Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming with Structure-of-Arrays Layout

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

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@acm.org

Abstract

Structure of Arrays (SOA) is a well-studied data layout technique for SIMD architectures. Previous work has shown that it can speed up applications in high-performance computing by several factors compared to a traditional Array of Structures (AOS) layout. However, most programmers are used to AOS-style programming, which is more readable and easier to maintain.

We present Ikra-Cpp, an embedded DSL for object-oriented programming in C++/CUDA. Ikra-Cpp's notation is very close to standard AOS-style C++ code, but data is layed out as SOA. This gives programmers the performance benefit of SOA and the expressiveness of AOS-style object-oriented programming at the same time. Ikra-Cpp is well integrated with C++ and lets programmers use C++ notation and syntax for classes, fields, member functions, constructors and instance creation.

CCS Concepts • **Software and its engineering → Object oriented languages**; *Data types and structures*; *Parallel programming languages*;

Keywords C++, CUDA, Object-oriented Programming, SIMD, Structure of Arrays, Template Metaprogramming

ACM Reference Format:
Matthias Springer and Hidehiko Masuhara. 2018. Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming with Structure-of-Arrays Layout. In *WPMVP'18 : Workshop on Programming Models for SIMD/Vector Processing, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3178433.3178439

WPMVP'18 , February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5646-6/18/02…$15.00
https://doi.org/10.1145/3178433.3178439

1 Introduction

Object-oriented programming (OOP) is a popular language paradigm in general-purpose computing, but not widely-used in high-performance SIMD computing due to insufficient compiler support. Object-oriented code is often several factors slower than tuned, non object-oriented code. In this paper, we present Ikra-Cpp, a DSL for object-oriented high-performance computing embedded in C++/CUDA. Ikra-Cpp allows programmers to write OOP-style code while, behind the curtain, storing data in a Structure of Arrays (SOA) representation; a well-studied best practice for SIMD architectures.

Why Object-oriented Programming? OOP is a well-established language paradigm: It can help programmers write more structured, modular, understandable code [23] with great expressiveness. There are numerous problems that can be expressed elegantly as object-oriented programs. For example, it is straightforward to model problems with real world entities such as particle simulations or agent-based simulations [8, 33] (e.g., traffic simulations [26]) in an object-oriented way. Furthermore, sometimes we want to calculate properties of a network of real-world entities (e.g., ranking/reachability/cluster detection in social networks). Previous work has studied (partly object-oriented) high-performance implementations of such problems [9, 17, 22, 30].

In this paper, we focus on the most basic OOP functionality in C++: Classes with member fields and (non-virtual) methods, instance creation with the `new` keyword and class pointer types. Inheritance/subclassing is not in the scope of this paper and future work.

Why Structure of Arrays? Unfortunately, many performance-critical applications are unacceptably slow when expressed in an object-oriented way due to the way virtually any modern compiler structures objects in main memory: *Arrays of Structures* (AOS). In AOS, every object is stored as a contiguous chunk of data. This is often not ideal for SIMD architectures, which operate on a vector of values.

In *Structure of Arrays* (SOA; *column stores* in database systems), all values of a field are grouped and stored contiguously across the entire object space. SOA is a well-studied, established data layout [4, 19, 20, 24, 28, 31, 39] and CUDA best practice which can save on memory access time (*memory coalescing* [16]), maximize cache usage and allow for vectorization via SIMD instructions [2, 14, 15].

```
1  class Body {
2   public:
3    double pos_x = 0.0;
4    double pos_y = 0.0;
5    double vel_x = 1.0;
6    double vel_y = 1.0;
7    Body(double x, double y)
8        : pos_x(x), pos_y(y) {}
9    void move(double dt) {
10     pos_x = pos_x + vel_x * dt;
11     pos_y = pos_y + vel_y * dt;
12   }
13 };
14 void create_and_move() {
15   Body* b = new Body(1.0, 2.0);
16   b->move(0.5);
17   assert(b->pos_x == 1.5);
18 }
```

(a) C++ Class (AOS Layout)

```
class Body : public SoaLayout<Body, 50> {
 public: IKRA_INITIALIZE_CLASS
  double_ pos_x = 0.0;
  double_ pos_y = 0.0;
  double_ vel_x = 1.0;
  double_ vel_y = 1.0;
  Body(double x, double y)
      : pos_x(x), pos_y(y) {}
  void move(double dt) {
    pos_x = pos_x + vel_x * dt;
    pos_y = pos_y + vel_y * dt;
  }
}; IKRA_HOST_STORAGE(Body);
void create_and_move() {
  Body* b = new Body(1.0, 2.0);
  b->move(0.5);
  assert(b->pos_x == 1.5);
}
```

(b) Ikra-Cpp: AOS Syntax, but SOA Layout

```
double Body_pos_x[50]; double Body_pos_y[50];
double Body_vel_x[50]; double Body_vel_y[50];
int Body_inst = 0;
int new_Body(double x, double y) {
  int id = Body_inst++;
  Body_pos_x[id] = x; Body_pos_y[id] = y;
  Body_vel_x[id] = Body_vel_y[id] = 1.0;
  return id;
}
void Body_move(int id, double dt) {
  Body_pos_x[id] += Body_vel_x[id] * dt;
  Body_pos_y[id] += Body_vel_y[id] * dt;
}
void create_and_move() {
  int b = new_Body(1.0, 2.0);
  Body_move(b, 0.5);
  assert(Body_pos_x[b] == 1.5);
}
```

(c) Hand-written SOA Layout in C++

Figure 1. Comparison of OOP Notation for a simplified 2D N-Body Simulation. Programmers want the notation of (a) but the performance of (c). With Ikra-Cpp, they get the performace of (c) with the notation of (b). A maximum of 50 objects are supported in this example.

Maintaining a SOA layout manually is troublesome. SOA code (Figure 1c) is less readable and expressive than AOS-style code (Figure 1a): Native OOP language constructs such as the `new` keyword or member access notation for fields cannot be used; instead, programmers must keep track of object allocations by themselves, implement constructor logic in global functions, and access field values through arrays. Methods require an explicit `this` parameter, and objects are referenced with integer IDs instead of class pointers.

Why a Library/DSL? Ikra-Cpp is a lightweight C++ library/embedded DSL [11] (around 2500 LoC) implemented entirely in C++ with template metaprogramming, operator overloading, helper classes and preprocessor macros. Our goal is to provide a mechanism that lets programmers write object-oriented AOS-style code (Figure 1b) while transparently laying out objects as SOA. Ikra-Cpp works with every modern C++14 compiler and the Nvidia CUDA Toolkit 9.0 or higher (in GPU mode). We originally designed Ikra-Cpp as an intermediate representation for Ikra-Ruby [32], a GPGPU library with object support for Ruby, but it can also be used standalone by C++/CUDA programmers, which is the focus of this paper. Ikra-Cpp could be implemented as a compiler extension. To the best of our knowledge, no such extension exists for a widely used language. We believe that this is due to the high engineering effort of writing a new compiler or such an invasive compiler extension [21].

Contributions and Outline The main contribution of this paper is twofold. First, to the best of our knowledge, Ikra-Cpp is the first C++ tool for SOA data layout that supports OOP features, most notably member function calls and constructors. Second, Ikra-Cpp supports many SOA features supported by projects discussed in the related work section (e.g., referencing objects with class/struct pointers instead of IDs; c.f. ispc), but with standard C++ syntax (c.f. SoAx) and without relying on an external tool or extending the language (c.f. ispc); everything is implemented in C++.

The remainder of this paper is structured as follows. Section 2 gives an overview of the architecture of Ikra-Cpp and describes its basic functionality. Section 3 explains how data is layed out, how addresses are computed and how a seamless notation can be achieved in C++. Sections 4 and 5 present first benchmarks and discuss related work. Finally, Sections 6 and 7 describe future work and conclude. The Ikra-Cpp source code and all examples are available online[1].

2 Language Overview

In this section, we describe the basic functionality of Ikra-Cpp, focusing on host (CPU) code.

2.1 Notation

A class that is layed out as SOA is called a *SOA class* and its instances are called *SOA objects*. In Ikra-Cpp, every SOA class (Figure 1b) must inherit from `SoaLayout`, a class that provides useful helper methods and type aliases. The maximum number of instances of a SOA class is a compile-time constant and template parameter of `SoaLayout` (Line 1).

SOA objects can only be created with the `new` keyword and must be referenced with pointers. Stack/static allocation

[1] https://github.com/prg-titech/ikra-cpp/tree/benchmark-cgo2018/

```
execute(&Body::move, /*dt=*/ 0.1);
```

(a) Execution on all Body objects.

```
std::array<Body*, 3> bodies =
    {{ Body::get(1), Body::get(5), Body::get(12) }};
execute(bodies, &Vertex::move, /*dt=*/ 0.1);
```

(b) Execution on a collection of Body objects, where the collection can be a `std::array` or a `std::vector`.

```
Body* start = Body::get(1);
int num = Body::size() + 1;
execute(start, num, &Vertex::move, /*dt=*/ 0.1);
```

(c) Execution on a collection of Body objects, given as start pointer and number of objects. In this particular example, all objects are selected.

Figure 2. Example: Running `Body::move` on multiple objects.

```
float Body::distance(float x, float y) {
  float dx = pos_x - x; float dy = pos_y - y;
  return std::sqrt(dx * dx + dy * dy);
}
auto reducer = [] (float acc, float next) {
  return acc + next;
}
float sum_dist = execute_and_reduce(
    reducer, &Body::distance, /*x=*/ 5.0, /*y=*/ 4.0);
float avg_dist = sum_dist / Body::size();
```

Figure 3. Example: Computing the average distance of all bodies from a given point in space. This listing defines a distance method, a *reducer function* that accumulates distance values and performs the execute and reduce parts on the host.

is not allowed, because the fields of an object are not stored as a consecutive chunk of data as in a traditional AOS layout. Programmers must use special *SOA field types* for field declarations, e.g., `double_` (Lines 3–6) instead of `double`.

Supported C++ OOP Features Ikra-Cpp currently supports many but not all OOP features of C++. This paragraph gives an overview of the most important ones. SOA classes are defined with standard C++ notation; templatization and inheritance will be possible in future versions of Ikra-Cpp. Member fields must be declared with SOA field types and must have primitive type (currently: `bool`, `char`, `double`, `float`, `int`). Pointers (of arbitrary base type) and non-primitive types are also possible, but not discussed in this paper[2]. SOA classes can have non-virtual member functions that may be templatized. Constructors (with and without field initializers) are supported. Member functions and constructors may be overloaded. Instance creation is only supported with the `new` keyword. Given a SOA object pointer, members can be accessed with the C++ *member of pointer* (arrow) operator from within and outside a class (Lines 10, 11, 16, 17).

2.2 Executor

The *executor API* allows programmers to perform an operation on a collection of objects of same type. Its usage is optional and programmers can achieve the same functionality manually with hand-written C++ loops or CUDA kernels. The benefits of the executor API are more compact code, abstraction from platform-specific parallelization constructs (e.g., programmers do not have to write CUDA kernels), and easy switching from CPU execution to GPU execution.

Do-All Executor The most basic operation of the executor API is method execution: Given a collection of objects of same type and a method name, execute the method for every object (Figure 2). In host mode, this done sequentially, but future versions of Ikra-Cpp may support thread pool execution. In device mode, Ikra-Cpp launches a CUDA kernel with (currently) one thread per object. Only methods that are annotated with the `__device__` keyword can be executed in device mode.

Execute and Reduce Ikra-Cpp provides an API for combined execute and reduce operations. For example, this is useful for termination detection of iterative algorithms where the termination criteria depends on a property of multiple objects. One concrete example is the standard, parallel, frontier-based breadth-first search algorithm [7] terminates if no new vertex is explored in an iteration, i.e., the frontier for the next iteration is empty. This can be written as a conjunction of boolean "Am I part of the frontier?" vertex values, which can be reduced to determine if the algorithm should terminate. In the N-Body example from this paper, execute and reduce can be used to calculate the average distance of all bodies from a given point in space (Figure 3).

The reduction part is not supported in device mode yet and performed on the host. Future versions of Ikra will perform parallel reductions with shared memory [18].

3 Implementation

Ikra-Cpp is based on four ideas: (a) Allocate a large *storage buffer* (`char` array) in which all data is stored (generated by `IKRA_HOST_STORAGE`). (b) Assign unique integer IDs to objects. (c) Reference objects with "fake pointers" that encode an object ID. (d) Override C++ operators to decode IDs, calculate addresses and acccess data in the storage buffer.

In this section, we explain those steps in more detail, using a more verbose, but less convoluted notation. The example class in Figure 4 is identical to the the one in Figure 1b (without constructor), but with expanded preprocessor macros.

[2]SOA field types similar to `double__` in Figure 5 can be defined for other (non-primitive) types as well.

```
class Body : public SoaLayout<Body> {
  const static int kMaxInst = 50;
  const static int kObjSize = 4 * 8;    four doubles = 32 bytes

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

  double__<1, 0> pos_x = 0.0;          field index = 1, offset = 0
  double__<2, 4> pos_y = 0.0;          field index = 2, offset = 4
  double__<3, 8> vel_x = 0.0;          field index = 3, offset = 8
  double__<4, 12> vel_y = 0.0;         field index = 4, offset = 12

  static Body* get(int id);                    Calculate Address

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

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

Figure 4. Example: Macro-expanded `Body` class from a Figure 1b. All objects are stored inside `storage`. Fields must be declared with special data types like `double__`.

```
1  template<class Self>          Template Parameter "Self": CRTP [6]
2  class SoaLayout {
3    template<int Index, int Offset>
4    using int__ = Field_<int, Index, Offset, Self>;
5    template<int Index, int Offset>
6    using double__ = Field_<double, Index, Offset, Self>;
7  };
8  template<typename T, int Index, int Offset, class Owner>
9  class Field_ {
10   Field_() {}
11   Field_(const T value) { this->operator=(value); }
12   operator T&() const { return *data_ptr(); }
13   void operator=(T value) { *data_ptr() = value; }
14   T* operator->() const { return data_ptr(); }
15   T* data_ptr() const;                    Calculate Address
16 };
```

Figure 5. Basic Implementation of Ikra-Cpp. SOA classes are subclasses of `SoaLayout` and only `Field_` instantiations may be used as types for SOA fields.

3.1 Overview

SOA object pointers (i.e., the result of a `new` expression) do not necessarily point to allocated data but are used to encode object IDs ("fake pointers"; similar to tagged pointers where the tag is the entire pointer). All objects of a SOA class `C` have a unique ID[3] between [1; *maxInst*(`C`)]: E.g., calling `new C` for the first time returns a `C*` pointer that encodes ID 1.

SOA field types behave like normal C++ types in most cases, but access data at a location inside the storage buffer. In particular, they must support the following operations.

- *Reading Value:* A `double__` value can be converted to a `double` value without an explicit typecast (*implicit conversion operator*[4], Figure 5, Line 12).
- *Writing Value:* A `double` value can be assigned to a `double__` field (*assignment operator*, Line 13).
- *Method Call:* For non-primitive types (does not apply to `double`), a method call on a SOA field is forwarded to the object at the data location (*member of pointer "arrow" operator*[5], Line 14).

SOA field types are defined in `SoaLayout` as template instantiations of `Field_` (Figure 5, Lines 3–6). This class provides the necessary operator implementations and calculates the address inside the storage buffer at which the field value of a certain object can be found (Line 15). In the most basic case, given the address of a field object *this*, the address of field `C::f` can be computed as follows, where *id* is a function that decodes the object ID from a field object address.

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

The first two lines in the equation compute the beginning of the *SOA array* storing all values of `C::f`. The third line computes the offset into that array. How exactly object IDs are encoded in SOA object addresses is determined by the *addressing mode*. Ikra-Cpp supports three different addressing modes, one of which must be chosen at compile time: *Zero Addressing* and two variants of *Valid Addressing*. The former one is more space-efficient but relies on non-standard C++ constructs, so it might not work with some compilers[6].

3.2 Addressing Modes

This section gives an overview of various addressing modes. Zero addressing and storage-relative zero addressing are implemented in Ikra-Cpp. In accordance with the C++ zero overhead principle [34], zero addressing is the default mode.

3.2.1 Zero Addressing

In this addressing mode (Figures 6, 7), an object of class `C` with ID *i* is referenced with a `C*` pointer pointing to memory address *i* (e.g., $\texttt{obj}_{10}$ has address `0xa`). Field values are grouped by field and stored from the beginning of the storage buffer. No field values are stored for object 0 (null pointer). Given a `C*` pointer *obj*, the memory location of field value `C::f`, i.e., `&obj->f`, is calculated as follows. Compile-time constants are in blue.

[3] ID 0 is reserved for null references.
[4] The `auto` keyword is not supported. E.g., a field value cannot be assigned to a variable declared as `auto` without an explicit type cast.
[5] Note for experienced C++ programmers: This is similar to how `std::unique_ptr` is implemented.
[6] We verified that it works with g++ 5.4.0, clang 3.8.0 and CUDA 9.0.

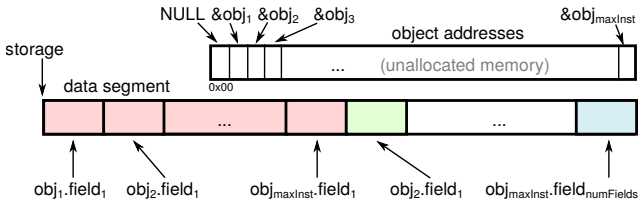

Figure 6. Storage Buffer Layout in Zero Addressing. Boxes in the upper part are 1 byte in size and used as SOA object addresses. Boxes in the data segment contain field values.

```
1  Body* Body::get(int id) {
2    return reinterpret_cast<Body*>(id)
3  }
4  template<typename T, int Index, int Offset, class Owner>
5  T* Field_<T, Index, Offset, Owner>::data_ptr() {
6    Owner* obj = reinterpret_cast<Owner*>(this);
7    return reinterpret_cast<T*>(Owner::storage
8        + Offset * Owner::kMaxInst - sizeof(T)
9        + sizeof(T) * reinterpret_cast<uintptr_t>(obj));
10 }
```

Figure 7. Address Computation in Zero Addressing Mode.

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

constant variable

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

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

One crucial assumption of zero addressing is that the size of the SOA class and the size of `Field_` instantiations are zero. Therefore, the address of a SOA object is equal to the addresses of all its fields[7], which we take advantage of in the definition of `data_ptr` (Figure 7, Line 6). Second, this allows us to use the `new` keyword for instance creation, even though it zero-initializes all memory before calling a constructor[8]. The size of a class or struct should be greater than zero (even if empty) according to the C++ standard [12], but many compilers can be instructed to use a size of zero. If this is not supported by a compiler, either valid addressing or a different mechanism for instance creation must be used.

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

[8] Zero-initializing a memory segment of zero bytes is a *no operation*, even if the segment starts at a bogus memory address ("fake pointer").

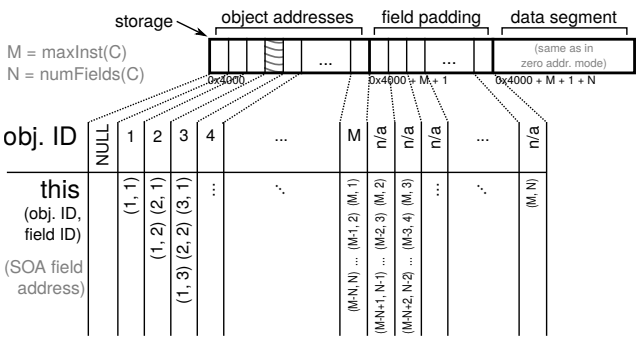

Figure 8. Address Computation in Storage-relative Zero Addressing. The table lists all SOA fields that share a certain address. E.g., `0x4003` is the address of $obj_3.field_1$, $obj_2.field_2$ and $obj_1.field_3$. Note that this is not where field values are actually stored. Such addresses are computed by `data_ptr` according to formula $addr_{valid}(this, \texttt{f})$.

3.2.2 Valid Addressing Mode

Since zero addressing does not conform to the C++ standard, Ikra-Cpp provides a different addressing mode. In valid addressing, the C++ size of every SOA field is one byte (e.g., `sizeof(double_) = 1`), consequently the C++ size of every SOA object is *numFields* bytes. In order to support the `new` keyword, the address of a SOA object must then point to valid (allocated) memory; thus the name *valid* addressing. The challenge of valid addressing is to add an as small as possible amount of padding (*wasted* memory) such that no data is overwritten by zero initialization. Programmers should use zero addressing if supported by their compiler, since it does not waste any memory. Even though we do not have measurements for valid addressing yet, we expect the same runtime performance as in zero addressing, because address computation is in both cases reduced to strided memory access after constant folding.

Storage-relative Zero Addressing In this addressing mode, an object of class `C` with ID i is referenced with a `C*` pointer pointing to ith byte of the storage buffer (Figure 8). E.g., if the storage buffer is allocated at address `0x4000`, then the address of obj_3 is `0x4003`. The *data segment*, where field values are stored, is identical to the one in zero addressing and starts at offset $padding = maxInst(\texttt{C}) + 1 + numFields(\texttt{C})$, i.e., *padding* many bytes are wasted in this addressing mode. In general, the memory location of a field `C::f` of object with address *obj* is then calculated as follows. Note that the formula is identical to the one in zero addressing, except for the offset of the data segment and the ID computation.

$$addr_{valid}(obj, \texttt{C::f}) = storage$$

data segment offset
$$+ maxInst(\texttt{C}) + 1 + numFields(\texttt{C})$$
$$+ maxInst(\texttt{C}) \cdot offset(\texttt{C::f})$$
$$- sizeof(\texttt{C::f})$$

ID computation
$$+ (obj - storage) \cdot sizeof(\texttt{C::f})$$

Since address computation is done inside SOA field classes (`Field_`, *not* `SoaLayout`), we have to express the above formula in terms of the address (*this* pointer) of a SOA field instead of the object address *obj*. The address of a SOA object *obj* inside of field `C::f` is defined as $obj = this - index(\texttt{C::f}) + 1$, where $index(\texttt{C::f})$ is the field index of `C::f`. E.g., the address of the third field `vel_x` of $\texttt{Body}_1$ is `0x4003` in Figure 8 (striped box). Consequently, $obj = \texttt{0x4003} - 3 + 1 = \texttt{0x4001}$. This object address can be used in the above formula. Putting both definitions together, the memory location of a field `C::f` with respect to its *this* pointer is then calculated as follows.

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

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

4 Preliminary Performance Evaluation

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

We benchmarked an iterative application of `Body::move` for all bodies (Figure 2a in a loop)[9]. The number of iterations was chosen such that every program ran for at least 5 sec. We calculated the average running time per iteration and report the minimum time out of 12 program runs.

Running Time Figures 9 and 10 show the running time on CPU and GPU. The upper subfigure shows the average running time of one entire iteration and the lower subfigure shows the average running time for a single Body instance.

In host mode, Ikra-Cpp's performance is almost identical to hand-written SOA code. AOS-32 is variant of AOS where 16 supplemental `double` fields were added to the Body class, similarly to the SoAx benchmark section [10]. We can think

[9]This benchmark is quite simple, but it clearly isolates the overheads of Ikra-Cpp, specifically address computation.

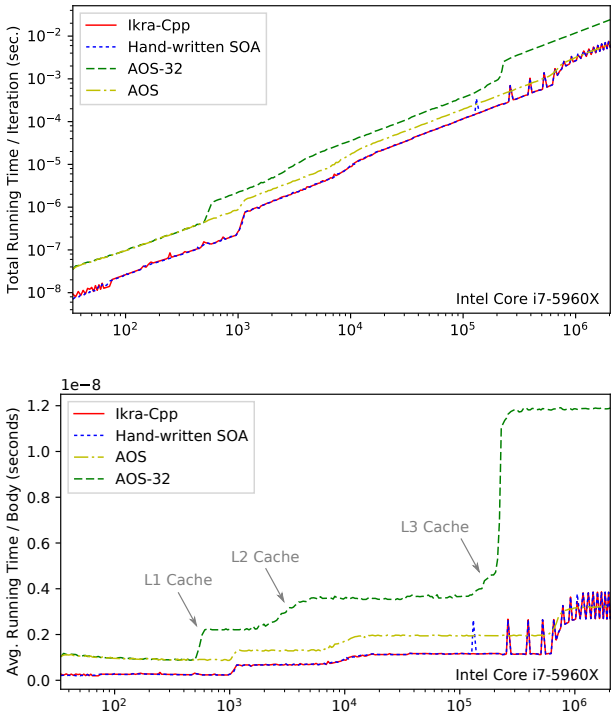

Figure 9. Host Mode Running Time in Seconds. The x axis measures the number of Body instances.

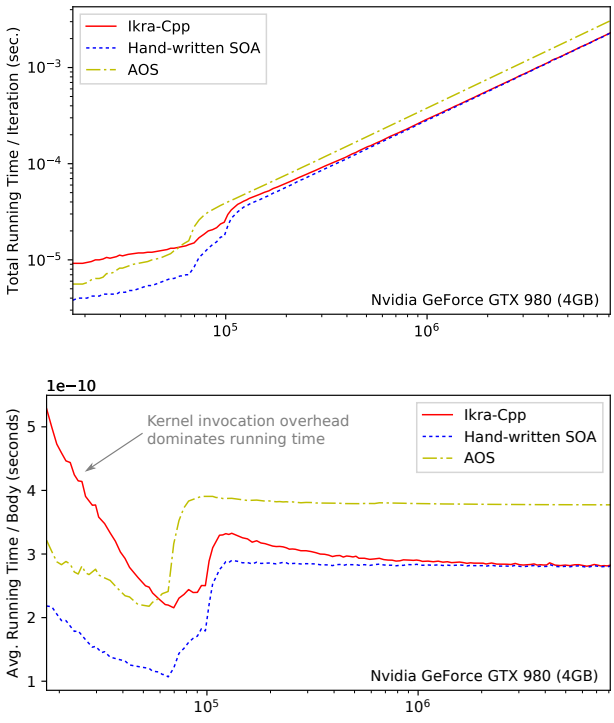

Figure 10. Device Mode (CUDA) Running Time in Seconds. The x axis measures the number of Body instances.

of such fields as additional properties of a body (e.g., mass or radius) that are not utilized in this particular computation. The AOS-32 graph of the lower subfigure clearly shows the effect of the L1, L2, L3 caches (32KB, 256KB, 20MB).

The performance difference in device mode is due to a higher kernel invocation overhead in Ikra-Cpp. More than 10000 iterations (kernel invocations) are performed for small problem sizes. With a larger number of bodies, we get closer to hand-written SOA code, because of fewer iterations.

Code Generation We verified that the generated binary code for reading/writing a single field from a SOA object is identical in Ikra-Cpp and hand-written SOA (gcc and clang). This shows that modern compilers can constant-fold the complex address computations in Ikra-Cpp.

Unfortunately, this is not always the case for other compiler optimizations. One example is automatic loop vectorization. In hand-written SOA code, gcc and clang vectorize the loop that calls `Body::move` for every `Body` instance. However, only gcc performs the equivalent loop vectorization with Ikra-Cpp. Clang is able to apply optimizations like loop unrolling but considers the operations involved in address computation[10] as potentially "dependent" memory operations and thus unsafe for vectorization.

There are three approaches to solve this problem. First, we can try rewriting the address computation part of Ikra-Cpp, in an attempt to give the compiler additional hints that trigger optimizations. This approach is fragile and could break at any time. Second, code can be vectorized manually, either with C++ SSE intrinsics or with a vectorization framework like *Sierra* [14, 15]. Considering that real applications, which exhibit code that is more complex than our example here, cannot be automatically vectorized (yet) with today's compilers, even if written in SOA style, this approach seems feasible to us. Third, Ikra-Cpp could be implemented as a compiler extension, which is the cleanest and most stable solution. We describe this approach in the context of the *Intel ispc Compiler* in more detail in Section 5.

5 Related Work

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

Homann and Laenen developed *SoAx* [10], a C++ library for AOS-style C/C++ programming with implicit SOA layout. Their implementation is based on preprocessor macros and template metaprogramming. SoAx does not support OOP concepts like classes or methods. SOA struct types are defined using `std::tuple` instantiations and a helper macro that defines every SOA array separately. Objects field values can only be accessed through a getter method of a SOA container object, which takes an object ID as argument, and not through SOA pointers. While such code is less expressive, it has two benefits: First, such code is easier to optimize for compilers than Ikra-Cpp code because it does not require decoding an object ID from a pointer. Second, it allows programmers to create multiple containers, each of which has its own ID range for objects.

Array of Structures eXtended (ASX) [35] is a library similar to SoAx. Objects in ASX can be allocated in an ASX containers and also on the stack (as single objects). ASX containers support both SOA and AOS data layout, one of which must be chosen as a template parameter.

The *Intel SPMD Program Compiler (ispc)* [3, 25] is an experimental C compiler with language features for better SIMD support. Among other features, it can layout an array of C structs in a hybrid SOA layout (also called *Array of Structures of Arrays* (AoSoA) [36, 38] or *Tiled AOS* [13]). If a struct type is annotated with the `soa<N>` keyword and used to declare an array (where N should be the SIMD width), then the array is layed out as hybrid SOA with a SOA length of N. Array elements can be accessed with the usual C syntax. Furthermore, it is possible to take the address of a SOA object and fields can be accessed using a SOA object pointer. From that perspective, ispc's functionality is very similar to Ikra-Cpp. We are not sure how that functionality is implemented internally and it would interesting to see how easily ispc can be extended to support OOP concepts like methods.

6 Future Work

There are two main tasks that must be solved in future versions of Ikra. First, Ikra-Cpp should support inheritance and virtual functions to take full advantage of object-oriented programming. To that end, SOA classes with virtual functions need a virtual method table (vtable) pointer SOA array. The vtable pointer can be seen as the first field. In the current implementation approach, this can only be done with first field addressing, because C++ compilers assume that an object pointer points to the beginning of an object, where the vtable pointer is stored.

Second, Ikra-Cpp must be tested with a wider variety of compilers, platforms and compiler options. At the moment, it seems like automatic loop vectorization is the only optimization affected by Ikra-Cpp, and we have yet to find a workaround in clang. While *performance DSLs* [1, 5, 27, 29, 37] are usually limited to a single domain, Ikra-Cpp addresses memory layout strategies and is applicable to a wide range of domains. This raises the question if Ikra-Cpp's functionality should better be implemented as part of a compiler like ispc[11], which could potentially solve such optimization issues.

[10] The problem is pointer casting. In the simplest case, an expression like `array[reinterpret_cast<uintptr_t>(id)]`, where `id` is a pointer encoding an integer array offset is already considered unsafe.

[11] ispc does not support OOP features like classes and methods, yet (01/2018).

7 Summary

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

References

[1] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for Physical Simulation on CPUs and GPUs. *ACM Trans. Graph.* 35, 2, Article 21 (May 2016), 12 pages.

[2] Paul Besl. 2015. *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.

[3] James Brodman, Dmitry Babokin, Ilia Filippov, and Peng Tu. 2014. Writing Scalable SIMD Programs with ISPC *(WPMVP '14)*. ACM, 25–32.

[4] E. Calore, A. Gabbana, J. Kraus, E. Pellegrini, S.F. Schifano, and R. Tripiccione. 2016. Massively Parallel Lattice-Boltzmann Codes on Large GPU Clusters. *Parallel Comput.* 58, C (Oct. 2016), 1–24.

[5] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. 2011. A Domain-specific Approach to Heterogeneous Parallelism *(PPoPP '11)*. ACM, 35–46.

[6] James O. Coplien. 1995. Curiously Recurring Template Patterns. *C++ Rep.* 7, 2 (Feb. 1995), 24–27.

[7] Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA *(HiPC'07)*. Springer-Verlag, 197–208.

[8] Dirk Helbing. 2012. Agent-Based Modeling. In *Social Self-Organization: Agent-Based Simulations and Experiments to Study Emergent Social Behavior*. Springer-Verlag, 25–70.

[9] Bruce Hendrickson and Jonathan W. Berry. 2008. Graph Analysis with High-Performance Computing. *Computing in Science and Engg.* 10, 2 (March 2008), 14–19.

[10] Holger Homann and Francois Laenen. 2017. SoAx: A generic C++ Structure of Arrays for handling Particles in HPC Codes. *ArXiv e-prints, to appear in Comm. Phys. Comm.* (Oct. 2017).

[11] Paul Hudak. 1998. Modular Domain Specific Languages and Tools *(ICSR '98)*. IEEE Computer Society, 134–142.

[12] ISO. 2012. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization. 1338 (est.) pages.

[13] Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. 2015. Automatic Data Layout Optimizations for GPUs *(Euro-Par 2015)*. Springer-Verlag, 263–274.

[14] Roland Leißa, Sebastian Hack, and Ingo Wald. 2012. Extending a C-like Language for Portable SIMD Programming *(PPoPP '12)*. ACM, 65–74.

[15] Roland Leißa, Immanuel Haffner, and Sebastian Hack. 2014. Sierra: A SIMD Extension for C++ *(WPMVP '14)*. ACM, 17–24.

[16] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (March 2008), 39–55.

[17] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing *(SIGMOD '10)*. ACM, 135–146.

[18] Harris Mark. 2008. Optimizing parallel reduction in CUDA. *Nvidia CUDA SDK* 2 (2008).

[19] Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. 2015. Columnar Objects: Improving the Performance of Analytical Applications *(Onward! 2015)*. ACM, 197–210.

[20] Gang Mei and Hong Tian. 2016. Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation. *SpringerPlus* 5, 1 (Feb. 2016).

[21] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-specific Languages. *ACM Comput. Surv.* 37, 4 (Dec. 2005), 316–344.

[22] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. *SIGPLAN Not.* 47, 8 (Feb. 2012), 117–128.

[23] Bertrand Meyer. 1997. *Object-oriented Software Construction (2nd Ed.)*. Prentice-Hall, Inc.

[24] Perhaad Mistry, Dana Schaa, Byunghyun Jang, David Kaeli, Albert Dvornik, and Dwight Meglan. 2011. Data Structures and Transformations for Physically Based Simulation on a GPU. In *High Performance Computing for Computational Science – VECPAR 2010: 9th Int. Conference, Revised Selected Papers*. Springer-Verlag, 162–171.

[25] Matt Pharr and William R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar)*. IEEE, 1–13.

[26] Viera K. Proulx. 1998. Traffic Simulation: A Case Study for Teaching Object Oriented Design *(SIGCSE '98)*. ACM, 48–52.

[27] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530.

[28] P. Richmond, S. Coakley, and D. M. Romano. 2009. A High Performance Agent Based Modelling Framework on Graphics Card Hardware with CUDA *(AAMAS '09)*. International Foundation for Autonomous Agents and Multiagent Systems, 1125–1126.

[29] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. Building-Blocks for Performance Oriented DSLs *(DSL '11)*. 93–117.

[30] Alban Rousset, Bénédicte Herrmann, Christophe Lang, and Laurent Philippe. 2016. A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Computer Science Review* 22, Supplement C (2016), 27–46.

[31] Jakob Siegel, Juergen Ributzka, and Xiaoming Li. 2009. CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator *(ICPPW '09)*. IEEE Computer Society, 174–181.

[32] Matthias Springer and Hidehiko Masuhara. 2016. Object Support in an Array-based GPGPU Extension for Ruby *(ARRAY 2016)*. ACM, 25–31.

[33] Benedikt Stefansson. 2000. Simulating Economic Agents in Swarm. In *Economic Simulations in Swarm: Agent-Based Modelling and Object Oriented Programming*. Springer US, 3–61.

[34] Bjarne Stroustrup. 2012. Foundations of C++ *(ESOP 2012)*. Springer-Verlag, 1–25.

[35] Robert Strzodka. 2012. Chapter 31 - Abstraction for AoS and SoA Layout in C++. In *GPU Computing Gems Jade Edition*, Wen-mei W. Hwu (Ed.). Morgan Kaufmann, 429 – 441.

[36] Robert Strzodka. 2012. Data Layout Optimization for Multi-valued Containers in OpenCL. *J. Parallel Distrib. Comput.* 72, 9 (Sept. 2012).

[37] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A

Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 134 (April 2014), 25 pages.

[38] Nicolas Weber and Michael Goesele. 2014. Auto-tuning Complex Array Layouts for GPUs *(PGV '14)*. Eurographics Association, 57–64.

[39] Jianlong Zhong and Bingsheng He. 2013. Parallel Graph Processing on Graphics Processors Made Easy. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1270–1273.

A First Field Addressing

This addressing mode is a variant of valid addressing. Its purpose is to reduce the amount of waste due to the padding area. It is also required for virtual function support in the future. An object of class `C` with ID i is referenced with a `C*` pointer pointing to the memory location of the value of the first field of object i (Figure 11). If the SOA class has at least one virtual function, the first field is the vtable pointer. If the number of fields of the SOA class is larger than the size of the first field, then the memory of the first field must be padded with $sizeof(\texttt{C::}first) - numFields(\texttt{C})$ bytes to avoid overwriting first field values of following objects due to zero initialization[12]. Given a `C*` pointer obj, the memory location of a field `C::f` is calculated as follows.

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

$sizeof^*$ and $offset^*$ take into account padding that might be added to the first field. The memory location of a field `C::f` with respect to its *this* pointer is calculated as follows.

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

$$\text{where } R = \frac{sizeof^*(\texttt{C::f})}{sizeof^*(\texttt{C::}first)}$$

[12]Future versions of Ikra-Cpp will allow object deallocation, in which case a newly allocated object might reuse *old* data.

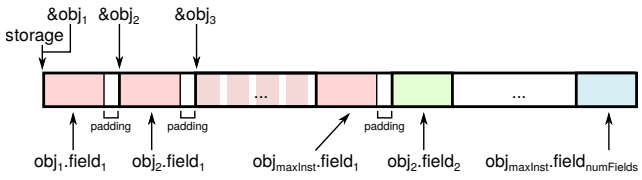

Figure 11. Storage Buffer Layout in First Field Addressing. The address of a SOA object is the location of its first field value, which may be padded.

Even though the definition of $addr_{first}$ contains a fraction, its value is always an integer. However, its calculation is not straightforward. On the one hand, address calculation can be optimized by isolating the variable parts. In the formula above, the only variable part *this* is multiplied by a compile-time constant and added to a compile-time constant (strided memory access). On the other hand, R might not be an integer and neither might be the constant-folded parts of the strided memory access. Floating point operations as part of the address computation should be avoided by all means. Ikra-Cpp supports first field addressing only for SOA classes where R is an integer, i.e., the size of every field is a multiple of the size of the first field. Furthermore, this addressing mode is superior to storage-relative zero addressing only if the field padding size is zero or one byte. Note that field padding is incurred for every instance, i.e., *maxInst* many times. First field addressing is optimal for the minimal N-Body example, where all fields have the same size and the number of fields equals the size of the first field.

B Additional Evaluation

We repeated the experiments from Section 4 on another machine with an Intel Core i7-6820HQ CPU (4x 2.70 GHz), 32 GB RAM and a GeForce 940MX GPU.