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

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

Hidehiko Masuhara∗
Tokyo Institute 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 laying out 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-Cpp1 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].

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 can be computed as follows, where offset(field_m) = \sum_{k=0}^{m-1} sizeof(field_k).

\[
\text{addr}(id, i) = \text{storage} + \text{maxInstances} \cdot \text{offset(field)} + id \cdot \text{sizeof(field)}
\]

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 handwritten SOA code2.

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.

1Academic Advisor
2Confirmed with gcc 5.4.0 and clang 3.8.0.
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.

References


