東京工業大学
Tokyo Institute of Technology

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

```
struct Body {
    float pos_x;
    float pos_y;
};
Body bodies[100];
```

- SOA: **S**tructure **of** **A**rrays
  - 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

```
namespace Body {
    float pos_x[100];
    float pos_y[100];
}
```

SOA array

Ikra-Cpp: A C++/CUDA DSL for SOA

東京工業大学
Tokyo Institute of Technology

- 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)

東京工業大学
Tokyo Institute of Technology

```
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);
}
```

```
namespace Body {
  double pos_x[50];
  double pos_y[50];
  double vel_x[50];
  double vel_y[50];
  int num_Body = 0;

  /* ... */

  void move(int id,
            double dt) {
    pos_x[id] += vel_x[id]*dt;
    pos_y[id] += vel_y[id]*dt;
  }
}
```

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

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

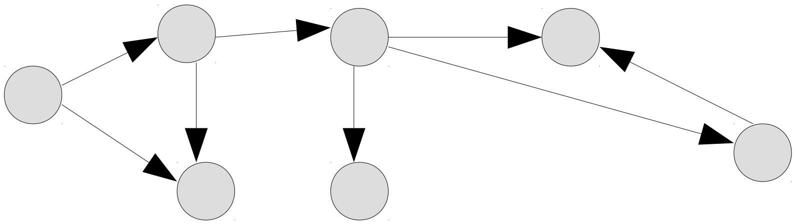
```
class Vertex {
  public:
    int distance;
    int num_neighbors;

    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

東京工業大学
Tokyo Institute of Technology

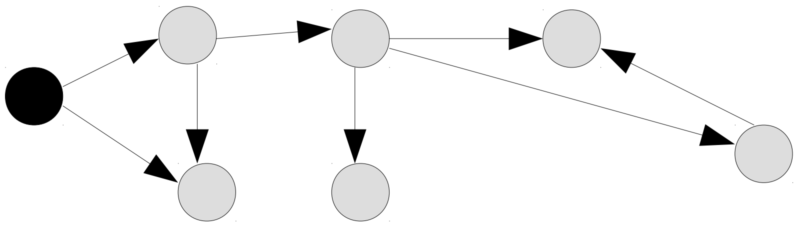


```
class Vertex {
  public:
    int distance;
    int num_neighbors;

    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

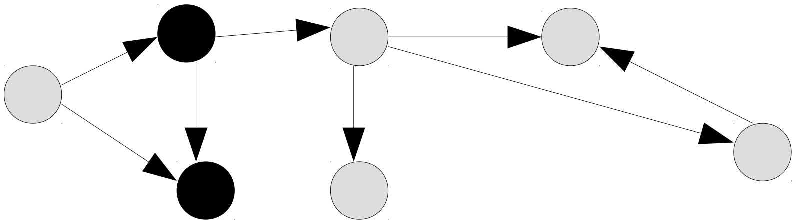


```
class Vertex {
  public:
    int distance;
    int num_neighbors;

    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

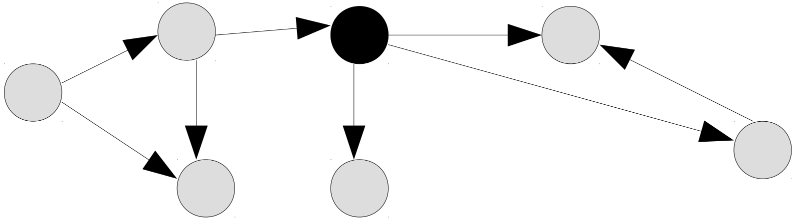


```
class Vertex {
  public:
    int distance;
    int num_neighbors;

    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

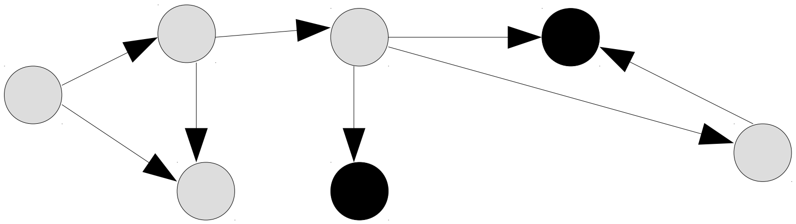


```
class Vertex {
  public:
    int distance;
    int num_neighbors;

    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

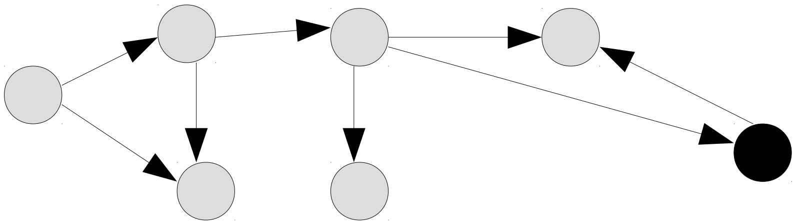


```
class Vertex {
  public:
    int distance;
    int num_neighbors;

    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

東京工業大学
Tokyo Institute of Technology

```
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>`

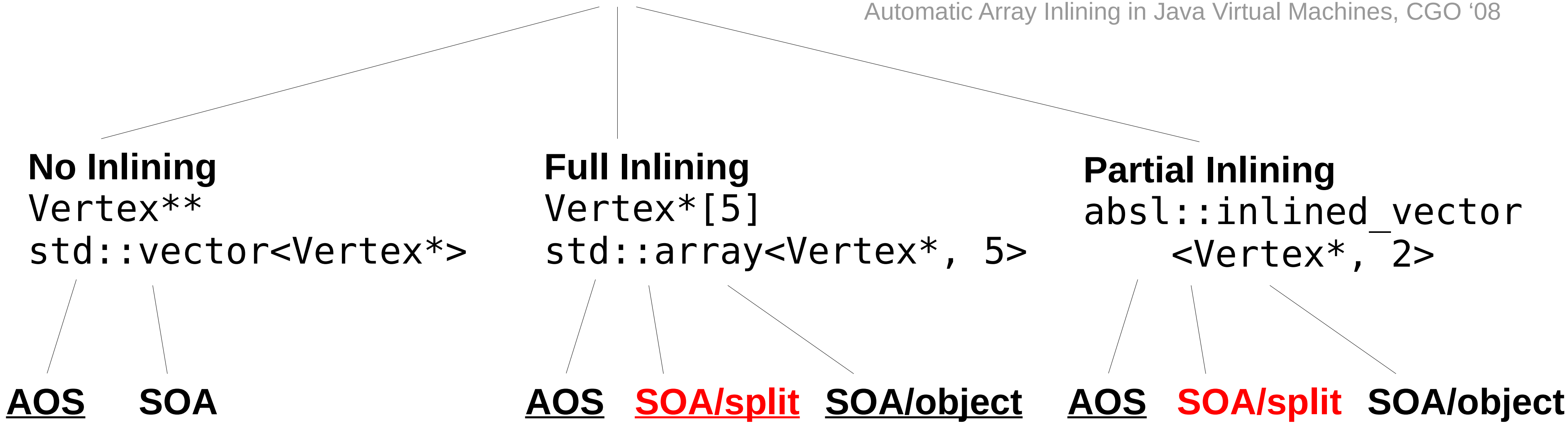
Partial Inlining

```
absl::inlined_vector
    <Vertex*, 2>
```

Reduce memory footprint

Layout of Array-typed Fields

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



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

Treat array as normal object (just a bunch bytes...)

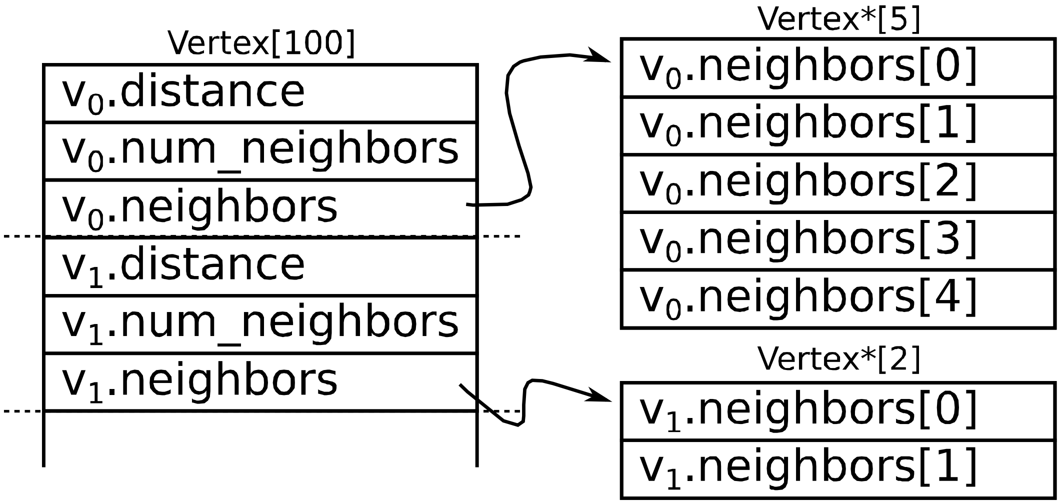
No Inlining, AOS

```
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



Vertex[100]

v_0.distance

v_0.num_neighbors

v_0.neighbors

v_1.distance

v_1.num_neighbors

v_1.neighbors

Vertex*[5]

v_0.neighbors[0]

v_0.neighbors[1]

v_0.neighbors[2]

v_0.neighbors[3]

v_0.neighbors[4]

Vertex*[2]

v_1.neighbors[0]

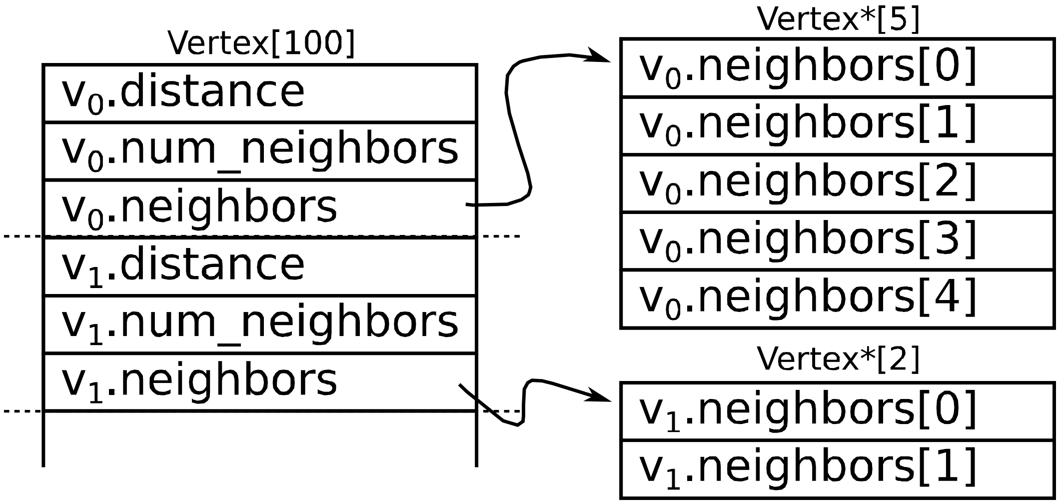
v_1.neighbors[1]

No Inlining, AOS

```
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)
```



Vertex[100]

v_0.distance

v_0.num_neighbors

v_0.neighbors

v_1.distance

v_1.num_neighbors

v_1.neighbors

Vertex*[5]

v_0.neighbors[0]

v_0.neighbors[1]

v_0.neighbors[2]

v_0.neighbors[3]

v_0.neighbors[4]

Vertex*[2]

v_1.neighbors[0]

v_1.neighbors[1]

+ Arrays: Can grow in size
+ Good cache utilization if all elements are accessed (cache line)
– Padding of objects due to alignment

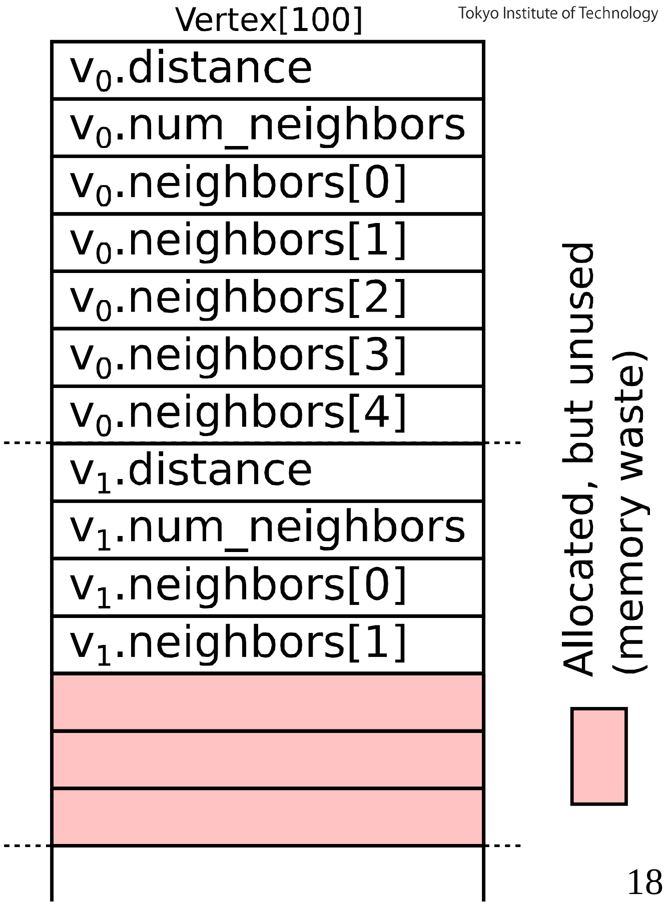
Full Inlining, AOS

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

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

Vertex vertices[100];
```

+ Arrays: Easy address computation
+ Arrays: No pointer indirection
– Arrays: High memory footprint
– Arrays: Cannot grow in size



Vertex[100]

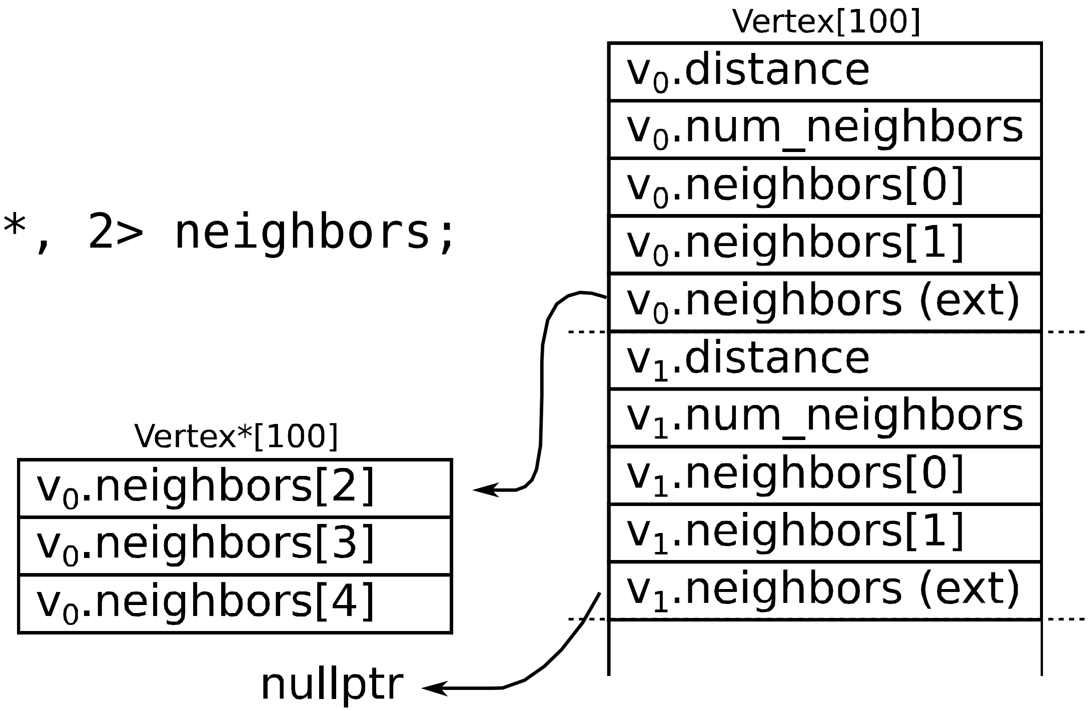
v_0.distance
v_0.num_neighbors
v_0.neighbors[0]
v_0.neighbors[1]
v_0.neighbors[2]
v_0.neighbors[3]
v_0.neighbors[4]
v_1.distance
v_1.num_neighbors
v_1.neighbors[0]
v_1.neighbors[1]

🟥 Allocated, but unused (memory waste)

Partial Inlining, AOS

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

    absl::inlined_vector<Vertex*, 2> neighbors;
};
Vertex vertices[100];
```



Vertex[100]

v_0.distance
v_0.num_neighbors
v_0.neighbors[0]
v_0.neighbors[1]
v_0.neighbors (ext)
v_1.distance
v_1.num_neighbors
v_1.neighbors[0]
v_1.neighbors[1]
v_1.neighbors (ext)

Vertex*[100]

v_0.neighbors[2]
v_0.neighbors[3]
v_0.neighbors[4]

nullptr

+ Arrays: No pointer indirection in most cases
+ Arrays: Can grow in size

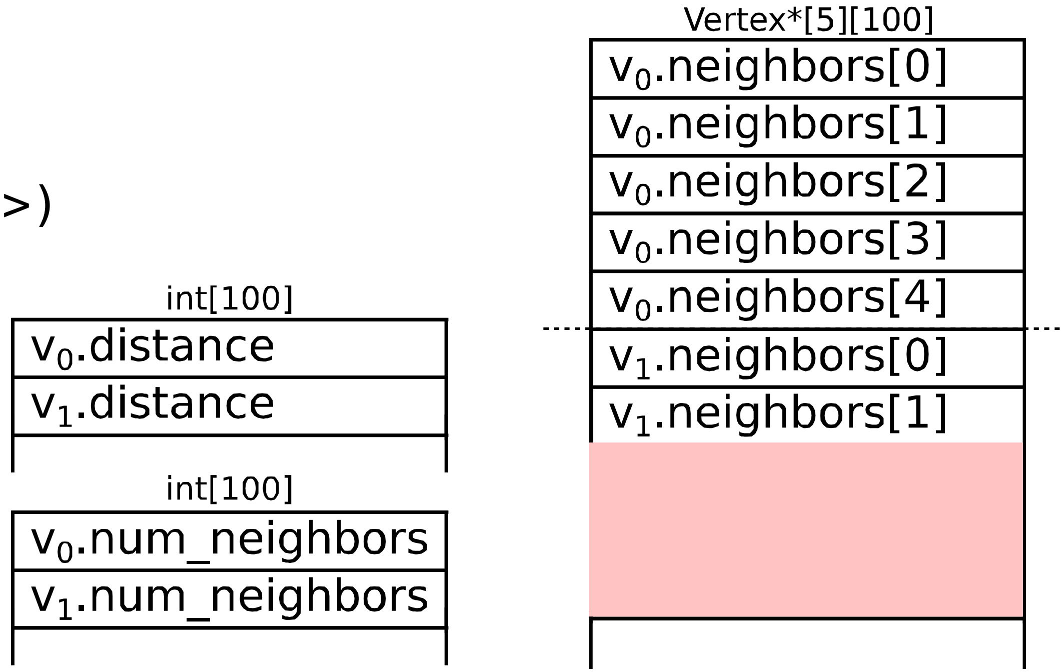
Full Inlining, SOA/object

```
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

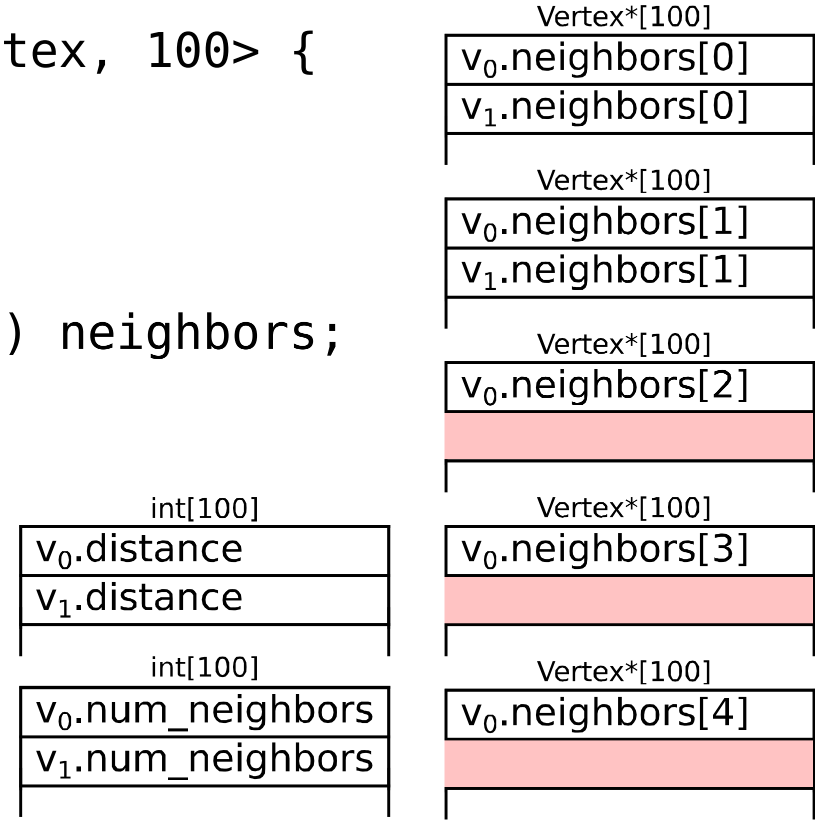


Full Inlining, SOA/split

```
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)
```



Vertex*[100]
v_0.neighbors[0]
v_1.neighbors[0]

Vertex*[100]
v_0.neighbors[1]
v_1.neighbors[1]

Vertex*[100]
v_0.neighbors[2]

int[100]
v_0.distance
v_1.distance

Vertex*[100]
v_0.neighbors[3]

int[100]
v_0.num_neighbors
v_1.num_neighbors

Vertex*[100]
v_0.neighbors[4]

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

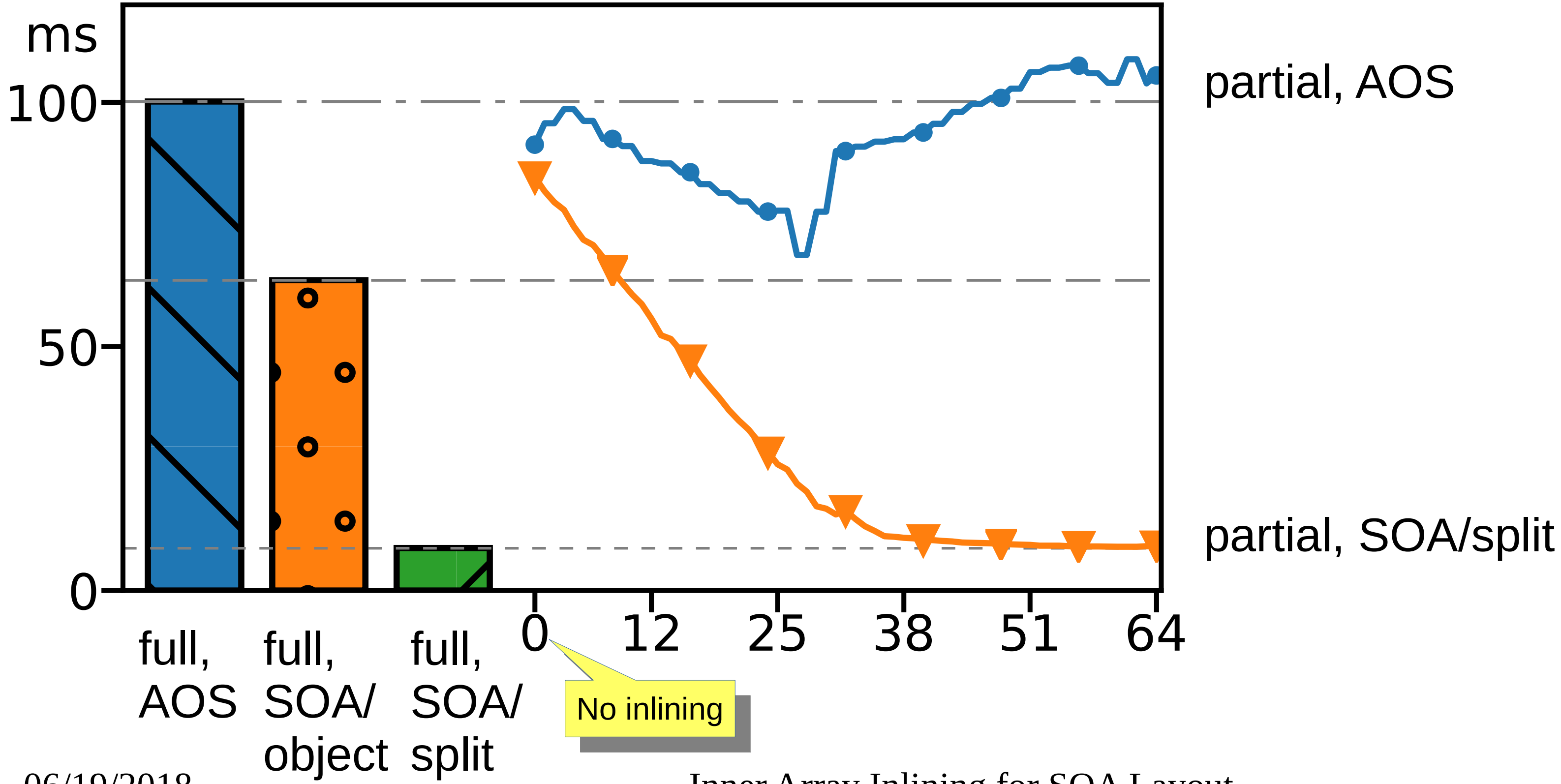
Synthetic Benchmark

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

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

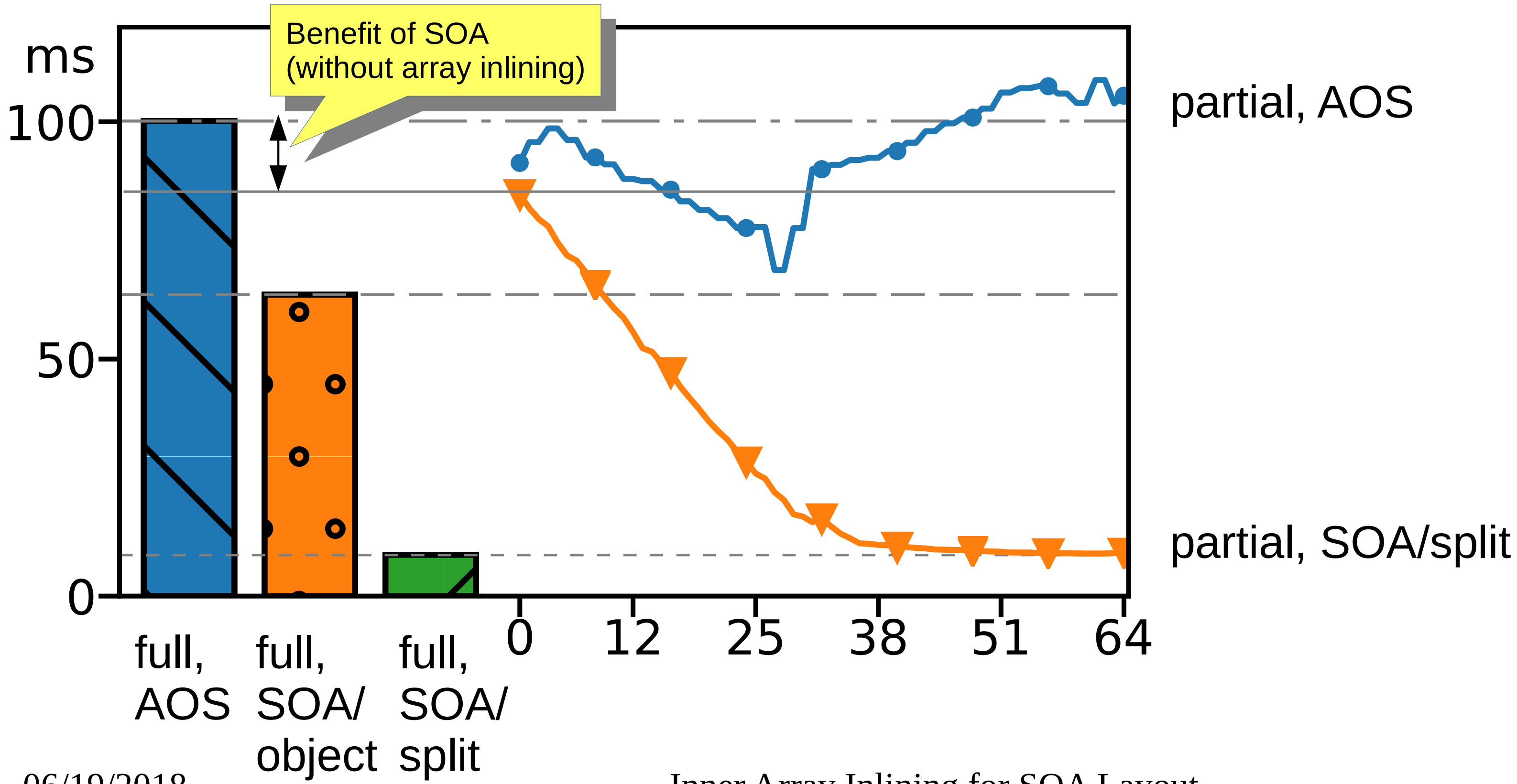
Size between 32 and 64, evenly distributed.

Synthetic Benchmark

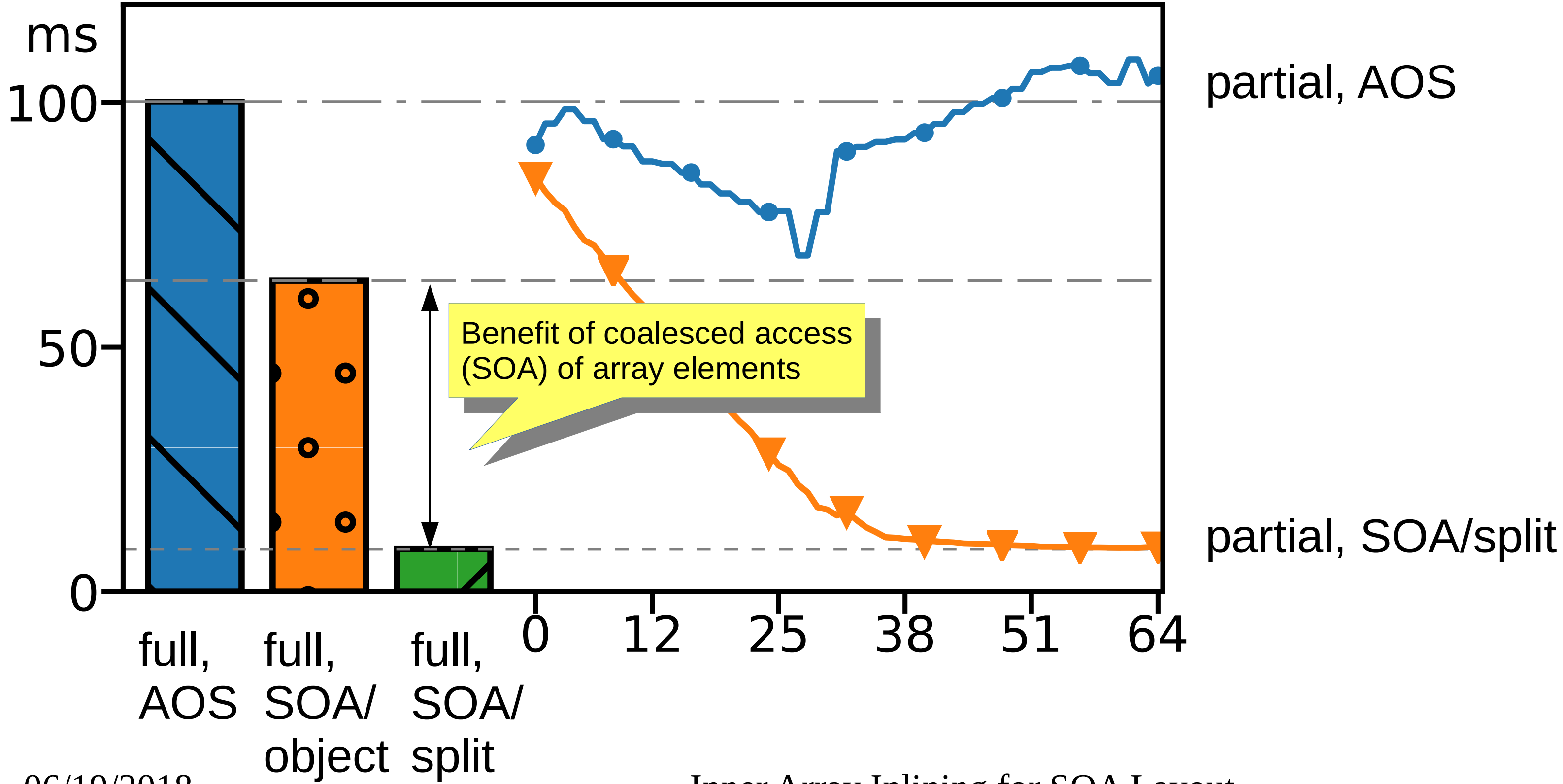


No inlining

Synthetic Benchmark

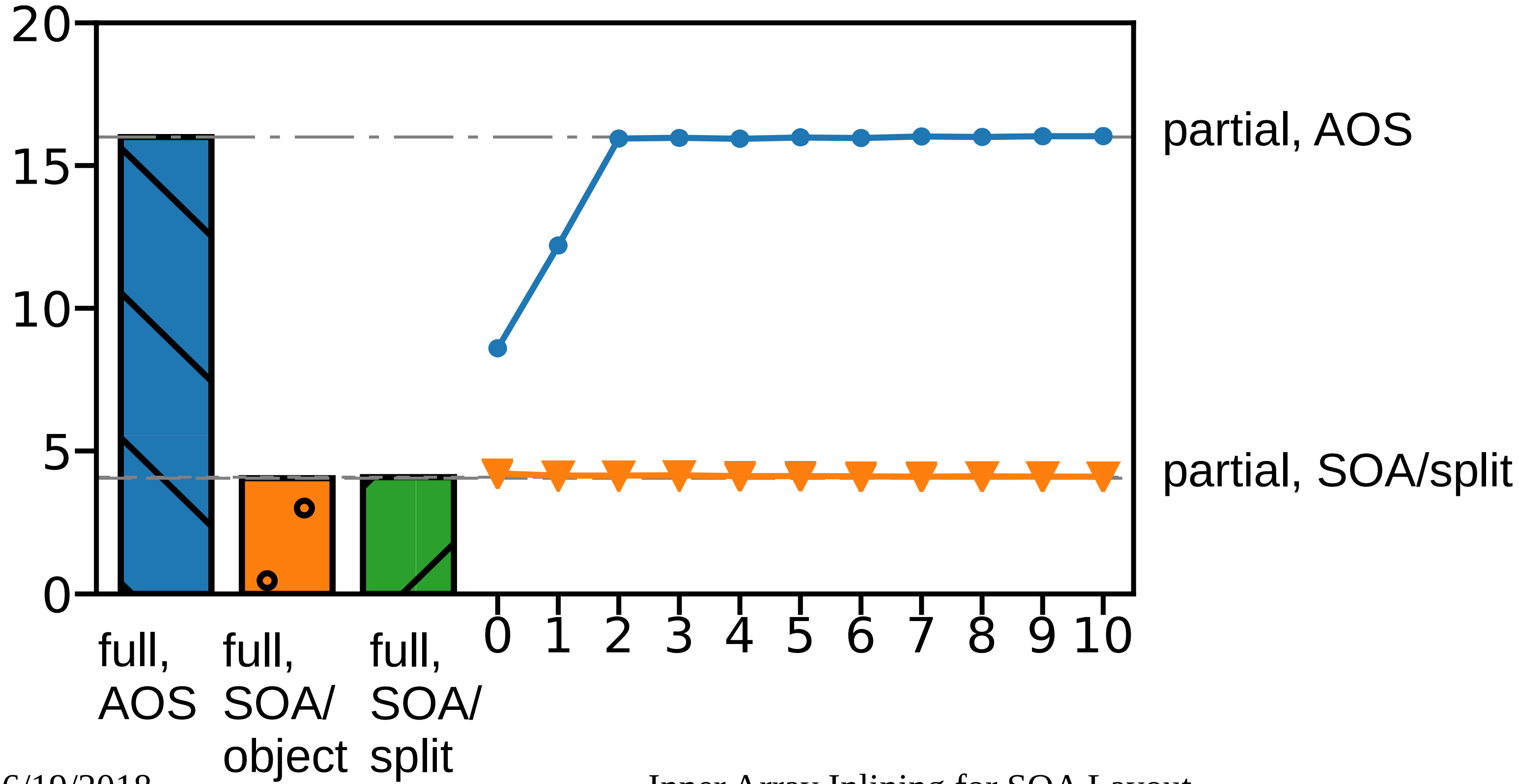


Synthetic Benchmark

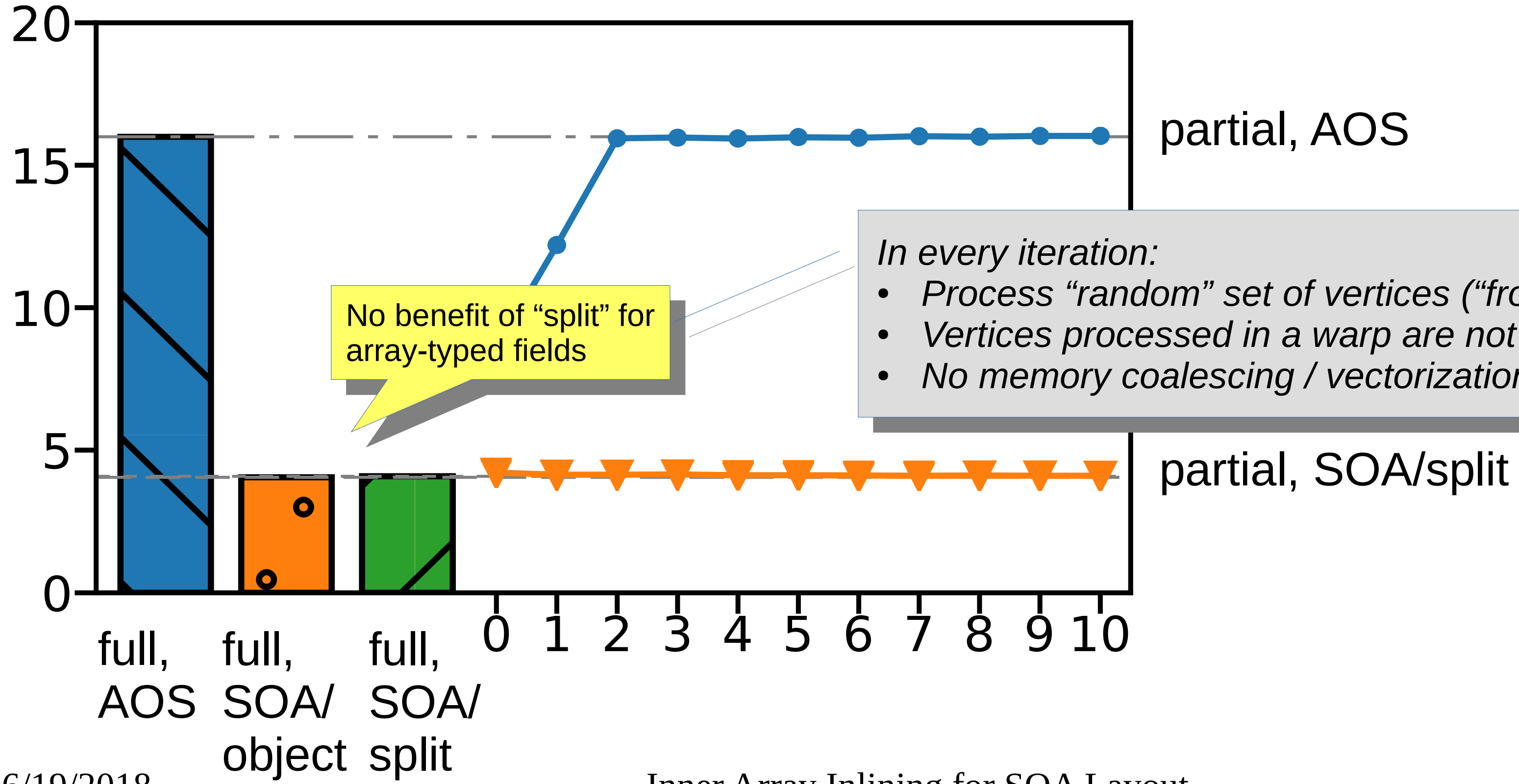


Benefit of coalesced access (SOA) of array elements

Frontier-based BFS Benchmark



Frontier-based BFS Benchmark



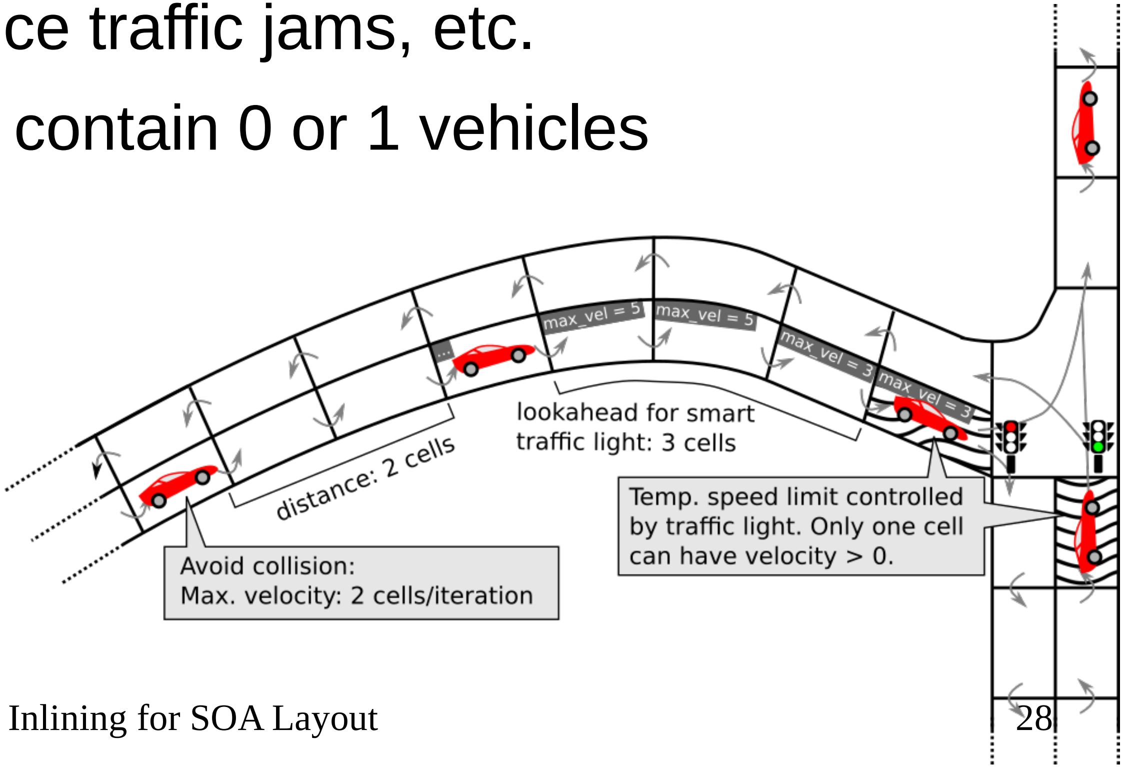
No benefit of “split” for array-typed fields

In every iteration:

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

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



distance: 2 cells

lookahead for smart traffic light: 3 cells

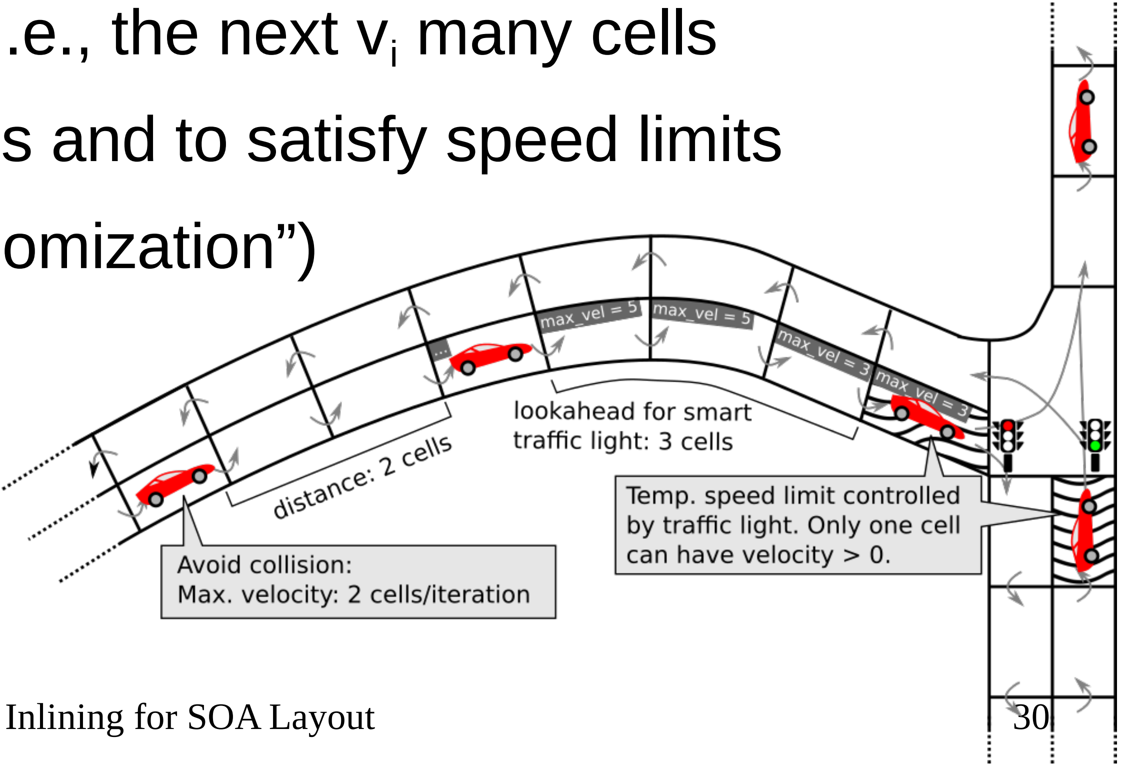
Avoid collision:
Max. velocity: 2 cells/iteration

Temp. speed limit controlled by traffic light. Only one cell can have velocity > 0.

```
76      6
77      7
78
79
80
81
82
83
84
85
86
88
89
90
91
92
93
94
95
97
98
99
10
10
10
11
11
11
11
11
11
11
14
14
14
15
18
21
22
22
63
Co
65
Ad
Nu
Nu
Nu
Nu
Nu
Nu
Nu
Us
```

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



distance: 2 cells

lookahead for smart traffic light: 3 cells

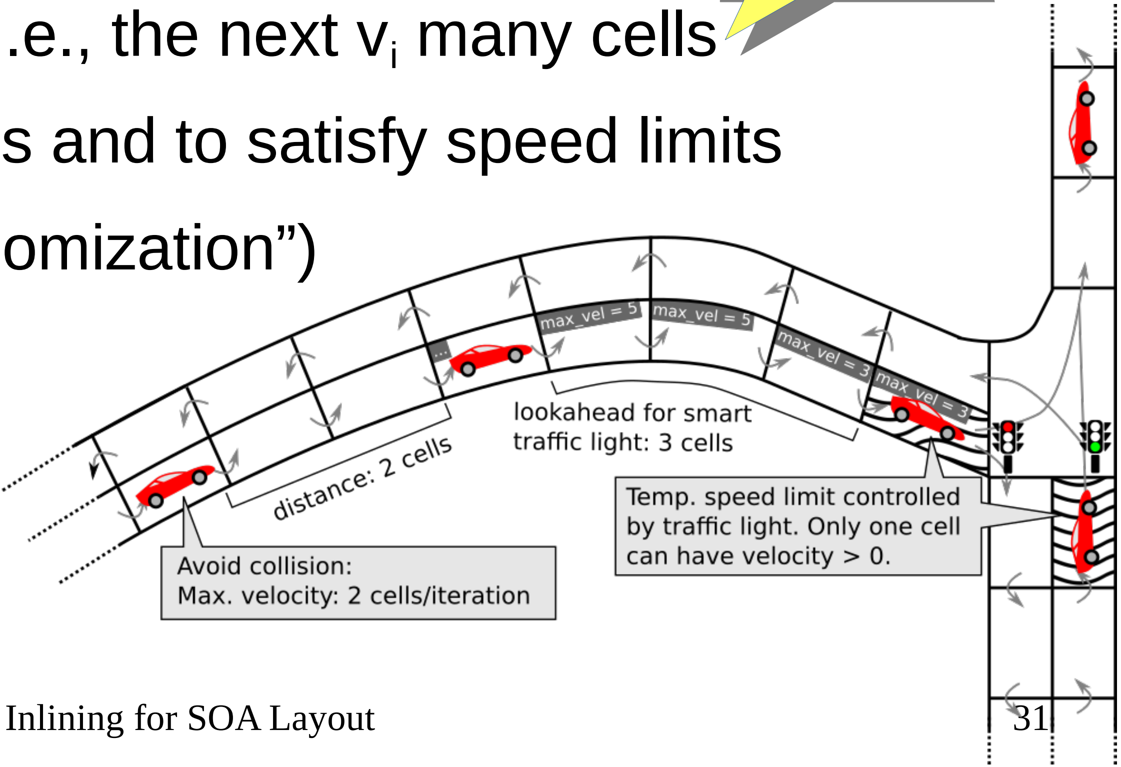
Temp. speed limit controlled by traffic light. Only one cell can have velocity > 0.

Avoid collision:
Max. velocity: 2 cells/iteration

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



lookahead for smart traffic light: 3 cells

Temp. speed limit controlled by traffic light. Only one cell can have velocity > 0.

Avoid collision:
Max. velocity: 2 cells/iteration

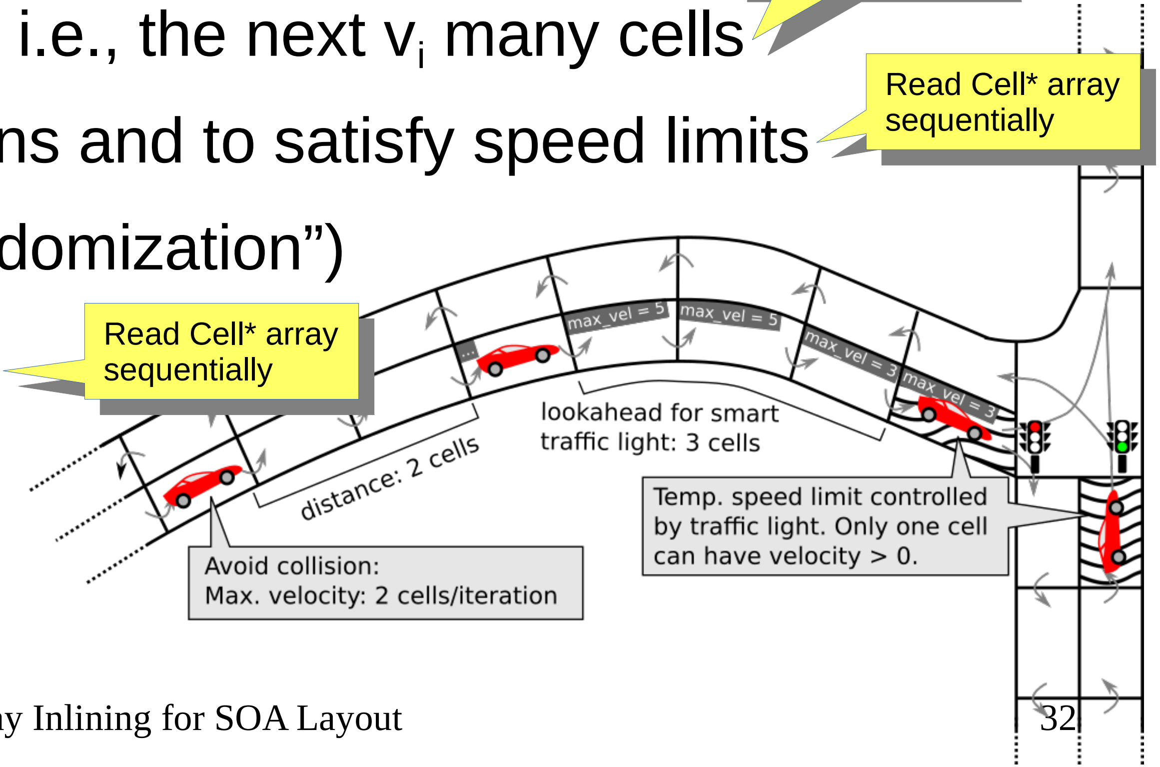
Nagel-Schreckenberg Iteration

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

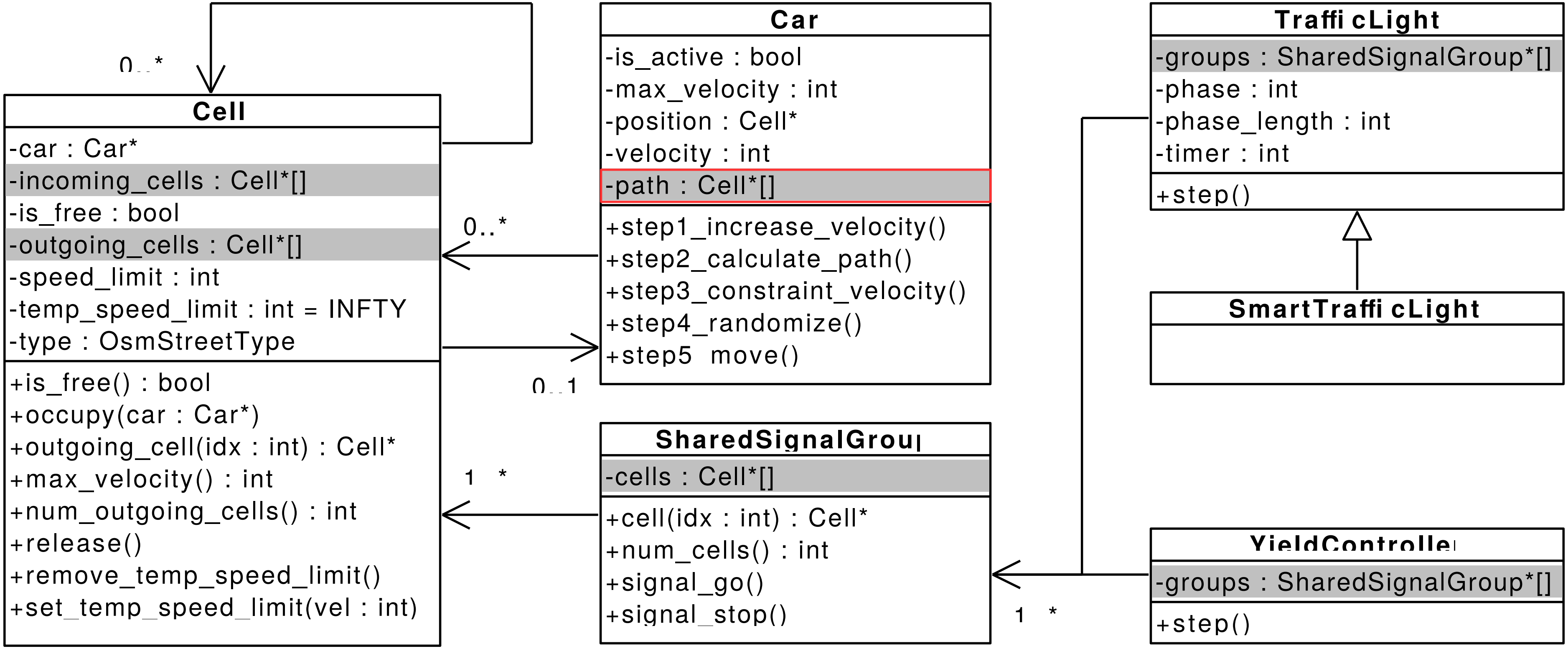


Avoid collision:
Max. velocity: 2 cells/iteration

Temp. speed limit controlled by traffic light. Only one cell can have velocity > 0.

Design of Traffic Flow Simulation

東京工業大学
Tokyo Institute of Technology



0..*

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*
+max_velocity() : int
+num_outgoing_cells() : int
+release()
+remove_temp_speed_limit()
+set_temp_speed_limit(vel : int)

0..*

0..1

1 *

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()

SharedSignalGrou

-cells : Cell*[]

+cell(idx : int) : Cell*
+num_cells() : int
+signal_go()
+signal_stop()

1 *

TrafficLight

-groups : SharedSignalGroup*[]
-phase : int
-phase_length : int
-timer : int

+step()

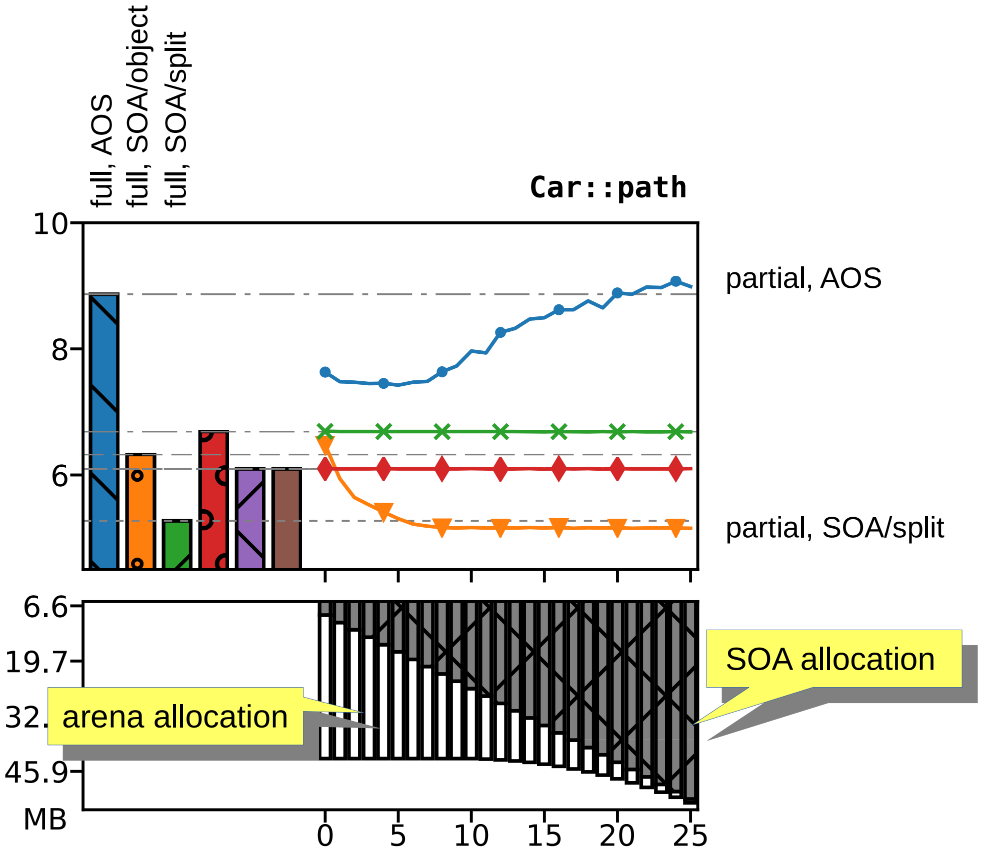
SmartTrafficLight

YieldControlle

-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

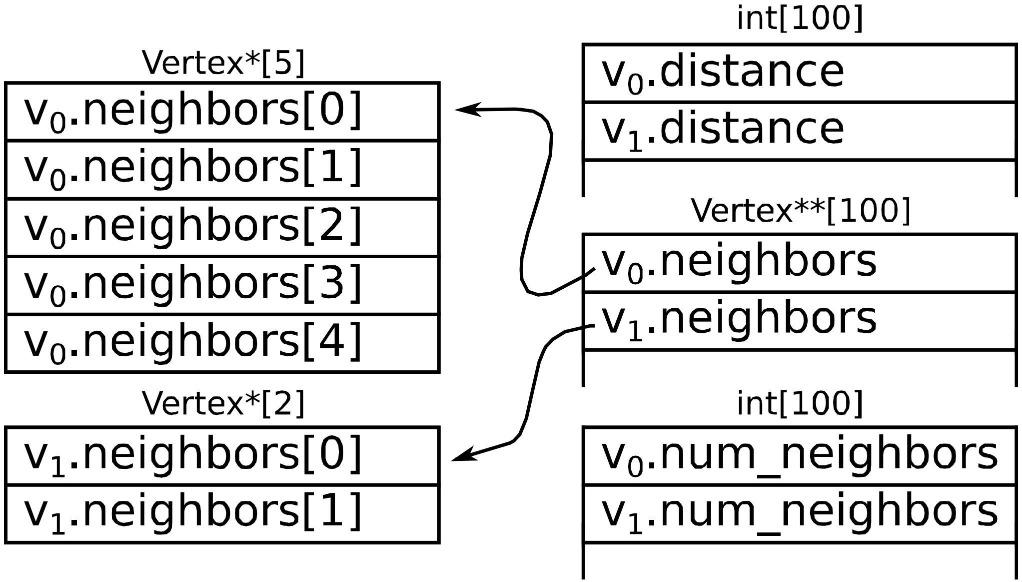
東京工業大学
Tokyo Institute of Technology

Appendix

No Inlining, SOA

```
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)

No Inlining, SOA

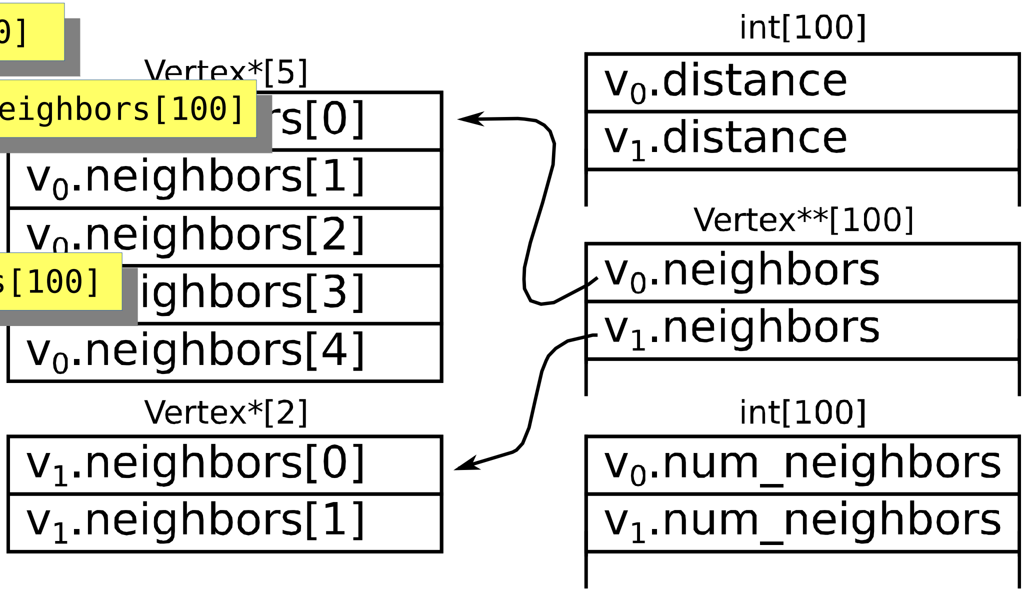
```
class Vertex : public SoaLayout<Vertex, 100> {
  public: IKRA_INITIALIZE_CLASS
    int_ distance;
    int_ num_neighbors;

    field_(Vertex**) neighbors;
    // std::vector<V
};   IKRA_DEVICE_STORA
```

`int distance[100]`

`int num_neighbors[100]`

`Vertex** neighbors[100]`



Vertex*[5]
rs[0]
v_0.neighbors[1]
v_0.neighbors[2]
ighbors[3]
v_0.neighbors[4]

Vertex*[2]
v_1.neighbors[0]
v_1.neighbors[1]

int[100]
v_0.distance
v_1.distance

Vertex**[100]
v_0.neighbors
v_1.neighbors

int[100]
v_0.num_neighbors
v_1.num_neighbors

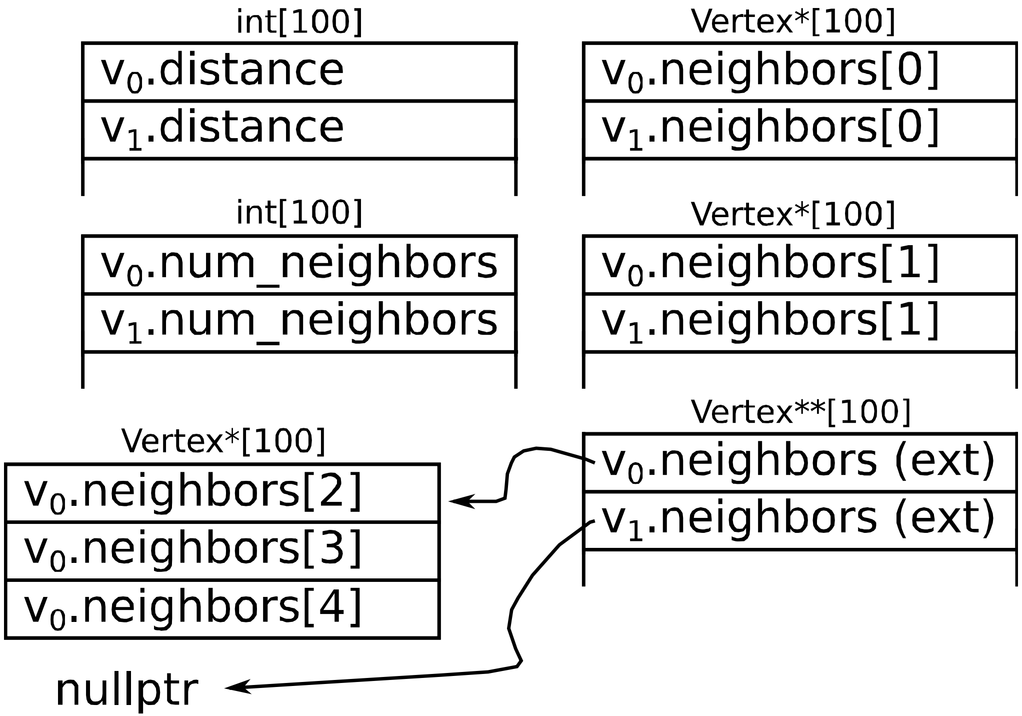
+ Potential for memory coalescing
+ Arrays: Good cache utilization if all elements are accessed (cache line)

Partial Inlining, SOA/split

```
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