Nested Object Support in a Structure-of-Arrays Dynamic Objector Allocator

(Extended Abstract)

Jizhe Chenxin
chenxinjizhe@prg.is.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

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

Abstract

DynaSOAr is a dynamic object allocator for GPGPU that enables object-oriented programming with an efficient structure-of-arrays (SOA) memory layout. One of the limitations in DynaSOAr is its poor support for nested objects. When a class has a field of another class, the fields of the inner class are allocated in an arrays-of-structure layout. This paper proposes a technique that translates nested class definitions into flat ones by inlining inner classes into top-level classes. We implemented this technique as a Sanajeh domain-specific language that translates Python class definitions into C++ classes using DynaSOAr. Our preliminary evaluation showed that Sanajeh executes a parallel benchmark program with nested objects at almost the same speed as the one with manually flatten classes.

Keywords: Python DSL, Parallel programming, GPGPU, Dynamic object allocation

1 Introduction

General-purpose computing on graphics processing units (GPGPU) is widely used for many types of applications, including numerical computation, simulation, deep learning, and cryptocurrencies.

While the majority of GPGPU programs are currently written in CUDA or OpenCL, there are a few high-level programming languages that attempt to support GPGPU programming with high-level programming abstractions such as the map-reduce [2, 3, 10], objects [7], and arrays [4].

DynaSOAr [7, 8] is a C++ domain-specific language (DSL) that provides object support for GPGPU. Its basic programming model is *single method, multiple objects (SMMO)*, where execution of a method on a large amount of objects of the same class is the source of parallelism. DynaSOAr has two notable features that makes this programming model feasible: (1) it supports dynamic object allocation—GPU threads can dynamically allocate objects in parallel; and (2) it supports structure-of-arrays (SOA) memory layouts, which enables faster memory accesses than a naive (i.e., arrays-of-structures) layout on GPUs. Several practical applications including n-body simulations and traffic simulations are successfully written in DynaSOAr with good performance.

This paper advances object support for GPGPU one step further by efficiently supporting *nested objects.* A nested object is an object that is allocated in a field of another object. In object-oriented programming (OOP), it is common to use nested objects for better maintainability. For example, in an n-body simulation, where each body has its position and velocity, we often define a vector class to represent 3D vectors, and define a body class with the fields for its position and velocity by using the vector class.

Since DynaSOAr does not efficiently support nested objects, we support nested objects by developing a translator, or a DSL called *Sanajeh*, from a program with nested objects into a DynaSOAr program without nested objects. In the rest of the paper, we first review SOA and AOS layouts and the features of DynaSOAr (Section 2), followed by the problem of nested objects in DynaSOAr (Section 3). We then present Sanajeh that transforms nested objects into flat representations (Section 4), whose performance is evaluated by running an n-body simulation (Section 5).

2 Background

2.1 Memory Layout of Objects and GPGPU Performance

Object-oriented programs for GPGPU perform the same computation on many objects of the same class. For example, an n-body simulation expresses each body by using an object allocated on the GPU memory, and updates positions of bodies by using many threads.

There are two kinds of memory layouts when placing many objects, namely the array-of-structures (AOS) layout and the structure-of-arrays (SOA) layout. Those two layouts result in different parallel memory access performances and different programming styles.

AOS is a straightforward layout when we declare an array of a class in CUDA/C++, where the memory of the array is split into contiguous blocks of array elements (i.e., objects of the element class), and each block stores the fields of the respective object. Listing 1 shows an example program.

Listing 1. AOS Layout in CUDA/C++

```
class Body {
  float pos_x;
  float pos_y;
  float vel_x;
  ...
} bodies[1000];

__device__ void update() {
  int i = ...compute index from thread ID...;
  ...bodies[i].pos_x...
}
```

On GPUs, AOS usually leads to lower computation performance due to non-coalesced memory access. Figure 1 shows memory accesses from the first few threads when they access `pos_x` in their respective objects. Since the addresses are interleaved, the memory accesses are slower when compared to the case with an SOA layout.

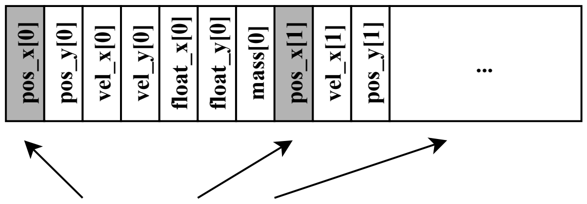

Figure 1. Non-Coalesced Memory Access With AOS Layout

An SOA layout split the memory into arrays of fields, where the fields of the nth object are allocated at the nth element of those arrays. To use SOA in CUDA/C++, we can declare a class that consists of arrays of fields as shown in Listing 2. Accessing the field `pos_x` of the `i`th object can be achieved by first accessing the array for `pos_x`, then accessing the `i`th element, as shown as the expression `bodies.pos_x[i]` in the listing.

Although SOA makes programs verbose, it is widely used in GPGPU applications because of its faster memory accesses than AOS [1, 6, 11]. As shown in Figure 2, when contiguous threads access the same field of their respective objects, the processor reads a contiguous memory region (so-called *memory coalescing*). This is much faster than the same access with AOS.

Listing 2. SOA Layout in CUDA/C++

```
class Body {
  float pos_x[1000];
  float pos_y[1000];
  float vel_x[1000];
  ...
} bodies;

__device__ void update() {
  int i = ...compute index from thread ID...;
  ...bodies.pos_x[i]...
}
```

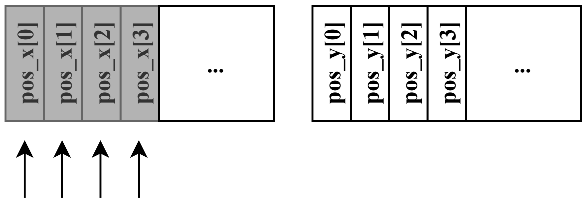

Figure 2. Coalesced Memory Access With SOA Layout

2.2 DynaSOAr

DynaSOAr is a dynamic object allocator for GPGPU [7, 8]. In DynaSOAr, fields of objects are stored in an SOA memory layout, while class declarations and field access expressions follow the standard C++ syntax. This means that, with the previous example, the programmer can write programs in a similar style to Listing 1[1], the memory layout and accesses become the one shown in Figure 2.

DynaSOAr allows GPU threads to allocate objects in parallel. To maximize parallelism, it uses the block-based SOA memory management, in which each block stores objects of the same class in an SOA layout. Though the rest of the paper discusses memory layouts by ignoring blocks for brevity, the discussions are valid for the case with the block-based management in DynaSOAr.

3 Nested Objects and Memory Layout

Nested objects are objects that contain other objects in their fields. From the viewpoint of class declarations, they are objects of classes that have a field of a class type. Below, we first review the advantages of nested objects (Section 3.1), and then explain how nested objects behave in DynaSOAr (Section 3.2).

[1]The actual class declaration in DynaSOAr is slightly more verbose due to its template-based implementation. In this paper, we use the standard C++ syntax for readability.

Listing 3. N-Body Simulation with Nested Objects

```
class Vector {
  float x;
  float y;
  Vector plus(Vector o){
    return *new Vector(x + o.x, y + o.y);
  }
  Vector times(float factor) { ... }
  Vector div(float factor) { ... }
}

class Body {
  Vector pos;
  Vector vel;
  Vector force;
  ...
  __device__ Vector update(Body o) {
    ...
    vel = vel.plus(force.times(kDt).div(mass));
    ...
  }
}
```

3.1 Advantages of Nested Objects

Nested objects are a natural consequence of OOP. In other words, since OOP promotes the use of object abstractions for better modularity, using objects inside of an object improves the modularity of programs. Listing 3 is an n-body simulation written with nested objects. Instead of having pairs of float fields for position, velocity, etc., `Body` uses `Vector` fields. Computation with those fields are also written with methods (or member functions) of `Vector` as shown in the `update` method.

As we can see in the example, nested objects make the intention of the program clearer as we can express a position as a single value instead of a pair of two values. They also make programs easier to be maintained. When we want to modify the algorithm for vectors, we only need to update the methods of the vector class. Nested objects promote program reusability. In the n-body example, `Vector::plus` can be used different places where vector addition is needed.

3.2 Nested Objects in DynaSOAr

There are three options when we want to use nested objects in DynaSOAr.

Nested class field. When a class (say `Body`) declares a field (`pos`) of another class (`Vector`), DynaSOAr allocates a contiguous array for the field (`pos`). However, the nested objects (`Vectors`) are stored in an AOS layout as illustrated in Figure 3.

Pointer to a dynamically allocated object. Since DynaSOAr is a dynamic object allocator, it is possible to allocate nested objects and to store the pointer to the allocated object into a field of the outer object. However, this is not a feasible solution as dynamic allocation incurs time and memory overheads, and requires manual deallocation.

Flattening. The last option is not to use nested objects, rather declare classes with fields of primitive types. This can be done merely by copying field declarations from nested classes to the outer class. In fact, the existing application programs in DynaSOAr are all written in this way to achieve maximum performance. However, this option sacrifices the advantages of nested objects discussed above.

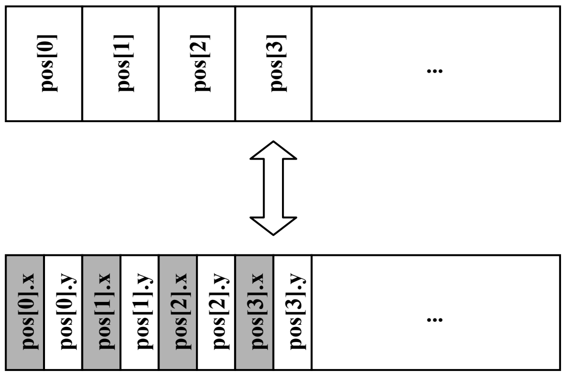

Figure 3. Memory Layout of Nested Object in DynaSOAr

4 Nested Object Support in Sanajeh

4.1 Overview

We propose Sanajeh[2], a Python DSL that efficiently supports nested objects. Though it is designed as a Python DSL, Sanajeh's programming model is identical to DynaSOAr as it serves as a one-to-one translator to DynaSOAr. Sanajeh however efficiently supports nested objects by inlining fields of inner classes when it translates into DynaSOAr classes.

Listing 4 is an excerpt of an n-body simulation in Sanajeh. It is a Python DSL that translates device code (i.e., the code to be executed on GPUs) into DynaSOAr class definitions. Roughly speaking, it offers the same APIs (e.g., for parallel method invocation and object allocation) to the DynaSOAr's; hence it can be considered as a Python wrapper for DynaSOAr. The Python classes that shall be executed on GPUs must have *type hints*[3] for fields, method parameters, and local variables. Type inferencing of local variables might be possible, yet left

[2]Sanajeh is a genus of snake whose fossil showed that it preyed hatching dinosaurs [12].

[3]https://www.python.org/dev/peps/pep-0484/

Listing 4. N-body Simulation in Sanajeh

```
class Vector:
  x: float
  y: float
  ...
  def plus(self, o: Vector) -> Vector:
    return Vector(self.x + o.x, self.y + o.y)
  def times(self, f: float) -> Vector: ...
  def div(self, f: float) -> Vector: ...
  ...

class Body:
  pos: Vector
  vel: Vector
  force: Vector
  ...
  def update(self):
    ...
    self.vel = self.vel.plus(
                 self.force.times(kDt)
                           .div(self.mass))
    ...
```

for future work. As type hints are ignored in the Python interpreters, Sanajeh programs can be interpreted as standard Python programs, if it is provided a library that has the compatible APIs.

4.2 Restrictions on Nested Objects

Sanajeh supports nested objects under the following assumptions on a top-level class declaration.

- Monomorphic: nested objects must have the same static and dynamic types. In other words, subclasses can be used only when the subclass is used of its own type.
- No circular class references: every field of the class is of primitive type or class without circular class references. In other words, (mutually) recursive types (e.g., linked lists) are not allowed for nested objects.
- Confined: a reference to a nested object is never returned from any method of the top-level class.
- No-aliasing: when a nested object is mutable, its reference must not be duplicated.

Currently, it is the programmer's responsibility to satisfy those assumptions. Automatic checking would be possible, which is left for future work.

4.3 Inlining Algorithm

Sanajeh inlines nested classes into flat ones in a preprocess of the transformation from Sanajeh/Python to DynaSOAr/CUDA. An alternative approach would do as a post-process, which we did not take due to a relatively larger syntax in C++, in comparison to Python.

The inlining process consisting of three steps, namely field inlining, expression normalization, method inlining, and field access rewriting. Although the algorithm can work for multi-level nesting (i.e., an inner class contains a field of another class), the following descriptions assume one level nesting for brevity.

4.3.1 Field Inlining. The first step inlines the field declarations in nested classes into the top-level class. Each field is renamed to a unique one. In this paper, we use a concatenation of the original field names.

For example, it converts the `Body` class in Listing 4 into the following one.

```
class Body:
  pos_x: float
  pos_y: float
  vel_x: float
  vel_y: float
  ...
```

4.3.2 Expression Normalization. The second step transforms expressions both in the nested classes and the top-level classes into the A-normal forms [5]. The a-normal form is a restricted style of expressions by replacing nested expressions with local variables. In our case, we replace method call expressions that appear as a parameter to another method call or the right-hand side of a field assignment.

For example, the A-normalization process transforms the assignment in Listing 4:

```
self.vel = self.vel.plus(
             self.force.times(kDt)
                       .div(self.mass))
```

into the following lines.

```
_v2: Vector = self.force.times(kDt)
_v1: Vector = _v2.div(self.mass)
_v0: Vector = self.vel.plus(_v1)
self.vel = _v0
```

Note that the declarations of the local variables `_v0`, `_v1` and `_v2` have types, which are obtained from the return type of the methods.

4.3.3 Method Inlining. Since all method calls are normalized, it simply replaces each method call expression with the `return` expression of the called method after inserted the preceding statements in the method. The formal parameters of the method and `self` are replaced with the actual parameters.

After method inlining, the example code becomes as follows.

```
_v2: Vector =
  Vector(self.force.x*kDt,self.force.y*kDt)
_v1: Vector =
  Vector(_v2.x/self.mass,_v2.y/self.mass)
_v0: Vector =
  Vector(self.vel.x+_v1.x,self.vel.y+_v1.y)
self.vel = _v0
```

4.3.4 Field Access Renaming.

The final step rewrites field accesses by (1) creating local variable declarations for fields of class-type local variables, (2) replacing each assignment expression of a class type with a sequence of assignment expressions of their fields, and (3) replacing each field access expression of a nested object with the access to the inlined field.

For the local variable declaration `_v2: Vector` in the above example, it creates local variables `_v2_x: float` and `_v2_y: float`. It then transforms the line

```
_v2: Vector = Vector(self.force.x*kDt, ...)
```

into

```
_v2.x = Vector(self.force.x*kDt, ...).x
_v2.y = Vector(self.force.x*kDt, ...).y
```

and further into the next lines by eliminating the constructor.

```
_v2.x = self.force.x*kDt
_v2.y = self.force.y*kDt
```

Finally, by replacing `_v2.x`, `self.force.x`, and so forth with `_v2_x`, `self.force_x`, and so forth, respectively, it produces the following lines.

```
_v2_x: float = self.force_x * kDt
_v2_y: float = self.force_y * kDt
_v1_x: float = _v2_x / self.mass
_v1_y: float = _v2_y / self.mass
_v0_x: float = self.vel_x + _v1_x
_v0_y: float = self.vel_y + _v1_y
self.vel_x = _v0_x
self.vel_y = _v0_y
```

4.4 Implementation

We implemented a prototype version of Sanajeh in Python, which is publicly available at https://github.com/prg-titech/Sanajeh/. The size of the translator and the runtime code is roughly three thousand lines of code.

5 Preliminary Evaluation

We evaluated the runtime performance of a program with nested objects by comparing the execution times of the following three versions of an n-body simulation program.

Manually flattened (DynaSOAr): The top-level class contains only primitive type fields after flattened nested objects by hand. As explained in Section 2.2, this version is optimal as objects and fields are allocated in an SOA layout. (Precisely, we obtained this version by manually translating the n-body simulation in DynaSOAr [7], which was originally written without using nested objects.)

Automatically flattened (Sanajeh): The source program that uses nested objects, which are automatically inlined in the Sanajeh-to-DynaSOAr translator. Since the automatic inlining should produce the same class structure as the class in the manually flattened version, we expect this version has the same performance.

Nested (DynaSOAr): The source program uses nested objects and compiled by DynaSOAr. This results in the AOS-in-SOA layout, where the fields of the `Body` class are allocated in the SOA layout, yet the fields of `Vector` in the `Body` are allocated in the standard C++ layout; i.e., the AOS layout. Though this version is written in DynaSOAr by hand, the same code should be produced if we implemented Sanajeh without the automatic inlining feature.

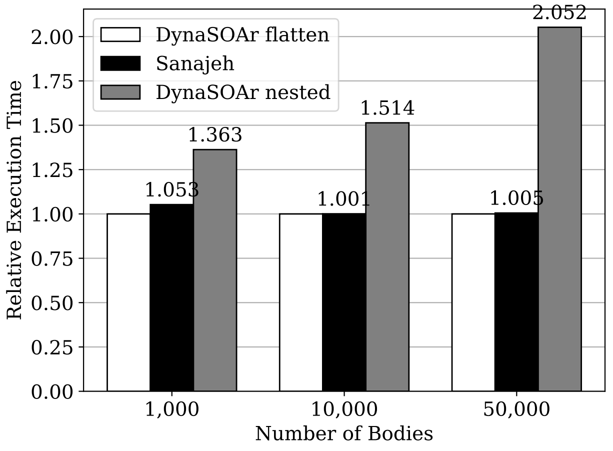

Figure 4. Execution Times of the N-Body Simulation Relative to the DynaSOAr Flatten Version (lower is better)

We executed the three versions with different numbers of bodies. For each configuration, we measured the execution time for 100 time-steps and calculated the average time for one step. We use an NVIDIA TITAN Xp GPU with 12 GB device memory attached to a host processor running Ubuntu 18.04.4. We used `nvcc`/CUDA Toolkit 10.1 with the `-O3` option and Python 3.7.4.

Figure 4 summarizes the execution times relative to the manually flattened cases. As we can see, the automatically flattened version has slight overheads (less than 6%), which are amortized with a larger number of bodies. The nested version are 36%–105% slower than the flatten version. The overheads increase as the number of bodies increases. Although we have not yet investigated the causes, GPU cache memory might be alleviating the AOS access overheads for smaller number of bodies.

6 Conclusion

This paper presents Sanajeh, a Python DSL for GPGPU with support for nested objects. We present the inlining algorithm that transforms a class declaration with nested classes into flat ones, by inlining fields of nested classes, inlining method calls, and renaming field accesses. With the nested object support, we confirmed that a benchmark program runs as fast as the program written without nested objects.

Although it is based on DynaSOAr, our work suggests that transforming classes with nested objects into inlined ones is possible with a simple inlining algorithm, which should also be useful to many OOP languages with SIMD parallelism. We believe the proposed technique will promote further use of OOP in array-based or data-parallel programming.

The current implementation of Sanajeh needs to be improved for practical usability. First, a type inferencing algorithm will let us use local variables without giving types. Second, inlining inner arrays of fixed sizes [9] will widen the flexibility of the language. Third, supporting inheritance in nested classes would increase the expressiveness of the language. Although dynamic dispatching is not feasible on GPUs, we believe that there are many use cases of inheritance that can be implemented with static dispatching.

Acknowledgments

The authors would like to thank Matthias Springer for his advice on this project. Lubis Luthfan helped us running the benchmark program. Youyou Cong, Yudai Tanabe, and the reviewers of the ARRAY 2021 workshop gave valuable comments on the draft version of the paper. The members of the Programming Research Group at Tokyo Institute of Technology gave many comments and suggestions throughout the research.

References

[1] Paul Besl. 2013. *Case study: Comparing Arrays of Structures and Structures of Arrays Data Layouts for a Compute-Intensive Loop.* Technical Report 392271. Intel Corporation. https://software.intel.com/content/www/us/en/develop/articles/a-case-study-comparing-aos-arrays-of-structures-and-soa-structures-of-arrays-data-layouts.html

[2] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming* (Austin, Texas, USA) *(DAMP '11)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

[3] Hidehiko Masuhara and Yusuke Nishiguchi. 2012. A Data-Parallel Extension to Ruby for GPGPU: Toward a Framework for Implementing Domain-Specific Optimizations. In *Proceedings of the 9th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'12)* (Beijing, China). ACM, 3–6. https://doi.org/10.1145/2237887.2237888

[4] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on ML Systems, colocated at the 31st Confernce on Neural Information Processing Systems (NIPS)*.

[5] Amr Sabry and Matthias Felleisen. 1993. Reasoning About Programs in Continuation-Passing Style. *Lisp and Symbolic Computation* 6, 3 (1993), 289–360.

[6] Matthias Springer and Hidehiko Masuhara. 2018. Ikra-Cpp: A C++/CUDA DSL for Object-OrientedProgramming with Structure-of-Arrays Layout. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing (WPMVP 2018)* (Vienna, Austria, 2018-02-24) *(WPMVP'18)*. ACM, New York, NY, USA, Article 6, 9 pages. https://doi.org/10.1145/3178433.3178439

[7] Matthias Springer and Hidehiko Masuhara. 2019. DynaSOAr: A Parallel Memory Allocator for Object-Oriented Programming on GPUs with Efficient Memory Access. In *Proceedings of of European Conference on Object-Oriented Programming (ECOOP'19)* (London, United Kingdom, 2019-07-18) *(Leibniz International Proceedings in Informatics (LIPICS), Vol. 134)*, Alastair Donaldson (Ed.). 17:1–17:37. https://doi.org/10.4230/LIPIcs.ECOOP.2019.17

[8] Matthias Springer and Hidehiko Masuhara. 2019. Massively Parallel GPU Memory Compaction. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM 2019)* (Phoenix, AZ, 2019-06-23), Harry Xu (Ed.). ACM, 14–26. https://doi.org/10.1145/3315573.3329979

[9] Matthias Springer, Yaozhu Sun, and Hidehiko Masuhara. 2018. Inner Array Inlining for Structure of Arrays Layout. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY@PLDI 2018)* (Philadelphia, Pennsylvania, USA). 50–58. https://doi.org/10.1145/3219753.3219760

[10] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 205–217. https://doi.org/10.1145/2784731.2784754

[11] Robert Strzodka. 2012. Abstraction for AoS and SoA Layout in C++ (Chapter 31). In *GPU Computing Gems Jade Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 429–441. https://doi.org/10.1016/B978-0-12-385963-1.00031-9

[12] Jeffrey A Wilson, Dhananjay M Mohabey, Shanan E Peters, and Jason J Head. 2010. Predation upon hatchling dinosaurs by a new snake from the Late Cretaceous of India. *PLoS Biol* 8, 3 (2010), e1000322. https://doi.org/10.1371/journal.pbio.1000322