DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access

Matthias Springer
Tokyo Institute of Technology, Japan
matthias.springer@acm.org

Hidehiko Masuhara
Tokyo Institute of Technology, Japan
masuhara@acm.org

Abstract

Object-oriented programming has long been regarded as too inefficient for SIMD high-performance computing, despite the fact that many important HPC applications have an inherent object structure. On SIMD accelerators, including GPUs, this is mainly due to performance problems with memory allocation and memory access: There are a few libraries that support parallel memory allocation directly on accelerator devices, but all of them suffer from uncoalesed memory accesses.

We discovered a broad class of object-oriented programs with many important real-world applications that can be implemented efficiently on massively parallel SIMD accelerators. We call this class *Single-Method Multiple-Objects* (SMMO), because parallelism is expressed by running a method on all objects of a type.

To make fast GPU programming available to domain experts who are less experienced in GPU programming, we developed DynaSOAr, a CUDA framework for SMMO applications. DynaSOAr consists of (1) a fully-parallel, lock-free, dynamic memory allocator, (2) a data layout DSL and (3) an efficient, parallel do-all operation. DynaSOAr achieves performance superior to state-of-the-art GPU memory allocators by controlling both memory allocation and memory access.

DynaSOAr improves the usage of allocated memory with a Structure of Arrays (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 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 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.

2012 ACM Subject Classification Software and its engineering → Allocation / deallocation strategies; Software and its engineering → Object oriented languages; Computer systems organization → Single instruction, multiple data

Keywords and phrases CUDA, Data Layout, Dynamic Memory Allocation, GPUs, Object-oriented Programming, SIMD, Single-Instruction Multiple-Objects, Structure of Arrays

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.17

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at
`https://dx.doi.org/10.4230/DARTS.5.2.2`
Source code: `https://github.com/prg-titech/dynasoar`

Acknowledgements This work was supported by a JSPS Research Fellowship for Young Scientists (KAKENHI Grant Number 18J14726). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the TITAN Xp GPU used for this research. We would also like to thank Hoang NT, Jonathon D. Tanks and the anonymous reviewers for their comments and suggestions on earlier versions of this paper.

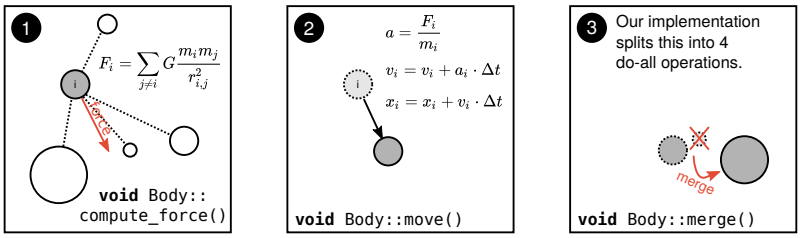


Figure 1 N-body Simulation with Collisions. The simulation consists of multiple do-all operations that are run in a loop for a fixed number of iterations (*time steps*)[1]. Every do-all operation runs in parallel and is a synchronization point: The next one can start only if the previous one has finished.

1 Introduction

General-purpose GPU computing has long been a tedious job, requiring programmers to write hand-optimized, low-level programs. In an attempt to make GPU computing available to a broader range of developers, our efforts are centered around bringing fast object-oriented programming (OOP) to low-level languages such as CUDA.

OOP has a wide range of applications in high-performance computing [9, 39, 6, 21, 17] but is often avoided due to bad performance [52]. Dynamic memory management and the ability/flexibility of creating/deleting objects at any time is one of the corner stones of OOP. 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. In recent years, fast, dynamic memory allocators have been developed for GPUs [60, 37, 69, 66, 7, 58, 25, 31] and demanded by application developers [70, 61, 45, 54, 55, 43, 44], 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, leading to poor data locality and memory bandwidth utilization when accessing allocated memory.

Single-Method Multiple-Objects (SMMO)

We identified a class of high-performance computing applications that can be expressed as object-oriented programs and implemented efficiently on SIMD architectures such as GPUs. We call this class *Single-Method Multiple-Objects* (SMMO). The most fundamental operation of SMMO is parallel *do-all*: Running one method in parallel on all existing objects of a type (*object set*). Such operations fit perfectly with the data-parallel SIMD execution model of GPUs and can be implemented very efficiently. The main challenge lies is the fact that the object set is dynamic: Objects can be created and deleted in GPU code. The main contribution of our work is the design and implementation of a dynamic memory allocator that works well with SMMO applications and runs entirely on the GPU.

SMMO is a broad class of problems with many real-world applications, such as social simulations [27], evacuation simulations [44], predicting wildfire spreading [57], (adaptive [30]) finite element methods [28] or particle systems, to name just a few. As an example, consider the n-body simulation with collisions in Fig. 1. Such simulations are used by astronomers to simulate the collision of galaxies or the formation of planets [4]. Every body is an object in SMMO and the simulation is a sequence of multiple do-all operations.

[1] We implement merging behavior with multiple do-all operations to avoid race conditions.

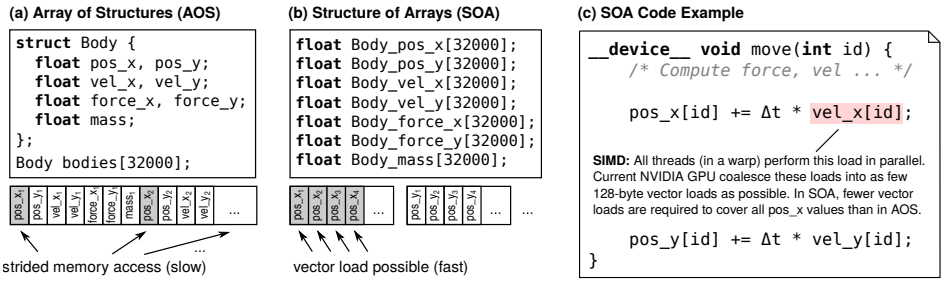


Figure 2 Data Layout of N-body Simulation in AOS and SOA. In SOA, multiple values of a field (e.g., `pos_x`$_1$ and `pos_x`$_2$) can be loaded into a vector register with a single vector load instruction. In AOS, a less efficient, strided memory load or multiple smaller memory loads are necessary, because accessed data is not contiguous.

Structure of Arrays Data Layout

Structure of Arrays (SOA) and Array of Structures (AOS) describe memory layouts for an object set [12] (Fig. 2). In AOS, the standard layout of most platforms, objects are stored as contiguous blocks of memory. In SOA, all values of a field are stored together. This allows for better cache utilization if not all fields are used in a computation. Moreover, it allows for efficient vector loads/stores on SIMD architectures. This is important, because SIMD architectures achieve parallelism by executing the same processor instruction on a *vector* register. Previous work has reported speedups over AOS by multiple factors (e.g., [36]).

Choosing the best data layout for an application is challenging and depends on the data access patterns of the application. Previous work has shown that a mixture of AOS and SOA can sometimes achieve the best performance [29, 40, 68]. How to find good data layouts has been studied before [40, 1] and is out of the scope of this paper. We are focusing on SOA in this work, but DynaSOAr could easily be extended to support other layouts in the future. Unfortunately, custom memory layouts come with a number of disadvantages:

Missing OOP Abstractions. In a hand-written SOA layout, programmers refer to an object with an *integer index* into SOA arrays (Fig. 2c). However, OOP language abstractions (e.g., encapsulation, member access, method calls, type checking, inheritance) only work on object pointers/classes in mainstream languages. To overcome such issues, new languages (e.g., Shapes [29]) and language dialects (e.g., ispc [53]) with built-in support for custom data layouts, as well as data layout libraries/DSLs for existing languages [63, 47, 59] have been developed.

Dynamic Object Set Size. SOA and AOS are not suitable for applications in which the number of objects changes over time, because programmers must specify a maximum object set size per type (e.g., 32,000 in Fig. 2) ahead of time. Dynamic memory allocation solves this problem. As one of our contributions, we show how to allocate memory dynamically while preserving the performance characteristics of SOA.

Subclassing/Inheritance. Inherited methods are shared between superclasses and subclasses. To allow a superclass method implementation to be used for a subclass, the subclass must use the same SOA arrays (and indices) as its superclass. In Columnar Objects, inherited SOA arrays are shared among all objects of all subclasses and newly introduced SOA arrays have a `null` value for objects of a super class [47]. This approach works, but it can waste a considerable amount of memory.

DynaSOAr: A Dynamic Allocator and C++/CUDA DSL for SOA Layout

In this work, we present DynaSOAr, a CUDA framework for SMMO applications. DynaSOAr is a parallel, lock-free, dynamic memory allocator, combined with an efficient do-all operation and an embedded C++/CUDA DSL to enable OOP abstractions with custom object layouts.

We are focusing on DynaSOAr's dynamic memory allocator and do-all operation in this work. DynaSOAr controls the data layout through its memory allocator and 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 (de)allocation is minimized with hierarchical bitmaps. This is important because fragmentation diminishes the benefit of the SOA layout through less efficient vectorized access (more vector transactions are need to access fragmented data) and adversely affects cache performance [32].
- 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 [37] and takes advantage of efficient bit operations/intrinsics.

Contributions and Outline

This paper makes the following contributions.

- The concept of Single-Method Multiple-Objects (SMMO) applications. We show that a variety of important HPC problems are SMMO applications.
- The design and implementation of DynaSOAr, a dynamic object allocator for CUDA; with fast (de)allocation and a 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.
- A concurrent, lock-free, hierarchical bitmap, based on atomic operations and retry loops.
- A comparison and evaluation of existing GPU memory allocators on SMMO applications.

The remainder of this paper is organized as follows. Sec. 2 gives an overview of the design goals of DynaSOAr, focusing on memory access considerations of GPUs. Sec. 3 describes the high-level architecture of DynaSOAr and Sec. 4 explains important optimizations such as hierarchical bitmaps. Sec. 5 compares the design of DynaSOAr with other allocators and Sec. 6 evaluates application performance and fragmentation using microbenchmarks and multiple SMMO applications. Finally, Sec. 7 concludes the paper. Additionally, we provide a systematic correctness analysis in the appendix.

2 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 [29, 64] or ispc [53]), 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, an embedded C++ DSL for object-oriented programming with SOA layout [59]. Its purpose is to make DynaSOAr easier to use for programmers. This paper is mainly about the memory allocator and the do-all operation.

2.1 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. All operations except for `parallel_do` and `parallel_new` are *device* functions that can only be called from GPU code.

- `HAllocatorHandle::parallel_do<T, &T::func>(args...)`: Launches a GPU kernel that runs a member function `T::func` for all objects of type T and subtypes[2] existing at launch time (*parallel do-all*). `T::func` may allocate new objects, but those are not enumerated by the same parallel do-all operation. `T::func` may deallocate any object of different type $U \neq T$, but the object it is bound to (`this`) is the only object of type T 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. In addition to `args...`, the constructor receives an ID i between 0 and $n-1$ (for the i^{th} object) as the first argument.
- `new(d_allocator) T(args...)`: Allocates a new object of type T and returns a pointer to the object. The *placement new* notation [10] 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 that was allocated with `d_allocator`[3].
- `DAllocatorHandle::device_do<T, &T::func>(args...)`: Runs a member function `T::func` for all objects of type T in the current GPU thread. 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 1 shows parts of the n-body simulation of Fig. 1 to illustrate DynaSOAr's API and DSL.

2.2 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:

[2] To avoid branch divergence, we launch a separate kernel for every type.
[3] There is no *placement delete* syntax, so it is a common pattern to provide a separate `destroy` function [62].

```
#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 idx)
      : Body(/*pos_x=*/ random_float(-kMaxPos, kMaxPos),
             /*pos_x=*/ random_float(-kMaxPos, kMaxPos), /* ... */) {}

  __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>();
  // Copy device handle to d_allocator handle.
  cudaMemcpyToSymbol(d_allocator, h_allocator->device_handle(),
                     cudaMemcpyHostToDevice);  // a bit simplified...

  // Create 65536 random body objects. We do not use the new keyword in this example.
  // Alternatively, we could run this in a kernel: new(d_allocator) Body(...)
  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.
  return 0;
}
```

Listing 1 DynaSOAr API Example: Excerpt from an n-body simulation with collisions.

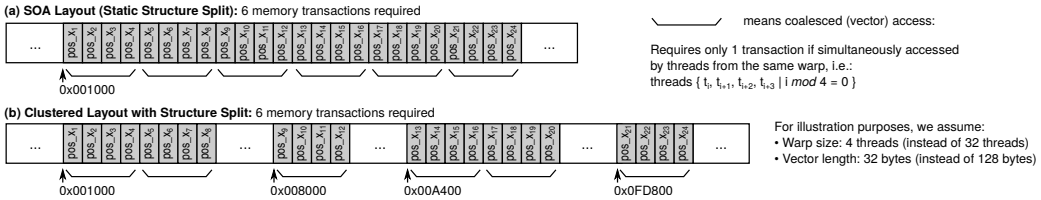


Figure 3 Data Layouts: Number of required memory transactions to read 24 floats simultaneously.

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* [67] (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 [38]. 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 [19].

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 [18].

DynaSOAr achieves good memory access performance with a 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.

2.3 High Density Memory Allocation

A SOA data layout (Fig. 3a) 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 (Fig. 3b) has the same cache/vector performance characteristics as a 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 are located. This gives DynaSOAr more freedom in the placement of allocations and is exploited by its allocation policy.

2.4 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 the same instruction on a vector register.

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

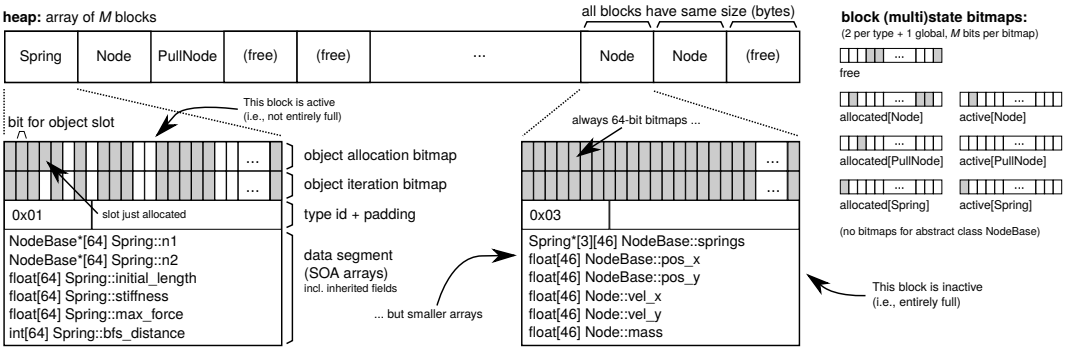


Figure 4 Example: Heap layout for a FEM simulation of a crack in a composite material. The heap is divided into M blocks of equal size. Every block has the same structure: an allocation bitmap, an iteration bitmap, and a type identifier, followed by a data segment storing objects in SOA layout.

To benefit from memory coalescing, the threads of a warp must access addresses on the same 128-byte L1 cache line. In a 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* (spatially local) 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 optimized.

2.5 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 [14] utilize private heaps, but such designs can lead to high memory consumption (*blowup*) [11] and are in feasible on massively parallel architectures with thousands of threads.

State-of-the-art GPU allocators such as ScatterAlloc [60] and Halloc [3] reduce collisions with hashing, which scatters allocations almost randomly on the heap. This would render a 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 paper, DynaSOAr can more than make up for slow allocations with more efficient memory access.

Previous CPU memory allocator designs emphasize mechanisms for reducing false sharing, which can degrade performance [11]. This 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/write to the shared L2 cache or global memory.

3 Architecture Overview

DynaSOAr manages a single, large heap in global memory on device. The heap is divided into M blocks of equal number of bytes. M is determined at compile time based on the block size. Multiple objects of the same type (C++ class/struct) are stored in a block in a Structure of Arrays (SOA) data layout (Fig. 4). Once a block is initialized (*allocated*) for a certain type, only objects of that type can be stored in that block until the block (and all its objects) is deallocated again and reinitialized to a different type.

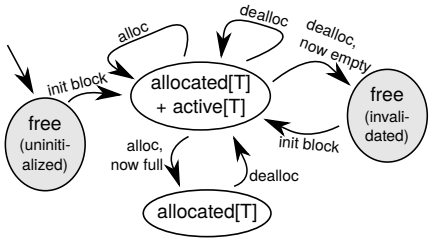


Figure 5 Block State Transitions. At first, blocks are in an uninitialized state. As part of allocation, new active blocks may be initialized (*allocated*). Active blocks become inactive when they are full. Inactive blocks become active again an object is deallocated. Active blocks are invalidated when their last object is deallocated. Invalidated blocks can be reinitialized (to any type) and are handled similar to uninitialized blocks.

The maximum number of objects in a block depends on its type, because different 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 (Fig. 5). Only if no active block could be found, a new block is allocated and becomes active (*slow path*).

3.1 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 do-all operation are not enumerated by the same 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.

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 [49]. 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 lock-free (de)allocation algorithms and simplifies *safe memory reclamation*.

3.2 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

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

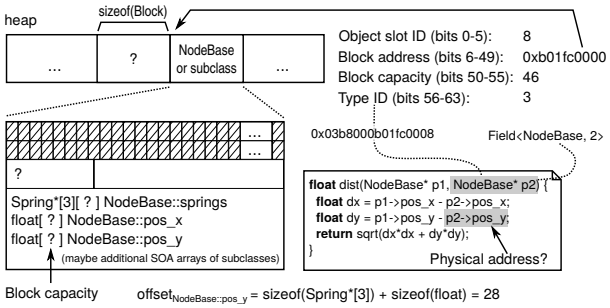


Figure 6 Object Pointer Example. The static type of `p2` is `NodeBase*`. The corresponding block has SOA arrays for `NodeBase` fields and for the additional fields of the runtime type of `p2`. The size of those arrays is not statically known and depends on the runtime type of `p2`.

$s = \text{argmin}_{i \in 1 \ldots n}\, size(T_i)$ is the index of the smallest type, then the capacity N_T of a block of type T is determined as follows.

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

A block of the smallest type T_s has capacity 64. Given a fixed heap size, the size of T_s determines the block size in bytes and thus the number of blocks M.

As soon as a type T is more than twice as big as T_s, the benefit of the SOA layout starts fading away for T, because $N_T < 32$. The maximum amount of memory coalescing can only be achieved with vector loads (cluster sizes) of 32 values (assuming 32-bit scalar types). Furthermore, 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.

3.3 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 object pointer.

In DynaSOAr, an object pointer is not a memory location, but a combination of various components (*fake pointer* [59]), similar to *global references* in *Shapes* [29]. 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 are implicitly converted to `T&` values [35], where `T` is the N-th predeclared field type of `B` [59]. This conversion is defined by our DSL and computes the actual, physical memory location within a data segment.

A DynaSOAr object pointer (Fig. 6) 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. 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[6]. We store additional information in these bits: The 8 most significant bits store the type identifier for fast instance-of checks. The next 6 bits store the capacity of the block. 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.

[6] We experimentally verified this on NVIDIA Maxwell and NVIDIA Pascal.

While in most object-oriented systems, runtime type information is only required for virtual function calls, DynaSOAr needs 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 Fig. 6 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`, so we encode both inside object pointers.

This computation, along with bit-shifting and bit-AND operations for extracting all components from an object pointer, is performed on every field read/write (Sec. B). 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.

3.4 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 maximum number of blocks on the heap.

- 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*.

3.5 Object Slot Allocation

When a new object is created, DynaSOAr allocates memory and runs the constructor on the object pointer. Alg. 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 Alg. 6). 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* in the active block bitmap (Line 12).

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, initialize object bitmaps.
6          allocated[T].set(bid);
7          active[T].set(bid);
8      alloc ← heap[bid].reserve();                  ▷ Reserve an object slot. See Alg. 6.
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);
4  if state = FIRST then
5      active[T].set(bid)
6  else if state = EMPTY then
7      if invalidate(bid) then
8          t ← heap[bid].type;
9          active[t].clear(bid);
10         allocated[t].clear(bid);
11         free.set(bid);
```

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[7].

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*.
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.

[7] We give a more systematic correctness argument in the appendix.

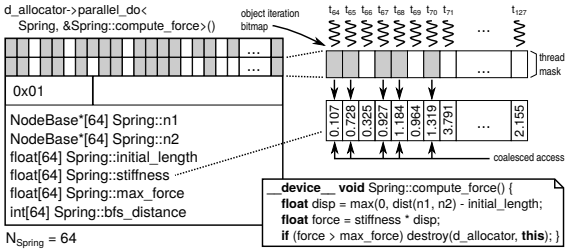


Figure 7 Thread Assignment Example. 64 threads with consecutive IDs (2 warps) are assigned to every allocated block of type `Spring`. Since not all object slots are in use, as indicated by the block iteration bitmap, some threads have no work to do. All other threads can benefit from memory coalescing when reading/writing fields of the object that they are assigned to.

3.6 Object Deallocation

When an object is deleted, DynaSOAr extracts its runtime type T from the object pointer. Then, DynaSOAr runs the C++ destructor and deallocates the memory (Alg. 2) as follows.

We first extract block and object slot IDs from the object pointer and free 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 [48]. 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.

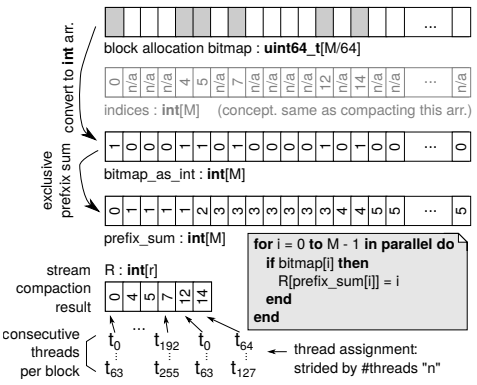
To avoid the block from being modified by other threads, we *invalidate* it. Block invalidation attempts to atomically flipping all bits in the object allocation bitmap from 0 to 1. If this atomic operation failed to flip at least one bit from 0 to 1 (because it was already 1), another thread must have reserved an object slot in the 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.

3.7 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. That method 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. In this case, all those memory accesses can be serviced by efficient 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 (Fig. 7). This leads to good memory coalescing on average. Perfect memory coalescing can be achieved if the following two conditions apply.



$$id_O(tid) = tid \ \% \ N_T \quad (assigned\ obj.\ slot\ idx.)$$

$$id_B(tid) = \left(R\left[\frac{tid + k \cdot n}{N_T}\right] \mid k \in [0; num_B(tid))\right) \quad (assigned\ block\ indices)$$

$$num_B(tid) = \left\lceil \frac{r \cdot N_T - tid}{n} \right\rceil \quad (number\ of\ assigned\ blocks)$$

Figure 8 Example: Compacting block allocation bitmap indices and assigning $n = 256$ threads to 6 allocated blocks with $N_T = 64$. This prefix sum-based implementation retains the order of indices (i.e., R is sorted), but this is not necessary for correctness.

- 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 at the same time.
- 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.

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. Our scheduling strategy always assigns the same object slot position $id_O(tid)$, but in multiple blocks $id_B(tid)$ (strided by the number of threads [34]), to a thread. In those formulas, R is an array of indices of all allocated blocks of type T, i.e., all blocks containing objects of type T. 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.

The array R is required because every thread must 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. DynaSOAr precomputes R before every parallel-do operation (Fig. 8). Conceptually, this is an application of stream compaction [8] and usually implemented with a prefix sum [56, 13]: Given a bitmap of size M, 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. Sec. 4.1 describes how this algorithm is further optimized with hierarchical bitmaps to avoid scanning empty bitmap parts.

4 Optimizations

This section describes performance optimizations that DynaSOAr applies in addition to the SOA data layout to achieve good (de)allocation performance.

4.1 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.

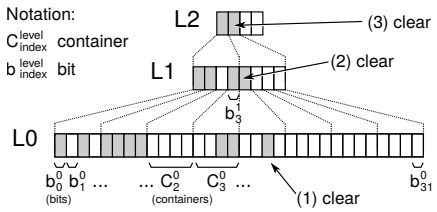


Figure 9 Example: Hierarchical bitmap of size 32 with container size 4 (instead of 64). This example illustrates how (1) a *clear*(18) operation triggers (2) a *clear*(4) operation in the nested bitmap, which triggers (3) a *clear*(1) operation in the next nested bitmap.

Our hierarchical bitmaps are structurally recursive (i.e., bitmaps nested in each other) and hide their hierarchy as an implementation detail from their interface. Such bitmaps are used in database systems [50] and garbage collectors [65], but we do not know of any hierarchical bitmaps that support concurrent modifications.

4.1.1 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_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 (Fig. 9). That bit is set if at least one bit is set in the container.

$$b_i^{l+1} = \bigvee_{k=0}^{63} b_{64 \cdot i+k}^l \qquad (\textit{container consistency})$$

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 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 [26], which have stronger runtime consistency guarantees and require more complex algorithms.

4.1.2 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 `while (!try_clear(pos)) {}`.
- `set(pos)` switches the bit at position `pos` from 0 to 1. Retries until the bit was changed.

Algorithm 3: Bitmap::try_clear(pos) : void. GPU

```
1 cid ← pos / 64;
2 offset ← pos % 64;
3 mask ← 1 << offset;
4 prev ← atomicAnd(&container[cid], ∼mask);
5 success ← (prev & mask) ≠ 0;
6 if success ∧ has_nested ∧ popc(prev) = 1 then
7 |   nested.clear(cid);
8 return success;
```

population cnt.: number of set bits

Algorithm 4: Bitmap::try_find_set() : int. GPU

```
1 if has_nested then
2 |   cid ← nested.try_find_set();
3 |   if cid = FAIL then return FAIL ;
4 else
5 |   cid ← 0;
6 offset ← ffs(container[cid]);
7 if offset = NONE then
8 |   return FAIL
9 else
10 |   return 64*cid + offset;
```

find first set: idx. of 1st set bit

- `try_find_set`() returns the position of an arbitrary bit that is set to 1 or *FAIL* if none was found. 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 identical to `while ((i = try_find_set()) != FAIL && try_clear(i)) {}; return i;`.
- `get`(`pos`) returns 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.

4.1.3 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 [20]. 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 (Fig. 9).

As an example, Alg. 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 that are still in process 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. *clear(pos)*/*set(pos)* ensure that the update is performed eventually by retrying (and spin-blocking the thread) until the update was successful.

Algorithm 5: Bitmap::indices() : int[N]. CPU

```
1  if has_nested then
2  |  selected ← nested.indices()
3  else
4  |  selected ← [0]
5  R ← array(N);
6  r ← 0;
7  for cid ∈ selected in parallel do ▷ GPU
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_bit(c, i);
12 return R.subarray(0, r);
```

idx. of i^{th} set bit in c

4.1.4 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 Alg. 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).

Even if the bitmap has set bits, this operation can fail if it reads an inconsistent combination of containers from different hierarchy levels. For example, consider that 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's performance is affected by such bitmap inconsistencies when searching for active blocks (Alg. 1, Line 2). While bitmap inconsistencies do not affect correctness, they lead to higher fragmentation because DynaSOAr will initialize additional blocks even though objects could be accommodated in already existing blocks. We analyze the effect of such bitmap inconsistencies in our benchmarks (Sec. 6.3).

4.1.5 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 Fig. 8). 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 (Alg. 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. Note that, in contrast to the prefix sum-based implementation of Sec. 3.7, this algorithm does not retain the order of indices and R and is not sorted.

4.2 Reducing Thread Contention

In Alg. 4 and 6, 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

Algorithm 6: Block::reserve() : (int, state). 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;
```

operations became considerably faster with recent GPU architectures [22, 2], 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 [37], 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 (optimized version of Alg. 6). 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 remaining allocation requests. This reduces 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 coalesced.

Bitmap Rotation. Instead of a plain *find first set* (ffs) in Alg. 4 and 6, 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 the same object slots in a block. This is a key optimization technique that improved performance by an order of magnitude.

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 Alg. 6). 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.

4.3 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, Alg. 6 shows how a single object slot is reserved. 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.

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. 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 apply $b \leftarrow b$ *&* *(b - 1)* in a loop $i-1$ times (to clear the first $i-1$ bits) and then calculate *ffs*(b). We omit the details of this optimization here, as it is only one example for a variety of similar low-level optimizations.

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 over the last years. 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 [3] 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 [37] is 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 [69], similar to Hoard [11]. It does not have a general *free* operation and can only deallocate entire heaps, so it is not suitable for SMMO applications.

CircularMalloc (CMalloc) [66] 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 optimization.

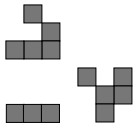
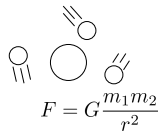
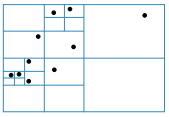
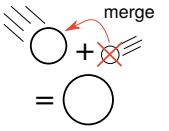
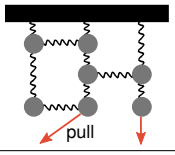
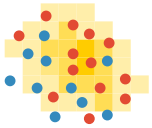
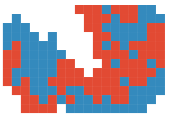
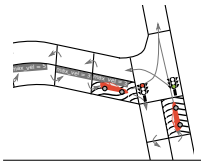
ScatterAlloc [60] hashes allocation requests to memory *pages* depending on their allocation size and the multiprocessor ID. Pages hold allocations of the same size, but slightly smaller requests can be accommodated, leading to internal fragmentation. While DynaSOAr uses hierarchical bitmaps, ScatterAlloc uses hashing with linear probing for finding pages during allocations. For benchmarks, we use mallocMC [24], 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.

6 Benchmarks

We evaluated DynaSOAr with multiple real-world SMMO applications that exhibit different memory allocation patterns (Table 1). We ran all benchmarks 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 them with nvcc (-O3) from the CUDA Toolkit 9.1 on Ubuntu 16.04.4.

Table 1 Description of Benchmark Applications.

	Benchmark Description	#par. do-all	#classes	alloc./dealloc.	smallest class	largest class
	**Game of Life:** A cellular automaton 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 active in the next iteration. Only alive-candidates and alive cells are processed with parallel do-all operations.	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. All objects are allocated upfront. 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 [16] (2D), to evaluate DynaSOAr with dynamic tree data structures. The running time is dominated by the construction/maintenance (i.e., frequent node insert-s/removals) of the quad tree via parallel top-down/bottom-up tree traversals. 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 a 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 [33] and removed. Literature describes extensions that would benefit from dynamic allocation [46].	3 / iteration	5 (4 dyn.)	✗ / ✓	32B, 6 fields	46B, 7 fields
	**Sugarscape:** An agent-based social simulation [27]. 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). Our simulation is quite simple. 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 [23]. 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 starve to death 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 [51]. This simulation can reproduce traffic jams and other real-world traffic 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 ever accessing the memory [42].	n/a	1 (1 dyn.)	✓ / ✓	4B, 1 field	(same)

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

We describe all benchmarks and their implementation in detail (incl. their SMMO structure) on GitHub[8]. 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), every object has a boolean `active` flag to prevent deleted objects from being enumerated in the future. In category (3), classes are merged with the underlying static cell data structure, which wastes memory in case of empty cells (Sec. 6.2). Category (4) applications cannot be implemented with 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 components maintain arrays for allocated and deleted objects of each type. Pointers are inserted into these arrays with atomic operations. At the end of a parallel do-all operation, deleted pointers are removed from the array of allocated pointers. Then, the array of allocated pointers is compacted with a prefix sum operation (same as Fig. 8).

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 show the amount of time spent on *parallel enumeration*. This 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 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* per type and one *free slot bitmap*. Allocation bitmaps are also used for `parallel_do` and `device_do`.

The main downside of BitmapAlloc is its inefficient memory usage. It supports only a single allocation size, potentially leading to high internal fragmentation.

[8] `https://github.com/prg-titech/dynasoar/wiki/Benchmark-Applications` (also see artifact)

Table 2 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

6.1 Performance Overview

Fig. 10 shows the running time of all benchmarked SMMO applications. 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.

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 in array [41].

In collision, DynaSOAr/BitmapAlloc enumerate objects with a bitmap scan of the object allocation bitmap (`device_do`; 1 bit/object), more efficently than other allocators. Other allocators read objects pointers from an array (8 bytes/object). The baseline versions 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 takes a long time. DynaSOAr and BitmapAlloc have much more efficient parallel-do operations than other allocators.

sugarscape and wa-tor exhibit a 2D grid structure of cells. Baseline versions take advantage of this geometric structure, leading to more coalesced memory access, while programmers have no control over where objects are placed in memory by dynamic allocators. 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 referred to with 64-bit object pointers, while all baseline versions use 32-bit integer indices. This penalizes especially benchmarks with small objects; their object sizes grow considerably just by switching from 32-bit integers indices to 64-bit pointers.

6.2 Space Efficiency

To evaluate how efficiently allocators manage memory, we gave them the same heap size and experimentally determined the max. problem size before running out of memory (Fig. 11).

For category (1) and (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.

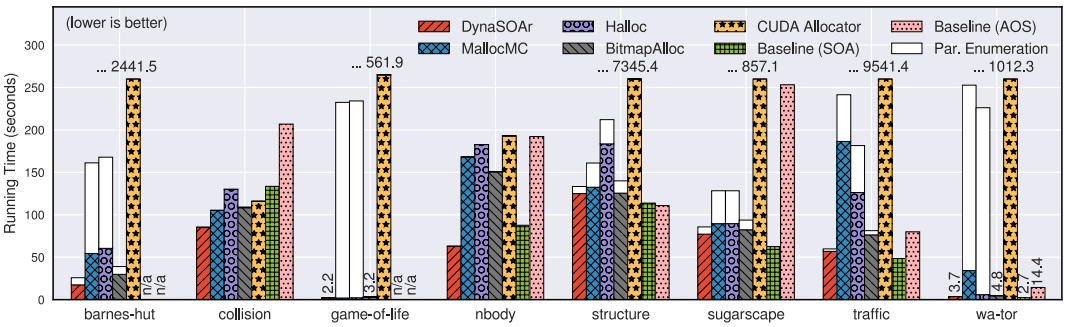


Figure 10 Running Time of SMMO Application Benchmarks. We gave every allocator some extra memory to avoid memory scarcity slowdowns: The heap size is 8 GiB, at least 4 times bigger than the maximum amount of all allocated memory at any point throughout the program execution.

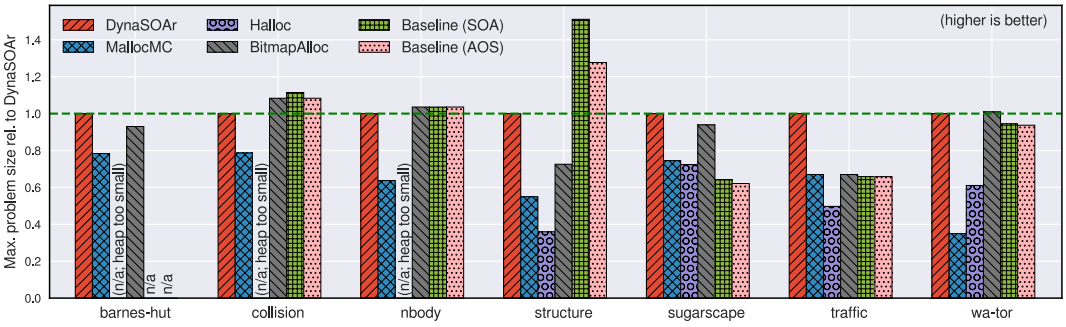
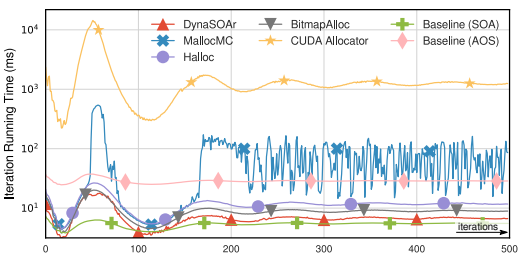
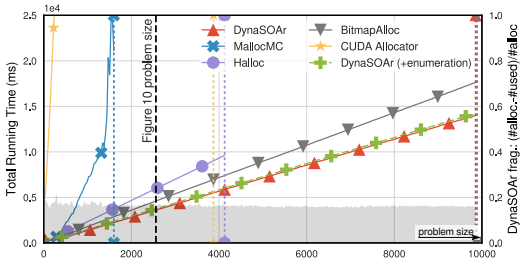


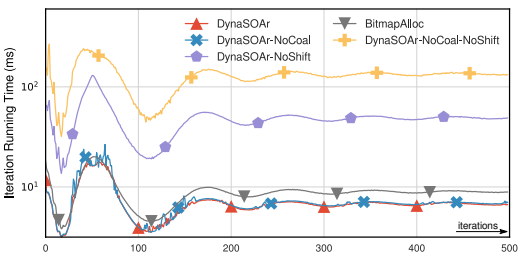
Figure 11 Space Efficiency. We measured the max. problem size of every allocator with the same heap size. Does not take into account enumeration arrays. Results are relative to DynaSOAr.



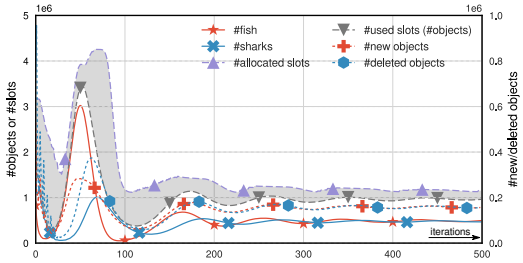
(a) Comparison with other allocators.



(b) Fixed heap size, increasing problem size.



(c) Isolating single DynaSOAr optimizations.



(d) Number of (de)allocations and fragmentation.

Figure 12 Detailed Analysis of wa-tor. Does not include 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 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. Here, DynaSOAr is not as fast as optimized SOA baseline implementations, but it can handle significantly larger problem sizes.

Out of all 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 and not every object.

6.3 Detailed Analysis of wa-tor

wa-tor 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 performance at a massive and at a lower number of concurrent (de)allocations. For a fair comparison of allocators, we do not include time spent on enumeration in this section.

Fig. 12A 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 fragmented heap.

In (B), 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 (B) and (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 (D), DynaSOAr has high fragmentation because many fish objects were deallocated. However, a block can only be deallocated when *all* of its objects are deallocated. The fragmentation level decreases gradually because 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 (B), fragmentation is independent of the problem size and constant at around 18% after 500 Wa-Tor iterations.

We implemented multiple DynaSOAr variants to pinpoint the source of DynaSOAr's speedup over other allocators (Fig. 12C. The most important optimization is the rotation-shifting of bitmaps. Without shifting (*-NoShift), performance degrades severely due to thread contention. 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).

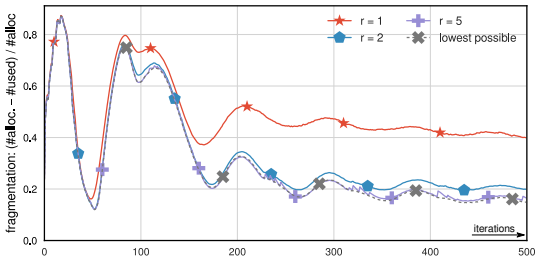
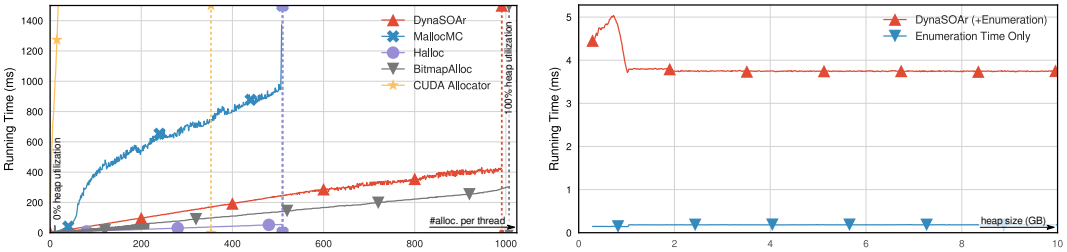


Figure 13 Memory fragmentation (wa-tor) by #active block lookup attemps r (Alg. 6, Line 2). With only 1 retry ($r = 2$), frag. is reduced by 50%. DynaSOAr uses $r = 5$ by default, which is close to the lowest achievable frag. level (i.e., without thread contention). Due to unfortunate alloc.-delloc. patterns, a frag. rate of 0% is not achievable without manually relocating objects or predicting future (de)allocations.



(a) Linux Scalability: Increasing #allocations. **(b)** Scaling study: Heap size (wa-tor).

Figure 14 Scaling Study: Number of Allocations and Heap Size.

In Fig. 13, we experiment with the number of active block lookup attempts before entering the slow path, which strongly affects fragmentation.

6.4 Raw Allocation Performance

The *Linux Scalability* microbenchmark [42] measures the raw (de)allocation time of 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. Allocated memory is never accessed. In Fig. 14A, the x-axis denotes the number of allocations per thread n 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, likely 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.

6.5 Parallel Object Enumeration

The overhead of object enumeration (parallel do-all) is negible in most benchmarks (Fig. 10, Fig. 12B). In Fig. 14B, 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.

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 [53].

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 benchchmarks show that DynaSOAr can achieve significant speedups over state-of-the-art allocators of more than 3x in application code with structured data, due to better memory access performance. DynaSOAr also has a significantly lower memory footprint than other allocators, mainly because DynaSOAr has no internal fragmentation by design 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.

In the future, we will investigate how DynaSOAr can be extended to support virtual functions and other custom object layouts.

References

1 James Abel, Kumar Balasubramanian, Mike Bargeron, Tom Craver, and Mike Phlipot. Applications Tuning for Streaming SIMD Extensions. *Intel Technology Journal*, Q2:13, May 1999.

2 Andy Adinets. CUDA pro tip: Optimized filtering with warp-aggregated atomics. `https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp- aggregated -atomics/`, 2017.

3 Andrew V. Adinetz and Dirk Pleiter. Halloc: A High-Throughput Dynamic Memory Allocator for GPGPU Architectures. In *GPU Technology Conference 2014*, 2014.

4 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`.

5 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`.

6 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, October 2010.

7 Saman Ashkiani, Martin Farach-Colton, and John D. Owens. A Dynamic Hash Table for the GPU. *CoRR*, abs/1710.11246, 2017. `arXiv:1710.11246`.

8 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`.

9 Stefania Bandini, Sara Manzoni, and Giuseppe Vizzari. Agent Based Modeling and Simulation: An Informatics Perspective. *Journal of Artificial Societies and Social Simulation*, 12(4):4, 2009.

10 Eli Bendersky. The many faces of operator new in C++. `https://eli.thegreenplace.net/2011/02/17/the-many-faces-of-operator-new-in-c`, 2011.

11 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`.

12 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.

13 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`.

14 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`.

15 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`.

16 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`.

17 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`.

18 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`.

19 NVIDIA Corporation. CUDA C best practices guide. `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#coalesced-access-to-global-memory`, 2018.

20 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`.

21 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`.

22 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, February 2019. IEEE Press. `doi:10.1109/CGO.2019.8661187`.

23 Alexander K. Dewdney. Computer Creations: Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. *Scientific American*, 251(6):14–26, December 1984.

24 Carlchristian H. J. Eckert. Enhancements of the massively parallel memory allocator ScatterAlloc and its adaption to the general interface mallocMC, October 2014. Junior thesis. Technische Universität Dresden. `doi:10.5281/zenodo.34461`.

25 Harold C. Edwards and Daniel A. Ibanez. Kokkos' Task DAG Capabilities. Technical Report SAND2017-10464, Sandia National Laboratories, Albuquerque, New Mexico, USA, September 2017. doi:10.2172/1398234.

26 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.

27 Joshua M. Epstein and Robert Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*, volume 1. The MIT Press, 1 edition, 1996.

28 Bruce W.R. Forde, Ricardo O. Foschi, and Siegfried F. Stiemer. Object-oriented finite element analysis. *Computers & Structures*, 34(3):355–374, 1990. doi:10.1016/0045-7949(90)90261-Y.

29 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.

30 Dietma Gallistl. The adaptive finite element method. *Snapshots of modern mathematics from Oberwolfach*, 13, 2016. doi:10.14760/SNAP-2016-013-EN.

31 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.

32 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.

33 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.

34 Mark Harris. CUDA pro tip: Write flexible kernels with grid-stride loops. https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/, 2013.

35 Kevlin Henney. Valued Conversions. *C++ Report*, 12:37–40, July 2000.

36 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.

37 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 *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1134–1139, June 2010. doi:10.1109/CIT.2010.206.

38 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.

39 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.

40 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.

41 Florian Lemaitre and Lionel Lacassagne. Batched Cholesky factorization for tiny matrices. In *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 130–137, October 2016. doi:10.1109/DASIP.2016.7853809.

42 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.

43 Xiaosong Li, Wentong Cai, and Stephen J. Turner. Efficient Neighbor Searching for Agent-Based Simulation on GPU. In *Proceedings of the 2014 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.

44 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.

45 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.

46 X. Lu, B.Y. Chen, V.B.C. Tan, and T.E. 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.

47 Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. Columnar Objects: Improving the Performance of Analytical Applications. In *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.

48 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.

49 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.

50 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*, pages 236–252, Berlin, Heidelberg, 2003. Springer-Verlag. doi:10.1007/978-3-540-39403-7_19.

51 Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *J. Phys. I France*, 2(12):2221–2229, September 1992. doi:10.1051/jp1:1992277.

52 Parag Patel. Object Oriented Programming for Scientific Computing. Master's thesis, The University of Edinburgh, 2006.

53 Matt Pharr and William R. Mark. ispc: A SPMD compiler for High-Performance CPU Programming. In *2012 Innovative Parallel Computing (InPar)*, pages 1–13. IEEE Computer Society, May 2012. doi:10.1109/InPar.2012.6339601.

54 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.

55 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.

56 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.

57 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.

58 Roy Spliet, Lee Howes, Benedict R. Gaster, and Ana Lucia Varbanescu. KMA: A dynamic memory manager for OpenCL. In *Proceedings of 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.

59 Matthias Springer and Hidehiko Masuhara. Ikra-Cpp: A C++/CUDA DSL for object-oriented programming with structure-of-arrays layout. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'18, pages 6:1–6:9, New York, NY, USA, 2018. ACM. doi:10.1145/3178433.3178439.

60 Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10. IEEE Computer Society, May 2012. doi:10.1109/InPar.2012.6339604.

61 Radek Stibora. Building of SBVH on Graphical Hardware. Master's thesis, Faculty of Informatics, Masaryk University, 2016.

62 Bjarne Stroustrup. Bjarne Stroustrup's C++ style and technique FAQ. is there a "placement delete"? http://www.stroustrup.com/bs_faq2.html#placement-delete, 2017.

63 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.

64 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.

65 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.

66 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.

67 Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, EECS Department, University of California, Berkeley, August 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html.

68 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.

69 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.

70 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.

Algorithm 7: 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;
```

A Concurrency and Correctness

CUDA has a weak consistency model for global memory access [5]. 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.

A.1 Object Slot Reservation/Freeing

Inside a block, object allocations are tracked with the 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 slot or if a contending thread was faster (Alg. 6, Line 5). Based on their return value, we also know if the current thread reserved the last slot (Line 11), in which case the block should be marked as inactive by the allocation algorithm.

A.1.1 Slot Reservation

`Block::reserve()` (Alg. 6) reserves a single object slot in the block. Our actual implementation 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, the resulting slot at 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).

Algorithm 8: DAllocatorHandle::initialize_block<T>(int bid) : void. GPU

```
1 heap[bid].type ← T;                                   ▷ Volatile write.
2 __threadfence();
3 heap[bid].bitmap ← 0;          ▷ Volatile write, assuming block capacity 64.
```

A.1.2 Slot Freeing

`Block::deallocate(pos)` (Alg. 7) frees a single object slot in the block. To support allocation request coalescing, we have a modified version of this function that can rollback multiple slots at once.

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).

A.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 (Alg. 7, 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 and epoch-based reclamation have been proposed in previous work [15, 48]. 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 in the same location. Other threads may still be able to find the block in the active block bitmap for a while, but object 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 (Alg. 8). Threads allocate objects optimistically and rollback changes should they detect a different block type (Alg. 1, Line 14; also see Sec. A.3).

Algorithm 9: DAllocatorHandle::invalidate(int bid) : bool. GPU

```
1  bitmap_ptr ← &heap[bid].bitmap;
2  before ← atomicOr(bitmap_ptr, 0xFF...F);   ▷ Invalidate (set) all obj. allocation 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[9] (Alg. 9) 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 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 activate the block (Alg. 2, Line 5). However, since we defered block deactivation, this *set(bid)* operation will spin until we deactivate the block (Alg. 9, Line 10). If invalidation rollback empties the block again, we try to invalidate the block one more time[10].

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 defered operation that should have happened earlier when the block type was known to be t.

[9] For presentation reasons, we assume a block capacity of 64 in all algorithms in this paper.
[10] Our actual implementation is iterative instead of recursive.

A.3 Object Allocation

The critical parts during allocations (Alg. 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[11] $t \neq T$. In this case, the allocation must be rolled back (Line 14). All blocks have the same basic structure, so no 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.

A.4 Object Deallocation

The critical part during deallocations (Alg. 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 bit must be set to active. If the current thread deleted 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 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 (Alg. 1, Line 12). This operation expects the bit to be in a set state and it will retry until t_1 sets the bit.

Block Deleted after Delete-First. A thread t_1 deleted the first object of a block. However, before marking the block 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.

[11] Block initialization (Alg. 8) 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 Replaced after Delete-First. A thread t_1 deleted the first object of a block. However, before marking the block 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 the block type.

A.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:*

$$b_i^{l+1} = \bigvee_{k=0}^{63} b_{64\cdot i+k}^l = \mathbb{1}\left(\sum C_{\lfloor i/64 \rfloor}^l > 0\right)$$

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.

▶ **Definition 2** (Semantics of Bitmap Operations). *Every bitmap L_l provides operations for setting and clearing bits (Sec. 4.1.2). These operations may update bits in the higher-level bitmap L_{l+1} if they **s**et the **f**irst bit ($SF_{\lfloor i/64 \rfloor}^l$) or **c**lear the **l**ast bit ($CL_{\lfloor i/64 \rfloor}^l$) of a container C_i^l, respectively:*

$$\underbrace{\mathit{set}(b_i^l) \text{ and } \mathbb{1}\left(\sum C_{\lfloor i/64 \rfloor}^l = 0\right)}_{\text{set-first: } SF_{\lfloor i/64 \rfloor}^l} \text{ then } \mathit{set}(b_{\lfloor i/64 \rfloor}^{l+1}) \qquad \forall i \in [0; 64)$$

$$\underbrace{\mathit{clear}(b_i^l) \text{ and } \mathbb{1}\left(\sum C_{\lfloor i/64 \rfloor}^l = 1\right)}_{\text{clear-last: } CL_{\lfloor i/64 \rfloor}^l} \text{ then } \mathit{clear}(b_{\lfloor i/64 \rfloor}^{l+1}) \qquad \forall i \in [0; 64)$$

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.

▶ **Definition 3** (Legal Bitmap Operations). *Let $\#\mathit{set}(b_i^l)$ and $\#\mathit{clear}(b_i^l)$ be the number of set and clear operations of b_i^l in a multiset of bitmap operations O_l. We call $\$(b_i^l) = \#\mathit{set}(b_i^l) - \#\mathit{clear}(b_i^l)$ the **set-surplus** of b_i^l. O_l is **legal** if it satifies the following conditions.*

1. *Overall bit operation is clear, remain or set:* $\mathbb{S}(b_i^l) \in \{-1, 0, 1\}$.
2. *Bit is in a cleared or set state afterwards:* $b_i^l + \mathbb{S}(b_i^l) \in \{0, 1\}$.

E.g., setting a cleared bit twice and clearing it once ($\mathbb{S} = 2 - 1 = 1$ and $0 + 1 = 1$) is OK, but setting the bit three times and clearing it once ($\mathbb{S} = 3 - 1 = 2$) would be an illegal usage of the bitmap data structure. 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 4.** *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 5.** *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_i^l. Let $\#SF_i^l$ be the number of times a first bit is set in the container and $\#CL_i^l$ be the number of times a last bit is cleared in the container. Then, according to Definition 2, $b_{\lfloor i/64 \rfloor}^{l+1}$ is set $\#SF_i^l$ times and cleared $\#CL_i^l$ times. We have to prove that the set-surplus $\mathbb{S}(b_{\lfloor i/64 \rfloor}^{l+1}) = \#SF_i^l - \#CL_i^l$ 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_k^l. Then, $\mathbb{S}(b_{\lfloor i/64 \rfloor}^{l+1}) = \mathbb{S}(b_k^l) \in \{-1, 0, 1\}$. 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 $b_{\lfloor i/64 \rfloor}^{l+1}$ is consistent with C_i^l after executing O_l. We consider two cases.

1. $b_{\lfloor i/64 \rfloor}^{l+1} = 0$. Therefore, due to initial consistency, $\sum C_{\lfloor i/64 \rfloor}^l = 0$. Therefore, $\#SF_i^l - \#CL_i^l \in \{0, 1\}$, otherwise, O_l would not be legal. Therefore, $b_{\lfloor i/64 \rfloor}^{l+1} + \mathbb{S}(b_{\lfloor i/64 \rfloor}^{l+1}) \in \{0, 1\}$.
   a. If $\#SF_i^l - \#CL_i^l = 0$, then $\vee_{k=0}^{63} b_{64 \cdot i+k}^l = 0$ after O_l. At the same time, $\mathbb{S}(b_{\lfloor i/64 \rfloor}^{l+1}) = 0$, so $b_{\lfloor i/64 \rfloor}^{l+1} = 0$ after O_l, which is consistent with the state of C_i^l after O_l.
   b. If $\#SF_i^l - \#CL_i^l = 1$, then $\vee_{k=0}^{63} b_{64 \cdot i+k}^l = 1$ after O_l. At the same time, $\mathbb{S}(b_{\lfloor i/64 \rfloor}^{l+1}) = 1$, so $b_{\lfloor i/64 \rfloor}^{l+1} = 1$ after O_l, which is consistent with the state of C_i^l after O_l.
2. $b_{\lfloor i/64 \rfloor}^{l+1} = 1$. Therefore, due to initial consistency, $\sum C_{\lfloor i/64 \rfloor}^l > 0$. Therefore, $\#SF_i^l - \#CL_i^l \in \{-1, 0\}$, otherwise, O_l would not be legal. Therefore, $b_{\lfloor i/64 \rfloor}^{l+1} + \mathbb{S}(b_{\lfloor i/64 \rfloor}^{l+1}) \in \{0, 1\}$.
   a. If $\#SF_i^l - \#CL_i^l = -1$, then $\vee_{k=0}^{63} b_{64 \cdot i+k}^l = 0$ after O_l. At the same time, $\mathbb{S}(b_{\lfloor i/64 \rfloor}^{l+1}) = -1$, so $b_{\lfloor i/64 \rfloor}^{l+1} = 0$ after O_l, which is consistent with the state of C_i^l after O_l.
   b. If $\#SF_i^l - \#CL_i^l = 0$, then $\vee_{k=0}^{63} b_{64 \cdot i+k}^l = 1$ after O_l. At the same time, $\mathbb{S}(b_{\lfloor i/64 \rfloor}^{l+1}) = 0$, so $b_{\lfloor i/64 \rfloor}^{l+1} = 1$ after O_l, which is consistent with the state of C_i^l 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 6.** *The bitmap data structure is initially in a consistent state. Furthermore, O_0 is legal. Otherwise, programmers use the bitmap data structure incorrectly.*

B Field Address Computation

This section describes a key implementation technique of the DynaSOAr DSL, that was taken from Ikra-Cpp [59]: Proxy Types. This technique allows us to implement custom data layouts in C++ 11 without breaking OOP abstractions or modifying the compiler.

Even though fields are declared with type `Field<B, N>`, they can be used almost like normal C++ types. There are certain limitations with respect to automatic type deduction (`auto` keyword). Internally, this is implemented with operator overloading, e.g.:

1. **Implicit Conversion Operator:** `Field<B, N>` values can be implicitly converted to the N-th predeclared type in `B`, without an explicit type cast. We call `B` the *base type*.
2. **Member of Object/Pointer Operators:** It is possible to call non-virtual member functions if the base type is (pointer to) a class or struct.
3. **Subscript Operator:** It is possible to use array access syntax (`[]`) for array base types.
4. **Indirection/Address-of Operators:** It is possible to dereference a value of pointer base type and to take the address of a field value.

Listing 2 shows the implementation of the implicit conversion operator. This code first extracts all components that are required for address computation from an object pointer. Then it returns a reference to an object of the base type at the computed memory location.

```
1  // Implicit conversion operator: E.g., convert Field<NodeBase, 2> to float& in Figure 6.
2  template<typename B, int N>
3  Field<B, N>::operator typename B::predeclared_type<N>&() {
4    int offset = ...;  // Computed with template metaprogramming. offset_{B::fieldname} in Figure 6.
5    auto obj_ptr = reinterpret_cast<uint64_t>(this) - 2;  // p2 in Figure 6.
6    // Bits 0-49 and clear 6 least significant bits.
7    auto* block_address = reinterpret_cast<char*>(obj_ptr & 0x3FFFFFFFFFFC0);
8    int obj_slot_id = obj_ptr & 0x3F;  // Bits 0-5
9    int block_capacity = (obj_ptr & 0xFC000000000000) >> 50;  // Bits 50-55
10   auto* soa_array = reinterpret_cast<typename B::predeclared_type<N>*>(
11       block_address + field_offset * block_capacity);
12   return soa_array[obj_slot_id];
13 }
```

Listing 2 Address Computation in Proxy Field Types.