objects$<T>(), lines 2–19)
20         heap[s_bid].data.forwarding_ptr[s_oid] ← t_ptr;
21     end                                               ▷ else: No work for this thread.
```

6.3.4 Storing Forwarding Pointers

After copying objects, pointers to the old memory location must be updated (*rewritten*). Most memory defragmentation systems do this with *forwarding pointers*: A pointer to the object's new memory location is stored at its old location.

We extended DynaSOAr to store forwarding pointers inside blocks. Every block may contain either a data segment or forwarding pointers. Listing 6.1 shows the data structure of a block of type `Fish` (also see lower left part of Figure 6.1).

Algorithm 17 shows how the forwarding pointers array is populated. This algorithm is identical to Algorithm 16, except for Line 20. These two algorithms cannot be merged into a single kernel because a forwarding pointer would then overwrite parts of the data segment[5].

6.3.5 Rewriting Pointers

The heap is now scanned for pointers that must be rewritten. If a pointer points to a location with a forwarding pointer, it is replaced with the forwarding pointer. We

[5]If the size of the first field is 8 bytes (same as a pointer), as in the example here, this is harmless. Otherwise, a thread would store a forwarding pointer into a memory location that is copied by another thread (race condition).

Algorithm 18: rewrite_pointer<T>(T* ptr) : T* GPU

```
1 s_bid ← extract_block_id(ptr);
2 if $s\_bid < R[B] \wedge defrag[T][s\_bid]$ then
3     s_oid ← extract_object_id(ptr);
4     return heap[s_bid].data.forwarding_ptr[s_oid];
5 else
6     return n/a;
```

LISTING 6.1: Example: Block structure for class `Fish`

```cpp
template<> struct Block<Fish> {
  uint64_t bitmap;

  union {
    struct {
      Cell* position[64];
      Cell* new_position[64];
      int random_state[64];
      int age[64];
      float spawn_probability[64];
    } data_segment; /* SOA arrays */

    Fish* forwarding_ptr[64];
  } data;
};
```

utilize the defragmentation candidate bitmap to quickly decide if a pointer must be rewritten, without reading the memory at the pointer location. *We read that location only if we are sure that it contains a forwarding pointer.* This is a key difference compared to other defragmentation systems.

Given a pointer `ptr`, Algorithm 18 returns the corresponding forwarding pointer or *n/a* if no forwarding pointer exists for `ptr`. We first extract the block ID s_bid of the object that `ptr` points to. This block is a source block if it is a defragmentation candidate (i.e., bit set in the defragmentation candidate bitmap) and if the block ID is smaller than $R[B]$ (Line 2). Recall that the blocks with IDs $R[i]$ with $i \in [0; B-1]$ are source blocks and R is sorted (Figure 6.4). Large parts of the *defrag[T]* bitmap will likely be cached by the L1/L2 caches, so we expect these bitmap lookups to be fast.

If the block is a source block, we extract the object ID s_oid from `ptr` and return the corresponding forwarding pointer. It is crucial that Algorithm 18 is efficient because it is executed for every pointer that is found during a heap scan.

Limiting Heap Scans Recent GPUs have up to 32 GB of memory, so scanning all allocated memory for pointers to rewrite is expensive. However, since classes and fields of application code are defined with DYNASOAR's DSL, COMPACTGPU can reflect on application classes and determine which parts of the heap may contain pointers that must be rewritten. As such, only a small part of the heap is scanned.

For example, when defragmenting `Fish` objects, we can avoid looking into blocks of type `Fish` or `Shark`, because those classes do not have fields of type *pointer to* `Fish` or *pointer to a superclass of* `Fish`. Only class `Cell` has a field of type `Agent*`, so we only scan the corresponding SOA arrays in the data segment of allocated blocks of type `Cell`. These are the pointers that are rewritten according to Algorithm 18.

In general, when defragmenting objects of type T, we first determine all classes U that have at least one field of type *pointer to* S, where $S :> T$ is a supertype of T or equal to T[6]. For every such type U, we scan allocated blocks of type U. For every allocated block, we scan the SOA arrays of fields of type *pointer to* S. Only these pointers are scanned and rewritten. This can exclude more than 95% of the heap. Moreover, reading the values from an SOA array is fast because those field reads are coalesced by the GPU.

Most other allocators have limited information about the structure of their allocations. If we were to implement COMPACTGPU's technique in such allocators, we cannot restrict the search space as described here. Previous work describes alternative techniques for limiting the search space based on intermediate results from a garbage collector [187].

6.3.6 Updating Block State Bitmaps

Finally, the state of source and target blocks must be updated in the corresponding block state bitmaps. Source blocks lose their *active*, *allocated* and *defrag* states and become *free*. Target blocks may lose their *active* and/or *defrag* states depending on their new fill level.

Moreover, the object allocation bitmaps of all target blocks must be updated. If m objects were relocated to a given target block, one thread flips the m first cleared bits to 1, using an algorithm that is similar to Algorithm 9.

6.3.7 Multiple Defragmentation Passes

A single defragmentation pass is guaranteed to delete all source blocks, i.e., $\frac{1}{n+1}$ of all defragmentation candidates. In addition, some target blocks may lose their defragmentation candidate state. However, a fragmentation level of $\frac{1}{n+1}$ can be achieved only if all candidates were eliminated (Section 6.3). This may require multiple defragmentation passes.

The efficiency of defragmentation passes decreases with a decreasing number of defragmentation candidates. For example, for $n = 1$, a single pass is guaranteed to eliminate 500 out of 1,000 total candidates. However, the next pass is only guaranteed to eliminate 250 out of 500 remaining candidates. Moreover, too much defragmentation can make allocations more expensive because DYNASOAR has to (re)initialize new blocks if there are not enough active blocks. To reduce runtime overheads, defragmentation should stop before eliminating all defragmentation candidates.

COMPACTGPU runs multiple defragmentation passes until all but k_1 defragmentation candidates were eliminated. The value of k_1 can be configured. Since defragmentation passes are very fast, the value of k_1 matters only in cases with a large number defragmentation passes or a massive number of (de)allocations.

Worst-case Analysis Let d be the number of defragmentation candidates. A single defragmentation pass reduces d at least by a fraction of $\frac{1}{n+1}$, so no more than $\frac{n}{n+1}$ of defragmentation candidates are left over. We can bound the number of defragmentation passes that are necessary in the worst case to eliminate all defragmentation candidates by:

[6]We also consider fields of type *array of pointer to* S etc. For simplicity, we mention only simple pointer types here.

$$\log_{\frac{n+1}{n}} d \qquad \textit{(worst-case number of defrag. passes)}$$

If all but $k_1 \geq 1$ defragmentation candidates should be eliminated, we can bound the number of defragmentation passes by:

$$\log_{\frac{n+1}{n}} d - \log_{\frac{n+1}{n}} k_1 = \log_{\frac{n+1}{n}} \frac{d}{k_1} \qquad \textit{(keep } k_1 \textit{ defrag. candidates)}$$

In reality, the number of required defragmentation passes is usually much lower. We experimentally analyze the actual number of defragmentation passes in Section 6.5.2.

6.3.8 Defragmentation Frequency

Memory defragmentation must be initiated by the programmer explicitly. Once initiated, COMPACTGPU may run multiple defragmentation passes depending on n, k_1 and the number of defragmentation candidates. We suggest one of the two following defragmentation policies for initiating defragmentation.

Every m Iterations Many GPU programs run a number of CUDA kernels iteratively in a loop. This policy initiates defragmentation every m iterations.

After Massive Deallocations Initiate defragmentation if there are at least k_2 many defragmentation candidates[7], where k_2 should be a large enough value. COMPACTGPU provides a helper function that lets programmers specify this threshold as an absolute number or as a percentage of the heap size and then initiates defragmentation if necessary. Internally, COMPACTGPU scales k_2 by $\frac{n}{n+1}$ to account for the fact that a larger value of n usually leads to more defragmentation candidates.

As a rule of thumb, we use the first policy for applications that experience a speedup from defragmentation, because even small compactions can lead to a performance gain. The second policy is useful for applications that mainly benefit from better space efficiency or see a slowdown from defragmentation. Future work will investigate how to automate defragmentation (choosing policies, parameters, etc.).

6.4 Pointer Rewriting Alternatives

Pointer rewriting is the most time-consuming step in applications with a large object set. In Algorithm 18, reading the forwarding pointer in Line 4 is a random memory access that cannot be coalesced and thus the most expensive operation of the algorithm. To get rid of this memory access, we implemented two alternatives that recompute forwarding pointers on-the-fly.

6.4.1 RECOMPUTE-GLOBAL: Recompute Forwarding Pointers

Instead of storing forwarding pointers in objects, we maintain *defragmentation records* (Figure 6.6). A defragmentation record is a tuple of source block ID, source object

[7]There is no global object counter. The number of defragmentation candidates approximates the number of allocated but unused object slots.

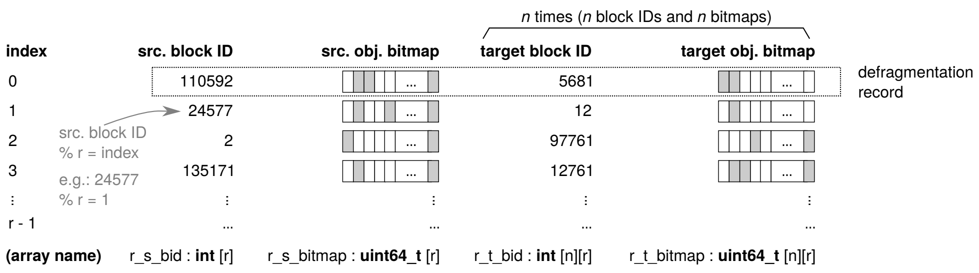


FIGURE 6.6: Example: Defragmentation records. Stored in SOA layout.

bitmap (i.e., object allocation bitmap), target block IDs and target object bitmaps. Based on a defragmentation record, all forwarding pointers for objects from the respective source block can be recomputed. Defragmentation records are stored in SOA layout (4 arrays for $n = 1$).

Source block IDs are stored in shared memory[8] and the remaining 3 arrays are stored in global memory. Current NVIDIA GPUs have 48 KB of shared memory, so we can have at most r = 48 KB / sizeof(**int**) = 12288 defragmentation records per defragmentation pass.

$$r = \frac{48 \cdot 1024 \text{ bytes}}{4 \text{ bytes}} = 12288 \qquad (\textit{number of defrag. record slots})$$

Choosing Source/Target Blocks During source/target block selection, we hash as many defragmentation records to defragmentation record array slots as possible and proceed with object copying. The defragmentation records data structure is effectively a hash table with the source block ID as key.

$$h(s\_bid) = s\_bid \;\%\; r \qquad (\textit{hash function})$$

The main challenge of selecting source/target blocks is to avoid hash collisions. Only a few defragmentation candidates have a suitable block ID s_bid, such that the block can be a source block stored in a given defragmentation record slot rid: Namely, those blocks whose hash code $h(s\_bid)$ is rid. If all of those blocks are assigned as target blocks, then this defragmentation record slot remains unused. Too many unused defragmentation record slots increase the number of required defragmentation passes.

To utilize as many defragmentation record slots as possible and to make best use of the limited amount of shared memory, we first assign source blocks and later target blocks. Algorithm 19 shows how source blocks are assigned. We run a CUDA kernel with one GPU thread for each defragmentation record array slot tid (thread ID). Each thread iteratively checks in the defragmentation candidate bitmap the bits of all blocks whose hash code equal the thread's assigned array slot. The algorithm chooses the first suitable block (Line 6). We should ensure that no more than $\frac{d}{n+1}$ are selected, where d is the number of defragmentation candidates; otherwise, there would not be enough target blocks for some source blocks. To that end, we maintain an atomic counter (Line 9). Finally, after storing the source block ID and bitmap in

[8]Shared memory is an explicitly programmable part of the L1 cache.

Algorithm 19: choose_source_blocks<T>() : void $\boxed{\text{CPU}}$

```
max_blocks ← $\left\lfloor \frac{d}{n+1} \right\rfloor$;                ▷ Leave enough blocks for target blocks.
for $tid$ ← 0 to $r$ in parallel do                           ▷ $\boxed{\text{GPU}}$ CUDA kernel
    s_bid ← $n/a$;
    for $bid$ ← $tid$ to $m$ by $r$ do                         ▷ Invariant: bid % r = tid
        if defrag[T][bid] then
            s_bid ← bid;                                  ▷ Choose this block as source.
            break;
    if s_bid ≠ $n/a$ then
        if $atomicSub$(&max_blocks, 1) > 0 then
            r_s_bid[tid] ← s_bid;
            r_s_bitmap[tid] ← heap[s_bid].bitmap;
            defrag[T].$clear$(s_bid);                       ▷ Block cannot be a target.
        else
            r_s_bid[tid] ← $n/a$;
    else
        r_s_bid[tid] ← $n/a$;
```

Algorithm 20: choose_target_blocks<T>() : void $\boxed{\text{CPU}}$

```
for $tid$ ← 0 to $r$ in parallel do                           ▷ $\boxed{\text{GPU}}$ CUDA kernel
    if r_s_bid[tid] ≠ $n/a$ then                              ▷ else: Slot not in use.
        for $i$ ← 0 to $n$ do                                 ▷ Choose $n$ target blocks.
            r_t_bid[i][tid] ← defrag[T].$clear$();
            r_t_bitmap[tid] ← heap[r_t_bid[i][tid]].bitmap;
```

the defragmentation records data structure, we clear the block's defragmentation candiate bit[9].

Finally, we assign target blocks (Algorithm 20). We assign n blocks to each source block by atomically finding and clearing a set bit in the defragmentation candidate bitmap. If this block is still a defragmentation candidate after this defragmentation pass, the bit must be set again.

Rewriting Pointers We place no forwarding pointers in blocks. During pointer rewriting (Algorithm 21), we check in shared memory if there is a defragmentation record for the block of ptr instead of checking the bit in the defragmentation candidate bitmap: We traded a global memory access for a much faster shared memory access. However, this approach limits the number of source blocks per defragmentation pass and, therefore, may increase the number of required defragmentation passes. Furthermore, we now have to recompute the forwarding pointer, which requires additional global memory accesses for reading the remaining defragmentation record values in case `ptr` must be rewritten. The computation of target pointers in Algorithm 21 is similar to Algorithm 16 and follows the same notation and variable names.

[9] In contrast to our original forwarding pointer technique, we clear this bit during source block selection instead of in a separate step (Section 6.3.6).

Algorithm 21: rewrite_pointer<T>(T* ptr) : T* $\boxed{\text{GPU}}$

```
s_bid ← extract_block_id(ptr);
if r_s_bid[h(s_bid)] = s_bid then                ▷ Matching defrag. record in shared memory.
    s_oid ← extract_object_id(ptr);
    s_bitmap ← r_s_bitmap[h(s_bid)];
    s_loc ← popc(((1 << s_oid) - 1) & s_bitmap);   ▷ Bit s_oid is the s_loc-th set bit.
    t_loc ← s_loc;
    for i ← 0 to n do
        t_bid ← r_t_bid[i][h(s_bid)];
        t_bitmap ← ~r_t_bitmap[i][h(s_bid)];
        t_slots ← popc(t_bitmap);
        if t_loc < t_slots then
            break;                                ▷ Target block t_bid determined.
        else
            t_loc ← t_loc − t_slots;
    t_oid ← nth_set_bit(t_bitmap, t_loc);
    t_ptr ← make_pointer(t_bid, t_oid);
    return t_ptr;
else
    return n/a;
```

6.4.2 RECOMPUTE-SHARED: Defrag. Records in Shared Memory

To further reduce the number of global memory accesses, we modified RECOMPUTE-GLOBAL to store the entire defragmentation records data structure in shared memory. The size of a defragmentation record is then $12 \cdot (n+1)$ bytes (4-byte block IDs and 8-byte bitmaps), so the shared memory can hold only 2048 defragmentation records in shared memory for $n = 1$ (and even less for larger n). This further increases the number of required defragmentation passes, but also reduces the number of global memory accesses during pointer rewriting.

$$r = \frac{48 \cdot 1024 \text{ bytes}}{(n+1) \cdot 12 \text{ bytes}} \qquad (\textit{number of defrag. record slots})$$

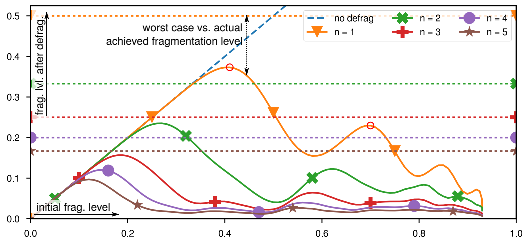
6.5 Evaluation

We evaluated COMPACTGPU with an NVIDIA TITAN Xp GPU (12 GB device memory). We compiled the programs with nvcc (-O3) from the CUDA Toolkit 10.1 on Ubuntu 16.04.4.

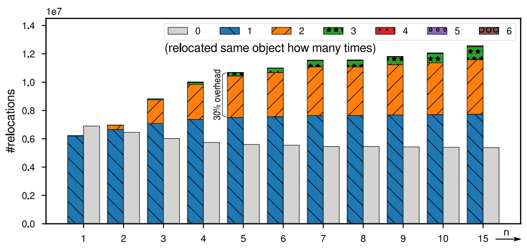
6.5.1 Defragmentation Quality

We first investigate how much fragmentation COMPACTGPU can eliminate. As described in Section 6.3, given a defragmentation factor n, the fragmentation level is guaranteed to be less than $\frac{1}{n+1}$ after defragmentation. However, in reality the fragmentation level is even lower.

We ran a synthetic benchmark that first allocates a very large number of objects and then randomly deallocates some objects. COMPACTGPU then defragments the heap with $k_1 = 0$. We measured the fragmentation level after defragmentation for different values of n. In Figure 6.7A, the x-axis denotes the initial heap fragmentation



(A) Defrag. quality by initial fragmentation



(B) Number of object relocations

FIGURE 6.7: Achieved fragmentation level and number of object relocations

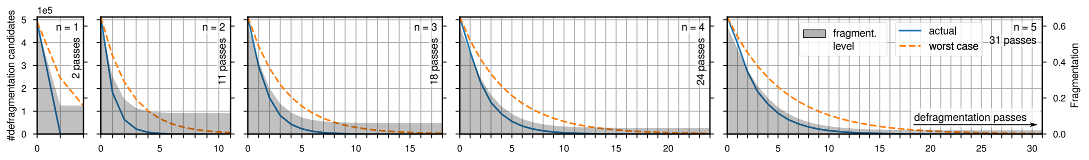


FIGURE 6.8: Number of defragmentation candidates by number of defragmentation passes

level (i.e., the percentage of deallocated objects) and the y-axis denotes the fragmentation level after defragmentation. Lower is better. The dotted lines indicate guaranteed fragmentation levels after defragmentation (fragmentation level $\frac{1}{n+1}$).

COMPACTGPU achieves its worst defragmentation quality at an initial fragmentation level that is slightly smaller than $\frac{n}{n+1}$ (e.g, 45% for $n = 1$). In this case, many blocks are at the boundary of becoming defragmentation candidates. There are a few more points with bad defragmentation quality. For example, around 70% for $n = 1$. In this case, a number of defragmentation candidates were eliminated in the first defragmentation pass, but the resulting blocks have unfortunate fill levels at the boundary of becoming a defragmentation candidate.

6.5.2 Number of Defragmentation Passes

We now investigate the number of defragmentation passes that are necessary to reach good fragmentation levels. The number of defragmentation passes is bounded by $\log_{\frac{n+1}{n}} d$, but in reality fewer passes are needed because some target blocks lose their state as defragmentation candidates.

We ran the same synthetic benchmark, but fixed the initial fragmentation level at 60%. Figure 6.8 shows the number of defragmentation candidates and the fragmentation level after every defragmentation pass. The length of the x-axis indicates the number of defragmentation passes required to eliminate all defragmentation candidates. The y-axis shows the number of remaining defragmentation candidates and the fragmentation level (gray area). Only a few passes bring down fragmentation to very low levels. Moreover, the number of required passes to eliminate all candidates is significantly lower than the theoretical upper bound.

Figure 6.7B shows the total number of object relocations for the synthetic benchmark (60% initial fragmentation level) at various defragmentation factors (x-axis). COMPACTGPU runs multiple defragmentation passes, so some objects may be relocated multiple times. The stacked bars classify relocations by the number of times an object is relocated (e.g., *0* = object not relocated, *1* = object relocated for the first time, etc.). All bars above "1" indicate an overhead of COMPACTGPU and could potentially be avoided by choosing different source/target blocks or with a different defragmentation strategy.

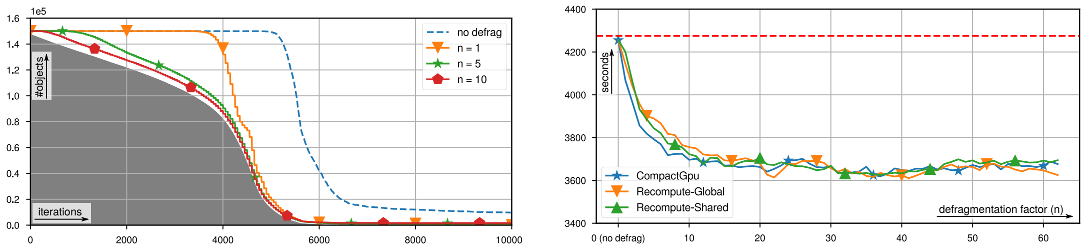


FIGURE 6.9: collision: N-body simulation with collisions

E.g., for $n = 5$, in 30.0% of all object relocations, an object was copied already for the second time or even more often. Even though 31 defragmentation passes are required for $n = 5$, no object was relocated more than 5 times.

6.5.3 Benchmark Applications

We evaluated COMPACTGPU with four SMMO applications (with $k_1 = 16$). Since dynamic memory allocation is not widely used on GPUs yet, there are no suitable standard benchmark suites. Our benchmarks are a subset of the DYNASOAR benchmarks and exhibit varying allocation patterns.

We measured the defragmentation quality and running time with different defragmentation factors. The defragmentation factor must be smaller than the capacity of a block, so every problem has a different maximum defragmentation factor. The dashed red lines in the running time graphs are baseline running times without defragmentation.

For every application, we also show a memory profile. The shaded area indicates the number of allocated objects (used object slots). Different colors indicate different C++ classes in the application. The lines indicate the actual memory usage (allocated object slots). The gap between the shaded area and a line is memory that is wasted due to fragmentation.

All applications except for wa-tor experience a speedup with memory defragmentation. wa-tor experiences a slowdown, but space savings.

collision This is an n-body simulation with collisions (Figure 6.9). A large number of body objects is allocated at the beginning. No other objects are allocated. When two body objects collide, they are merged and one object is deallocated. The fragmentation level increases gradually with every deallocated body. The worst fragmentation is reached around iteration 5,000, when most objects were already deallocated but most blocks are still allocated due to a few remaining objects in each block.

We initiated defragmentation every 50 iterations. Defragmentation had a very small overhead and led to a performance improvement of 12.2% for $n = 36$. This is because of more efficient vector load/store instructions (more coalescing) and due to better cache utilization. The initial dataset size is 5.7 MB. Towards the end of the simulation, only few objects remain, and if they are stored in a dense way, they fit into GPU caches.

structure This is a simulation of a fracture in a composite material (Figure 6.10), modeled as a mesh of finite elements. The simulation exerts a force on some elements and connections between two elements break if the force between them exceeds a certain threshold. A BFS pass identifies elements that are disconnected from the remaining simulation and deallocates them.

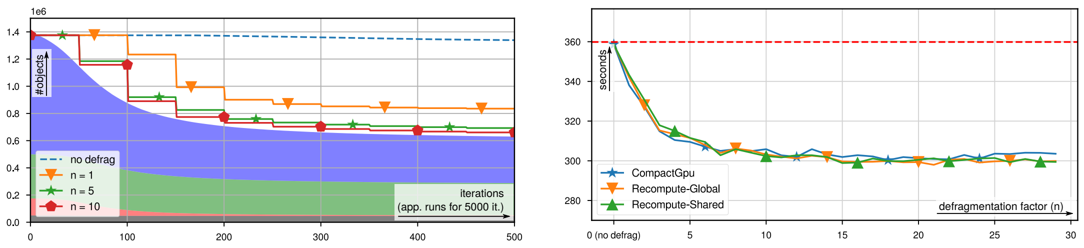


FIGURE 6.10: structure: Simulation of a fracture in a composite material (FEM)

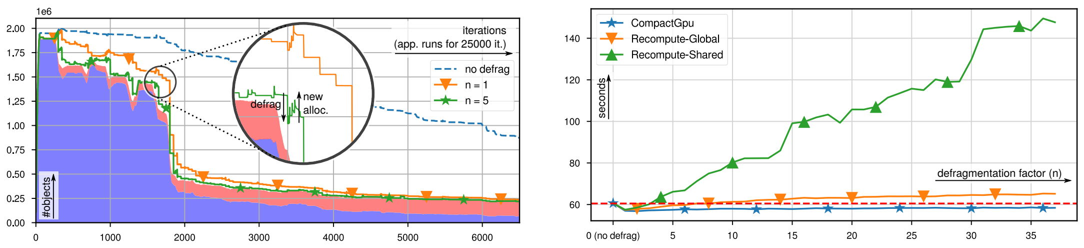


FIGURE 6.11: generation: Generational cellular automaton

Similar to collision, this simulation exhibits only deallocations. However, this simulation has four classes. Even though many objects are already deallocated at the end of the simulation, most blocks are still allocated and overall memory consumption has barely decreased.

We initiated defragmentation every 50 iterations. Defragmentation achieved a peak speedup of 16.3% for $n = 18$.

generation This is an adaptation of Game of Life (rule 0235678/3468/255, similar to "Burst" [204]; Figure 6.11). After a cell dies, it stays around for a few more iterations and blocks the cell. This implementation simulates only alive cells by allocating objects for alive cells and cells that may become alive in the next iteration.

This simulation contains both allocation and deallocation of objects. After iteration 2,000, most cells are dead and the simulation converges into a mostly static pattern.

We initiated defragmentation every 50 iterations for a speedup of 6.3% ($n = 2$). Higher defragmentation factors led to *overfitting*: E.g., the line for $n = 5$ follows the number of allocated objects very closely, even into small local minima. This does not give any additional performance benefit.

We chose $k_1 = 16$, i.e., 16 defragmentation candidates are excluded from defragmentation. It is important to retain a few fragmented (non-full) blocks because DYNASOAR first looks for active (non-full) blocks during allocations (*fast path*) and has to initialize a new block if none were found (*slow path*). Too small values of k_1 led to *overcompaction*: Consider the enlarged part of the memory profile in Figure 6.11. At first, defragmentation lowers the overall memory usage. However, allocations in the next few iterations immediately increase the fragmentation level again due to new block initializations, bringing it almost back to the initial fragmentation level. A higher value of k_1 would likely speed up allocations and increase the performance of the overall application a little bit.

wa-tor This is the fish-and-sharks running example. Fish and sharks appear in waves until an equilibrium is reached [49].

This simulation experiences a slowdown from defragmentation (Figure 6.13). At $n = 10$, the slowdown is 8.1%. This slowdown is *not* due to defragmentation

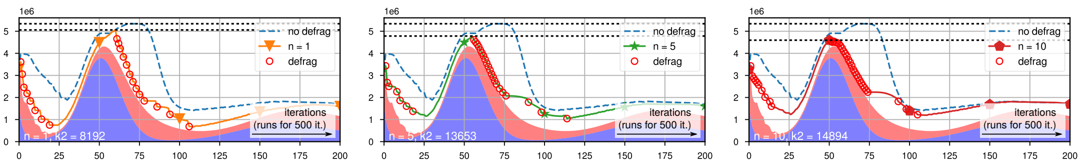


FIGURE 6.12: wa-tor: An agent-based fish-and-sharks simulation

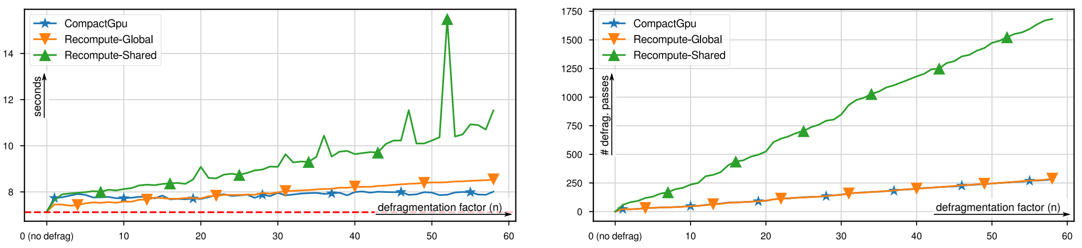


FIGURE 6.13: Running time and number of defragmentation passes for wa-tor

runtime overhead. The main reason is that COMPACTGPU is not order-preserving: Objects from a source block are scattered into multiple target blocks. This leads to less coalesced memory accesses when certain fields are accessed. Similar slowdowns have been reported on certain benchmarks in CPU systems [5]. We will further investigate this effect in future work.

The benefit of defragmentation in wa-tor is a lower memory footprint. In Figure 6.12, the dotted lines indicate the maximum memory usage throughout the simulation. Circles indicate defragmentation runs. For $n = 10$, the overall memory consumption is reduced by 14%, so that programmers can run larger problem sizes on the same hardware.

We use the *After Massive Deallocations* heuristic to initiate defragmentation. Initiating defragmentation every few iterations would incur a higher slowdown. To save memory, defragmentation is most important around iteration 50. At that time, many fish objects (red area) are deallocated and a large number of shark objects are allocated. New shark objects can reuse deallocated memory locations of fish objects as soon as the corresponding blocks are deallocated. Defragmentation eliminates many non-full fish blocks by compaction.

Pointer Rewriting Alternatives COMPACTGPU is faster than RECOMPUTE variants in most cases. The high number of bitwise operations for recomputing a memory pointer (Algorithm 21), combined with divergent execution (some pointer are rewritten, some are not) led to a high slowdown compared to our forwarding pointer method.

Furthermore, with increasing n, RECOMPUTE-SHARED requires a larger number of defragmentation passes (Figure 6.13) because the shared memory is very small, limiting the number of defragmentation candidates per pass.

For collision and structure, this slowdown is negible because very little time is spent on defragmentation overall, but we can see clear difference for generation and wa-tor.

6.5.4 Runtime Overhead

To evaluate the efficiency of our implementation, we measured the runtime overhead of COMPACTGPU. There are two kind of overheads.

Benchmark	Alloc. Size	#Rewr. Fields	n	#Defrag	#Passes	Total Runtime	Defrag	Scan	Copy	Rewrite
Synthetic (60% frag.)	2,097.2 MB	1	3	1	18	n/a	44.4	4.0	6.7	33.3
collision	5.7 MB	1	10	200	186	3,698,945	36	17	7	8
generation	57.4 MB	1	2	500	537	56,830	191	80	17	85
structure	58.9 MB	3	10	100	368	305,846	140	54	16	65
wa-tor	1,107.6 MB	1	9	38	43	7,729	49	7	14	20

TABLE 6.1: Benchmark characteristics and running time (right side; milliseconds) for selected defragmentation factors

First, COMPACTGPU extends (de)allocation procedures of DYNASOAR to maintain *defrag[T]* bitmaps (Section 6.3.1). To measure this overhead, we compare the running time without defragmentation (red dashed line) and *no defrag* values in the running time graphs. In *no defrag*, we maintain a defragmentation candidate bitmap but never initiate defragmentation. There is almost no measurable overhead for maintaining these bitmaps.

Second, COMPACTGPU has three potentially expensive steps: (a) Generating/compacting an indices array R from a defragmentation candidate bitmap (*scan*), (b) copying objects and placing forwarding pointers (*copy*) and (c) scanning the heap and rewriting pointers (*rewrite*). In Table 6.1, we show the time spent in each step, as well as the overall time spent on defragmentation (*defrag*), which includes additional overheads such as block state updates. If *#Passes* < *#Defrag*, the programmer initiated defragmentation but there were not enough defragmentation candidates to start a defragmentation pass.

In every benchmark, defragmentation takes only a very small fraction of the overall application running time. wa-tor has the largest overhead: The application spends 0.6% of its running time in defragmentation.

Synthetic Benchmark The synthetic benchmark isolates the runtime overhead for one defragmentation. We added a second class to the benchmark and made objects of both classes point to each other randomly. There are initially 32,768,000 objects of each class (object size 32 bytes). The benchmark deletes 60% of the objects of one class and then initiates defragmentation.

The performance of the *scan* phases mainly depends on the efficiency of the prefix sum operations (CUB library) and can thus not be further optimized.

The *copy* phases copy (read+write) 282.1 MB of object data and write 70.5 MB of forwarding pointers in 6.7 milliseconds (94.7 GB/s). This is 17.3% of the global memory bandwidth of our TITAN Xp GPU.

The *rewrite* step is most time consuming: COMPACTGPU has to check 32,768,000 pointers (262.1 MB) per pass (18 $\times$ 262.1 MB = 4,717.2 MB in total). Out of these pointers, COMPACTGPU rewrites only a small part (read+write 70.3 MB of forwarding pointers) because many objects were deleted. COMPACTGPU finishes the rewrite step in 33.3 milliseconds, resulting in a memory transfer rate of 145.9 GB/s (not taking into account other memory accesses). This is 26.6% of the global memory bandwidth of our TITAN Xp GPU. The Nvidia Profiler shows that 32% of all global memory accesses in this step hit the L1 cache (64 KB) and 63% hit the L2 cache (3,072 KB), indicating that defragmentation candidate bitmaps are largely cached.

COMPACTGPU achieves a high performance because most memory reads/writes have good coalescing. Overall, our benchmark results show that COMPACTGPU is highly optimized with little room for improvement.

6.6 Related Work

A vast number of memory defragmentation systems have been developed for CPU systems in the past. A main difference on GPU architectures is that it is easier to decide where to relocate objects to, because there are only a small number of object sizes. This pattern is reflected in the design of many GPU dynamic memory allocators: Many allocators maintain containers for objects of the same size [7, 71]. On CPU systems, there are typically many different allocation sizes.

The only existing GPU memory defragmentation system was developed by Veldema and Philippsen [187]. Their work consists of an allocator and a defragmentation system[10]. To compact the memory, their defragmentation system selects 10% of all memory regions that are less than 75% full as source regions. They use their memory allocator to allocate a target location in another region. This is problematic because allocation is expensive and requires some sort of synchronization between threads. Runtime overheads of their defragmentation system range from 0.5% to 33%, higher than the overhead of COMPACTGPU.

Veldema and Philippsen also propose a technique for limiting the search space during pointer rewriting based on additional data collected by a garbage collector. This technique could be used in COMPACTGPU instead of relying on class structure metainformation of DYNASOAR.

To the best of our knowledge, there are no other defragmentation systems for GPUs. We believe that this is because of limited support for dynamic memory allocation. The default CUDA dynamic memory allocator is known to be slow and unreliable [175], so most programmers avoid dynamic memory management entirely. It is still a common practice to allocate a large chunk of memory statically and manage it manually. Out of the few custom memory allocators that exist, many (e.g., ScatterAlloc [175], Halloc [7]) use a hashing approach to scatter allocations in the heap almost randomly, in order to avoid collisions among allocating threads. Not only do they miss important opportunities for vectorization (e.g., SOA layout), but they are also known to incur the negative effects of high fragmentation [7].

Many efficient CPU memory defragmentation systems divide the heap into two areas: Objects are copied from a *from-space* to a *to-space* [101, 111]. Both spaces are swapped before every defragmentation pass. In such an approach, only half of the memory space is usable by the allocator. This is acceptable on virtual memory architectures because the virtual memory space is much larger than the physical memory space. Current GPU architectures do not have virtual memory and even the amount of physical memory is much smaller than on CPU systems. Cutting the available memory by half would be unacceptable on GPUs.

Pointer Rewriting without Forwarding Pointers Some memory defragmentation systems use data structures other than forwarding pointers [5, 101]. For example, the *Compressor* uses a markbit vector to recompute forwarding pointers on-the-fly during pointer rewriting [101]. A markbit vector is a bit vector where bits for the first and last heap word of an allocated object are set. Since forwarding pointers are not read from memory, only two accesses (read pointer, replace with new pointer) are required to rewrite a pointer, assuming the markbit vector is cached. We experimented with similar techniques (RECOMPUTE-GLOBAL, RECOMPUTE-SHARED; Section 6.4), but they did not lead to a performance improvement.

[10]The source code of this system is not available, so we could not compare it with COMPACTGPU.

6.7 Conclusion

We presented CompactGpu, a memory defragmentation system for GPUs. CompactGpu is able to (a) speed up applications through better cache utilization and vector load/store efficiency on allocated memory and (b) lower the overall memory consumption of an application.

CompactGpu achieves low runtime overheads through careful SIMD-friendly design considerations and implementation efforts: CompactGpu utilizes bitmaps to select source/ target blocks and to quickly decide if a pointer must be rewritten. Furthermore, CompactGpu exhibits mostly regular control flow, accesses memory in coalescing-friendly patterns and requires no synchronization between threads.

Our main takeaways are that (a) memory defragmentation on GPUs is feasible and able to deliver speedups, (b) too much defragmentation does not pay off (due to overfitting and overcompaction) and can even be detrimental to performance due to less efficient allocations, and (c) careful design considerations are necessary to achieve good performance on GPUs; many good CPU designs, such as recomputing forwarding pointers on-the-fly, are not efficient on GPUs.

Chapter 7

SMMO Examples

In this chapter, we present examples of Single-Method Multiple-Objects (SMMO) applications. We used these applications to evaluate DynaSOAr and CompactGpu. We show the data structure of each application and highlight their SMMO structure, i.e., which parallel do-all operations they consist of.

Contents

Benefits of Object-oriented Programming To analyze the performance of DynaSOAr, we implemented most SMMO applications with and without dynamic memory allocation (*baseline* versions; Section 5.6). SOA baselines store objects in a hand-written SOA data layout (Listing 2.5). We no longer consider such a layout as *object-oriented* because C++ abstractions for object-oriented programming cannot be used. We noticed that missing OOP abstractions made the development of the SOA baseline versions more tedious than their counterpart versions that utilize the DynaSOAr data layout DSL and dynamic memory management, in particular:

No Type Safety With respect to typing, the implementation of SOA baselines felt like programming in an untyped/dynamically-typed language because all object references are of type *integer*. This has two disadvantages. First, many programming errors are not caught by the type checker at compile time, which slowed down the development process. To make matters worse, some programming errors do not even cause the program to crash at runtime, but simply produce a wrong result (Listing 7.1 and 7.2). Second, types are also a form of code documentation and their absence makes code harder to read and understand [58].

No Dynamic Memory Allocation Many applications are more difficult to implement without dynamic allocation. Category 2 applications require an additional boolean field to keep track of active/allocated objects (Section 5.6). Category 3 applications required changes to the data structures within the application, which breaks abstractions.

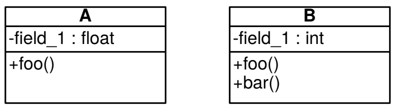


FIGURE 7.1: Example: Dummy classes

LISTING 7.1: Example: Dummy classes with hand-written SOA layout (no OOP)

```
float A_field_1[10000];
int B_field_1[10000];

void A_foo(int id) { printf("A: %f\n", A_field_1[id]); }
void B_foo(int id) { printf("B: %d\n", B_field_1[id]); }
void B_bar(int id) { printf("B_bar: %d\n", B_field_1[id]); }

int get_A() { /* Return ID of an object of type A */ }

int main() {
  A_foo(get_A()); // OK
  B_foo(get_A()); // Typo: B_foo instead of A_foo. But code compiles and runs.
  B_bar(get_A()); // Calling opeation of B on A. But code compiles and runs.
}
```

LISTING 7.2: Example: Dummy classes in AOS layout (with OOP)

```
class A {
 public:
  float field_1;
  void foo() { printf("A: %f\n", field_1); }
};
A objects_A[10000];

class B {
 public:
  int field_1;
  void foo() { printf("A: %d\n", field_1); }
  void bar() { printf("B_bar: %d\n", field_1[id]); }
};
B objects_B[10000];

A* get_A() { /* Return pointer to an object of type A */ }

int main() {
  get_A()->foo(); // OK
  get_A()->foo(); // It is impossible to make the same mistake as in Listing 7.1.
  get_A()->bar(); // Compile error
}
```

Notation of Chained Field Accesses Some applications exhibit a pattern of chained field accesses in their source code, e.g.: `obj->f1->f2`. The order in which field name tokens appear in hand-written SOA code is inversed and counter-intuitive: `B_f2[A_f1[obj]]`. Moreover, this notation requires programmers to repeat the type/class of objects/entities (`A` and `B` in this example).

In this chapter, we present the design and implementation of various SMMO applications with DynaSOAr. These implementations are object-oriented and do not suffer from the above mentioned shortcomings. To further highlight the benefits of object-oriented programming, we discuss interesting design and implementation choices of SOA baseline versions for certain SMMO applications.

Publications This chapter is in part based on the following papers.

- Matthias Springer, Hidehiko Masuhara. **"DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access (Artifact)."** In: *Dagstuhl Artifacts Series.* Vol. 5, Iss. 2, Art. 2. Leibniz-Zentrum für Informatik, Dagstuhl Publishing, 2019. `doi:10.4230/DARTS.5.2.2`.

7.1 nbody: N-body Simulation

`nbody` is a 2D particle system simulation. Such simulations are used by astronomers to simulate the collision of galaxies or the formation of planets [9]. This example is a very simple SMMO application with only one class. The entire application consists of only around 100 lines of code.

`nbody` simulates a large number of bodies. Each body has a position, velocity and mass. According to Newton's theory of gravity, two bodies with a masses m_1 and m_2 and a distance of r pull each other closer with a gravitational force F.

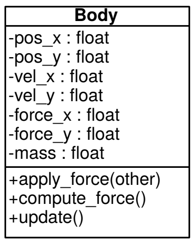
$$F = G\frac{m_1 m_2}{r^2} \qquad \textit{(gravitational force)}$$

The goal of `nbody` is to calculate the future position and velocity of all bodies. `nbody` is an iterative algorithm: Each iteration advances the simulation by a small time step Δt. We assume that the gravitational forces remain constant during an iteration, so we can compute each body's new velocity and position as follows.

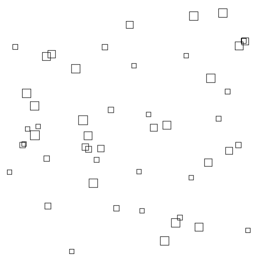
$$\begin{aligned} v &\leftarrow v + \frac{F}{m} \cdot \Delta t \qquad \textit{(simulation time step)} \\ x &\leftarrow x + v \cdot \Delta t \end{aligned}$$

7.1.1 Data Structure

Every body is an object of class `Body` (Figure 7.2, Listing 7.3). This class has fields for position, velocity, mass, as well as helper variables for accumulating the force that acts on the body.



(A) Data structure



(B) Screenshot

FIGURE 7.2: Data structure and screenshot of nbody. Bodies pull each other closer with graviational force. Apart from the gravitational force, there is no interaction between bodies, making this example the simplest one in this chapter.

LISTING 7.3: Data structure of nbody

```
#include "dynasoar.h"

class Body;
using AllocatorT = SoaAllocator<kNumObjects, Body>;

class Body : public AllocatorT::Base {
 public:
  declare_field_types(
      Body,
      float, // pos_x_
      float, // pos_y_
      float, // vel_x_
      float, // vel_y_
      float, // force_x_
      float, // force_y_
      float) // mass_

 private:
  Field<Body, 0> pos_x_;
  Field<Body, 1> pos_y_;
  Field<Body, 2> vel_x_;
  Field<Body, 3> vel_y_;
  Field<Body, 4> force_x_;
  Field<Body, 5> force_y_;
  Field<Body, 6> mass_;

 public:
  __device__ Body(float pos_x, float pos_y, float vel_x, float vel_y, float mass);

  // Constructor for parallel_new.
  __device__ Body(int index);

  __device__ void apply_force(Body* other);

  __device__ void compute_force();

  __device__ void update();
};
```

7.1.2 Application Implementation

This application consists of two parallel do-all operations and one parallel new operation (Listing 7.5). All member functions of `Body` are device functions (annotated with CUDA keyword `__device__`) because they are executed on the GPU.

1. **Initialization:** Create a few random `Body` objects. Parallel new: `Body`.
2. **Iterative Algorithm:** Each iteration consists of the following steps.
   (a) **Computing Forces:** Compute forces between all pairs of bodies with a nested (sequential) do-all iteration. Parallel do-all: `Body::compute_force`.
   (b) **Acceleration and Movement:** Compute a new velocity and position for each body. Parallel do-all: `Body::update`.

In the following paragraphs, we describe each step of this simulation in more detail.

Allocator Initialization Before the actual application code begins, we have to create an allocator. We first create a new *allocator handle* which can be used from host (CPU) code. This internally allocates a large chunk of global GPU memory which contains the heap of our allocator. We then store a device allocator pointer in variable `device_allocator`. This variable is a device variable, so it is only visible from GPU code. We use this pointer to interact with DynaSOAr from GPU code. Every DynaSOAr program initializes the memory allocator in this way, so we will skip over this part in the remaining examples.

Step 1: Simulation Initialization At the beginning, we initialize the simulation with 65,536 random `Body` objects. The parallel new operation invokes the second constructor of `Body` in parallel, passing an index value between 0 and 65,535 as argument. This constructor initializes the position, velocity and mass of the `Body` object with random values using the cuRAND library.

Step 2: Iterative Algorithm The simulation invokes two parallel do-all operations iteratively. The first method `compute_force()` computes the total force that is acting on a body and stores it in the fields `force_x_` and `force_y_`. This requires a nested *for* loop because we have to compute and sum the forces between all pairs of bodies. In DynaSOAr, we express such a nested *for* loop with `device_do`, which is similar to `parallel_do` but runs sequentially.

Listing 7.4 illustrates this nested loop structure. The outer loop is a CUDA kernel and runs in parallel. The inner loop runs sequential. While the implementation of `b2->apply_force(b1)` could conceptually sum the force in either `b1` or `b2` (the same force acts in both directions), it has to modify `b1` to avoid race conditions. If `apply_force` were to modify `b2`, then there would be multiple GPU threads modifying `b2` concurrently, because the outer loop runs in parallel. To avoid race conditions, this would require an atomic add operation. Instead, `apply_force` modifies the force of `b1`, which is now only accessed by one GPU thread.

The second method `update()` computes a new velocity and position based on the force value that was computed by the first method. If a body goes out of bounds of the simulation space, we invert its velocity. This only for visualization reasons; a real n-body simulation would not do this.

LISTING 7.4: Nested loop structure (conceptually)

```
for (Body* b1 : AllocatorT::all_objects<Body>) { // parallel_do
  for (Body* b2 : AllocatorT::all_objects<Body>) { // device_do
    // Compute gravitational force between b1 and b2. Accumulate the total
    // gravitational force (of all bodies) in force_x_ and force_y_ values.
    // But should apply_force modify b1 or b2?
    b2->apply_force(b1); // Or: b1->apply_force(b2)
  }
}
```

Both methods run in separate parallel do-all operations (and thus separate CUDA kernels) to ensure that the simulation is free of race conditions and to ensure that the application produces the same result on every run, assuming the random number generator is initialized with the same seed. If both methods were to run in one CUDA kernel, the implemention of `apply_force` may read an updated or not-yet updated position of another body, depending on the scheduling of threads.

7.1.3 Further Optimizations

Related work describes three additional techniques to further optimize this n-body simulation [146]. To keep this benchmark simple, these optimizations are not implemented in our n-body simulation.

- **Shared Memory:** Since we compute forces between all pairs of bodies, every CUDA thread reads the position and mass fields of all bodies. To reduce the amount of data read from global memory, we can read those fields only once per CUDA block and store the values in shared memory.
- **Nested Parallelism:** Instead of parallelizing only the outer *for* loop, also partly parallelize the inner *for* loop. This technique is particular useful for improving occupancy if the number of bodies is small.
- **Loop Unrolling:** Especially on older GPU architectures, unrolling the inner *for* loop can improve instruction-level parallelism (ILP) and thus better utilize the GPU's resources (e.g., through *dual-issue*, Section 2.1.1).

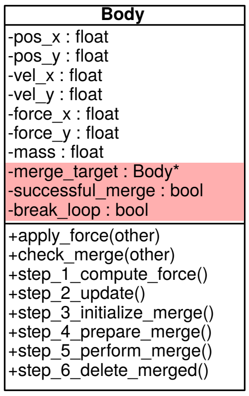
To be able to implement these optimizations, we have to extend DYNASOAR's API in future work. In particular, there is currently no way of unrolling `device_do` iterations. This could be simplified if range-based *for* loops could be used in lieu of `device_do` iterations (Section 7.2.1). Moreover, the current API makes it difficult to partly parallelize object enumeration. In essence, what we require is a mixture of `parallel_do` and `device_do`. Such an API and its implementation could be inspired by virtual warp-centric programming [86].

7.2 collision: N-Body Simulation with Collisions

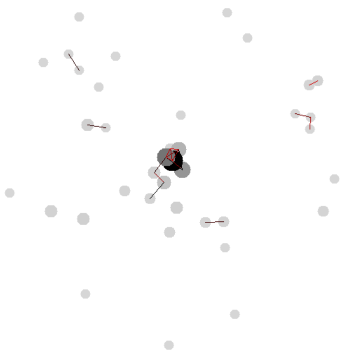
We now extend the n-body simulation of Section 7.1 with collisions. Two bodies are merged into one large body if their distance is below a certain threshold. This example is interesting because it exhibits more complex object interactions and stores a pointer to another object in a field, as opposed to only primitively-typed values in the n-body simulation of Section 7.1.

LISTING 7.5: Application logic of nbody

```
// Allocator handles.
AllocatorHandle<AllocatorT>* allocator_handle;
__device__ AllocatorT* device_allocator;

__device__ Body::Body(float pos_x, float pos_y,
                      float vel_x, float vel_y, float mass)
    : pos_x_(pos_x), pos_y_(pos_y), vel_x_(vel_x), vel_y_(vel_y), mass_(mass) {}

__device__ Body::Body(int index) {
  curandState rand_state;
  curand_init(kSeed, index, 0, &rand_state);
  // Initialize with random float between -1 and 1.
  pos_x_ = 2 * curand_uniform(&rand_state) - 1;
  /* Similarly for the pos_y_, vel_x_, vel_y_, mass_... */
}

__device__ void Body::apply_force(Body* other) {
  if (other != this) {
    // To avoid race conditions: Update other instead of this.
    float dx = pos_x_ - other->pos_x_;
    float dy = pos_y_ - other->pos_y_;
    float dist = sqrt(dx*dx + dy*dy);
    float F = kGravityConstant * mass_ * other->mass_ / (dist * dist);
    other->force_x_ += F*dx / dist;
    other->force_y_ += F*dy / dist;
  }
}

__device__ void Body::compute_force() {
  force_x_ = force_y_ = 0.0f;
  device_allocator->device_do<Body>(&Body::apply_force, this);
}

__device__ void Body::update() {
  vel_x_ += force_x_ * kDt / mass_;
  vel_y_ += force_y_ * kDt / mass_;
  pos_x_ += vel_x_ * kDt;
  pos_y_ += vel_y_ * kDt;

  // Bounce off the walls.
  if (pos_x_ < -1 || pos_x_ > 1) { vel_x_ = -vel_x_; }
  if (pos_y_ < -1 || pos_y_ > 1) { vel_y_ = -vel_y_; }
}

int main() {
  // Create new allocator.
  allocator_handle = new AllocatorHandle<AllocatorT>();
  AllocatorT* dev_ptr = allocator_handle->device_pointer();
  cudaMemcpyToSymbol(device_allocator, &dev_ptr, sizeof(AllocatorT*), 0,
                     cudaMemcpyHostToDevice);

  // Create 65536 new Body objects.
  allocator_handle->parallel_new<Body>(65536);

  // Simulation loop.
  for (int i = 0; i < kNumIterations; ++i) {
    allocator_handle->parallel_do<Body, &Body::compute_force>();
    allocator_handle->parallel_do<Body, &Body::update>
  }
}
```



(A) Data structure



(B) Screenshot

FIGURE 7.3: Data structure and screenshot of collision, an extension of nbody. In the screenshot, the darkness of a body indicates its mass. When bodies are getting close, we connect them with lines. A red line indicates that two bodies are almost within merging distance.

This simulation is an extension of the previous n-body simulation. It computes forces between bodies and accelerates/moves bodies in the same way. However, two bodies are merged according to the physical law of perfectly inelastic collision if they become too close. In that case, the lighter body b_2 is merged into the heavier one b_1 ($m_1 > m_2$). According to the law of perfectly inelastic collision, the new velocity of the heavier body b_1 is the weighted sum of the velocities of both bodies.

$$\begin{aligned} v_1 &\leftarrow \frac{v_1 \cdot m_1 + v_2 \cdot m_2}{m_1 + m_2} \\ x_1 &\leftarrow \frac{x_1 + x_2}{2} \\ m_1 &\leftarrow m_1 + m_2 \end{aligned} \qquad (\textit{perfectly inelastic collision})$$

7.2.1 Data Structure

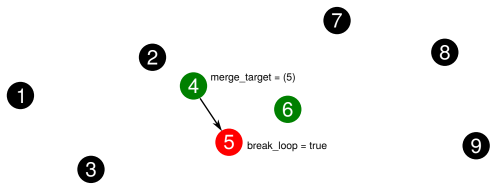
We added three fields to class `Body` to implement body merging semantics (Figure 7.3, red color). `merge_target` points to the `Body` object into which a given body should be merged. `successful_merge` is set to *true* if a body was successfully merged and can be deleted. `break_loop` is another boolean flag for breaking out of a `device_do` iteration. This flag is necessary because the C++ `break` keyword, which is used to break out of loops, cannot be used with `device_do` iterations.

We will provide a C++ iterator for enumerating allocated objects in future versions of DYNASOAR. Programmers can than use range-based *for* loops (Listing 7.7) instead of `device_do`. In that case, the field `break_loop` will no longer be necessary.

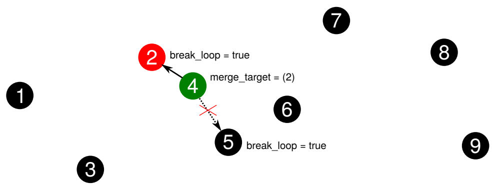
7.2.2 Application Implementation

The initialization of the simulation is identical to the previous n-body simulation and omitted in this section. The actual simulation invokes six parallel do-all operations iteratively. The first two parallel do-all operations are identical to the previous n-body simulation.

1. **Compute Forces:** Compute and accumulate the force exerted by all other bodies on a given body. Parallel do-all: `Body::step_1_compute_force`.



(A) b_5 selects b_4 to merge.



(B) Afterwards, b_2 also selects b_4.

FIGURE 7.4: Preparing a body merge operation

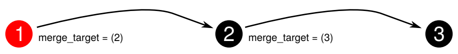


FIGURE 7.5: Merge target being merged itself

2. **Acceleration and Movement:** Accelerate and move a body. Parallel do-all: `Body::step_2_update`.
3. **Reset Merge Fields:** Initialize the three new fields that will be used during merging. Parallel do-all: `Body::step_3_initialize_merge`.
   (a) `merge_target` $\leftarrow$ `nullptr`: No merge target was selected yet for this body.
   (b) `successful_merge` $\leftarrow$ `false`: This body was not merged yet.
   (c) `break_loop` $\leftarrow$ `false`
4. **Prepare Merge:** For a given body, check if and which other body can be merged into it. This step implements a *pull* semantics: Instead of looking for body into which a given body can be merged, we are looking for a body that can be merged into a given body. Parallel do-all: `Body::step_4_prepare_merge`.
5. **Perform Merge:** If a merge target was selected for a given body in the previous step, perform the perfectly inelastic collision. This step implements a *push* semantics. Parallel do-all: `Body::step_5_update_merge`.
6. **Delete Body:** If a given body was merged in the previous step, delete it. Parallel do-all: `Body::step_6_delete_merged`.

Steps 4 and 5 (Listing 7.6) are the interesting ones and we will describe them in more detail in the following paragraphs.

Step 4: Prepare Merge This method determines if another body should be merged into a given body. A body b_y can be merged into b_x if the distance between b_y and b_x is below a certain threshold and if $m_x > m_y$. This method operates from the perspective of a *receiving* body b_x (*pull* semantics) and iterates over all other bodies (`device_do`) to find a body that satisfies these requirements. If such a body b_y was found, the merge target of b_y is set to b_x. Furthermore, the `break_loop` flag of b_x is set. This does not really stop the loop, but future iterations of `check_merge` will immediately return.

Consider the example in Figure 7.4a. We observe the process from the perspective of the red body b_5, i.e., `step_4_prepare_merge` is bound to b_5. The green bodies are within merging range. b_5 checks b_4 and decides to merge it into itself. The `device_do` loop breaks at this point and b_5 does not even consider b_6, which is also in range.

Since `step_4_prepare_merge` runs as a parallel do-all operation, multiple bodies are concurrently looking for merge partners. In Figure 7.4b, b_2 selects the same body

LISTING 7.6: Additional application logic of collision

```
__device__ void Body::check_merge(Body* other) {
  // Only merge into larger body.
  if (!other->break_loop_ && mass_ < other->mass_) {
    float dx = pos_x_ - other->pos_x_;
    float dy = pos_y_ - other->pos_y_;
    float dist_square = dx * dx + dy * dy;

    if (dist_square < kMergeThreshold * kMergeThreshold) {
      // Try to merge this into other. There is a race condition here:
      // Multiple threads may try to merge this body. Only one can win.
      this->merge_target_ = other;
      other->break_loop_ = true;
    }
  }
}

__device__ void Body::step_4_prepare_merge() {
  device_allocator->device_do<Body>(&Body::check_merge, this);
}

__device__ void Body::step_5_update_merge() {
  Body* m = merge_target_;
  if (m != nullptr) {
    if (m->merge_target_ == nullptr) {
      // Perform merge.
      float new_mass = mass_ + m->mass_;
      float new_vel_x = (vel_x_ * mass_ + m->vel_x_ * m->mass_) / new_mass;
      float new_vel_y = (vel_y_ * mass_ + m->vel_y_ * m->mass_) / new_mass;
      m->mass_ = new_mass;
      m->vel_x_ = new_vel_x;
      m->vel_y_ = new_vel_y;
      m->pos_x_ = (pos_x_ + m->pos_x_) / 2;
      m->pos_y_ = (pos_y_ + m->pos_y_) / 2;

      successful_merge_ = true;
    }
  }
}

__device__ void Body::step_6_delete_merged() {
  if (successful_merge_) { destroy(device_allocator, this); }
}
```

LISTING 7.7: Alternative: C++ range-based *for* loop instead of `device_do`

```
__device__ void Body::step_4_prepare_merge() {
  for (Body& other : device_allocator->iterator<Body>()) {
    if (other.mass_ < mass_) {
      float dx = other.pos_x_ - pos_x_;
      float dy = other.pos_y_ - pos_y_;
      float dist_square = dx * dx + dy * dy;

      if (dist_square < kMergeThreshold * kMergeThreshold) {
        other.merge_target_ = this;
        break;
      }
    }
  }
}
```

b_4 as b_5 and writes a pointer to itself into `merge_target`. This is a race condition: Which body b_4 will be merged into depends on thread scheduling. In this example, b_2 wins. Note that b_5 cannot select a new body now because it already decided to break from the loop. It would be difficult to let b_5 select a new body in such cases. First, b_5 may already be done executing the `device_do`, so this would potentially require another `device_do` iteration. Second, b_5 is not even guaranteed to notice that `merge_target` was overwritten by another thread because GPU caches are not coherent.

We accept the race condition in this step. If the merging threshold is sufficiently small, then such race conditions are less likely to appear. Furthermore, even if a body is not merged due to such a race condition, it is likely getting merged in one of the next iterations, because a body will likely spend multiple iterations within the merging threshold.

Step 5: Perform Merge This method performs a merge of a given body in case a merge target was selected in the previous step. It operates from the perspective of a *giving* body (*push* semantics).

There is another potential race condition in this method. Consider the example in Figure 7.5. b_1 should be merged into b_2 and b_2 should be merged into b_3. Since `step_5_perform_merge` runs as a parallel do-all operation, both bodies are merged concurrently. This is problematic because the thread of b_1 writes fields of b_2 and the thread of t_2 reads those fields concurrently. Depending on the thread scheduling, the thread of t_2 may read the original or updated values. This race condition is more problematic than the one in the previous step because a body could effectively disappear from the simulation.

To avoid this race condition, we merge b_1 into b_2 only if b_2 does not have a merge target itself. After the merge, the `successful_merge` flag of b_1 is set to *true*, so that it will be deleted from the simulation in `step_6_delete_merged`.

7.2.3 Benefits of Object-oriented Implementation

We would like to highlight two benefits of object-oriented programming in this implementation.

- **Field Access Notation:** C++'s object field access notation is more readable compared to a hand-written SOA layout. For example, Listing 7.8 shows Line 27 of Listing 7.6 in a hand-written SOA layout. There are two problems with this notation: First, we cannot use C++'s member access (arrow) operator. Second, we have to specify an index for each SOA array access, whereas methods are bound to an object and field accesses without an explictly specified object refer to the bound object (`this/id`).

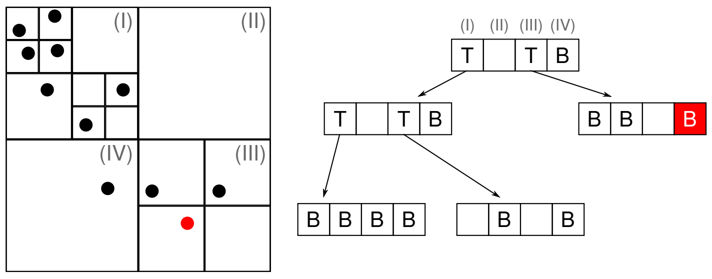
- **Active Flag:** Objects that were deleted (Listing 7.6, Line 41) are no longer enumerated by subsequent parallel do-all operations. To implement the same semantics without dynamic memory allocation (baselines AOS/SOA), we had to add an extra field `active` to class `Body`, which is initially *true* but later set to *false* if the `Body` object was merged. CUDA kernels process only active objects.

7.2.4 Further Optimizations

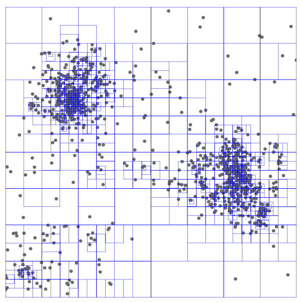
The most time-consuming steps of this n-body simulation are *computing forces* (Step 1) and *finding merge partners* (Step 4). Both steps are implemented with a `device_do`

LISTING 7.8: Field access notation in hand-written SOA layout

```
float new_vel_x = (Body_vel_x[id] * Body_mass[id] + Body_vel_x[m] * Body_vel_x[m]) ↩
    / new_mass;
```



(A) Quad tree structure



(B) Screenshot

FIGURE 7.6: Barnes-Hut quad tree

iteration, resulting in N^2 body-body computations, where N is the number of bodies. Step 1 can be approximated with the Barnes-Hut algorithm (Section 7.3). Step 4 is a common problem in physical simulations and could be optimized with spatial subdivision [112].

7.3 barnes-hut: Approximating N-Body with a Quad Tree

Barnes-Hut [17] is an approximation of the n-body simulation in Section 7.1. N-body is computationally expensive because it computes gravitational forces between all pairs of Body objects. Barnes-Hut recursively divides the simulation space into quadrants by building a quad tree (Figure 7.6). If a body is far enough away from a quadrant, the force of the quadrant on the body can be approximated by treating the entire quadrant as a single larger body instead of computing exact forces with every body inside the quadrant. The device_do iteration of the original n-body simulation is then replaced with a quad tree traversal. We are designing a 2D n-body simulation in this section. A 3D n-body simulation would utilize an octree instead of a quad tree.

7.3.1 Data Structure

This application has three classes. An abstract class NodeBase and two subclasses BodyNode and TreeNode.

The tree is made up of TreeNodes. Every TreeNode stores up to four children in the children array field. This array has one slot per quadrant (e.g., quadrant 1 = index 0). A child can be either another TreeNode or a BodyNode. Furthermore, every node stores a pointer to its parent TreeNode. child_idx is the position of the node within its parent's children array. The root of the tree has no parent (nullptr).

Every node has a position and a mass. In case of a BodyNode, this is the position and mass of the body. In case of a TreeNode, this is the center of gravity and accumulated mass of all bodies in the subtree (*summary*). As the position of bodies changes throughout the simulation, their position within the tree also changes. Therefore, tree summaries must be recomputed from time to time.

TreeNodes store two additional coordinates: p1 is the left upper corner and p2 is the lower right corner of the node. Moreover, TreeNodes have two boolean flags

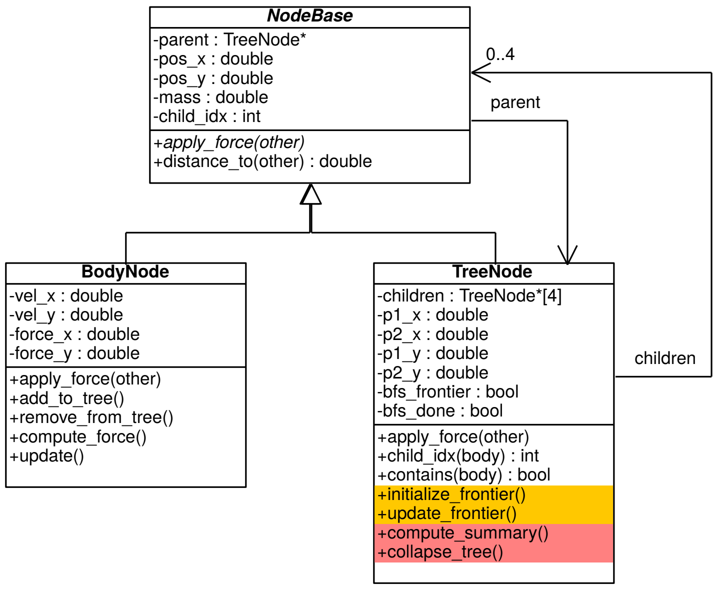


FIGURE 7.7: Data structure of barnes-hut

`bfs_frontier` and `bfs_done` that are used to implement bottom-up tree traversals, as described later. These field names start with `bfs_` because the tree traversal is conceptually similar to the parallel frontier-based breadth-first search algorithm.

7.3.2 Application Implementation

This simulation consists of a larger number of parallel do-all operations. We first give a high-level overview of the simulation and then discuss selected do-all operations in more detail in the following paragraphs.

1. **Tree Initialization:** Create the root of the tree (`TreeNode` object) and store it in a top-level variable `tree`. The tree has no children yet at this point.
2. **Body Initialization:** Create a few random `BodyNode` objects with a parallel new operation, similar to the previous simulations. These bodies are not yet part of the tree, as indicated by `parent = nullptr`.
3. **Inserting Bodies:** Insert those bodies into the tree that are not yet part of the tree (all bodies at this point). Parallel do-all: `BodyNode::add_to_tree`.
4. **Iterative Algorithm:** Each iteration consists of the following steps.
   (a) **Computing Summaries:** Compute `TreeNode` summaries with a bottom-up tree traversal. Multiple parallel do-all operations.
   (b) **Computing Forces:** Compute forces between bodies with top-down tree traversal. Parallel do-all: `BodyNode::compute_force`.
   (c) **Acceleration and Movement:** Compute a new velocity and position for each body. This step is identical to the second step of the original n-body simulation. Parallel do-all: `BodyNode::update`.
   (d) **Removing Bodies:** Remove bodies from the tree if they moved into a different quadrant. Parallel do-all: `BodyNode::remove_from_tree`.
   (e) **Reinserting Bodies:** Insert those bodies back into the tree that were just removed. Parallel do-all: `BodyNode::add_to_tree`.

(f) **Collapsing Tree:** Remove empty `TreeNode`s and collapse `TreeNode`s with only one `BodyNode` child. This is another bottom-up tree traversal and implemented with multiple parallel do-all operations.

In the remainder of this section, we describe the design and implementation of all steps that require tree traversals or tree modifications. The latter ones are difficult to implement because multiple threads may be concurrently modifying the tree.

Step 4a: Computing Tree Summaries Before gravitational forces can be computed, we have to ensure that tree summaries are up to date. Step 4a and Step 4f are both implemented with a bottom-up tree traversal. Listing 7.9 shows the structure of such traversals.

Many parallel tree/graph traversals (e.g., BFS) are iterative, frontier-based algorithms. In such a traversal, each node has a boolean *frontier flag* which indicates if a node is part of the frontier. Only frontier nodes are processed in an iteration. When all frontier nodes finished processing, the frontier is advanced, usually based on the results of the processing step. Alternatively, instead of a boolean flag, the frontier is sometimes defined as all nodes that have a certain property; e.g., all nodes with a certain distance (Section 4.3.2).

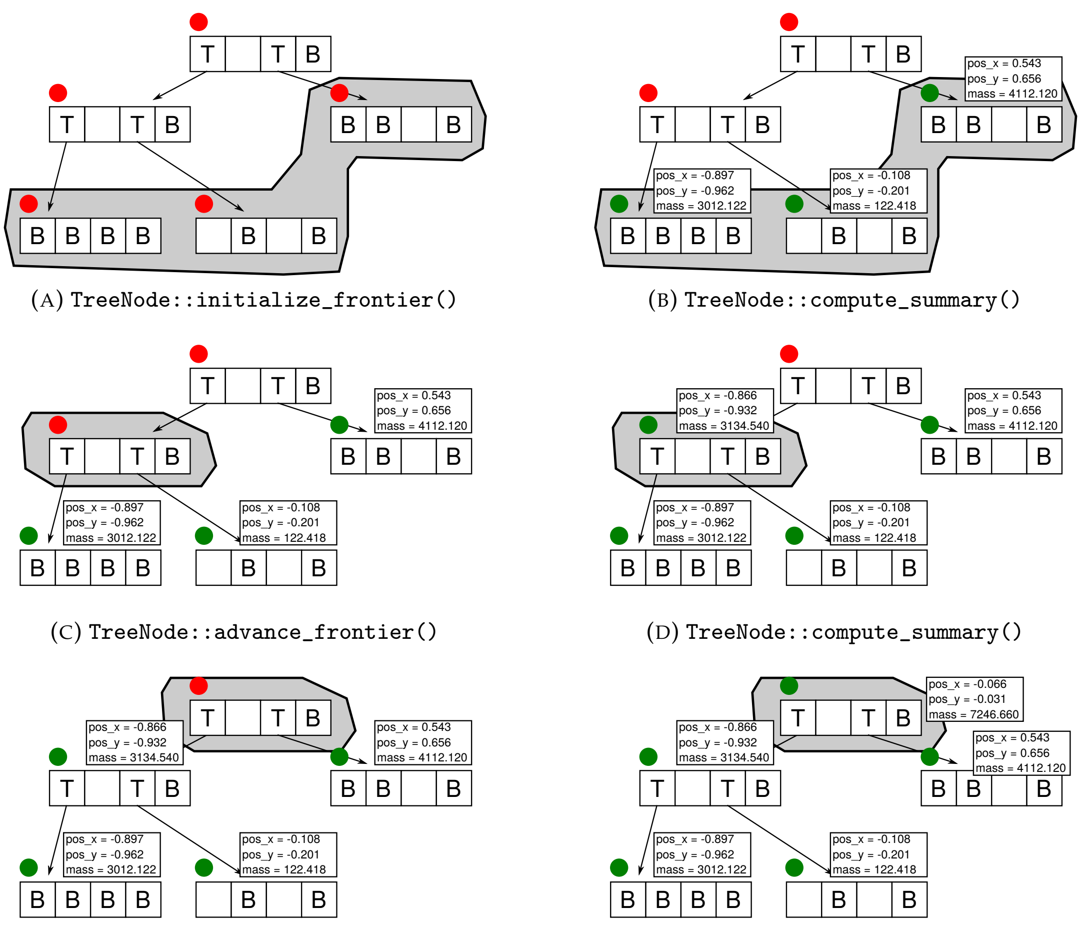
In our bottom-up tree traversal, every node has two flags: A boolean frontier flag `bfs_frontier` and a boolean *done flag* `bfs_done` which is set to *true* after processing to ensure that a node is not processed multiple times. Our traversal algorithm visits only `TreeNode`s. All `TreeNode`s without `TreeNode` children (tree leaves) are part of the initial frontier. A processing step computes tree summaries for all frontier nodes and sets their done flag to *true*. If a `TreeNode` has not been processed yet but all of its children are done processing, it will become part of the next frontier. As a side note, this is a common pattern of parallel graph processing and fits well with the vertex-based *bulk-synchronous model* [186], which is the foundation of graph processing frameworks such as Gunrock [197].

Figure 7.8 illustrates this algorithm with an example. The circles indicate the state of the `bfs_done` flag (red = *false*, green = *true*). Initially, only nodes without `TreeNode` children are part of the frontier (Subfigure A), as indicated by the shaded area. The first processing step computes tree node summaries of those nodes and sets their done flag (Subfigure B). Then, the frontier is advanced: Only the left middle node becomes part of the next frontier because it is the only unprocessed node whose children are all processed (Subfigure C). The algorithm proceeds in this pattern until the tree root was processed (Subfigure F).

Step 4b: Computing Forces Barnes-Hut is faster than a regular n-body simulation because the computation of gravitational forces has a lower computational complexity. In n-body, the total force of all other bodies acting on a given body is computed with a `device_do` operation (complexity $\theta(n)$, where n is the number of bodies). In Barnes-Hut, this force is computed with a top-down tree traversal, which has the same worst-case complexity but a lower expected complexity. Our implementation uses a *preorder depth-first traversal*, but other traversals would also work.

Listing 7.10 shows how gravitational forces are computed with a quad tree traversal. We assume that all `TreeNode` summaries (i.e., fields `pos_x`, `pos_y`, `mass`) are up to date. Instead of a `device_do`, we start the tree traversal at the root of the tree (Line 21).

`NodeBase::apply_force` is a pure virtual function. Let `this` be the object that the function is bound to. If `this` is a `BodyNode`, the function adds the gravitational



(A) `TreeNode::initialize_frontier()` (B) `TreeNode::compute_summary()`

(C) `TreeNode::advance_frontier()` (D) `TreeNode::compute_summary()`

(E) `TreeNode::advance_frontier()` (F) `TreeNode::compute_summary()`

FIGURE 7.8: Compute summaries of quad tree

LISTING 7.9: Bottom-up quad tree traversal pattern

```
__device__ void TreeNode::initialize_frontier() {
  bfs_frontier_ = true; bfs_done_ = false;
  for (int i = 0; i < 4; ++i) {
    if (children_[i]->cast<TreeNode>() != nullptr) { bfs_frontier_ = false; break; }
  }
}

__device__ void TreeNode::advance_frontier() {
  if (!bfs_done_) {
    for (int i = 0; i < 4; ++i) {
      TreeNode* child = children_[i]->cast<TreeNode>();
      if (child != nullptr && !child->bfs_done_) return; // Unprocessed child found
    }
    bfs_frontier_ = true;
  } else { bfs_frontier_ = false; }
}

void bottom_up_traversal() {
  allocator_handle->parallel_do<TreeNode, &TreeNode::initialize_frontier>();
  do {
    // Do something: e.g., compute TreeNode summary or collape TreeNode
    allocator_handle->parallel_do<TreeNode, &TreeNode::advance_frontier>();
  } while (!host_tree->bfs_done_);
}
```

LISTING 7.10: Force computation via quad tree traversal

```
__device__ TreeNode* tree; // Pointer to tree root
TreeNode* host_tree; // Pointer to tree root in host memory

__device__ void BodyNode::apply_force(BodyNode* other) {
  // Same as Body::apply_force in n-body.
}

__device__ void TreeNode::apply_force(BodyNode* other) {
  if (contains(other) || distance_to(other) <= kDistThreshold) {
    // Too close or inside. Recurse.
    for (int i = 0; i < 4; ++i) {
      if (children_[i] != nullptr) { children_[i]->apply_force(other); }
    }
  } else {
    // Far enough away to use approximation. Same as BodyNode::apply_force.
  }
}

__device__ void BodyNode::compute_force() {
  force_x_ = force_y_ = 0.0f;
  tree->apply_force(this);
}
```

force induced by the body to the `force_` fields of `other`, similar to the original n-body simulation. If `this` is a `TreeNode`, there are two possibilites:

1. If `other` is contained in `this` or if `other` is close to the `this`'s center of gravity, we have to recurse and do a more accurate computation with every child of `this`. Using a Barnes-Hut approximation would result in a too large error.

2. If `other` is not within the bounds of `this` and far enough away from its center of gravity, we can approximate the gravitational force induced by the entire `TreeNode` using the `this`'s summary.

Since DYNASOAR does not support virtual function calls yet, we had to implement this function with a hand-written *switch-case* statement (Section 7.3.3). We plan to auto-generate such code in future versions of DYNASOAR.

Step 4d: Removing Bodies We are now removing `BodyNode`s from the tree if they moved into a different quadrant of their parent `TreeNode` or if they moved into a different `TreeNode`. This is easy to detect based on the position of the body and corner points `p1`, `p2` of the `TreeNode`. Such bodies are removed from the `children` array and their `parent` pointer is reset to `nullptr`.

Figure 7.9 shows the quad tree of Figure 7.6A after bodies were moved (Step 4c). The red bodies moved into a different quadrant or parent and are removed from the tree. This may leave some `TreeNode`s empty.

Step 3 / Step 4e: Inserting Bodies We are now (re)inserting `BodyNode`s into the quad tree which have no parent. In Step 3, these are all newly created bodies. In Step 4e, these are all bodies that were removed in the previous step. This step is the most challenging part of barnes-hut, because the structure of the quad tree is modified concurrently by multiple threads.

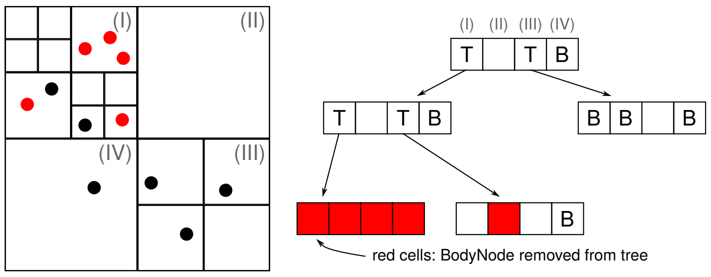


FIGURE 7.9: Removing BodyNodes from the quad tree

Listing 7.11 shows the insertion algorithm. Only BodyNodes without a parent are processed. The algorithm maintains a `current` pointer to the `TreeNode` into which the body should be inserted. This pointer is initialized to the root of the tree. The *while* loop of Line 4 traverses the tree structure by determining the `children` array slot into which the body should be inserted (Line 5) and updating the `current` pointer if necessary. At this point, we have to consider three cases.

1. The selected **array slot is empty** (Line 7, Figure 7.10). In this case, we can directly insert the body into the array. However, since multiple threads may be attempting to insert into the same slot, we insert the body with an atomic compare-and-swap operation, such that only one thread can succeed. Afterwards, we set the `parent` pointer with an atomic exchange operation. This is to ensure that the modification to the `parent` pointer is guaranteed to become visible to other threads in the CUDA kernel (*volatile write*). If a thread fails to insert a body because another thread succeeded with a contending compare-and-swap operation, the thread retries with another *while* loop iteration.

2. The selected **array slot contains a** `TreeNode` (Line 12). In this case, we try to insert the body into that `TreeNode` in the next *while* loop iteration.

3. The selected **array slot contains a** `BodyNode`, denoted by `other` (Line 14, Figure 7.11). In this case, we have to insert a new `TreeNode` into the tree. The method `make_child_ tree_node` creates a new `TreeNode` with the correct `p1`, `p2` and `parent` values, but does not insert it into `current` yet. We first insert `other` into the newly created (empty) `TreeNode`[1]. Now we swap in the new `TreeNode` with an atomic compare-and-swap operation. Similar to Case 1, only one thread can succeed in modifying the respective `children` array slot. If successful, we update the `parent` pointer of `other`. However, this body may still be in the process of insertion (Case 1 or `other` in Case 3) and its `parent` pointer may not have been set yet. Therefore, we require another *while* loop to retry until `parent` was updated. Now we can insert the original body into the new `TreeNode` by running another iteration of the outer *while* loop.

Note that the outer *while* loop of Listing 7.11 maintains a boolean flag `is_true` instead of an infinite loop with a break or return statement. This is to avoid deadlocks, since the structure of this loop is similar to the one of the critical section implementation of Section 2.1.3.

[1]The thread fence in Line 21 ensures that other threads see `other`'s slot in the `children` array of the new `TreeNode` as occupied before the new `TreeNode` becomes visible through the CAS in Line 23.

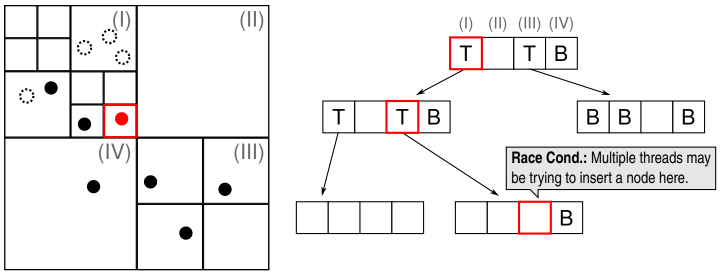

(I)
(II)
(III)
(IV)
T
B
Race Cond.: Multiple threads may be trying to insert a node here.

FIGURE 7.10: Inserting a BodyNode into an empty slot

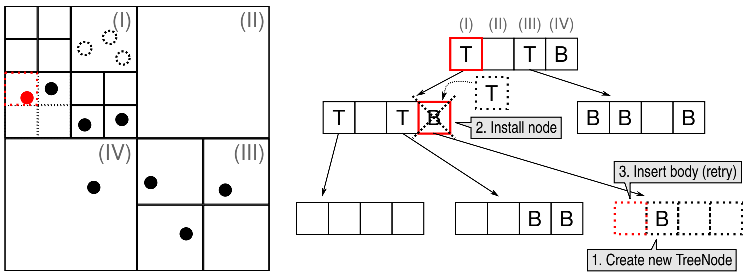


FIGURE 7.11: Inserting a BodyNode into a slot with another BodyNode

LISTING 7.11: Inserting a body into the quad tree

```
__device__ void BodyNode::add_to_tree() {
  if (parent_ == nullptr) {
    TreeNode* current = tree; bool is_done = false;
    while (!is_done) { // Check where to insert in this node.
      int c_idx = current->child_idx(this);
      auto*& child_ptr = current->children_[c_idx];
      if (child == nullptr) { // Slot not in use.
        if (atomicCAS<NodeBase>(&child, nullptr, this) == nullptr) {
          atomicExch(&parent_, current);
          child_index_ = c_idx; is_done = true; // Done inserting.
        }
      } else if (child->cast<TreeNode>() != nullptr) { // There is a subtree here.
        current = static_cast<TreeNode*>(child);
      } else { // There is a body "other" here.
        BodyNode* other = static_cast<BodyNode*>(child);
        // Replace BodyNode with TreeNode.
        auto* new_node = current->make_child_tree_node(c_idx);
        // Insert other into new node.
        int other_c_idx = new_node->child_idx(other);
        new_node->children_[other_c_idx] = other;
        __threadfence();
        // Try to install the new TreeNode. (Retry.)
        if (atomicCAS<NodeBase>(&child, other, new_node) == other) {
          // It may take a while until we see the correct parent, because
          // another may not be done inserting this node yet.
          TreeNode* parent_before = nullptr;
          do {
            parent_before = atomicCAS<TreeNode>(&other->parent_, current, new_node);
          } while (parent_before != current);
          other->child_index_ = other_c_idx;
          current = new_node; // Now insert body into new_node.
        } else { destroy(device_allocator, new_node); } // Rollback.
      }
    }
  }
}
```

Step 4f: Collapsing Tree As the final step of a barnes-hut iteration, we clean up the quad tree. Empty `TreeNode`s are removed and `TreeNode`s with only one child, which is a `BodyNode`, are collapsed with the parent node. This step is implemented as a bottom-up tree traversal, similar to Step 4a. Listing 7.12 shows the *processing* step of the tree traversal. Only frontier `TreeNode`s f are processed. We now have to consider four cases.

1. The node f has **zero children** (Line 16). In this case, we unregister f from its parent and delete it.
2. The node f has **one child** c **which is a** `BodyNode` (Line 24). In this case, we store the c in the `children` array slot of the parent where f is currently stored. We then delete f.
3. The node f has **one child** c **which is a** `TreeNode`. We cannot collapse this node. The subtree c must contain more than one `BodyNode`; otherwise, it would have been collapsed already by the previous bottom-up iteration. Since every `TreeNode` level divides the space into four equally-sized quadrants, collapsing f would require changes to the `TreeNode`s within c. Moreover, at least two `BodyNode`s would end up as direct children in the same quadrant of some `TreeNode` within c, which is forbidden.
4. The node f has **more than one child**. We cannot collapse this node.

Note that removing nodes from a tree is much simpler than adding nodes. No atomic operations or other synchronization primitives are necessary.

7.3.3 Virtual Function Calls

Unfortunately, DynaSOAr does not yet support C++ abstractions for virtual functions. Therefore, we have to implement the dispatch logic of the virtual function `NodeBase::apply_force` in Step 4b by hand. Our implementation consists of an explicit runtime type check (`cast<T>`) and an *if-then-else* statement that dispatches to the correct method implementation (Listing 7.13). `cast<T>` is a method provided by DynaSOAr's data layout DSL. Its semantics are similar to C++'s `dynamic_cast<T*>`, but it determines the runtime type of an object from its address (fake pointer; Section 5.2.3). This implementation is more efficient than C++'s `dynamic_cast<T*>` because it does not have to read a vtable pointer from memory.

A *switch-case* statement-based implementation of a virtual function call is compiled to code that is more efficient than a vtable-based implementation, because it allows compilers to inline virtual method calls. This results in more efficient GPU binary code.

GPU programs usually consist of a single compilation unit. Therefore, it is possible to enumerate and inline all possible runtime types (subtypes of the static receiver type) in the source code. We plan to automate this process in the future by auto-generating *switch-case* statements as part of the compilation process.

7.3.4 Benefits of Object-oriented Programming

Implementing barnes-hut without object-oriented programming is tedious. One particular problem is the dynamic allocation of new `TreeNode`s in Line 17 of Listing 7.11. Our implementation optimistically alloates a new `TreeNode` and tries to insert it with an atomic compare-and-swap operation. If this operation fails, we delete the object

LISTING 7.12: Collapsing a TreeNode

```
__device__ void TreeNode::collapse_tree() {
  if (bfs_frontier_) {
    bfs_frontier_ = false;

    // Count children.
    int num_children = 0;
    NodeBase* single_child = nullptr;

    for (int i = 0; i < 4; ++i) {
      if (children_[i] != nullptr) {
        ++num_children;
        single_child = children_[i];
      }
    }

    if (num_children == 0) { // Remove node without children.
      if (parent_ != nullptr) { // Do not remove the root.
        parent_->children_[child_idx_] = nullptr;
        destroy(device_allocator, this);
        return;
      }
    } else if (num_children == 1) { // Collapse TreeNodes with 1 child...
      if (parent_ != nullptr) {
        if (single_child->cast<BodyNode>() != nullptr) {
          // ... but only if the node is a body.
          single_child->parent_ = parent_;
          single_child->child_index_ = child_index_;
          parent_->children_[child_index_] = single_child;

          destroy(device_allocator, this);
          return;
        } // else: TreeNode child cannot be collapsed.
      }
    }

    // Done processing this node.
    bfs_done_ = true;
  }
}
```

LISTING 7.13: Handwritten virtual method call

```
__device__ void NodeBase::apply_force(BodyNode* other) {
  if (cast<BodyNode>() != nullptr) {
    static_cast<BodyNode*>(this)->apply_force(other);
  } else {
    assert(cast<TreeNode>() != nullptr);
    static_cast<TreeNode*>(this)->apply_force(other);
  }
}
```

again. Related work describes an alternative implementation without dynamic memory allocation but with a lightweight lock that prevents two threads from inserting a node into the same location [28]. While such an implementation may be a bit faster than our implementation, locking is problematic on GPUs and such locks must be carefully designed and implemented to avoid deadlocks (Section 2.1.3).

Implicit type information is another benefit of object-oriented programming in this application. A non-OOP implementation would have to store an explicit type identifier in addition to an object ID for each child in `TreeNode::children`. This is necessary because a child could be a `BodyNode` or another `TreeNode`. We do not have native support for virtual functions in DYNASOAR yet, but virtual function calls would be another benefit of an OOP-based implementation.

Finally, consider the field access in Line 28 of Listing 7.12. Listing 7.14 shows the same functionality in a hand-written SOA layout. Without OOP abstractions, such code is much harder to write/maintain, especially due to the nested array accesses.

LISTING 7.14: barnes-hut: Field access notation in hand-written SOA layout

```
TreeNode_children[TreeNode_child_idx[id]][TreeNode_parent[id]] = single_child;
```

7.3.5 Further Optimizations

Related work describes an optimization to speed up Step 4b, which computes forces with a quad tree traversal. The performance of this step can be improved by assigning spatially local bodies to the same warp [28]. This could be achieved by physically rearranging/sorting `Body` objects. Bodies which are spatially local traverse similar parts of the quad tree, which improves memory accesses (threads of a warp access similar addresses) and reduces warp divergence (similar tree traversals result in similar control flow). Apart from COMPACTGPU's memory defragmentation, DYNASOAR does currently not allow programmers to rearrange allocations in memory. Future work could extend DYNASOAR in that direction.

Another problem of our barnes-hut implementation is the recursive nature of Step 4b. If the mass and position of a `TreeNode` cannot be approximated, we have to recursively visit up to 4 child nodes (Listing 7.10, Line 12). Unfortunately, this pattern is *not* tail recursive. Therefore, the compiler has to allocate stack frames for the recursive method calls in local memory, which resides in the (slow) global memory. However, the only state that we conceptually have to maintain is a pointer to the current `TreeNode` and an array index into the `children` array. This information is sufficient for implementing a preorder tree traversal, so it should be possible to transform this recursion into an iterative implementation that does not allocate new stack frames.

7.4 structure: Finite Element Method

structure is a finite element method (FEM), inspired by a problem in material science: Simulating the formation of a crack in a composite material [125]. structure simulates a mesh of elements, which can be seen as an undirected graph. Every graph node is connected by springs with up to three other nodes.

The simulation has three types of nodes: Ordinary *nodes*, *anchor nodes* and *pull nodes*. Pull nodes move into a certain direction with a constant velocity and *anchor*

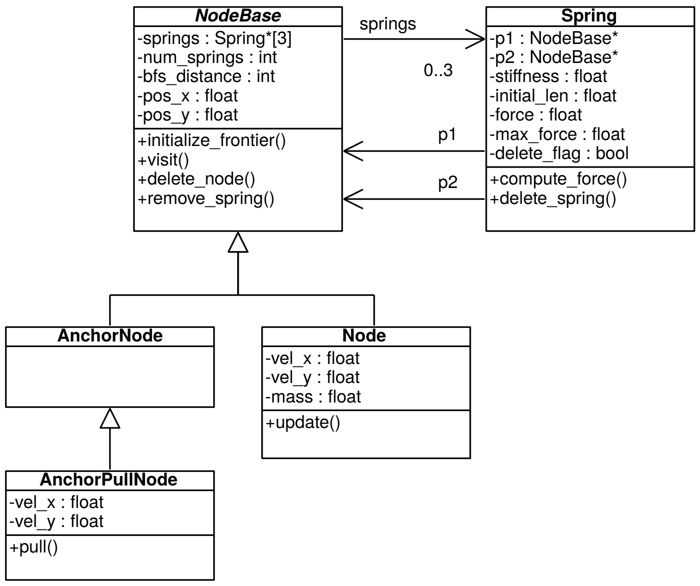


FIGURE 7.12: Data structure of structure

nodes are fixed at a certain position. Ordinary nodes can move freely. Springs have an initial length and a stiffness k. If a spring is stretched beyond its initial length, it exerts a pulling force F on both node endpoints according to Hooke's law.

$$F = k \cdot \Delta x \qquad \text{(Hooke's law)}$$

Pull nodes stretch the entire mesh in a certain direction. As soon as the force F between two nodes exceeds a certain threshold, the spring breaks, forming a crack in the material.

7.4.1 Data Structure

This application consists of five classes (Figure 7.12): An abstract class `NodeBase` with three subclasses `AnchorNode`, `AnchorPullNode` and `Node`, and a class `Spring`.

Every node is connected with up to three springs. These are the edges of the graph and stored in an array `springs` (adjacency list), along with the actual number of springs `num_springs`. Every node has a 2D position. In addition, `AnchorPullNodes` and `Nodes` have a velocity. In the former case, this velocity is constant. In the latter case, the velocity changes based on the forces induced by the springs.

`Springs` store pointers to their node endpoints. If at any point the spring is stretched so far that `force` $>$ `max_force`, the spring breaks.

7.4.2 Application Implementation

Before running the main simulation loop, we have to load or generate a mesh network. Our current implementation generates a random graph with a configurable percentage of each node type. structure is an iterative algorithm and consists of the following steps.

1. **Pull Nodes:** Update the position of pull nodes based on their velocity. Parallel do-all: `AnchorPullNode::pull`.

2. **Compute Steps:** Repeat 40 times.
   (a) **Compute Forces:** Compute the force of each spring based on Hooke's law. If this force exceeds a spring's threshold, delete the spring from the simulation. Parallel do-all: `Spring::compute_force`.
   (b) **Update Velocity/Position:** Sum computed forces exerted by springs for each node. Accelerate and move nodes based on this force. Parallel do-all: `Node::update`.
3. **Remove Disconnected Nodes:** Find disconnected nodes with a BFS and remove them from the simulation. Implemented with multiple parallel do-all operations.

Step 2 is somewhat similar to an n-body simulation. We are computing forces between elements, however, according to Hooke's law instead of Newton's gravitational law. Furthermore, we only consider direct neighbors in the graph structure.

Step 3: Removing Disconnected Nodes This step is the most challenging part. `Nodes` and `AnchorPullNodes` are removed from the simulation if they are no longer connected to an `AnchorNode`. `Springs` whose endpoint(s) were deleted are also removed from the simulation.

This step is based on a parallel breadth-first graph traversal. The traversal starts at `AnchorNodes` (excl. `AnchorPullNodes`) and marks every visited node. Unvisited nodes are removed after the traversal. We use the standard frontier-based BFS algorithm, which is known to work well on GPUs (Section 4.3.2).

Listing 7.15 shows how disconnected nodes are detected and removed. Nodes do not have a boolean frontier flag in our implementation. Instead, every node has a `distance` field, which is initialized to infinity (`kMaxDistance`). In BFS iteration i, the frontier consists of the nodes with `distance` i.

In the beginning, all `AnchorNodes` are initialized to a distance of 0. A BFS iteration (`NodeBase::visit`) iterates over the neighboring nodes of frontier nodes and updates their distance if they are still unvisited (Line 16). We keep iterating until no vertices were visited in an iteration, as indicated by the `continue_bfs` flag.

Finally, we process all unvisited nodes with `NodeBase::delete_node`. This method does not delete those nodes yet, but sets a flag on springs that are connected to them. Springs are deleted in `Spring::delete_spring`. Before a spring deletes itself, it unregisters itself from its endpoints (`remove_spring`). As part of this process, we atomically decrement the spring counter of the node. Once the counter reaches zero, the node is deleted. This process requires an atomic operation, because multiple GPU threads may be concurrently unregistering springs from a node.

7.5 traffic: Traffic Flow Simulation

Traffic flow simulations are important tools in transportation planning [130]. They can guide the design and construction of city street networks. traffic is an agent-based microsimulation that simulates single vehicles (*agents*) which move on a street network. It is based on the Nagel-Schreckenberg model [145], a simple model based on cellular automata which can reproduce real-world traffic phenomena [195, 196] such as traffic jams. Compared to other SMMO applications, traffic is quite complex: It does not only model agents, but also more advanced street features such as traffic

LISTING 7.15: Removing disconnected nodes with BFS

```
__device__ bool continue_bfs;

__device__ void NodeBase::initialize_frontier() {
  distance_ = this->cast<AnchorNode>() == nullptr ? kMaxDistance : 0;
}

__device__ void NodeBase::visit(int distance) {
  if (distance == distance_) {
    continue_bfs = true;
    for (int i = 0; i < kMaxDegree; ++i) {
      auto* spring = springs_[i];
      if (spring != nullptr) {
        // Neighboring vertex.
        auto* n = spring->p1() == this ? spring->p2() : spring->p1();
        // Set distance on neighboring vertex if unvisited.
        if (n->distance_ == kMaxDistance) { n->distance_ = distance + 1; }
      }
    }
  }
}

__device__ void NodeBase::delete_node() {
  if (distance_ == kMaxDistance) {
    for (int i = 0; i < 3; ++i) {
      if (springs_[i] != nullptr) { springs_[i]->delete_flag_ = true; }
    }
  }
}

__device__ void Spring::delete_spring() {
  if (delete_flag_) {
    p1_->remove_spring(this);
    p2_->remove_spring(this);
    destroy(device_allocator, this);
  }
}

__device__ void NodeBase::remove_spring(Spring* s) {
  for (int i = 0; i < 3; ++i) {
    if (springs_[i] == s) {
      springs_[i] = nullptr;
      if (atomicSub(&num_springs_, 1) == 1) { destroy(device_allocator, this); }
      return;
    }
  }
}

void delete_disconnected_nodes() {
  allocator_handle->parallel_do<NodeBase, &NodeBase::initialize_frontier>();

  for (int dist = 0; continue_bfs /*read with cudaMemcpyFromSymbol*/; ++dist) {
    continue_bfs = false; // write with cudaMemcpyToSymbol
    allocator_handle->parallel_do<NodeBase, int, &NodeBase::visit>(dist);
  }

  allocator_handle->parallel_do<NodeBase, &NodeBase::delete_node>();
  allocator_handle->parallel_do<Spring, &Spring::delete_spring>();
}
```

lights or yield signs. We describe this application only on a high level. Related work describes many variantions and how to implement them [214].

By default, this application generates a random street network upon startup. Alternatively, real-world street networks can be imported from OpenStreetMap (OSM) dumps in GraphML file format. Such dumps include street properties such as position, shape, connections to other streets, speed limits and OSM street type.

7.5.1 Data Structure

In the Nagel-Schreckenberg model, a street (*link*) is divided into equally-sized `Cells`, each of which can contain up to one agent (Figure 7.13). An agent (class `Car`; Figure 7.14) can move onto a neighboring cell only if it is free. Every agent has a velocity, measured in cells per iteration. Both agents and cells have a maximum velocity; the latter one can be used to model speed limits on streets.

Agents precompute their `path` of movement per iteration. The `path` array contains the next `velocity` many cells onto which the agent is about to move. All of these cells must be empty. This is to ensure that the agent does not crash into other agents.

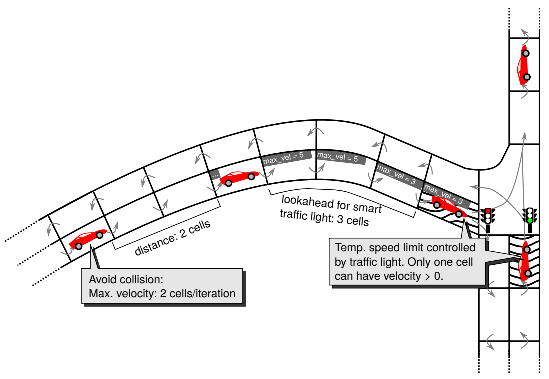
Intersections and Traffic Lights Every cell has zero, one, or multiple outgoing cells, forming a directed graph. In the first case, the cell is a *sink*, i.e., a street leaves the simulation area. Agents entering a sink will be randomly redistributed. The second case is most common and represents a regular street cell. The third case appears at intersections, where a cell is connected to the first cell of every outgoing[2] street.

It is important to ensure that only one car enters an outgoing street (i.e., first cell of the street) at an intersection in one iteration, even if multiple cars from different incoming streets are waiting. To that end, a *traffic controller* can impose temporary speed limits on cells, e.g., a speed limit of zero, corresponding to a red light [60]. Traffic controllers set and remove speed limits for the last cells of all incoming streets such that only one incoming street has a green light at a time. We implemented three kinds of controllers.

- A *traffic light* imposes a temporary speed limit of zero on all incoming streets, except for one street which has a green *phase* for a certain number of iterations (*phase length*). Green phases are scheduled round-robin among all incoming streets.
- A *smart traffic light* works like a normal traffic light but assigns a green phase to an incoming street immediately if this street is the only incoming street with a waiting car. Real traffic lights have sensors/cameras to provide such behavior. All traffic lights in the benchmark section are smart.
- A *yield controller* corresponds to a yield traffic sign, which is often found at the end of merge lanes of highway entrances. Given n incoming streets, it assigns a temporary speed limit of zero to all streets $i > s$ if street s has a car, i.e., incoming streets/cells in the controller should be ordered by priority.

The last two traffic controllers should ensure that traffic does not have to stop or slow down in front of an intersection. Thus, controllers must check all cells from which a car could cross an intersection in one iteration (not only the closest incoming cell), as indicated by the maximum allowed speed limit on a street (*lookahead*). This is done

[2]Streets in this simulation are one-way streets. Two-way streets consists of one incoming and one outgoing street.


max_vel = 5
max_vel = 5
max_vel = 3
max_vel = 3
lookahead for smart traffic light: 3 cells
distance: 2 cells
Avoid collision:
Max. velocity: 2 cells/iteration
Temp. speed limit controlled by traffic light. Only one cell can have velocity > 0.

(A) Example: Cells and intersections

(B) Screenshot: Imported OpenStreetMap data

FIGURE 7.13: Representation of street networks in traffic

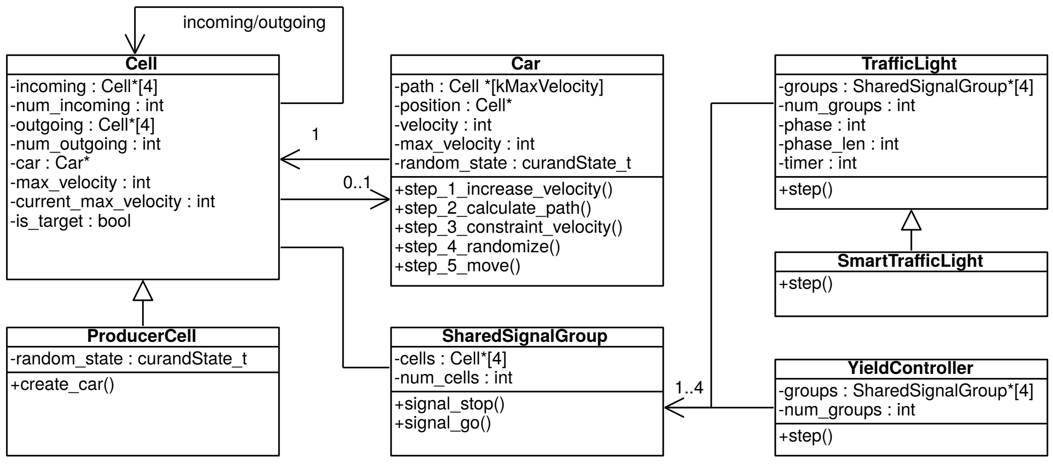


FIGURE 7.14: Data structure of traffic

with a graph traversal on back edges (*incoming cells*), which terminates when a car was found within lookahead range.

Turn Lanes To allow the traffic to flow more smoothly, we generate turn lanes from each incoming street to each outgoing street at intersections. To implement a red traffic light signal for an incoming street, we now have to impose a speed limit on the last cell of each turn lane. Those cells are grouped in a `SharedSignalGroup` because they should all have the same traffic light signal[3].

7.5.2 Application Implementation

This application consists of the following parallel do-all operations.

1. **Advance Traffic Light State:** Increment a timer. Once it reaches the phase length, propagate a red signal to the currently green shared signal group, as indicated by `phase` (index into `groups`). Now, the next shared signal group receives a green light phase and we reset the counter. Smart traffic lights function similar but can change their phase even before the timer reaches the phase length, in case a car is waiting at only one shared signal group with a red signal. Parallel do-all: `TrafficLight::step`

2. **Advance Yield Controller State:** Among all shared signal groups, find the first one that has a car that can cross within the next iteration. That signal group receives a green light and all other signal groups receive a red light. Parallel do-all: `YieldController::step`

3. **Nagel-Schreckenberg Iteration:** In the following steps, we denote the agent that is processed in a parallel do-all operation by a.

   (a) **Acceleration:** Increase the agent's velocity unless it is already driving at its maximum velocity: $v_a \leftarrow min(v_a + 1, v\_max_a)$.
   Parallel do-all: `Car::step_1_increase_velocity`

   (b) **Compute Path:** Determine the agent's path of movement of length v_a, i.e., the next v_a many cells that it will pass through. A *navigation strategy* determines the next cell at an intersection with multiple outgoing cells. Our current implementation uses random walk, biased towards large streets. Parallel do-all: `Car::step_2_calculate_path`

   (c) **Adjust Velocity:** Avoid collisions with other agents and enforce speed limits. To avoid collisions, reduce the speed v_a to the largest possible value such that the first v_a many cells on the calculated path are free. To follow speed limits, reduce v_a to the largest possible value, such that $v_a \leq v\_max_c$ for each cell c among the first v_a many cells on the calculated path.
   Parallel do-all: `Car::step_3_constraint_velocity`

   (d) **Randomization:** Reduce the agent's velocity by one unit with a probability of 20%. Parallel do-all: `Car::step_4_randomize`

   (e) **Update Position:** Move the agent from its current location by v_a many cells according to the calculated path. Parallel do-all: `Car::step_5_move`

[3]Traffic lights could be extended such that more than one street (for certain directions) has a green light at a time.

In our DynaSOAr benchmarks, we, furthermore, distinguish between three kinds of cells: Regular cells, producer cells and sink cells. Producer cells create new agents with given probability if the cell is empty. Sink cells remove agents with a given probability. This is to simulate dynamic (de)allocation.

In a real traffic simulation, we would generate new agents in certain city areas (e.g., residential areas). Agents would move to certain waypoints according to a *schedule* which can be generated from real-world traffic data. Agents would be removed from the simulation when they reach their final waypoint.

Step 3c: Adjust Velocity Listing 7.16 illustrates how traffic avoids collisions and speed limit violations in a Nagel-Schreckenberg iteration. Let us assume that we decided to move the agent at a certain `velocity` in Step 3a and precomputed the same number of cells in `path` as part of Step 3b. Now, we check every cell on the precomputed path. The speed limit of an agent has to be reduced in two cases.

1. A cell on the path is occupied by another agent (Line 7). In this case, the speed limit has to be reduced such that the cell is not entered.

2. A cell on the path has a speed limit of less than the agent's intended velocity (Line 13). Depending on which option would make more progress, we either reduce the speed limit to the cell's speed limit (Line 15) or decide to stop before entering the cell (Line 18). We make more progress in the latter case if the cell has such a low speed limit that we would not even reach the cell in this iteration. E.g., this is the case when approaching a red traffic light (speed limit zero).

Listing 7.16: Avoiding collisions and speed limit violations

```
 1 __device__ void Car::step_3_constraint_velocity() {
 2   for (int distance = 1; distance <= velocity_; ++distance) {
 3     // Invariant: Movement of up to distance - 1 many cells at velocity_ is allowed.
 4     Cell* next_cell = path_[distance - 1];
 5 
 6     // Avoid collision.
 7     if (next_cell->car_ != nullptr) {
 8       // Cannot enter cell.
 9       velocity_ = distance - 1;
10       break;
11     } // else: Can enter next cell.
12 
13     if (velocity_ > next_cell->current_max_velocity_) {
14       // Car is too fast for this cell.
15       if (next_cell->current_max_velocity_ > distance - 1) {
16         // Even if we slow down, we would still make progress.
17         velocity_ = next_cell->current_max_velocity_;
18       } else {
19         // Do not enter the next cell.
20         velocity_ = distance - 1;
21         break;
22       }
23     }
24   }
25 }
```

Notice how the `Car::path` array is accessed. For each car, we access array elements sequentially, starting from index 0 up to index `velocity` − 1. We can optimize this access by storing this inner array in SOA layout.

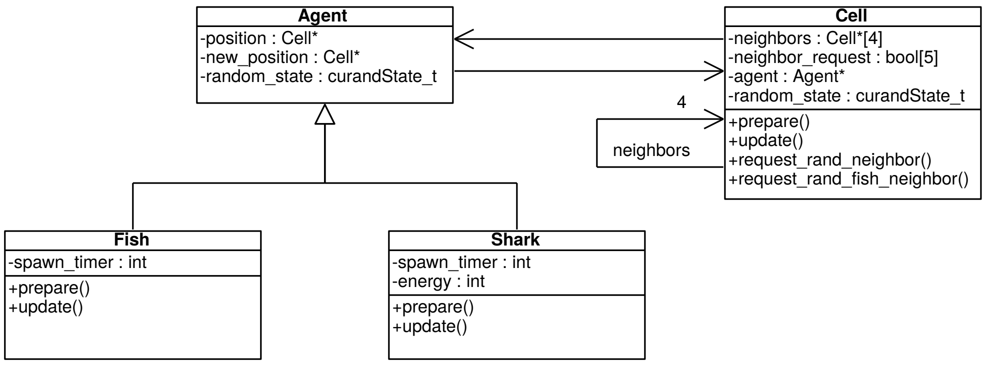


FIGURE 7.15: Data structure of wa-tor

7.5.3 Object-oriented Traffic Simulations

Previous work has demonstrated that traffic simulations can be expressed very well with object-oriented programming, because many real-world entities such as different kinds of vehicles, streets, intersections, traffic lights, etc. can be directly mapped to objects/classes [189, 87, 107]. Traffic simulations are even used for teaching object-oriented software design to students [156].

7.6 wa-tor: Fish and Sharks Simulation

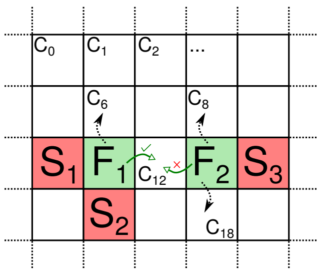
wa-tor is an ecosystem of fish and sharks that occupy a 2D space in a predator-prey relationship [49]. Such ecosystems can be described with the Lotka-Volterra equations [74]. In this application, we simulate wa-tor with a cellular automaton (CA). Cellular automatons are well suited for GPU execution [72]. There are typically thousands of cells and all cells feature the same or similar computations, based on the state of neighboring cells.

wa-tor simulates a hypothetical, torus-shaped (2D space, wrapping around at the borders) planet made up of cells. Each cell can be occupied by up to one agent (fish or shark). Sharks are predators who are eating the fish (the prey). In each iteration, an agent can move to a neighboring cell. Fish can only move to empty cells, but sharks can move to empty cells or cells containing a fish, thereby consuming the fish. Sharks have an energy level that decreases with each iteration and increases when consuming a fish. Both fish and sharks reproduce after a certain number of iterations.

7.6.1 Data Structure

Figure 7.15 illustrates the data structure of wa-tor. There is an abstract class `Agent` with two subclasses `Fish` and `Shark`, and another class `Cell`.

Agents store a pointer to their current position (`position`) and have a field `new_position` which is used for determining the next cell to move to. Cells have pointers to all four neighboring cells (`neighbors`). Furthermore, there is an array `neighbor_request` with a boolean flag per neighbor, indicating whether an agent from a neighboring cell is attemping to enter this cell. This data structure allows us to implement the movement of agents without expensive synchronization mechanisms such as atomic operations.



F_1::prepare
Candidates: $\{C_6, C_{12}\}$
Selected: C_{12}

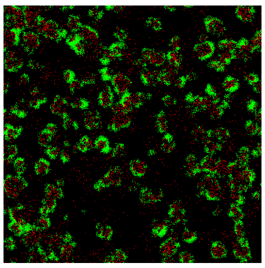
F_2::prepare
Candidates: $\{C_8, C_{12}, C_{18}\}$
Selected: C_{12}

C_{12}::decide
Candidates: $\{F_1, F_2\}$
Selected: F_1

F_1::update
Move to C_{12}

F_2::prepare
Stay on C_{13}

(A) Cell interaction



(B) Screenshot

FIGURE 7.16: Cell interaction and screenshot of wa-tor. Green areas indicate fish and red areas indicate sharks.

7.6.2 Application Implementation

wa-tor is an iterative algorithm. An iteration consists of the following steps.

1. **Reset Cells:** Initialize the array `neighbor_requests` to *false*. This array is used by agents of neighboring cells to indicate that they wish to enter this cell. Parallel do-all: `Cell::prepare`

2. **[Fish] Select Outgoing Cell:** Among all four neighboring cells, select an empty cell at random and set the corresponding `neighbor_request` to *true*. This is the cell that the agent is planning to move to. Parallel do-all: `Fish::prepare`

3. **Select Incoming Fish:** Among all agents that are trying to move to this cell, as indicated by the `neighbor_request` array, select one at random. Parallel do-all: `Cell::decide`

4. **[Fish] Move to New Cell:** Move to the selected cell if approval was granted. Leave a new `Fish` at the current location with a certain probability. Parallel do-all: `Fish::update`

5. **Reset Cells:** Same as Step 1. Parallel do-all: `Cell::prepare`

6. **[Shark] Select Outgoing Cell:** Same as Step 2, but sharks are also allowed to and prefer to move onto cells that contain a fish. Parallel do-all: `Shark::prepare`

7. **Select Incoming Shark:** Same as Step 3. Parallel do-all: `Cell::decide`

8. **[Shark] Move to New Cell:** Same as Step 4, but sharks consume a fish if present on the target cell. Parallel do-all: `Shark::update`

Figure 7.16 illustrates the selection of cells by fish. In this figure, red blocks indicate sharks and green blocks indicate fish. F_1 scans its neighborhood for empty cells to move to[4]. Possible candidates are C_6 and C_{12}. Among those, F_1 randomly selects C_{12}. However, F_2 also selects C_{12} in the same iteration. Only one agent is allowed to move onto the cell. Among those two, `Cell::decide` randomly decides that F_1 is allowed to enter the cell. As a consequence, F_1 moves to C_{12} and F_2 remains at its original position.

Listing 7.17A shows how the simulation logic for `Fish` (Steps 1–4) is implemented. The simulation logic for `Shark`s is implemented in a similar way. Requests to move to a cell are stored in `neighbor_request`. This array has five slots: One for each

[4]Fish and sharks cannot move diagonally.

LISTING 7.17: Simulation logic for `Fish`

```
__device__ void Cell::prepare() {
  for (int i = 0; i < 5; ++i) { neighbor_request_[i] = false; }
}

__device__ void Fish::prepare() {
  ++spawn_timer_;
  position_->request_random_free_neighbor();
}

__device__ void Cell::request_random_free_neighbor() {
  int candidates[4]; int num_candidates = 0;

  for (int i = 0; i < 4; ++i) {
    if (neighbors_[i]->agent_ == nullptr) { candidates[num_candidates++] = i; }
  }

  if (num_candidates == 0) { // Stay on this cell.
    neighbor_request_[4] = true;
  } else {
    int selected_index = curand(&random_state) % num_candidates; // cuRAND library
    int selected = candidates[selected_index];
    int neighbor_index = (selected + 2) % 4;
    neighbors_[selected]->neighbor_request_[neighbor_index] = true;
  }
}

__device__ void Cell::decide() {
  if (neighbor_request_[4]) { // This cell has priority.
    agent_->new_position_= this;
  } else { // Select random agent among requesting neighbors.
    int candidates[4]; int num_candidates = 0;

    for (int i = 0; i < 4; ++i) {
      if (neighbor_request_[i]) { candidates[num_candidates++] = i; }
    }

    if (num_candidates > 0) {
      int selected_index = curand(&random_state_) % num_candidates;
      neighbors_[candidates[selected_index]]->agent_->new_position_ = this;
    }
  }
}

__device__ void Fish::update() {
  Cell* old_position = position_;
  if (old_position != new_position_) {
    position_ = new_position_;
    position_->agent_ = this;

    if (spawn_timer_ > kSpawnThreshold) { // Spawn offspring.
      auto* new_fish = new(device_allocator) Fish();
      new_fish->position_ = old_position;
      old_position->agent_ = new_fish;
      spawn_timer_ = 0;
    } else { old_position->agent_ = nullptr; }
  }
}
```

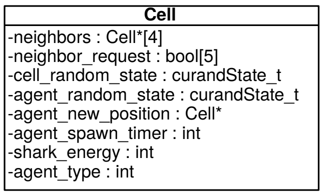

Cell
-neighbors : Cell*[4]
-neighbor_request : bool[5]
-cell_random_state : curandState_t
-agent_random_state : curandState_t
-agent_new_position : Cell*
-agent_spawn_timer : int
-shark_energy : int
-agent_type : int

FIGURE 7.17: Data structure of wa-tor (merged agents into `Cell`), without methods

neighbor and one for the cell itself, indicating that an agent wishes to stay on its current cell. This is the case if all neighboring cells are occupied. After examining all requests, `Cell::decide` sets `new_position` of the agent that is allowed to enter the cell. `Fish::update` moves to this cell and leaves a new `Fish` object at its old location every `kSpawnThreshold` iterations.

7.6.3 Benefits of Object-oriented Programming and Dynamic Allocation

Complex ecosystems such as predatory-prey ecosystems or population ecosystems are often modelled with object-oriented programming [97, 93, 63] because real or abstract entities can be mapped directly to objects/classes. Previous work describes in detail why object-oriented programming is suitable and an intuitive way of modelling such ecosystems [170].

wa-tor is difficult to implement without dynamic object allocation. `Fish` and `Shark` objects are created and deleted all the time. However, every cell contains at most one `Fish` or `Shark` object at a time. To implement wa-tor with only static allocation (baseline), we merged all classes of wa-tor into a single class `Cell` (Figure 7.17). The field `agent_type` indicates whether a cell contains an agent and if so, what the type of the agent is (fish or shark).

This example highlights the importance of dynamic object allocation. Our baseline versions with only static allocation break abstractions of our object-oriented implementation because class `Cell` now contains fields that logically describe both cells and agents.

7.7 sugarscape: Simulation of Population Dynamics

Sugarscape is an agent-based social simulation, originally presented in Epstein and Axtell's book *Growing Artificial Societies* [59]. It is a celluar automaton that simulates the behavior and interaction of agents (male/female) on a 2D grid. Agents require a certain amount of sugar to survive an iteration (*metabolism*). Cells grow and accumulate sugar over time and agents can harvest sugar by moving onto a cell.

Many variants of Sugarscape have been implemented in the past. Those variants can simulate complex social dynamics [100] such as trade, wars, diseases, etc. Our sugarscape implementation is rather simple and can simulate ageing and reproduction.

7.7.1 Data Structure

sugarscape consists of a 2D grid of cells with agents moving upon them, so the data structure (Figure 7.19) is similar to wa-tor. The main difference is that the grid structure is *not* encoded with adjacency lists in class `Cell`. Instead, there is a global array `cells` which stores pointers to all cells from left to right, top to bottom. Either implementation, adjacency list or global array, is feasible.

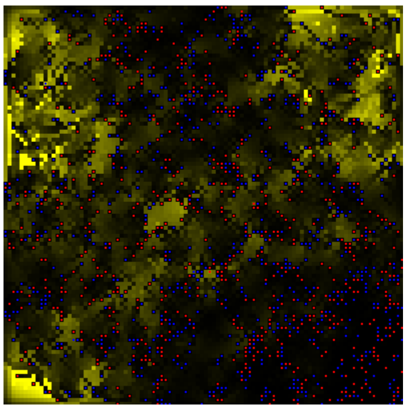


FIGURE 7.18: Screenshot of sugarscape. Yellow areas indicate sugar levels. Sugar diffuses to neighboring cells over time. Male/female agents are colored in blue/red.

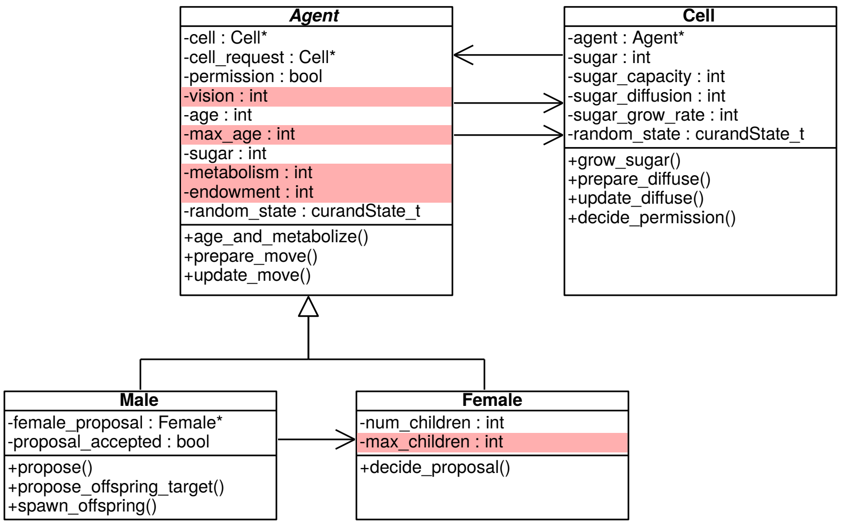


FIGURE 7.19: sugarscape: Data structure

Agents have a number of *genetic properties*, as indicated by the red color in Figure 7.19. Those properties are passed down to offspring.

7.7.2 Application Implementation

This application consists of a large number of parallel do-all operations. More complex Sugarscape simulations would require even more do-all operations. The operations that implement the moving behavior of agents are similar to the respective operations of wa-tor.

1. **Grow Sugar:** Increase the amount of sugar of this cell by `sugar_grow_rate`. Parallel do-all: `Cell::grow_sugar`
2. **Precompute Sugar Diffusion:** Compute the amount of sugar to be diffused to neighboring cells. Parallel do-all: `Cell::prepare_diffuse`
3. **Diffuse Sugar:** Take sugar from neighboring cells and add to this cell. Parallel do-all: `Cell::update_diffuse`
4. **Age and Metabolize:** Increase the age of the agent and reduce the sugar level of the agent by `metabolism_rate`. If the age exceeds `max_age` or if the sugar level drops below zero, the agent dies. Parallel do-all: `Agent::age_and_metabolize`
5. **Select Outgoing Cell:** Among all neighboring cells with distance of up to `vision`, select the empty cell with the largest amount of sugar. Store a pointer to that cell in `cell_request`. Parallel do-all: `Agent::prepare_move`
6. **Select Incoming Agent:** Among all agents that are trying to move to this cell, as indicated by the surrounding agents' `cell_request`, select one at random. Parallel do-all: `Cell::decide_permission`
7. **Move to New Cell:** Move to the selected cell if approval was granted. Consume all sugar of that cell. Parallel do-all: `Agent::update_move`
8. **Male** $\xrightarrow{\textbf{Propose}}$ **Female:** If a `Male` is ready to spawn offspring, i.e., sugar level $>$ endowment: Among all neighboring `Female` agents with distance of up to `vision`, select the agent with the largest amount of sugar. Parallel do-all: `Male::propose`
9. **Female** $\xrightarrow{\textbf{Accept Proposal}}$ **Male:** Among all neighboring, proposing `Male` agents with distance of up to `vision`, accept the agent with the largest amount of sugar. A `Female` cannot accept any more proposals once `num_children` $\geq$ `max_children`. Parallel do-all: `Female::decide_proposal`
10. **Choose Offspring Target:** If the proposal was accepted, select a neighboring cell with distance of up to `vision` at random. This is similar to Step 5. Parallel do-all: `Male::propose_offspring_target`
11. **Select Agent:** Among all `Male` agents that are tyring to spawn offspring on this cell, select one agent at random. This is similar to Step 6. Parallel do-all: `Cell::decide_permission`
12. **Generate Offspring:** Spawn a new agent on the selected cell if approval was granted. The gender is selected randomly. The genetic properties of the new

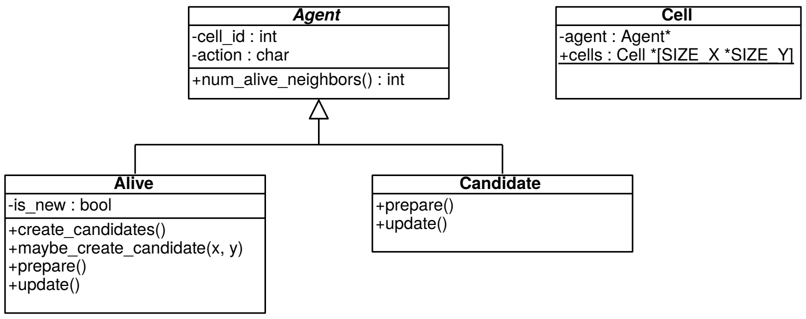


FIGURE 7.20: Data structure of gol

agent (vision, endowment, metabolism, maximum age) are average values of both parents. The new agent receives an initial sugar level of half the endowment of each parent. Parallel do-all: `Male::spawn_offspring`

Previous work describes how to implement Sugarscape efficiently on parallel architectures, in particular on GPUs [53]. These Sugarscape implementations differ in details, but show that GPUs are well-suited for agent-based modelling [1, 129] and simulating a massive number of agents.

7.8 gol: Game of Life

Game of Life is a cellular automaton due to John H. Conway. The game follows a simple set of rules, but can exhibit complex behavior and has even been shown to be Turing complete [159].

gol consists of a 2D grid of cells. A cell can either be alive or dead. In every iteration, the new state of a cell is decided as follows: Living cells remain alive only if they are surrounded by two or three alive neighbors. Dead cells become alive if they are surrounded by three alive neighbors.

The most straightforward Game of Life implementation calculates the new state of *every* cell depending on its neighborhood. gol follows a different approach. We only simulate cells that are alive or may become alive in the next iteration (*alive-candidates*). Candidates are dead neighbors of alive cells. This approach has a lower expected runtime complexity because most cells are dead most of the time.

7.8.1 Data Structure

This application has four classes (Figure 7.20). An abstract class `Agent` with two subclasses `Alive` and `Candidate`, and another class `Cell`. Cells are created at the beginning of the simulation and stored in a global array (static class variable) `Cell::cells`. A cell can contain an agent or be empty. Agents store references to their cell in the form of an integer index into the global cells array. This requires less memory than an 8-byte pointer and makes it easier to determine all neighboring cells without storing explicit adjencency lists as in wa-tor.

The main problem of gol is space efficiency. A typical Game of Life implementation requires only 1 bit to encode the state of a cell. In gol, every cell requires at least 8 bytes in the form of an `Agent` pointer. Therefore, even though gol has a lower expected runtime complexity in terms of the number of processed cells, a naive implementation that spawns one GPU thread per cell and encodes the cell state with 1 bit/byte is usually much faster. However, our gol data structure and computation strategy can serve as a blueprint for other cellular automata that have to maintain a more complex state for each cell [45, 211, 109].

7.8.2 Application Implementation

Before entering the main application loop, gol loads the initial game state from a PBM (*Portable BitMap*) file. Such files can be created with common image manipulation programs such as GIMP. We benchmarked DYNASOAR with Rendell's universal turing machine pattern [159]. Every loop iteration of gol consists of four parallel do-all operations.

1. **Prepare Candidate Action:** Decide whether a candidate should be deleted, upgraded to an `Alive` cell or remain as is. Parallel do-all: `Candidate::prepare`
2. **Prepare Alive Action:** Decide whether an alive cell should be deleted (downgraded to `Candidate`) or stay as is. Parallel do-all: `Alive::prepare`
3. **Perform Candidate Action:** Perform the action determined in Step 1. Parallel do-all: `Candidate::update`
4. **Perform Alive Action:** Perform the action determined in Step 2. If this is a newly created object (Step 3), create `Candidates` on all surrounding cells. Requires special handling to ensure that no two objects create a `Candidate` on the same cell. Parallel do-all: `Alive::update`

Step 4: Perform Alive Action This step is the most challenging part of the application (Listing 7.18). Newly created `Alive` objects must spawn `Candidate` objects around them, but we have to ensure that we do not create multiple objects per cell.

Figure 7.21 illustrates this problem with an example. A_2 and A_3 are newly created `Alive` objects. These must create `Candidate` objects on all empty, surrounding cells. C_{12} is a neighbor of both A_2 and A_3, but only one of them should create a `Candidate` at that location.

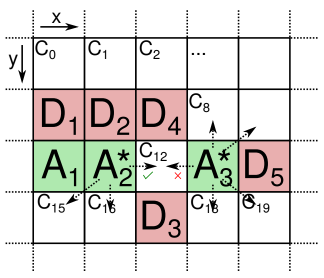
To that end, we specify an order among `Alive` objects that determines which object is creating a `Candidate` object: top to bottom, left to right. E.g., with respect to C_{12}, the `Alive` object on C_6 should create the candidate. However, that cell does not have a newly created `Alive` object. Therefore, we check C_{11} next. This cell contains A_2, a newly created `Alive` object, so it is that object's responsibility to create a candidate on C_{12}. Every newly created `Alive` object can by itself scan the neighborhood of every surrounding, dead (empty) cell, to determine if it should create a candidate at that location.

7.8.3 generation: Generational Cellular Automaton

generation is an extension of gol. When an `Alive` cell dies, it stays around for a constant number of iterations before it is replaced with a `Candidate` and can subsequently become alive again.

This application is based on the rule *Burst* [204] (0235678/3468/255): An alive cell remains alive if it has 0, 2, 3, 5, 6, 7 or 8 alive, neighboring cells. Otherwise, the `Alive` object dies but blocks the cell for 255 more iterations. A dead cell becomes alive if it has 3, 4, 6 or 8 alive, neighboring cells.

Figure 7.22 shows a screenshot of generation. The red areas indicate `Candidates`. The black areas indicate `Alive` objects. `Alive` objects that are already dead but still block a cell are shown in gray.

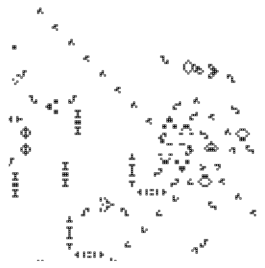


A_2: maybe_create_candidate at:
{C_{12}, C_{15}, C_{16}}

A_3: maybe_create_candidate at:
{C_8, C_9, C_{12}, C_{18}, C_{19}}

Which object should create C_{12}?
First new Alive object among on cells:
(C_6, C_{11}, C_{16}, C_7, C_{17}, C_8, C_{13}, C_{18})
$\Rightarrow$ C_{11}: A_2

(A) Cell interaction: Creating new `Candidates`



(B) Screenshot (UTM)

FIGURE 7.21: Cell interaction and screenshot of gol. Green blocks = `Alive`, red blocks = `Candidate`, star = newly created.

LISTING 7.18: Creating `Candidate` objects on dead cells

```
__device__ void Alive::update() {
  if (is_new_) { create_candidates(); }
  else {
    if (action_ == kActionDie) { // Replace with Candidate.
      Cell::cells[cell_id_]->agent_ = new(device_allocator) Candidate(cell_id_);
      destroy(device_allocator, this);
    }
  }
}

__device__ void Alive::create_candidates() {
  for (int dx = -1; dx < 2; ++dx) {
    for (int dy = -1; dy < 2; ++dy) {
      int nx = cell_id_ % SIZE_X + dx;
      int ny = cell_id_ / SIZE_X + dy;
      if (nx > -1 && nx < SIZE_X && ny > -1 && ny < SIZE_Y) {
        if (Cell::cells[ny*SIZE_X + nx]->agent_ == nullptr) {
          maybe_create_candidate(nx, ny);
        }
      }
    }
  }
}

__device__ void Alive::maybe_create_candidate(int x, int y) {
  // Check neighborhood of cell to determine who should create Candidate.
  for (int dx = -1; dx < 2; ++dx) {
    for (int dy = -1; dy < 2; ++dy) {
      int nx = x + dx;
      int ny = y + dy;
      if (nx > -1 && nx < SIZE_X && ny > -1 && ny < SIZE_Y) {
        Alive* alive = Cell::cells[ny*SIZE_X + nx]->agent_->cast<Alive>();
        if (alive != nullptr && alive->is_new_) {
          if (alive == this) {
            Cell::cells[y*SIZE_X + x]->agent_ =
                new(device_allocator) Candidate(y*SIZE_X + x);
          } // else: Created by another Alive.
          return;
        }
      }
    }
  }
}
```

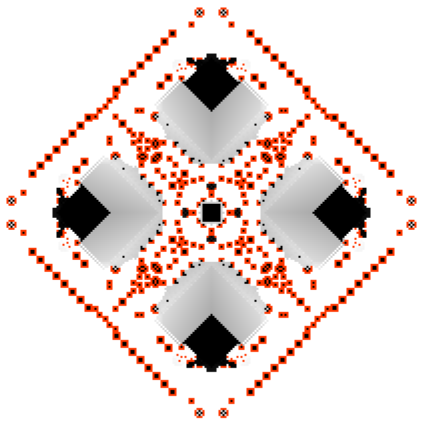


FIGURE 7.22: Screenshot of generation

7.9 Conclusion

In this chapter, we presented the design and implementation of nine applications from different domains with the object-oriented SMMO programming model. This chapter speaks to the importance and broadness of the SMMO model.

This chapter also highlights the benefits of object-oriented programming: In most applications, object-oriented abstractions greatly improve the code readability compared to a hand-written SOA data layout, mostly due to C++'s member-of operators. An object-oriented programming style also provides stronger type guarantees in C++, which can improve developer productivity. Some applications are more difficult to implement without dynamic object allocation, which we consider an essential feature of object-oriented programming. Finally, all applications exhibit an inherent object structure of abstract or real-world entities and, needless to say, we would like the application source code to reflect this structure.

Chapter 8

Conclusion

GPU programming is challenging. For best performance, programmers have to write GPU programs in low-level C/C++ dialects and adopt a SIMD/GPU-specific programming style to avoid slowdowns due to inefficient memory access or control flow divergence. Even though high-level GPU programming languages and libraries exist, these systems often fail to deliver the performance of hand-tuned CUDA/OpenCL code.

Our goal is to make GPU programming available to a wider range of developers by providing better support for object-oriented programming (OOP) in low-level and high-level programming languages. Object-oriented programming is often seen as too inefficient for high-performance computing, but as we have shown in this thesis, object-oriented code can achieve competitive performance if properly optimized. Inefficient device memory access is often the biggest performance problem of object-oriented GPU code and we have demonstrated through various prototypes how to achieve good memory coalescing and cache utilization on GPUs.

- IKRA-RUBY: We developed a Ruby library for array-based GPU computing in a high-level, object-oriented language. Kernel fusion of functional, parallel array operations such as `map`, `reduce` or `stencil` allows programmers to compose complex computations from small building blocks in a modular, object-oriented way, without losing performance compared to a single monolithic CUDA kernel. We also showed how to express kernel fusion as part of the static type inference process.
- *Single-Method Multiple-Objects (SMMO)*: In the object-oriented SMMO programming model, a computation is expressed as running a method on all existing objects of a type. Such computations are well suited for SIMD parallelism and they achieve good performance on GPUs, because the expected amount of branch divergence is low. We demonstrated that a variety of applications and programming patterns from different domains can be expressed in SMMO, ranging from social/physical simulations over BFS graph traversals to dynamic tree updates/constructions.
- IKRA-CPP: We developed a C++/CUDA library for SMMO applications. Most notably, IKRA-CPP provides an embedded data layout DSL for the Structure of Arrays (SOA) data layout. The SOA data layout is a well-studied best practice for GPU programmers and results in a significant speedup of SMMO application code compared to a traditional AOS data layout. While standard C++/CUDA does not allow programmers to use object-oriented abstractions with custom data layouts such as SOA, IKRA-CPP programmers can enjoy the benefits of object-oriented programming together with the performance improvements of the SOA data layout.

- DYNASOAR: Dynamic memory allocation is one of the corner stones of object-oriented programming. However, it is not supported well on GPUs. Most notably, existing dynamic memory allocators care only about data placement and miss key optimizations for efficient access of allocations. Building on top of IKRA-CPP, DYNASOAR is a lock-free, dynamic memory allocator that optimizes the access of allocated memory with an SOA data layout and an efficient parallel do-all operation. Compared to other state-of-the-art allocators, DYNASOAR improves the performance of SMMO applications by a factor of up to 3x.

- COMPACTGPU: Unfortunate allocate-deallocate patterns can lead to increased memory fragmentation of dynamically allocated objects. On GPUs, fragmentation can have an adverse effect on memory coalescing and cache utilization. Therefore, we extended DYNASOAR with a memory defragmentation system COMPACTGPU, which compacts the heap by merging partly occupied memory blocks. In our benchmarks, COMPACTGPU could lower the memory consumption of SMMO applications by up to 14% and improve the runtime performance of SMMO applications by up to 16%.

With our three main prototypes IKRA-CPP, DYNASOAR and COMPACTGPU, GPU programmers can achieve SMMO application performance that is close to, sometimes even faster than, non-OOP CUDA code. At the same time, they get all the benefits of object-oriented programming such as good abstraction, expressiveness, modularity and developer productivity.

We plan to improve IKRA-CPP and IKRA-RUBY in the future. It is our vision that IKRA-CPP will eventually become a part of IKRA-RUBY, such that programmers can develop SMMO applications in a high-level programming language.

Bibliography

[1] Brandon G. Aaby, Kalyan S. Perumalla, and Sudip K. Seal. Efficient Simulation of Agent-based Models on Multi-GPU and Multi-core Clusters. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, pages 29:1–29:10, ICST, Brussels, Belgium, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). doi:10.4108/ICST.SIMUTOOLS2010.8822.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[3] Martín Abadi, Luca Cardelli, Benjamin Pierce, Benjamin Pierce, and Gordon Plotkin. Dynamic Typing in a Statically Typed Language. *ACM Trans. Program. Lang. Syst.*, 13(2):237–268, April 1991. doi:10.1145/103135.103138.

[4] James Abel, Kumar Balasubramanian, Mike Bargeron, Tom Craver, and Mike Phlipot. Applications Tuning for Streaming SIMD Extensions. *Intel Technology Journal*, (Q2), May 1999.

[5] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An Efficient Parallel Heap Compaction Algorithm. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 224–236, New York, NY, USA, 2004. ACM. doi:10.1145/1028976.1028995.

[6] Andy Adinets. CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics, 2017. URL: https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/.

[7] Andrew V. Adinetz and Dirk Pleiter. Halloc: A High-Throughput Dynamic Memory Allocator for GPGPU Architectures. In *GPU Technology Conference*, GTC '14, 2014. URL: http://fliphtml5.com/wihc/nvsq/basic.

[8] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In Pierre Cointe, editor, *ECOOP '96 — Object-Oriented Programming*, pages 142–166, Berlin, Heidelberg, 1996. Springer-Verlag. doi:10.1007/BFb0053060.

[9] Stephen G. Alexander and Craig B. Agnor. N-Body Simulations of Late Stage Planetary Formation with a Simple Fragmentation Model. *Icarus*, 132(1):113–124, 1998. doi:10.1006/icar.1998.5905.

[10] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU Concurrency: Weak Behaviours and Programming Assumptions. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 577–591, New York, NY, USA, 2015. ACM. doi:10.1145/2694344.2694391.

[11] Robert J. Allan. Survey of Agent Based Modelling and Simulation Tools. Technical Report DL-TR-2010-007, Science and Technology Facilities Council, Warrington, United Kingdom, Oct 2010. URL: http://purl.org/net/epubs/work/50398.

[12] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. A Dynamic Hash Table for the GPU. *CoRR*, abs/1710.11246, 2017. arXiv:1710.11246.

[13] Undisclosed Authors. Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture. Technical Report 253665-060US, Intel Corporation, Sept. 2016.

[14] Darius Bakunas-Milanowski, Vernon Rego, Janche Sang, and Chansu Yu. Efficient Algorithms for Stream Compaction on GPUs. *International Journal of Networking and Computing*, 7(2):208–226, 2017. doi:10.15803/ijnc.7.2_208.

[15] Stefania Bandini, Sara Manzoni, and Giuseppe Vizzari. Agent Based Modeling and Simulation: An Informatics Perspective. *Journal of Artificial Societies and Social Simulation*, 12(4), 2009. URL: http://jasss.soc.surrey.ac.uk/12/4/4.html.

[16] Rajkishore Barik, Rashid Kaleem, Deepak Majeti, Brian T. Lewis, Tatiana Shpeisman, Chunling Hu, Yang Ni, and Ali-Reza Adl-Tabatabai. Efficient Mapping of Irregular C++ Applications to Integrated GPUs. In *Proceedings of 2014 Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 33:33–33:43, New York, NY, USA, 2014. ACM. doi:10.1145/2581122.2544165.

[17] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, 1986. doi:10.1038/324446a0.

[18] Eli Bendersky. The many faces of operator new in C++, 2011. URL: https://eli.thegreenplace.net/2011/02/17/the-many-faces-of-operator-new-in-c.

[19] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM. doi:10.1145/378993.379232.

[20] Paul Besl. A case study comparing AoS (Arrays of Structures) and SoA (Structures of Arrays) data layouts for a compute-intensive loop run on Intel Xeon processors and Intel Xeon Phi product family coprocessors. Technical report, Intel Corporation, 2013.

[21] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient Stream Compaction on Wide SIMD Many-core Architectures. In *Proceedings of the Conference on High*

Performance Graphics 2009, HPG '09, pages 159–166, New York, NY, USA, 2009. ACM. doi:10.1145/1572769.1572795.

[22] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, September 1999. doi:10.1145/324133.324234.

[23] Jeff Bonwick. The Slab Allocator: An Object-caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference*, USTC '94, Berkeley, CA, USA, 1994. USENIX Association.

[24] Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM. doi:10.1145/97945.97982.

[25] James Brodman, Dmitry Babokin, Ilia Filippov, and Peng Tu. Writing Scalable SIMD Programs with ISPC. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 25–32, New York, NY, USA, 2014. ACM. doi:10.1145/2568058.2568065.

[26] Trevor Brown. *Techniques for Constructing Efficient Lock-free Data Structures*. PhD thesis, University of Toronto, 2017. arXiv:1712.05406.

[27] Trevor Alexander Brown. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, New York, NY, USA, 2015. ACM. doi:10.1145/2767386.2767436.

[28] Martin Burtscher and Keshav Pingali. Chapter 6 – An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm. In Wen mei W. Hwu, editor, *GPU Computing Gems Emerald Edition*, Applications of GPU Computing Series, pages 75–92. Morgan Kaufmann, Boston, 2011. doi:10.1016/B978-0-12-384988-5.00006-1.

[29] John R. Cary, Svetlana G. Shasharina, Julian C. Cummings, John V.W. Reynders, and Paul J. Hinker. Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, 105(1):20–36, 1997. doi:10.1016/S0010-4655(97)00043-X.

[30] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A Domain-specific Approach to Heterogeneous Parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM. doi:10.1145/1941553.1941561.

[31] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 285–299, New York, NY, USA, 1995. ACM. doi:10.1145/217838.217868.

[32] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious Structure Definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 13–24, New York, NY, USA, 1999. ACM. doi:10.1145/301618.301635.

[33] James O. Coplien. Curiously Recurring Template Patterns. *C++ Report*, 7(2):24–27, February 1995.

[34] NVIDIA Corporation. NVIDIA GameWorks Documentation: Memory Statistics - Global, 2015. URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/memorystatisticsglobal.htm.

[35] NVIDIA Corporation. NVIDIA Tesla P100 Whitepaper, 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[36] NVIDIA Corporation. CUDA C Best Practices Guide: Coalesced Access to Global Memory, 2018. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#coalesced-access-to-global-memory.

[37] NVIDIA Corporation. CUDA ZONE Forum: Constant Cache, 2018. URL: https://devtalk.nvidia.com/default/topic/1035182/cuda-programming-and-performance/constant-cache/.

[38] NVIDIA Corporation. Press Release: NVIDIA-Accelerated Supercomputers Hit New Highs on TOP500 List, 2018. URL: https://nvidianews.nvidia.com/news/nvidia-accelerated-supercomputers-hit-new-highs-on-top500-list.

[39] NVIDIA Corporation. CUDA C Programming Guide: Compute Capability 6.x: Global Memory, 2019. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#global-memory-6-x.

[40] NVIDIA Corporation. CUDA C Programming Guide: Device Memory Access, 2019. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#device-memory-accesses.

[41] NVIDIA Corporation. CUDA Toolkit Documentation: PTX Writer's Guide to Interoperability, 2019. URL: https://docs.nvidia.com/cuda/ptx-writers-guide-to-interoperability.

[42] NVIDIA Corporation. Pascal Tuning Guide: Instruction Scheduling, 2019. URL: https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#sm-scheduling.

[43] NVIDIA Corporation. Pascal Tuning Guide: Unified L1/Texture Cache, 2019. URL: https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#l1-cache.

[44] Cederman Daniel, Gidenstam Anders, Ha Phuong, Sundell Hkan, Papatriantafilou Marina, and Tsigas Philippas. *Lock-Free Concurrent Data Structures*, chapter 3, pages 59–79. Wiley-Blackwell, 2017. doi:10.1002/9781119332015.ch3.

[45] Debasis Das. A Survey on Cellular Automata and Its Applications. In P. Venkata Krishna, M. Rajasekhara Babu, and Ezendu Ariwa, editors, *Global Trends in Computing and Communication Systems*, pages 753–762, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-29219-4_84.

[46] Kei Davis and Jörg Striegnitz. Parallel Object-Oriented Scientific Computing Today. In Frank Buschmann, Alejandro P. Buchmann, and Mariano A. Cilia, editors, *Object-Oriented Technology. ECOOP 2003 Workshop Reader*, pages 11–16, Berlin, Heidelberg, 2004. Springer-Verlag. doi:10.1007/978-3-540-25934-3_2.

[47] Simon Garcia De Gonzalo, Sitao Huang, Juan Gómez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu. Automatic Generation of Warp-level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 73–84, Piscataway, NJ, USA, Feb 2019. IEEE Press. doi:10.1109/CGO.2019.8661187.

[48] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008. doi:10.1145/1327452.1327492.

[49] Alexander K. Dewdney. Computer Creations: Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. *Scientific American*, 251(6):14–26, 12 1984.

[50] Jose J. Dolado, Mark Harman, Mari C. Otero, and Lin Hu. An Empirical Investigation of the Influence of a Type of Side Effects on Program Comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, July 2003. doi:10.1109/TSE.2003.1214329.

[51] Julian Dolby and Andrew Chien. An Automatic Object Inlining Optimization and Its Evaluation. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '00, pages 345–357, New York, NY, USA, 2000. ACM. doi:10.1145/349299.349344.

[52] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a High-level Language for GPUs: (via Language Support for Architectures and Compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM. doi:10.1145/2254064.2254066.

[53] Roshan M D'Souza, Mikola Lysenko, and Keyvan Rahmani. SugarScape on Steroids: Simulating Over a Million Agents at Interactive Rates. In *Proceedings of the Agent 2007 Conference on Complex Interaction and Social Emergence*, number ANL/DIS-07-2, 2007.

[54] Carlchristian H. J. Eckert. Enhancements of the massively parallel memory allocator ScatterAlloc and its adaption to the general interface mallocMC. Junior thesis. Technische Universität Dresden, October 2014. doi:10.5281/zenodo.34461.

[55] Harold C. Edwards and Daniel A. Ibanez. Kokkos' Task DAG Capabilities. Technical Report SAND2017-10464, Sandia National Laboratories, Albuquerque, New Mexico, USA, 9 2017. doi:10.2172/1398234.

[56] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. SNZI: Scalable NonZero Indicators. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 13–22, New York, NY, USA, 2007. ACM. doi:10.1145/1281100.1281106.

[57] Ahmed ElTantawy and Tor M. Aamodt. MIMD Synchronization on SIMT Architectures. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 11:1–11:14, Piscataway, NJ, USA, 2016. IEEE Press. doi:10.1109/MICRO.2016.7783714.

[58] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How Do API Documentation and Static Typing Affect API Usability? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 632–642, New York, NY, USA, 2014. ACM. doi:10.1145/2568225.2568299.

[59] Joshua M. Epstein and Robert Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. The MIT Press, 1 edition, 1996.

[60] Jörg Esser and Michael Schreckenberg. Microscopic Simulation of Urban Traffic Based on Cellular Automata. *International Journal of Modern Physics C*, 08(05):1025–1036, 1997. doi:10.1142/S0129183197000904.

[61] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 216–225, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/ICPP.2011.45.

[62] Matthias Felleisen. Functional Objects. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, pages 269–269, Berlin, Heidelberg, 2004. Springer-Verlag. doi:10.1007/978-3-540-24851-4_12.

[63] J. Gomes Ferreira. ECOWIN – an object-oriented ecological model for aquatic ecosystems. *Ecological Modelling*, 79(1):21–34, 1995. doi:10.1016/0304-3800(94)00033-E.

[64] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing*, 71(10):3934–3957, 2015. doi:10.1007/s11227-015-1483-z.

[65] Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. You Can Have It All: Abstraction and Good Cache Performance. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 148–167, New York, NY, USA, 2017. ACM. doi:10.1145/3133850.3133861.

[66] Keir Fraser. *Practical lock-freedom*. PhD thesis, February 2004. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[67] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 60–73, New York, NY, USA, 2017. ACM. doi:10.1145/3050748.3050761.

[68] Juan José Fumero, Michel Steuwer, and Christophe Dubach. A Composable Array Function Interface for Heterogeneous Computing in Java. In *Proceedings of the 2014 ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY '14, pages 44:44–44:49, New York, NY, USA, 2014. ACM. doi:10.1145/2627373.2627381.

[69] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. XBFS: EXploring Runtime Optimizations for Breadth-First Search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, pages 121–131, New York, NY, USA, 2019. ACM. doi:10.1145/3307681.3326606.

[70] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[71] Isaac Gelado and Michael Garland. Throughput-oriented GPU Memory Allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 27–37, New York, NY, USA, 2019. ACM. doi:10.1145/3293883.3295727.

[72] Michael J. Gibson, Edward C. Keedwell, and Dragan A. Savić. An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware. *Journal of Parallel and Distributed Computing*, 77:11–25, 2015. doi:10.1016/j.jpdc.2014.10.011.

[73] Paul D. Gilbert. Creating Stand-Alone Smalltalk Applications. Master's thesis, University of Illinois, 1988. URL: https://apps.dtic.mil/docs/citations/ADA197217.

[74] Narendra S. Goel, Samaresh C. Maitra, and Elliott W. Montroll. On the Volterra and Other Nonlinear Models of Interacting Populations. *Rev. Mod. Phys.*, 43:231–276, Apr 1971. doi:10.1103/RevModPhys.43.231.

[75] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the Cache Locality of Memory Allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 177–186, New York, NY, USA, 1993. ACM. doi:10.1145/155090.155107.

[76] Christopher Haine. *Kernel optimization by layout restructuring*. PhD thesis, 2017. URL: http://www.theses.fr/2017BORD0639.

[77] Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *High Performance Computing – HiPC 2007*, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag. doi:10.1007/978-3-540-77220-0_21.

[78] Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC '07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag. doi:10.1007/978-3-540-77220-0_21.

[79] Mark Harris. Optimizing Parallel Reduction in CUDA. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[80] Mark Harris. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops, 2013. URL: https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.

[81] Kevlin Henney. Valued Conversions. *C++ Report*, 12:37–40, July 2000.

[82] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. Design and GPGPU Performance of Futhark's Redomap Construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY '16, pages 17–24, New York, NY, USA, 2016. ACM. doi:10.1145/2935323.2935326.

[83] Johannes Hofmann, Jan Treibig, Georg Hager, and Gerhard Wellein. Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 57–64, New York, NY, USA, 2014. ACM. doi:10.1145/2568058.2568068.

[84] Eric Holk, Ryan Newton, Jeremy Siek, and Andrew Lumsdaine. Region-based Memory Management for GPU Programming Languages: Enabling Rich Data Structures on a Spartan Host. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 141–155, New York, NY, USA, 2014. ACM. doi:10.1145/2660193.2660244.

[85] Holger Homann and Francois Laenen. SoAx: A generic C++ Structure of Arrays for handling particles in HPC codes. *Computer Physics Communications*, 224:325–332, 2018. doi:10.1016/j.cpc.2017.11.015.

[86] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM. doi:10.1145/1941553.1941590.

[87] Ryota Horiguchi, Masahiko Katakura, Hirokazu Akahane, and Masao Kuwahara. A development of a traffic simulator for urban road networks: AVENUE. In *Vehicle Navigation and Information Systems Conference*, VNIS '94, pages 245–250. IEEE Computer Society, Aug 1994. doi:10.1109/VNIS.1994.396833.

[88] Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen-Mei Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *Proceedings of the 10th IEEE International Conference on Computer and Information Technology*, pages 1134–1139. IEEE Computer Society, June 2010. doi:10.1109/CIT.2010.206.

[89] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. Expression Templates Revisited: A Performance Analysis of Current Methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69, 2012. doi:10.1137/110830125.

[90] Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proceedings of the 1986 Conference on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 347–349, New York, NY, USA, 1986. ACM. doi:10.1145/28697.28732.

[91] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. Compiling and Optimizing Java 8 Programs for GPU Execution. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 419–431, Washington, DC, USA, 2015. IEEE Computer Society. doi:10.1109/PACT.2015.46.

[92] ISO/IEC. *ISO International Standard 14882:2011 – Information technology – Programming languages – C++*. International Organization for Standardization, February 2012.

[93] Hans J. M. Baveco and Arnold M. W. Smeulders. Objects for Simulation: Smalltalk and Ecology. *SIMULATION*, 62(1):42–56, 1994. doi:10.1177/003754979406200106.

[94] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, January 2011. doi:10.1109/TPDS.2010.107.

[95] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. Technical report, Citadel Enterprise Americas LLC, 2018. arXiv:1804.06826.

[96] Mark S. Johnstone and Paul R. Wilson. The Memory Fragmentation Problem: Solved? In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, pages 26–36, New York, NY, USA, 1998. ACM. doi:10.1145/286860.286864.

[97] B. Jones, W. Sterner, and J. Schank. Biota: An Object-Oriented Tool for Modeling Complex Ecological Systems. *Mathematical and Computer Modelling*, 20(8):31–48, 1994. doi:10.1016/0895-7177(94)90229-1.

[98] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM. doi:10.1145/165854.165874.

[99] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A Performance Comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581, 2010. arXiv:1005.2581.

[100] Joseph Kehoe. The Specification of Sugarscape. *CoRR*, abs/1505.06012, 2015. arXiv:1505.06012.

[101] Haim Kermany and Erez Petrank. The Compressor: Concurrent, Incremental, and Parallel Compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, New York, NY, USA, 2006. ACM. doi:10.1145/1133981.1134023.

[102] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Efficient Warp Execution in Presence of Divergence with Collaborative Context Collection. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 204–215, New York, NY, USA, 2015. ACM. doi:10.1145/2830772.2830796.

[103] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976. doi:10.1145/360248.360252.

[104] Fredrik Kjolstad, Danny Dig, Gabriel Acevedo, and Marc Snir. Transformation for Class Immutability. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 61–70, New York, NY, USA, 2011. ACM. doi:10.1145/1985793.1985803.

[105] Fredrik Berg Kjolstad and Marc Snir. Ghost Cell Pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 4:1–4:9, New York, NY, USA, 2010. ACM. doi:10.1145/1953611.1953615.

[106] Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic Data Layout Optimizations for GPUs. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, pages 263–274, Berlin, Heidelberg, 2015. Springer-Verlag. doi:10.1007/978-3-662-48096-0_21.

[107] Iisakki Kosonen. *HUTSIM - Simulation Tool for Traffic Signal Control Planning*. PhD thesis, Helsinki University of Technology, 1996.

[108] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[109] Jiří Kroc. Editorial. *Advances in Complex Systems*, 10(supp01):1–3, 2007. doi:10.1142/S0219525907001057.

[110] Sandia National Laboratories. VTK-m Wiki: Virtual Methods in the Execution Environment, 2017. URL: http://m.vtk.org/index.php/Virtual_Methods_in_the_Execution_Environment#Virtual_Methods_in_CUDA.

[111] Bernard Lang and Francis Dupont. Incremental Incrementally Compacting Garbage Collection. In *Papers of the Symposium on Interpreters and Interpretive Techniques*, SIGPLAN '87, pages 253–263, New York, NY, USA, 1987. ACM. doi:10.1145/29650.29677.

[112] Scott Le Grand. Chapter 32 - Broad-Phase Collision Detection with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison-Wesley Professional, first edition edition, 2007.

[113] Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. Vectorized Production Path Tracing. In *Proceedings of High Performance Graphics*, HPG '17, pages 10:1–10:11, New York, NY, USA, 2017. ACM. doi:10.1145/3105762.3105768.

[114] Roland Leißa, Sebastian Hack, and Ingo Wald. Extending a C-like Language for Portable SIMD Programming. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 65–74, New York, NY, USA, 2012. ACM. doi:10.1145/2145816.2145825.

[115] Roland Leißa, Immanuel Haffner, and Sebastian Hack. Sierra: A SIMD Extension for C++. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, pages 17–24, New York, NY, USA, 2014. ACM. doi:10.1145/2568058.2568062.

[116] Florian Lemaitre and Lionel Lacassagne. Batched Cholesky Factorization for tiny matrices. In *Proceedings of the 2016 Conference on Design and Architectures for Signal and Image Processing*, DASIP '16, pages 130–137. IEEE Computer Society, Oct 2016. doi:10.1109/DASIP.2016.7853809.

[117] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *CoRR*, abs/0810.1355, 2008. arXiv:0810.1355.

[118] Chuck Lever and David Boreham. Malloc() Performance in a Multithreaded Linux Environment. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, Berkeley, CA, USA, 2000. USENIX Association.

[119] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 109–118, New York, NY, USA, 2015. ACM. doi:10.1145/2751205.2751232.

[120] Xiaosong Li, Wentong Cai, and Stephen J. Turner. Efficient Neighbor Searching for Agent-Based Simulation on GPU. In *Proceedings of the IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '14, pages 87–96, Washington, DC, USA, 2014. IEEE Computer Society. doi:10.1109/DS-RT.2014.19.

[121] Xiaosong Li, Wentong Cai, and Stephen J. Turner. Cloning Agent-based Simulation on GPU. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '15, pages 173–182, New York, NY, USA, 2015. ACM. doi:10.1145/2769458.2769470.

[122] Xiaosong Li, Wentong Cai, and Stephen J. Turner. Supporting efficient execution of continuous space agent-based simulation on GPU. *Concurrency and Computation: Practice and Experience*, 28(12):3313–3332, 2016. doi:10.1002/cpe.3808.

[123] Yuan Lin and Vinod Grover. NVIDIA Developer Blog: Using CUDA Warp-Level Primitives, 2018. URL: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/.

[124] Barbara H. Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994. doi:10.1145/197320.197383.

[125] Xin Lu, Bo-Yang Chen, Vincent B. C. Tan, and Tong-Earn Tay. Adaptive floating node method for modelling cohesive fracture of composite materials. *Engineering Fracture Mechanics*, 194:240–261, 2018. doi:10.1016/j.engfracmech.2018.03.011.

[126] Justin Luitjens. Global Memory Usage and Strategy, July 2011. GPU Computing Webinar 7/12/2011, Accessed: 2019-02-28. URL: https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GlobalMemory.pdf.

[127] Ching lung Su, Po-Yu Chen, Chun-Chieh Lan, Long-Sheng Huang, and Kuo-Hsuan Wu. Overview and comparison of OpenCL and CUDA technology for GPGPU. In *Proceedings of the 2012 IEEE Asia Pacific Conference on Circuits and Systems*, APCCAS '12, pages 448–451. IEEE Computer Society, Dec 2012. doi:10.1109/APCCAS.2012.6419068.

[128] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An Effective GPU Implementation of Breadth-first Search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM. doi:10.1145/1837274.1837289.

[129] Mikola Lysenko and Roshan M. D'Souza. A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. *Journal of Artificial Societies and Social Simulation*, 11(4):10, 2008. URL: http://jasss.soc.surrey.ac.uk/11/4/10.html.

[130] Sven Maerivoet and Bart De Moor. Transportation Planning and Traffic Flow Models. *ArXiv Physics e-prints*, July 2005. arXiv:physics/0507127.

[131] Deepak Majeti, Rajkishore Barik, Jisheng Zhao, Max Grossman, and Vivek Sarkar. Compiler-Driven Data Layout Transformation for Heterogeneous Platforms. In Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L. Scott, and Josef Weidendorfer, editors, *Euro-Par 2013: Parallel Processing Workshops*, pages 188–197, Berlin, Heidelberg, 2014. Springer-Verlag. doi:10.1007/978-3-642-54420-0_19.

[132] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. doi:10.1145/1807167.1807184.

[133] Naoya Maruyama and Takayuki Aoki. Optimizing Stencil Computations for NVIDIA Kepler GPUs. In *First International Workshop on High-Performance Stencil Computations*, HiStencils '14, 2014.

[134] Amrita Mathuriya, Ye Luo, Anouar Benali, Luke Shulenburger, and Jeongnim Kim. Optimization and Parallelization of B-Spline Based Orbital Evaluations in QMC on Multi/Many-Core Shared Memory Processors. In *2017 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '17, pages 213–223. IEEE Computer Society, May 2017. doi:10.1109/IPDPS.2017.33.

[135] Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. Columnar Objects: Improving the Performance of Analytical Applications. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 197–210, New York, NY, USA, 2015. ACM. doi:10.1145/2814228.2814230.

[136] Xinxin Mei, Kaiyong Zhao, Chengjian Liu, and Xiaowen Chu. Benchmarking the Memory Hierarchy of Modern GPUs. In Ching-Hsien Hsu, Xuanhua Shi, and Valentina Salapura, editors, *Network and Parallel Computing*, pages 144–156, Berlin, Heidelberg, 2014. Springer-Verlag. doi:10.1007/978-3-662-44917-2_13.

[137] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-specific Languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005. doi:10.1145/1118890.1118892.

[138] Duane Merrill and Michael Garland. Single-pass Parallel Prefix Scan with Decoupled Look-back. Technical Report NVR-2016-002, NVIDIA Corporation, Mar. 2016.

[139] Duane Merrill, Michael Garland, and Andrew Grimshaw. High-Performance and Scalable GPU Graph Traversal. *ACM Trans. Parallel Comput.*, 1(2):14:1–14:30, February 2015. doi:10.1145/2717511.

[140] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st ed. edition, 1988.

[141] Maged M. Michael. Safe Memory Reclamation for Dynamic Lock-free Objects Using Atomic Reads and Writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 21–30, New York, NY, USA, 2002. ACM. doi:10.1145/571825.571829.

[142] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004. doi:10.1109/TPDS.2004.8.

[143] Maged M. Michael. Scalable Lock-free Dynamic Memory Allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 35–46, New York, NY, USA, 2004. ACM. doi:10.1145/996841.996848.

[144] Mikołaj Morzy, Tadeusz Morzy, Alexandros Nanopoulos, and Yannis Manolopoulos. Hierarchical Bitmap Index: An Efficient and Scalable Indexing Technique for Set-Valued Attributes. In Leonid Kalinichenko, Rainer Manthey, Bernhard Thalheim, and Uwe Wloka, editors, *Advances in Databases and Information Systems*, ADBIS '13, pages 236–252, Berlin, Heidelberg, 2003. Springer-Verlag. doi:10.1007/978-3-540-39403-7_19.

[145] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *J. Phys. I France*, 2(12):2221–2229, Sept. 1992. doi:10.1051/jp1:1992277.

[146] Lars Nyland, Mark Harris, and Jan Prins. Chapter 31 - Fast N-Body Simulation with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison-Wesley Professional, first edition edition, 2007.

[147] Nathaniel Nystrom, Derek White, and Kishen Das. Firepile: Run-time Compilation for GPUs in Scala. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 107–116, New York, NY, USA, 2011. ACM. doi:10.1145/2047862.2047883.

[148] Amos O. Olagunju and Bassey Akpan. The Benefits of Object-oriented Methodology for Software Development. *International Journal of Information and Computer Science*, 4:39–46, Mar 2015. doi:10.14355/ijics.2015.04.007.

[149] Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. Mostly Concurrent Compaction for Mark-sweep GC. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 25–36, New York, NY, USA, 2004. ACM. doi:10.1145/1029873.1029877.

[150] Undisclosed Authors (SPARC Internal White Paper). *Understanding the Application Binary Interface*, pages 373–378. Springer-Verlag, New York, NY, 1991. doi:10.1007/978-1-4612-3192-9_25.

[151] Parag Patel. Object Oriented Programming for Scientific Computing. Master's thesis, The University of Edinburgh, 2006.

[152] Matt Pharr and William R. Mark. ispc: A SPMD compiler for High-Performance CPU Programming. In *2012 Innovative Parallel Computing*, InPar '12, pages 1–13. IEEE Computer Society, May 2012. doi:10.1109/InPar.2012.6339601.

[153] Everett H. Phillips and Massimiliano Fatica. Implementing the Himeno benchmark with CUDA on GPU clusters. In *Proceedings of the 2010 IEEE International*

Symposium on Parallel Distributed Processing, IPDPS '10, pages 1–10. IEEE Computer Society, April 2010. doi:10.1109/IPDPS.2010.5470394.

[154] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1 edition, 2002.

[155] Max Plauth, Frank Feinbube, Frank Schlegel, and Andreas Polze. A Performance Evaluation of Dynamic Parallelism for Fine-Grained, Irregular Workloads. *International Journal of Networking and Computing*, 6(2):212–229, 2016. doi:10.15803/ijnc.6.2_212.

[156] Viera K. Proulx. Traffic Simulation: A Case Study for Teaching Object Oriented Design. In *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '98, pages 48–52, New York, NY, USA, 1998. ACM. doi:10.1145/273133.273160.

[157] Dan Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000. doi:10.1142/S0129626400000214.

[158] Mohaned Qunaibit, Stefan Brunthaler, Yeoul Na, Stijn Volckaert, and Michael Franz. Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization. In Todd Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:29, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2018.16.

[159] Paul Rendell. *Game of Life Universal Turing Machine*. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-19842-2_5.

[160] Jon Peddie Research. Marktanteile der führenden Hersteller am Absatz von Grafikchips weltweit vom 3. Quartal 2009 bis zum 1. Quartal 2019. Statista - Das Statistik-Portal, 2019. URL: https://de.statista.com/statistik/daten/studie/36476/umfrage/marktanteile-der-hersteller-von-grafikkarten-seit-dem-2-quartal-2009/.

[161] Dirk Riehle. Value Object. In *Proceedings of the 2006 Conference on Pattern Languages of Programs*, PLoP '06, pages 30:1–30:6, New York, NY, USA, 2006. ACM. doi:10.1145/1415472.1415507.

[162] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-Blocks for Performance Oriented DSLs. In Olivier Danvy and Chung chieh Shan, editors, *Proceedings of the 2011 IFIP Working Conference on Domain-Specific Languages*, DSL '11, pages 93–117, 2011. doi:10.4204/EPTCS.66.5.

[163] Sami Rosendahl. CUDA and OpenCL API comparison. T-106.5800 Seminar on GPGPU Programming, 2010.

[164] Nikolay Sakharnykh. Everything You Need to Know about Unified Memory, 2018. URL: http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf.

[165] Shigeyuki Sato and Hideya Iwasaki. A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 79–94, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-10672-9_8.

[166] Henry Schäfer, Benjamin Keinert, and Marc Stamminger. Real-time Local Displacement Using Dynamic GPU Memory Management. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 63–72, New York, NY, USA, 2013. ACM. doi:10.1145/2492045.2492052.

[167] Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A Work-Efficient Step-Efficient Prefix Sum Algorithm. In *Workshop on Edge Computing Using New Commodity Architectures*, 2006.

[168] Jie Shen, Ana Lucia Varbanescu, Xavier Martorell, and Henk Sips. A Study of Application Kernel Structure for Data Parallel Applications. Technical Report PDS-2015-001, Delft University of Technology, 2015.

[169] Jakob Siegel, Juergen Ributzka, and Xiaoming Li. CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator. *Journal of Algorithms & Computational Technology*, 5(2):341–362, 2011. doi:10.1260/1748-3018.5.2.341.

[170] William Silvert. Object-oriented ecosystem modelling. *Ecological Modelling*, 68(1):91–118, 1993. Theoretical Modelling Aspects. doi:10.1016/0304-3800(93)90110-E.

[171] Abhayendra Singh, Shaizeen Aga, and Satish Narayanasamy. Efficiently Enforcing Strong Memory Ordering in GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 699–712, New York, NY, USA, 2015. ACM. doi:10.1145/2830772.2830778.

[172] Hark-Soo Song and Sang-Hee Lee. Effects of wind and tree density on forest fire patterns in a mixed-tree species forest. *Forest Science and Technology*, 13(1):9–16, 2017. doi:10.1080/21580103.2016.1262793.

[173] Tyler Sorensen and Alastair F. Donaldson. Exposing Errors Related to Weak Memory in GPU Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 100–113, New York, NY, USA, 2016. ACM. doi:10.1145/2908080.2908114.

[174] Roy Spliet, Lee Howes, Benedict R. Gaster, and Ana Lucia Varbanescu. KMA: A Dynamic Memory Manager for OpenCL. In *Proceedings of 7th Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 9:9–9:18, New York, NY, USA, 2014. ACM. doi:10.1145/2576779.2576781.

[175] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. In *2012 Innovative Parallel Computing*, InPar '12, pages 1–10. IEEE Computer Society, May 2012. doi:10.1109/InPar.2012.6339604.

[176] Radek Stibora. Building of SBVH on Graphical Hardware. Master's thesis, Faculty of Informatics, Masaryk University, 2016.

[177] Bjarne Stroustrup. Foundations of C++. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 1–25, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-28869-2_1.

[178] Bjarne Stroustrup. Bjarne Stroustrup's C++ Style and Technique FAQ. Is there a "placement delete"?, 2017. URL: http://www.stroustrup.com/bs_faq2.html#placement-delete.

[179] Robert Strzodka. Chapter 31 - Abstraction for AoS and SoA Layout in C++. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 429–441. Morgan Kaufmann, Boston, 2012. doi:10.1016/B978-0-12-385963-1.00031-9.

[180] Robert Strzodka. Data Layout Optimization for Multi-valued Containers in OpenCL. *J. Parallel Distrib. Comput.*, 72(9):1073–1082, September 2012. doi:10.1016/j.jpdc.2011.10.012.

[181] Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. A Source-to-Source OpenACC Compiler for CUDA. In Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L. Scott, and Josef Weidendorfer, editors, *Euro-Par 2013: Parallel Processing Workshops*, pages 178–187, Berlin, Heidelberg, 2014. Springer-Verlag. doi:10.1007/978-3-642-54420-0_18.

[182] Alexandros Tasos, Juliana Franco, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. Extending SHAPES for SIMD Architectures: An Approach to Native Support for Struct of Arrays in Languages. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '18, pages 23–29, New York, NY, USA, 2018. ACM. doi:10.1145/3242947.3242951.

[183] Hayo Thielecke. An introduction to C++ template programming. Technical report, University of Birmingham, Jan 2016.

[184] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a Next-Generation Open Source Framework for Deep Learning. In *Proceedings of Workshop on Machine Learning Systems in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, LearningSys, 2015. URL: http://learningsys.org/papers/LearningSys_2015_paper_33.pdf.

[185] Katsuhiro Ueno, Atsushi Ohori, and Toshiaki Otomo. An Efficient Non-moving Garbage Collector for Functional Languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 196–208, New York, NY, USA, 2011. ACM. doi:10.1145/2034773.2034802.

[186] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, August 1990. doi:10.1145/79173.79181.

[187] Ronald Veldema and Michael Philippsen. Parallel Memory Defragmentation on a GPU. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '12, pages 38–47, New York, NY, USA, 2012. ACM. doi:10.1145/2247684.2247693.

[188] Todd Veldhuizen. C++ Gems. chapter Expression Templates, pages 475–487. SIGS Publications, Inc., New York, NY, USA, 1996.

[189] Kanagaraj Venkatesan, Asaithambi Gowri, and R. Sivanandan. Development of Microscopic Simulation Model for Heterogeneous Traffic using Object Oriented Aproach. *Transportmetrica*, 4(3):227–247, 2008. doi:10.1080/18128600808685689.

[190] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. Object-Relative Addressing: Compressed Pointers in 64-Bit Java Virtual Machines. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 79–100, Berlin, Heidelberg, 2007. Springer-Verlag. doi:10.1007/978-3-540-73589-2_5.

[191] Marek Vinkler and Vlastimil Havran. Register Efficient Dynamic Memory Allocator for GPUs. *Comput. Graph. Forum*, 34(8):143–154, December 2015. doi:10.1111/cgf.12666.

[192] Vasily Volkov. Better Performance at Lower Occupancy. *GPU Technology Conference*, 2010. URL: https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf.

[193] Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html.

[194] Mohamed Wahib and Naoya Maruyama. Scalable Kernel Fusion for Memory-bound GPU Applications. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 191–202, Piscataway, NJ, USA, 2014. IEEE Press. doi:10.1109/SC.2014.21.

[195] Joachim Wahle, Jörg Esser, Lutz Neubert, and Michael Schreckenberg. A Cellular Automaton Traffic Flow Model for Online-Simulation of Urban Traffic. In S. Bandini, R. Serra, and F. Suggi Liverani, editors, *Cellular Automata: Research Towards Industry*, pages 185–193, London, 1998. Springer-Verlag. doi:10.1007/978-1-4471-1281-5_17.

[196] Joachim Wahle, Lutz Neubert, Jörg Esser, and Michael Schreckenberg. A cellular automaton traffic flow model for online simulation of traffic. *Parallel Computing*, 27(5):719 – 735, 2001. Cellular automata: From modeling to applications. doi:10.1016/S0167-8191(00)00085-5.

[197] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A High-performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 11:1–11:12, New York, NY, USA, 2016. ACM. doi:10.1145/2851141.2851145.

[198] Nicolas Weber and Michael Goesele. Auto-tuning Complex Array Layouts for GPUs. In *Proceedings of the 14th Eurographics Symposium on Parallel Graphics and Visualization*, PGV '14, pages 57–64, Aire-la-Ville, Switzerland, Switzerland, 2014. Eurographics Association. doi:10.2312/pgv.20141085.

[199] Nicolas Weber and Michael Goesele. MATOG: Array Layout Auto-Tuning for CUDA. *ACM Trans. Archit. Code Optim.*, 14(3):28:1–28:26, August 2017. doi:10.1145/3106341.

[200] Peter Wegner. Concepts and Paradigms of Object-oriented Programming. *SIGPLAN OOPS Mess.*, 1(1):7–87, August 1990. doi:10.1145/382192.383004.

[201] Sven Widmer, Dominik Wodniok, Nicolas Weber, and Michael Goesele. Fast Dynamic Memory Allocator for Massively Parallel Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 120–126, New York, NY, USA, 2013. ACM. doi:10.1145/2458523.2458535.

[202] Wikipedia contributors. Type punning — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 23-May-2019]. URL: https://en.wikipedia.org/w/index.php?title=Type_punning&oldid=889434798.

[203] Christian Wimmer and Hanspeter Mössenböck. Automatic Array Inlining in Java Virtual Machines. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 14–23, New York, NY, USA, 2008. ACM. doi:10.1145/1356058.1356061.

[204] Mirek Wójtowicz. Mirek's Cellebration: Cellular Automata Rules Lexicon, 2002. Accessed: 2019-02-22. URL: http://psoup.math.wisc.edu/mcell/rullex_gene.html.

[205] Carolin Wolf, Georg Dotzler, Ronald Veldema, and Michael Philippsen. Object Support for OpenMP-style Programming of GPU Clusters in Java. In *27th International Conference on Advanced Information Networking and Applications Workshops*, pages 1405–1410. IEEE Computer Society, March 2013. doi:10.1109/WAINA.2013.62.

[206] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems Software*, ISPASS '10, pages 235–246. IEEE Computer Society, March 2010. doi:10.1109/ISPASS.2010.5452013.

[207] Cliff Woolley. GPU Optimization Fundamentals, 2013. URL: https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf.

[208] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 107–118, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/MICRO.2012.19.

[209] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. Gpucc: An Open-source GPGPU Compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 105–116, New York, NY, USA, 2016. ACM. doi:10.1145/2854038.2854041.

[210] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. doi:10.1145/2509578.2509581.

[211] Naigong Yu, Mingai Li, and Xiaogang Ruan. Applications of Cellular Automata in Complex System Study. *International Journal of Information and Systems Sciences*, 1(3–4):302–310, 2005.

[212] Mengchi Zhang, Roland Green, and Timothy G. Rogers. Characterizing the Runtime Effects of Object-Oriented Workloads on GPUs. In *Proceedings of the 2018 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '18, pages 109–110. IEEE Computer Society, April 2018. doi:10.1109/ISPASS.2018.00019.

[213] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 31–43, New York, NY, USA, 2017. ACM. doi:10.1145/3018743.3018755.

[214] Hong Zheng, Young-Jun Son, Yi-Chang Chiu, Larry Head, Yiheng Feng, Hui Xi, Sojung Kim, and Mark Hickman. A Primer for Agent-based Simulation and Modeling in Transportation Applications. Technical Report FHWA-HRT-13-054, Federal Highway Administration, U.S. Department of Transportation, 11 2013.

[215] Yutao Zhong, Maksim Orlovich, Xipeng Shen, and Chen Ding. Array Regrouping and Structure Splitting Using Whole-program Reference Affinity. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 255–266, New York, NY, USA, 2004. ACM. doi:10.1145/996841.996872.

[216] Xiangyuan Zhu, Kenli Li, Ahmad Salah, Lin Shi, and Keqin Li. Parallel Implementation of MAFFT on CUDA-enabled Graphics Hardware. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 12(1):205–218, January 2015. doi:10.1109/TCBB.2014.2351801.