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

Matthias Springer, Hidehiko Masuhara
Tokyo Institute of Technology

WPMVP 2018
Outline

1. Introduction
2. Ikra-Cpp API and Example
3. Implementation Outline
4. Addressing Modes
5. Performance Evaluation
6. Related Work + Future Work
7. Summary
Introduction

• **AOS: Array of Structures**
  ```
  struct { float x, y, z; } arr[100];
  ```

• **SOA: Structure of Arrays**
  ```
  struct {
    float x[100], y[100], z[100];
  } s;
  ```
  – Good for caching, vectorization, parallelization

• **Hybrid SOA (SoAoS)**

• **Ikra-Cpp:** Embedded C++/CUDA DSL for SOA
  – Notation close to standard C++
  – **Support OOP features:**
    Member functions, pointers, constructors, virtual functions

Figures: ispc: A SPMD Compiler for High-Performance CPU Programming
class Body : public SoaLayout<Body> {
    
    public: IKRA_INITIALIZE_CLASS
    float_ pos_x = 0.0;
    float_ pos_y = 0.0;
    float_ vel_x = 1.0;
    float_ vel_y = 1.0;

    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;
    }
};

IKRA_HOST_STORAGE(Body, 128);
class Body : public SoaLayout<Body> {
    public: 
        IKRA_INITIALIZE_CLASS
        float_ pos_x = 0.0;
        float_ pos_y = 0.0;
        float_ vel_x = 1.0;
        float_ vel_y = 1.0;

        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;
        }
    }
};

IKRA_HOST_STORAGE(Body, 128);
class Body : public SoaLayout<Body> { 
  public: IKRA_INITIALIZE_CLASS
    float_ pos_x = 0.0;
    float_ pos_y = 0.0;
    float_ vel_x = 1.0;
    float_ vel_y = 1.0;

    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;
  }
};

IKRA_HOST_STORAGE(Body, 128);

Use this class like any other C++ class:

void create_and_move() {
  Body* b = new Body(1.0, 2.0);
  b->move(0.5);
  assert(b->pos_x == 1.5);
}
Ikra-Cpp “Executor” API

- A few extra functions to have a uniform API for CPU and GPU computation
- Construct multiple objects:
  ```cpp
  Body* b = Body::make(10, /*x=*/1.0, /*y=*/2.0)
  ```
- for-all execution:
  ```cpp
  ikra::execute(&Body::move, b, 10, /*dt=*/0.5);
  ```
- GPU versions: `cuda_make`, `cuda_execute`
- Automatic memory transfer between CPU/GPU
Implementation Outline

- Statically allocated storage buffer
- “Fake pointers” encode object IDs
- Special SOA field types (e.g., float_)
  - Overloaded Operators: Decode object ID, calculate location inside the storage buffer
- Implementation: preprocessor macros, template metaprogramming, operator overloading (achieving close to standard C++ notation without special tools/compilers)
Field Types

float pos_x = 0.0;
float pos_y = 0.0;
float vel_x = 1.0;
float vel_y = 1.0;

float<0, 0> pos_x = 0.0;
float<1, 4> pos_y = 0.0;
float<2, 8> vel_x = 1.0;
float<3, 12> vel_y = 1.0;

- Implicit conversion operator: float x = body->pos_x;
- Assignment operator: body->pos_x = 10.5;
- Member of pointer operator: vertex->neighbor->visit();
Addressing Modes

- Multiple techniques for encoding IDs

  Zero Addressing.: \&obj_{id} = id

  Storage-relative Zero Addr.: \&obj_{id} = buffer + id

  First Field Addr.: \&obj_{id} = buffer + \text{sizeof}(T) * id
Zero Addressing Mode

Zero Addressing: \&obj_{id} = id

(float) b1->pos_y

```
0x600000
0x600200
0x600400
0x600600
```

float Body_pos_x[128]
float Body_pos_y[128]
float Body_vel_x[128]
float Body_vel_y[128]

```
B0.pos_y
b1.pos_y
b2.pos_y
b3.pos_y
b127.pos_y
```

Body_pos_y[1]
Zero Addressing: \&obj_{id} = id

(float) b1->pos_y

IKRA_HOST_STORAGE(Body, 128);

char buffer[128*(4 + 4 + 4 + 4)]

IKRA_HOST_STORAGE(Body, 128);

float* Body_pos_y = (float*) (buffer + offset*num_obj);
Zero Addressing Mode

Zero Addressing: \&obj_id = id

(float) b1->pos_y
Zero Addressing Mode

Zero Addressing: $\&obj_{id} = id$

(float) $b1->pos_y$

(char) $buffer[128*(4 + 4 + 4 + 4)]$

$sizeof(float) == 1$

$0x600000$

$0x0$

$0x4$

$0x7C$

$0x600000$
Zero Addressing Mode

Zero Addressing: \&obj_{id} = id

(float) b1->pos_y
Zero Addressing Mode

Zero Addressing: \&obj_{id} = id

(float) ptr->pos_y
Zero Addressing Mode

Zero Addressing: \&obj_{id} = id

(float) ptr->pos_{y}

Can be constant-folded to strided memory access

float_{index, offset}::operator float&() {
    float* arr = (float*) buffer + offset*128;
    return arr[(int) this - index];
}
Addressing Modes

• Multiple techniques for encoding IDs

Zero Addressing.: \&obj_{id} = id

Storage-relative Zero Addr.: \&obj_{id} = buffer + id

First Field Addr.: \&obj_{id} = buffer + sizeof(T) * id
Performance Evaluation

- Reading/writing single field in zero addressing
  - Same assembly code as hand-written SOA code
  - Verified with gcc 5.4.0 and clang 3.8.0 / 5.0 in -O3
Performance Evaluation

- Reading/writing single field in zero addressing
  - Same assembly code as hand-written SOA code
  - Verified with gcc 5.4.0

- Consecutive field access in a loop
  - gcc: same performance as hand-written SOA
    (constexpr necessary in implementation)
  - clang: loop vectorization fails
  - nvcc: same performance as hand-written SOA,
    minus kernel invocation overhead (unoptimized)
Performance Evaluation

![Graphs showing performance evaluation results with gcc and nvcc compiler.](image)

- The graphs compare the running time of different methods:
  - Ikra-Cpp
  - Hand-written SOA
  - AOS
  - AOS-32

- The x-axis represents the input size, ranging from $10^2$ to $10^6$.

- The y-axis represents the average running time per body in seconds.

- Key observations:
  - Kernel invocation overhead dominates running time for nvcc.
  -直观地展示了不同方法的运行时间对比。
Related Work


```cpp
template <ASX::ID t_id = ASX::ID_value>
struct Body {
    typedef ASX::ASAGroup<float, t_id> ASX_ASA;
    union {
        float pos_x; ASX_ASA dummy1;
    };
    union {
        float pos_y; ASX_ASA dummy2;
    };
    union {
        float vel_x; ASX_ASA dummy3;
    };
    union {
        float vel_y; ASX_ASA dummy4;
    };
};

typedef ASX::Array<Body, 100, ASX::SOA> Container;
Container container;

void move(Container::reference body, float dt) {
    body.pos_x = body.pos_x + body.vel_x * dt;
    body.pos_y = body.pos_y + body.vel_y * dt;
}
```

Must all have same size (or be combined in groups of same size)
Related Work


```cpp
SOAX_ATTRIBUTE(pos, "position");
SOAX_ATTRIBUTE(vel, "velocity");

typedef std::tuple<pos<float>, 2>, vel<float>, 2>> Body;
Soax<Body> container(100);

void move(int id, float dt) {
    container.pos(id, 0) += container.vel(id, 0) * dt;
    container.pos(id, 1) += container.vel(id, 1) * dt;
}

void move_all(float dt) {
    container.posArr(0) += container.velArr(0) * dt;
    container.posArr(1) += container.velArr(1) * dt;
}
```

Layout specified with std::tuple
Related Work


```cpp
struct Body {
    float pos_x, pos_y, vel_x, vel_y;
};

soa<100> struct Body bodies[100];

void move(uniform soa<100> Body* varying body, uniform float dt) {
    body->pos_x += body->vel_x * dt;
    body->pos_y += body->vel_y * dt;
}
```
Related Work

• No support for object-oriented programming
  – Pointers instead of IDs
  – Member Functions
  – Constructors, Dynamic Allocation (**new** keyword)

• Support for multiple containers

• ASX and SoAx: non-standard C++ notation
Future Work

• Utilize ROSE Compiler
  – Powerful C++ preprocessor (source-to-source)
  – “Optimization of low-level abstractions can be (and frequently is) not handled well by the compiler” (ROSE Manual)
  – Use mixture of techniques shown today and ROSE code transformation rules

• Support more OOP features: subclassing, virtual function calls
Summary

- **Ikra-Cpp**: C++/CUDA DSL for OOP with SOA Data Layout
- Implementation in C++, no external tools required
- Notation close to standard C++
- Address computation is as easy as in normal array access, but can be difficult for compilers to optimize
- **Future work**: Reimplementation with ROSE, more OOP features
Appendix
Field Types

float pos_x = 0.0;
float pos_y = 0.0;
float vel_x = 1.0;
float vel_y = 1.0;

Field<float, 0, 0> pos_x = 0.0;
Field<float, 1, 4> pos_y = 0.0;
Field<float, 2, 8> vel_x = 1.0;
Field<float, 3, 12> vel_y = 1.0;
Storage Buffer

```cpp
float_ pos_x = 0.0;
float_ pos_y = 0.0;
float_ vel_x = 1.0;
float_ vel_y = 1.0;
```

```cpp
Field<float, 0, 0> pos_x = 0.0;
Field<float, 1, 4> pos_y = 0.0;
Field<float, 2, 8> vel_x = 1.0;
Field<float, 3, 12> vel_y = 1.0;
```

Inside Field<float>:
- Calculate address into storage buffer.

```
IKRA_HOST_STORAGE(Body, 100);
```

```
char buffer[100 * Body::ObjectSize];
```
C++ Operators

```cpp
float_ pos_x = 0.0;
float_ pos_y = 0.0;
float_ vel_x = 1.0;
float_ vel_y = 1.0;
```

```
Field<float, 0, 0> pos_x = 0.0;
Field<float, 1, 4> pos_y = 0.0;
Field<float, 2, 8> vel_x = 1.0;
Field<float, 3, 12> vel_y = 1.0;
```

```cpp
pos_x = 1.0; /* type error: cannot assign float to Field<float>*/
```
C++ Operators

```cpp
float_ pos_x = 0.0;
float_ pos_y = 0.0;
float_ vel_x = 1.0;
float_ vel_y = 1.0;

void Field<T, Index, Offset>::operator=(float value) {
    *data_ptr() = value;
}
```

SOA field array

How to calculate ID? Later...

```cpp
T* Field<T, Index, Offset>::data_ptr() {
    T* arr = (T*) buffer + 100*Offset;
    return arr + id();
}
```
C++ Operators

```cpp
float_ pos_x = 0.0;
float_ pos_y = 0.0;
float_ vel_x = 1.0;
float_ vel_y = 1.0;

Field<float, 0, 0> pos_x = 0.0;
Field<float, 1, 4> pos_y = 0.0;
Field<float, 2, 8> vel_x = 1.0;
Field<float, 3, 12> vel_y = 1.0;

float x = pos_x;   // type error: cannot convert Field<float> to float*/
```
C++ Operators

float pos_x = 0.0;
float pos_y = 0.0;
float vel_x = 1.0;
float vel_y = 1.0;

T* Field<T, Index, Offset>::data_ptr() {
    T* arr = (T*) buffer + 100*Offset;
    return arr + id();
}

Field<T, Index, Offset>::operator T&() {
    return *data_ptr();
}
Encoding IDs in Pointers

- Object pointers do not point to meaningful addresses.

```cpp
Body* b = new Body(/*x=*/1.0, /*y=*/2.0);
```

```cpp
t* Field<T, Index, Offset>::data_ptr() {
    t* arr = (t*) buffer + 100*Offset;
    return arr + id();
}
```

What is the value of `b`? (Encoding ID in pointer)

How to calculate ID? (Decoding ID from this)
Zero Addressing Mode

- Multiple techniques for encoding IDs

**Zero Addressing:** \&obj\_id = id

```cpp
int Field<T, Index, Offset>::id() {
    Body* ptr = (Body*) (((char*) this – Index)*sizeof(Field<...>));
    return (int) ptr;
}
```

```cpp
void* Body::operator new() {
    return (void*) size++;
}
```
Zero Addressing Mode

- Multiple techniques for encoding IDs

**Zero Addressing.:** \&obj\textsubscript{id} = id

```cpp
int Field<T, Index, Offset>::id() {
    Body* ptr = (Body*) ((char*) this – Index);
    return (int) ptr;
}

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

![Diagram showing memory allocation and object addresses]

WPMVP’18
Example: Address Computation

Zero Addressing.: \&obj_{id} = id

class TestClass : public SoaLayout<TestClass> {
    public: IKRA_INITIALIZE
        int_ field0;
        int_ field1;
};

IKRA_HOST_STORAGE(Body, 100);

int get_field1(TestClass* o) {
    return o->field1;
}
Example: Address Computation

Zero Addressing.: \( \&_{\text{id}} \text{id} \)

class TestClass : public SoaLayout<TestClass> {
    public: IKRA_INITitalize
        int_ field0;
        int_ field1;
};

IKRA_HOST_STORAGE(Body, 100);

int get_field1(TestClass* o) {
    Field<int, 1, 4>& f = o->field1;
    return (int) f;
}
Example: Address Computation

Zero Addressing.: \&obj_{id} = id

class TestClass : public SoaLayout<TestClass> {
    public: IKRA_INITIALIZER
        int_ field0;
        int_ field1;
};

IKRA_HOST_STORAGE(Body, 100);

int get_field1(TestClass* o) {
    Field<int, 1, 4>& f = o->field1;
    int* data_ptr = f.data_ptr();
    return *data_ptr;
}
Example: Address Computation

Zero Addressing.: &obj_{id} = id

class TestClass : public SoaLayout<TestClass> {
    public: IKRA_INITIALIZE
        int_ field0;
        int_ field1;
};

IKRA_HOST_STORAGE(Body, 100);

int get_field1(TestClass* o) {
    Field<int, 1, 4>& f = o->field1;
    int* arr = (int*) (buffer + 100*4);
    int* data_ptr = arr + f.id();
    return *data_ptr;
}
Example: Address Computation

Zero Addressing: \&obj_{id} = id

class TestClass : public SoaLayout<TestClass> {
    public: IKRA_INITIATE
        int field0;
        int field1;
};

IKRA_HOST_STORAGE(Body, 100);

int get_field1(TestClass* o) {
    Field<int, 1, 4>& f = o->field1;
    int* arr = (int*) (buffer + 100*4);
    int* data_ptr = arr + (int)&f - 1;
    return *data_ptr;
}
Example: Address Computation

Zero Addressing: \&obj_{id} = id

class TestClass : public SoaLayout<TestClass> {
  public: IKRA_INITIALIZE
    int_ field0;
    int_ field1;
};

IKRA_HOST_STORAGE(Body, 100);

int get_field1(TestClass* o) {
  int* arr = (int*) (buffer + 100*4);
  int* data_ptr = arr + (int) o + 1 - 1;
  return *data_ptr;
}
Example: Address Computation

Zero Addressing: \& obj_{id} = id

class TestClass : public SoaLayout<TestClass> {
    public: IKRA_INITIALIZE
        int_ field0;
        int_ field1;
};

IKRA_HOST_STORAGE(Body, 100);

int get_field1(TestClass* o) {
    int* data_ptr = (int*) (buffer + 400 + ((int) o)*4);
    return *data_ptr;
}

Field access can be as efficient as in hand-written SOA layout!
If your compiler can optimize this...

Strided memory access: const + 4*var

gcc and clang: yes
Addressing Modes

- Multiple techniques for encoding IDs

**Zero Addressing.** \&obj\_id = id

```cpp
new Body(1.0, 2.0);
void* Body::operator new() {
    return (void*) size++;
}
```

Problem with Zero-Initialization

a) Workaround: Set size of C++ object to 0
   ```cpp
   char _[0];
   ```
   b) Use a different encoding scheme where pointers point to allocated memory
Addressing Modes

• Multiple techniques for encoding IDs

**Storage-relative Zero Addr.**: \&obj\textsubscript{id} = buffer + id

\begin{verbatim}
int Field<T, Index, Offset>::id() {
    Body* ptr = (Body*) ((char*) this – Index);
    return (int) ptr – (int) buffer;
}
\end{verbatim}

\begin{verbatim}
void* Body::operator new() {
    return (char*) buffer + size++;
}
\end{verbatim}

Again, field access can be as efficient as in handwritten SOA layout!
Addressing Modes

- Multiple techniques for encoding IDs

**Storage-relative Zero Addr.**: \&obj\_id = buffer + padding + id

```cpp
int Field<T, Index, Offset>::id() {
    Body* ptr = (Body*) ((char*) this – Index);
    return (int) ptr – (int) buffer – padding;
}
```

```cpp
void* Body::operator new() {
    return (char*) buffer + padding + size++;
}
```

Again, field access can be as efficient as in handwritten SOA layout!