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

Matthias Springer (Tokyo Institute of Technology)

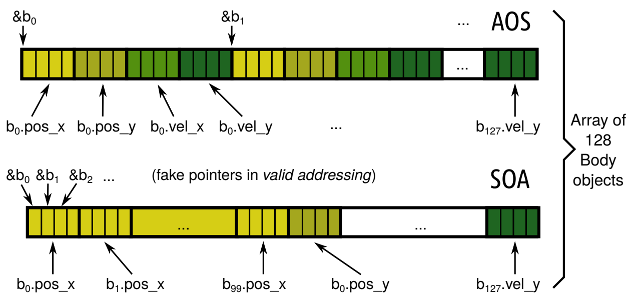
東京工業大学 Tokyo Institute of Technology

https://github.com/prg-titech/ikra-cpp

Context: HPC with uniformly structured data (e.g., n-body simulation, traffic flow simulation)

Goal: SOA memory layout (good for caching, vectorization, parallelization) with C++ Notation

[Pointer insteads of IDs, Method Calls, **new** Keyword, Templates, Member of Object/Pointer Operator, Future work: virtual functions]



Object Creation:

```
Body *p = new Body(1.0, 2.0);
Body *q = Body::make(10, 1.0, 2.0);
```

Field Access:

```
p->vel_x = p->vel_y = 1.5;
```

Member Functions:

```
p->move(0.5);
forall(&Body::move, q, 10, 0.5);
```

```
class Body : public SOA<Body> {
 public: INITIALIZE_CLASS
  float_ pos_x; float_ pos_y;
  float_ vel_x; float_ vel_y;

  Body(float x, float y)
    : pos_x(x), pos_y(y) {}

  void move(float dt) {
    pos_x = pos_x + vel_x * dt;
    pos_y = pos_y + vel_y * dt;
  }
};
HOST_STORAGE(Body, 128);
```

GPU mode: Use DEVICE_STORAGE. Max. #objects

```
char buffer[128 * 16];
```

Large enough to store 128 objects (four float[128] arrays)

Related Work

Robert Strzodka. **Abstraction for AoS and SoA Layout in C++.** GPU Computing Gems Jade Edition, pp. 429-441, 2012. [First DSL approach in C++. Supports easy change between AOS and SOA layout. Complicated notation. Potentially large mem. footprint if fields have different size.]

Holger Homann, Francois Laenen. **SoAx: A generic C++ Structure of Arrays for handling particles in HPC code.** Comp. Phys. Comm., Vol. 224, pp. 325-332, 2018. [Simpler notation than [Strzodka12] for single struct. Still not like standard C++. *Expression Templates* to avoid memory allocation for temporary results in large arith. expressions.]

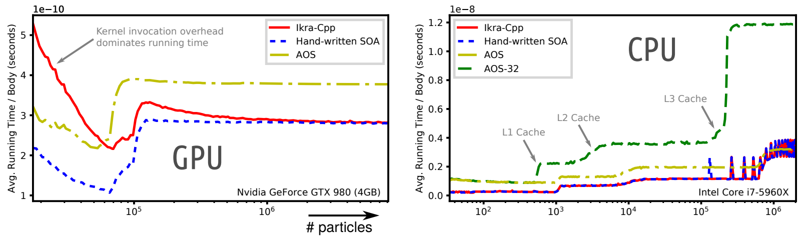
Results & Main Insights

- ★ Field access (decoding object IDs + calculating memory addresses) is as efficient as array access in hand-written SOA code (*strided mem. access*).
  Zero Addressing: $data\_ptr_{vel\_x}(ptr) = 0x600400 + 4 * ptr$
  Valid Addressing: $data\_ptr_{vel\_x}(ptr) = -0x11FFC00 + 4 * ptr$

```
float test(Body* ptr) {
  return ptr->vel_x;
}
```

- ★ Main Limitation: How well can the compiler optimize this code? Experiments performed with Zero Addressing.
  gcc 5.4.0: compiler hints necessary (constexpr)
  clang 3.8, 5.0: works for simple examples, loop vectorization fails (because of single buffer array)
  Future work: Reimplement with ROSE Compiler (a powerful C++ preprocessor)

Microbenchmark: Iterative Application of Body::move(0.5), Zero Addressing Mode, gcc 5.4.0 (-O3). Identical assembly.



So, how does it work?

"Fake" Pointers encode Object IDs

There are various encoding techniques. Need to specify both an encoder (object construction) and a decoder (field access).

More OOP Features, but harder to optimize

a) **Zero Addressing:** $\&obj_{id} = id$

```
void* Body::operator new() {
  return (void*) size++;
}

int field<T, idx, offset>::id() {
  Body* ptr = ((char*) this)
    - idx*sizeof(field<...>));
  return (int) ptr;
}
```

encoder

decoder

new keyword

virtual functions

b) **Valid Addressing:** $\&obj_{id} = buffer + id$

c) **First Field Addressing:** $\&obj_{id} = buffer + sizeof(T) * id$

SOA field types

index offset

```
float_ vel_x;
field<float, 2, 8> vel_x;
```

Make field<float> behave like a float

Implement implicit conversion and assignment operator.

```
field<T, idx, offset>::operator T&() {
  return *data_ptr();
}
```

Calculate memory location of field value

```
T* field<T, idx, offset>::data_ptr() {
  T* arr = (T*) (buffer + 128*offset);
  return arr + id();
}
```

(padding area needed for *valid* + *first field addressing*)

Mem. Layout Example

(Zero Addr., (**float**) b_5.vel_x)