Inner Array Inlining for Structure of Arrays Layout

Matthias Springer, Yaozhu Sun, Hidehiko Masuhara
Tokyo Institute of Technology

ARRAY 2018
06/19/2018
Data Layout: AOS / SOA

- **AOS: Array of Structures**
  - All field values of a struct/object stored together
  - Standard layout in most programming languages/compilers
  - *Benefits*: easy to understand, simple memory management

- **SOA: Structure of Arrays**
  - All values of a field stored together
  - Best practice in SIMD programming
  - *Benefits*: Cache + memory bandwidth utilization, vectorization
  - * Downsides*: Tedious to implement, lacks OOP features
Ikra-Cpp: A C++/CUDA DSL for SOA

• An embedded data layout DSL in C++/CUDA
• Focus on object-oriented programming and GPU programming
  – Standard C++ notation for OOP features: Member functions, field access, (future work: virtual member functions, inheritance)
  – Abstractions for launching CUDA kernels: Execute member function for all objects
  – This talk: Focus on GPUs, but also works on CPUs (vectorizing compiler)
• Implemented with advanced C++ features: template meta-programming, operator overloading, macros, type punning
**Ikra-Cpp: Example (n-body Simulation)**

```cpp
class Body : public SoaLayout<Body, 50> {
    public: IKRA_INITIALIZE_CLASS
    double_ pos_x = 0.0;  double_ pos_y = 0.0;
    double_ vel_x = 0.0;  double_ vel_y = 0.0;

    __device__ Body(double x, double y)
        : pos_x(x), pos_y(y) {}

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

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

Implementation Paper: M. Springer, H. Masuhara:
Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming
with Structure-of-Arrays Layout, WPMVP ’18

**06/19/2018**

Inner Array Inlining for SOA Layout
What about Array-typed Fields?

- How to handle array-typed fields in a SOA layout?
- What kind of layout is best for performance?
- Outline of this work
  - Overview of array data layout strategies for AOS and SOA
  - Performance study: synthetic benchmark, BFS, traffic flow simulation
Example: Vertex class of BFS

class Vertex {
    public:
        int distance;
        int num_neighbors;

    Vertex** neighbors;     // or: std::vector<Vertex*>

    void visit(int iteration) {    // call iteratively for all vertices
        if (distance == index) {
            for (int i = 0; i < num_neighbors; ++i) {
                neighbors[i]->distance = index + 1;
            }
        }
    }
};
Example: Vertex class of BFS

class Vertex {
    public:
        int distance;
        int num_neighbors;

    Vertex** neighbors;  // or: std::vector<Vertex*>  

    void visit(int iteration) {    // call iteratively for all vertices
        if (distance == index) {
            for (int i = 0; i < num_neighbors; ++i) {
                neighbors[i]->distance = index + 1;
            }
        }
    }
};
Example: Vertex class of BFS

```cpp
class Vertex {
    public:
        int distance;
        int num_neighbors;

        Vertex** neighbors; // or: std::vector<Vertex*>

    void visit(int iteration) {
        // call iteratively for all vertices
        if (distance == index) {
            for (int i = 0; i < num_neighbors; ++i) {
                neighbors[i]->distance = index + 1;
            }
        }
    }
};
```
Example: Vertex class of BFS

```cpp
class Vertex {
    public:
        int distance;
        int num_neighbors;

    Vertex** neighbors;  // or: std::vector<Vertex>*

    void visit(int iteration) {
        if (distance == index) {
            for (int i = 0; i < num_neighbors; ++i) {
                neighbors[i]->distance = index + 1;
            }
        }
    }
};
```
Example: Vertex class of BFS

class Vertex {
    public:
        int distance;
        int num_neighbors;

    Vertex** neighbors;  // or: std::vector<Vertex*> 

    void visit(int iteration) {
        // call iteratively for all vertices
        if (distance == index) {
            for (int i = 0; i < num_neighbors; ++i) {
                neighbors[i]->distance = index + 1;
            }
        }
    }
};
Example: Vertex class of BFS

class Vertex {
    public:
    int distance;
    int num_neighbors;

    Vertex** neighbors;  // or: std::vector<Vertex*> 

    void visit(int iteration) {
        // call iteratively for all vertices
        if (distance == index) {
            for (int i = 0; i < num_neighbors; ++i) {
                neighbors[i]->distance = index + 1;
            }
        }
    }
};

06/19/2018
Example: Vertex class of BFS

class Vertex {
    public:
        int distance;
        int num_neighbors;

    Vertex** neighbors; // or: std::vector<Vertex*>(

    void visit(int iteration) { // call iteratively for all vertices
        if (distance == index) {
            for (int i = 0; i < num_neighbors; ++i) {
                neighbors[i]->distance = index + 1;
            }
        }
    }
};
Example: Vertex class of BFS

class Vertex {
    public:
        int distance;
        int num_neighbors;
        Vertex** neighbors; // or: std::vector<Vertex>*

    void visit(int iteration) { // call iteratively for all vertices
        if (distance == index) {
            for (int i = 0; i < num_neighbors; ++i) {
                neighbors[i]->distance = index + 1;
            }
        }
    }
};
Layout of Array-typed Fields

- **No Inlining**
  - `Vertex**
  - `std::vector<Vertex*>`

- **Full Inlining**
  - `Vertex*[5]`
  - `std::array<Vertex*, 5>`

- **Partial Inlining**
  - `absl::inlined_vector<Vertex*, 2>`

Reduce memory footprint
Layout of Array-typed Fields

No Inlining
- Vertex**
- std::vector<Vertex*>  
  
AOS | SOA

Full Inlining
- Vertex*[5]
- std::array<Vertex*, 5>
  
AOS  | SOA/split | SOA/object

Partial Inlining
- absl::inlined_vector<Vertex*, 2>
  
AOS  | SOA/split | SOA/object

Split array by indices; one SOA array per array slot

Previous work on Array Inlining in JVM: C. Wimmer, H. Mössenböck: Automatic Array Inlining in Java Virtual Machines, CGO '08

Treat array as normal object (just a bunch bytes...)
class Vertex {
    public:
    int distance;
    int num_neighbors;
    Vertex** neighbors;
    // std::vector<Vertex>*
};

Vertex vertices[100];

+ Arrays: Can grow in size
+ Good cache utilization if all elements are accessed (cache line)
– Padding of objects due to alignment
class Vertex : public AosLayout<Vertex, 100> {
    public: IKRA_INITIALIZE_CLASS
    int_ distance;
    int_ num_neighbors;

    field_(Vertex**) neighbors;
    // std::vector<Vertex*> ]
};

IKRA_DEVICE_STORAGE(Vertex)

+ Arrays: Can grow in size
+ Good cache utilization if all elements are accessed (cache line)
– Padding of objects due to alignment
class Vertex {
public:
    int distance;
    int num_neighbors;

    Vertex*[5] neighbors;
    // std::array<Vertex*, 5>
};

Vertex vertices[100];
**Partial Inlining, AOS**

```cpp
class Vertex {
public:
    int distance;
    int num_neighbors;

    absl::inlined_vector<Vertex*, 2> neighbors;
};

Vertex vertices[100];
```

- Arrays: No pointer indirection in most cases
- Arrays: Can grow in size
class Vertex : public SoaLayout<Vertex, 100> {
  public: IKRA_INITIALIZE_CLASS
    int_ distance;
    int_ num_neighbors;

    field_(std::array<Vertex*, 5>)
      neighbors;
};  IKRA_DEVICE_STORAGE(Vertex)

+ Arrays: No pointer indirection
+ Arrays: Suitable for nested parallelism
   (coalesced array access)
– Arrays: High memory footprint
– Arrays: Cannot grow in size

<table>
<thead>
<tr>
<th>Vertex*[5][100]</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0.neighbors[0]</td>
</tr>
<tr>
<td>v0.neighbors[1]</td>
</tr>
<tr>
<td>v0.neighbors[2]</td>
</tr>
<tr>
<td>v0.neighbors[3]</td>
</tr>
<tr>
<td>v0.neighbors[4]</td>
</tr>
<tr>
<td>v1.neighbors[0]</td>
</tr>
<tr>
<td>v1.neighbors[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int[100]</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0.distance</td>
</tr>
<tr>
<td>v1.distance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int[100]</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0.num_neighbors</td>
</tr>
<tr>
<td>v1.num_neighbors</td>
</tr>
</tbody>
</table>
class Vertex : public SoaLayout<Vertex, 100> {
  public: IKRA_INITIALIZE_CLASS
    int_ distance;
    int_ num_neighbors;

    fully_inlined_array_(Vertex*, 5) neighbors;
};  IKRA_DEVICE_STORAGE(Vertex)

+ Arrays: Easy address computation
+ Arrays: No pointer indirection
+ Arrays: Potential for memory coalescing
– Arrays: High memory footprint
Synthetic Benchmark

class DummyClass {
  public:
    int increment;
    int array_size;
    int* array;

  void benchmark() {
    for (int i = 0; i < array_size; ++i) {
      array[i] += increment;
    }
  }
};

Size between 32 and 64, evenly distributed.
Synthetic Benchmark

Benefit of SOA (without array inlining)

0 12 25 38 51 64 ms

partial, AOS

partial, SOA/split
Synthetic Benchmark

Benefit of coalesced access (SOA) of array elements

partial, AOS

partial, SOA/split

06/19/2018

Inner Array Inlining for SOA Layout
Frontier-based BFS Benchmark
Frontier-based BFS Benchmark

In every iteration:
- Process “random” set of vertices (“frontier”)
- Vertices processed in a warp are not consecutive
- No memory coalescing / vectorization

No benefit of “split” for array-typed fields
Example: Traffic Flow Simulation

- Based on Nagel-Schreckenberg model (cellular automaton)
- Simple model, can reproduce traffic jams, etc.
- Divide streets in cells: may contain 0 or 1 vehicles
Nagel-Schreckenberg Iteration

1. Increase velocity $v_i$ of vehicle (#cells / iteration)
2. Compute movement path, i.e., the next $v_i$ many cells
3. Reduce $v_i$ to avoid collisions and to satisfy speed limits
4. Randomly reduce $v_i$ (“randomization”)
5. Move vehicle acc. to path
1. Increase velocity $v_i$ of vehicle (#cells / iteration)
2. Compute movement path, i.e., the next $v_i$ many cells
3. Reduce $v_i$ to avoid collisions and to satisfy speed limits
4. Randomly reduce $v_i$ ("randomization")
5. Move vehicle acc. to path

---

Write Cell* array sequentially
Nagel-Schreckenberg Iteration

1. Increase velocity $v_i$ of vehicle (#cells / iteration)
2. Compute movement path, i.e., the next $v_i$ many cells
3. Reduce $v_i$ to avoid collisions and to satisfy speed limits
4. Randomly reduce $v_i$ (“randomization”)
5. Move vehicle acc. to path

Write Cell* array sequentially

Read Cell* array sequentially

Read Cell* array sequentially

Write Cell* array sequentially

Avoid collision: Max. velocity: 2 cells/iteration

Temp. speed limit controlled by traffic light. Only one cell can have velocity > 0.
### Design of Traffic Flow Simulation

#### Cell
- **car** : Car*
- **incoming_cells** : Cell*
  - **is_free** : bool
- **outgoing_cells** : Cell*
  - **speed_limit** : int
  - **temp_speed_limit** : int = INFTY
  - **type** : OsmStreetType

  + **is_free()** : bool
  + **occupy(car : Car*)**
  + **outgoing_cell(idx : int) : Cell**

  + **step1_increase_velocity()**
  + **step2_calculate_path()**
  + **step3_constraint_velocity()**
  + **step4_randomize()**
  + **step5_move()**

#### Car
- **is_active** : bool
- **max_velocity** : int
- **position** : Cell*
- **velocity** : int

  + **path** : Cell*

  + **step1_increase_velocity()**
  + **step2_calculate_path()**
  + **step3_constraint_velocity()**
  + **step4_randomize()**
  + **step5_move()**

#### SharedSignalGroup
- **cells** : Cell*

  + **cell(idx : int) : Cell**

  + **step1_increase_velocity()**
  + **step2_calculate_path()**
  + **step3_constraint_velocity()**
  + **step4_randomize()**
  + **step5_move()**

#### TrafficLight
- **groups** : SharedSignalGroup*

  + **step()**

#### SmartTrafficLight
- **groups** : SharedSignalGroup*

  + **step()**

#### YieldController
- **groups** : SharedSignalGroup*

  + **step()**
Performance of Traffic Simulation

- full, SOA/split significantly faster than full, SOA/object: memory coalescing
- AOS performance degrades with high inlining size because most vehicles have a low velocity
- Increasing memory footprint for inlining size > 12 (every vehicle has a max. velocity of at least 12)
Summary

• Extension of SOA layout to array-typed fields: Group array elements by index instead of object/struct (SOA/split)

• Partial Inlining: Reduce high memory footprint caused by “outliers”, but maintain overall performance.

• Limitation: Coalesced access only if all objects/structs are read “in sequence” (not the case for many graph algorithms like BFS)

• SOA/object is useful for nested parallelism (c.f. Virtual Warp-centric Programming paper; future work)

• Ikra-Cpp: Layout is chosen manually, but easy to change
Appendix
class Vertex : public SoaLayout<Vertex, 100> {
    public: IKRA_INITIALIZE_CLASS
    int_ distance;
    int_ num_neighbors;

    field_(Vertex**) neighbors;  // std::vector<Vertex**>
};  IKRA_DEVICE_STORAGE(Vertex)

+ Potential for memory coalescing
+ Arrays: Good cache utilization if all elements are accessed (cache line)
class Vertex : public SoaLayout<Vertex, 100> {
    public: IKRA_INITIALIZE_CLASS
    int distance;
    int num_neighbors;
    field_(Vertex**) neighbors; // std::vector<Vertex*>}
    IKRA_DEVICE_STORAGE(Vertex)

+ Potential for memory coalescing
+ Arrays: Good cache utilization if all elements are accessed (cache line)
Partial Inlining, SOA/split

```cpp
class Vertex : public SoaLayout<Vertex, 100> {
    public: IKRA_INITIALIZE_CLASS
    int_ distance;
    int_ num_neighbors;

    inlined_array_(Vertex*, 2)
    neighbors;
};  IKRA_DEVICE_STORAGE(Vertex)
```

+ Arrays: No pointer indirection in most cases
+ Arrays: Potential for memory coalescing
+ Arrays: Can grow in size