A C++/CUDA DSL for Object-oriented Programming with Structure-of-Arrays Data Layout

CGO'18 Student Research Competition – Extended Abstract

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

Hidehiko Masuhara*
Tokyo Instiute of Technology
masuhara@acm.org

Abstract

Object orientation 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-OOP code. We propose Ikra-Cpp, a CUDA/C++ DSL for object-oriented high-performance computing that lets programmers write object-oriented code with standard C++ notation, while storing data in the well-studied Structure of Arrays (SOA) layout. This gives programmers the performance benefit of SOA and the expressiveness of object-oriented programming at the same time.

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

Extended Abstract

Previous work has demonstrated that Structure of Arrays (SOA) is an effective technique for achieving high performance on SIMD architectures such as GPUs or CPUs with vector instructions [2, 4, 5, 7, 9, 10, 12]. Given an array of C struct or class instances, SOA groups all values of a field together (Figure 1). This is contrary to traditional Array of Structures (AOS), where the fields of every object are grouped together, and allows for more efficient memory access (parallel vector register loads, *memory coalescing* on GPUs) and cache usage.

The goal of our research is to provide a mechanism that lets programmers write object-oriented code in a readable and concise AOS style (Figure 2a), while automatically layouting data as SOA in the background. Existing C++ libraries and language extensions like *SoAx* [3, 11] and the *Intel SPMD Program Compiler (ispc)* [1, 8] provide that functionality for C structs without methods. Our research extends that work to object-oriented programming. The contributions of our research prototype Ikra-Cpp[1] are support for classes, methods and constructors with minimal changes to standard C++ notation (Figure 2b) in both C++ and CUDA. Our approach is closer to SoAx than to ispc: We are developing a lightweight C++ library/DSL instead of a new C++ compiler or invasive compiler extension, due to the high engineering effort of compiler construction [6].

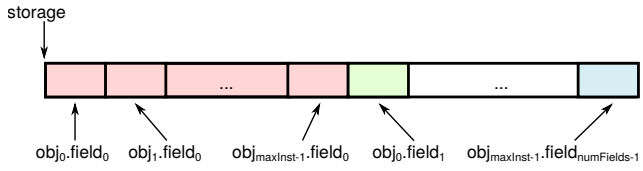

Figure 1: Storage buffer (SOA). All values of a field are stored consecutively.

High-level Idea Ikra-Cpp stores all data of a SOA class in a large statically allocated *storage buffer* (Figure 1). Upon field access, Ikra-Cpp calculates and accesses an address inside the storage buffer. Our main insight is that this can be achieved efficiently and seamlessly (with AOS-style notation) in C++ with template metaprogramming, operator overloading and preprocessor macros. Given an object ID *id*, the address $\&obj_{id}.field_i$ can be computed as follows, where $offset(field_m) = \sum_{k=0}^{m-1} sizeof(field_k)$.

$$\begin{aligned} addr(id, i) = {} & storage \\ & + maxInstances \cdot offset(field_i) \\ & + id \cdot sizeof(field_i) \end{aligned}$$

All terms except for *id* are compile-time constants. After constant folding, this computation is compiled to an efficient `mov` instruction with strided memory access when used in field reads/writes. The binary code is identical to hand-written SOA code[2].

Implementation Outline SOA objects are always referenced with pointers and cannot be stack allocated. Pointers to SOA objects do not point to actual data but are used to encode object IDs ("fake pointers"). For example, the address `0x1` is used to reference the first object. Fields must be declared with special types (e.g., `double_`). Their implicit conversion and assignment operators compute the object ID from their `this` pointer and use the formula above to access the storage buffer at the correct location.

*Academic Advisor
[1] https://github.com/prg-titech/ikra-cpp/

[2] Confirmed with gcc 5.4.0 and clang 3.8.0.

```cpp
class Body {
 public:
  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;
  }
};

void create_and_move() {
  Body* b = new Body(1.0, 2.0);
  b->move(0.5);
  assert(b->pos_x == 1.5);
}
```

(a) C++ Class (AOS Layout)

```cpp
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

```cpp
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 2: 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). Ikra-Cpp also provides constructs for parallel method invocation (not discussed here).

Developing Ikra-Cpp is challenging from a technical point of view: Due to the complex combination of operator overloading and address computation, it is harder for C++ compilers to apply optimizations. While Ikra-Cpp is now on par with hand-written SOA code in gcc (Figure 3), clang still has trouble performing automatic loop vectorization.

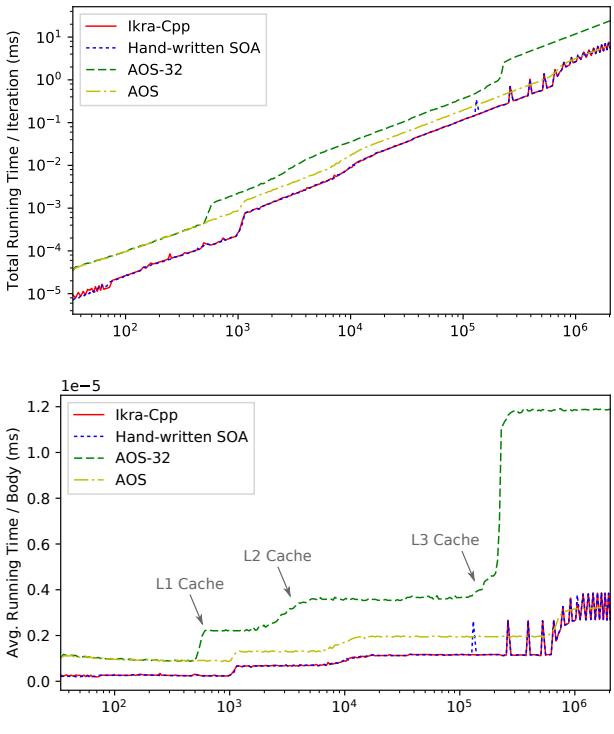

Figure 3: Running time for updating the position of every body (`Body::move`). The x and y axes measure the number of bodies and average running time values for one iteration. Every program ran iteratively for at least 5s on a machine with an Intel i7-5960X CPU, Ubuntu 16.04.1 and gcc 5.4.0 (-O3). Reported values are minimums of 12 program runs. To allow for easier comparison with SoAx [3], we also show AOS-32, where `Body` has 8 additional `double` fields. The lower diagram clearly shows the effect of the L1 cache (32KB), the L2 cache (256KB) and the L3 cache (20MB).

References

[1] J. Brodman, D. Babokin, I. Filippov, and P. Tu. 2014. Writing Scalable SIMD Programs with ISPC. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing (WPMVP '14)*. ACM, New York, NY, USA, 25–32.

[2] 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.

[3] H. Homann and F. Laenen. 2017. SoAx: A generic C++ Structure of Arrays for handling Particles in HPC Codes. *ArXiv e-prints, submitted to Computer Physics Communications* (Oct. 2017). arXiv:physics.comp-ph/1710.03462

[4] T. Mattis, J. Henning, R. Rein, R. Hirschfeld, and M. Appeltauer. 2015. 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! 2015)*. ACM, 197–210.

[5] G. Mei and H. Tian. 2016. Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation. *SpringerPlus* 5, 1 (Feb. 2016), 104.

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

[7] P. Mistry, D. Schaa, B. Jang, D. Kaeli, A. Dvornik, and D. Meglan. 2011. *Data Structures and Transformations for Physically Based Simulation on a GPU.* Springer Berlin Heidelberg, 162–171.

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

[9] J. Ributzka, X. Li, and J. Siegel. 2009. CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator. In *2009 International Conference on Parallel Processing Workshops (ICPPW '09)*. IEEE Computer Society, 174–181.

[10] 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.

[11] R. 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.

[12] J. Zhong and B. He. 2013. Parallel Graph Processing on Graphics Processors Made Easy. *Proceedings of the VLDB Endowment* 6, 12 (Aug. 2013), 1270–1273.