Inner Array Inlining for Structure of Arrays Layout

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

Yaozhu Sun
Tokyo Institute of Technology
Japan
sunyaozhu@prg.is.titech.ac.jp

Hidehiko Masuhara
Tokyo Institute of Technology
Japan
masuhara@acm.org

Abstract

Previous work has shown how the well-studied and SIMD-friendly Structure of Arrays (SOA) data layout strategy can speed up applications in high-performance computing compared to a traditional Array of Structures (AOS) data layout. However, a standard SOA layout cannot handle structures with inner arrays; such structures appear frequently in graph-based applications and object-oriented designs with associations of high multiplicity.

This work extends the SOA data layout to structures with array-typed fields. We present different techniques for inlining (embedding) inner arrays into an AOS or SOA layout, as well as the design and implementation of an embedded C++/CUDA DSL that lets programmers write such layouts in a notation close to standard C++. We evaluate several layout strategies with a traffic flow simulation, an important real-world application in transport planning.

CCS Concepts • **Software and its engineering → Object oriented languages**; *Data types and structures*; *Parallel programming languages*;

Keywords CUDA, Data Inlining, Inner Arrays, Object-oriented Programming, SIMD, Structure of Arrays

ACM Reference Format:
Matthias Springer, Yaozhu Sun, and Hidehiko Masuhara. 2018. Inner Array Inlining for Structure of Arrays Layout. In *Proceedings of 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'18).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3219753.3219760

1 Introduction

In recent years a variety of applications [19] have been implemented on GPUs in order to utilize their massive parallelism, in areas such as database systems, machine learning, mathematics or real-world simulations (e.g., traffic simulations [23, 24]). Our recent focus is on applications that are amenable to object-oriented programming such as agent-based simulations (e.g., traffic flow simulations [21]) or graph processing algorithms. Due to conceptual differences in the programming models and hardware architectures between CPUs and GPUs, developing performant GPU programs is not straightforward for most programmers. There are a number of best practices for achieving good performance on GPUs, ranging from control flow optimizations to data layout optimizations; such optimizations are often tedious to implement, lead to less readable code and interfere with programming abstractions. One well-studied best practice for CPUs and GPUs is *Structure of Arrays* (SOA).

AOS and SOA Well-organized programs make frequent use of structs or classes. *Array of Structures* (AOS) and Structure of Arrays (SOA) describe techniques for organizing multiple structs/objects in memory. In AOS, the standard technique in most compilers, all fields of a struct are stored together. In SOA, all values of a field are stored together. SOA can benefit cache utilization [4] and is useful in SIMD programs: Two values of the same field but different object can be loaded into a vector register in a single instruction. Similarly, GPUs can combine multiple simultaneous global memory requests within a warp (group of 32 consec. threads) into fewer *coalesced* requests, if data is spatially local [6, 18].

SOA works well with simple data structures, but cannot be applied easily to structs that contain arrays or other collections of different size. Such arrays must be allocated at separate locations, requiring an additional pointer indirection (Fig. 1). In this work, we study two important real-world

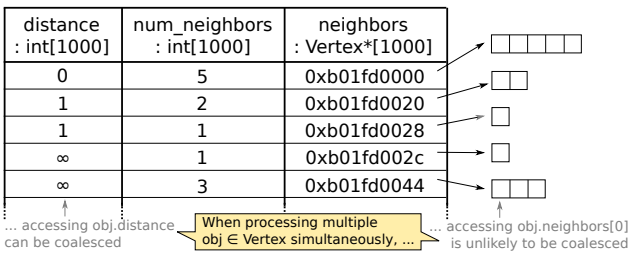


Figure 1. Example: Vertex class in SOA memory layout with 1000 objects. Values for `distance` and `num_neighbors` are stored in SOA arrays, but `neighbor` arrays must be stored in a different location, because every array may have a different size. This requires an additional pointer indirection and simultaneous accesses into `neighbor` arrays are unlikely to be coalesced.

ARRAY'18, June 19, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5852-1/18/06…$15.00
https://doi.org/10.1145/3219753.3219760

applications: breadth-first search (BFS) and an agent-based, object-oriented traffic flow simulation. In graph processing, most vertices have multiple neighbors and adjacency lists are the preferred representation for BFS on GPUs [5]. The traffic flow simulations exhibits graph-based features for representing street networks and utilizes array-based data structures within the simulation logic. Algorithms that iterate over arrays one-by-one or in a fashion that is *uniform* among all objects are particularly interesting, because their memory access can be optimized. While a standard SOA layout groups together all elements of an array per object, a different, more SIMD-friendly layout can group elements by array index and benefit from memory coalescing.

Ikra-Cpp Even though SOA can lead to better memory performance, it cannot be combined with object-oriented programming (OOP) in CUDA and many other programming languages while maintaining OOP abstractions [22]. C++ libraries like *SoAx* [7] and *ASX* [25], or the *ispc* compiler [20] lay out data as SOA while providing an AOS-style programming interface, but they do not support object-oriented programming. Neither do they support SIMD-friendly array-typed fields. The goal of the Ikra-Cpp project is to provide a set of language abstractions and optimizations for object-oriented high-performance computing on CPUs and GPUs. The focus of this paper is on array-typed fields.

Contributions and Outline This paper makes the following contributions.

- An analysis and performance evaluation of data layout strategies for array-typed fields in a SOA data layout.
- The design and implementation of Ikra-Cpp, an embedded C++/CUDA DSL that allows programmers to design object-oriented programs with array-typed fields in AOS-style notation, while storing data in SOA layout with inner array inlining.
- A parallel implementation of a traffic flow simulation, an important real-world application in transport planning, based on cellular automata on GPUs.

In the remainder of this paper, Sections 2 and 3 discuss layout strategies for array-typed field and their API/implementation in Ikra-Cpp. Section 4 shows how they can be applied in a larger project, a traffic flow simulation. Section 5 presents a performance evaluation with various layout strategies. Finally, Sections 6 and 7 discuss related work and conclude.

2 Data Layout Strategies for Inner Arrays

This section describes seven inner array layout strategies, focusing on C++-style arrays with fixed size after allocation. Fig. 2–5 visualize thesee strategies, using the `Vertex` class of a breath-first algorithm (Fig. 6(a)) as an example.

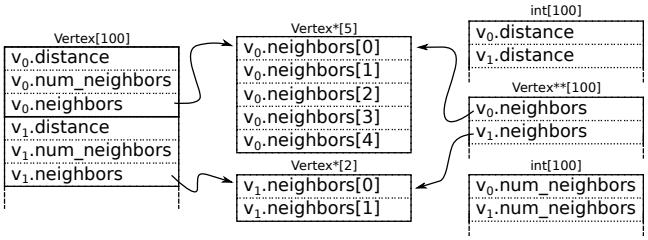


Figure 2. No Inlining: AOS (left side) and SOA (right side). Low memory footprint, but inner arrays must be accessed through a pointer indirection.

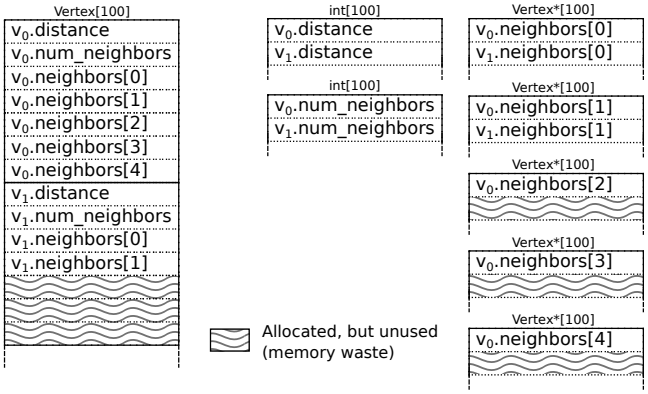


Figure 3. Full Inlining: AOS (left side) and SOA (middle, right side). High memory footprint if inner arrays have different sizes, but arrays can be accessed without pointer indirection.

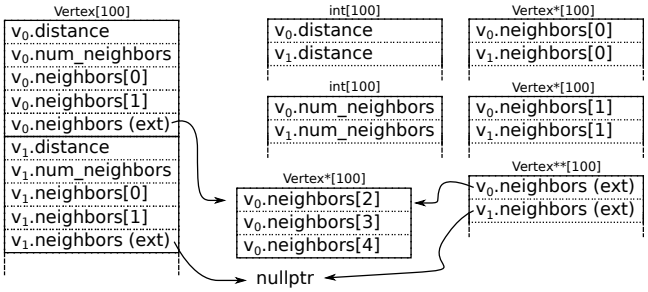


Figure 4. Partial Inlining: AOS (left side) and SOA (middle, right side). The first two inner array elements can be accessed without pointer direction, but a conditional branch is required to check if an element is stored in the object (inlined) or in the external storage.

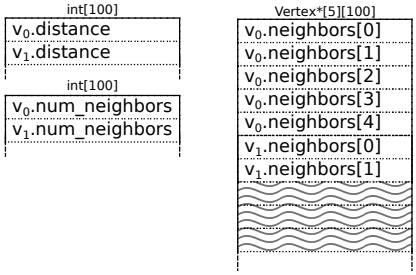


Figure 5. SOA, Array as Object. High memory footprint if inner arrays have different sizes, but arrays can be accessed without pointer indirection.

2.1 Data Layouts

We consider three kinds of layout strategies: *AOS*, *SOA* and *Array as Object* (i.e., SOA without handling inner arrays specially). For each one, inner arrays can either be *fully inlined*, *partially inlined* or not inlined at all (*without inlining*).

AOS without Inlining (Fig. 2 (left)) This is the default data layout that many programmers choose intuitively. Objects are laid out as AOS, i.e., all fields of an object are stored together. Since inner arrays can have different sizes, they are allocated at different locations and referenced with pointers. This can either be done manually (using `malloc`/`new`) or with a helper class like `std::vector<T>`. Depending on the structure, compilers may have to add padding to ensure that all fields are aligned properly. "AOS without Inlining" is identical to "AOS with Partial Inlining" with an inlining size of zero; the conditional branch is optimized away.

AOS with Full Inlining (Fig. 3 (left)) This layout strategy still lays out objects as AOS, but inlines inner arrays fully into objects, such that they can be accessed more efficiently without a pointer indirection. Furthermore, small inner arrays may be able to share cache lines with other fields, and thus benefit cache utilization. However, this approach wastes memory; all inner arrays must have the same size, i.e., the largest size among all inner arrays, to be able to hold all elements. The amount of wasted memory depends on the variance among inner array sizes. This layout can be implemented with a helper class like `std::array<T, N>`.

AOS with Partial Inlining (Fig. 4 (left)) This layout is a mixture of the previous two strategies. Up to N many inner array elements are inlined into objects, where N is a compile-time constant. Elements with an index $\geq N$ are stored externally at a different location. The benefit of this approach is efficient access to the first N elements. Access to elements on the external storage is as expensive as with "No Inlining". Furthermore, this strategy requires an additional conditional branch to determine whether an element is stored in the inline storage or on the external storage. This imposes almost no overhead on GPUs, which do generally not execute instructions speculatively. This layout can be implemented with a helper class like `absl::InlinedVector<T, N>`[1].

SOA without Inlining (Fig. 2 (right)) This layout stores objects as SOA (all values of a field together, sorted by object ID). This approach has a number of benefits: First, if not all fields are used all the time, it can improve cache utilization because those fields will not occupy cache entries. Second, it allows for efficient loads/stores from/into vector registers if objects with consecutive IDs are processed. Third, it allows for memory coalescing on GPUs if objects with consecutive IDs are processed within a warp. Finally, no memory is wasted for object padding; only the field arrays themselves must be aligned. With respect to inner arrays, the same advantages and disadvantages as in the first strategy apply. This layout is supported by Ikra as `inlined_array_(T, 0)`.

SOA with Full Inlining (Fig. 3 (right)) This layout treats every inner array slot as a separate field and allocates values for every array index in a separate SOA array. It provides opportunities for vectorized operations and memory coalescing not only for primitive fields but also when accessing inner array elements. This is possible if objects with consecutive IDs are processed in the same warp and inner array elements with the same indices are accessed. Many parallel graph algorithms [15] on GPUs do not benefit much from additional memory coalescing in this layout. Even though reading the pointers in an adjacency list can be coalesced, data reads/writes on neighboring vertices are still uncoalesced, because vertex IDs of neighbors are usually *random* and not consecutive. Similar to the second strategy, this layout provides more efficient array access without a pointer redirection but can waste memory. It is supported by Ikra as `fully_inlined_array_(T, N)`.

SOA with Partial Inlining (Fig. 4 (right)) Similar to the third strategy, this layout is a mixture between *no* and *full* inlining. The first N inner array elements can be accessed efficiently and may benefit from vectorized operations and memory coalescing. This layout is supported by Ikra as `inlined_array_(T, N)`.

SOA with Array as Object (Fig. 5) This layout treats inner arrays as normal C++ objects and does not perform any data layout transformations on them. There is *one* SOA array for every field, including inner arrays. This layout is useful if the total data size is small and all elements of an inner array are always used together. In that case, all inner arrays can fit into the cache and accessing the first inner array element will prefetch the next ones. Furthermore, if an application exhibits nested parallelism with respect to inner arrays, those accesses can be vectorized and coalesced. This layout is supported by Ikra as `field_(std::array<T, N>)`.

From an inlining perspective, this layout inlines the inner array fully. It could easily be adapted to *partial inlining* (`field_(absl:: InlinedVector<T, N>)`) or *without inlining* (`field_(std::vector <T>)`), but we do not analyze such layouts any further in this work.

2.2 Choosing a Layout Strategy

Programmers have a variety of layout strategies to choose from. Which strategy is best depends on the hardware architecture, the data access patterns of the application and the characteristics of the dataset. With Ikra-Cpp, programmers still have to take these factors into account, but switching between strategies is now much easier. As a rule of thumb, we suggest to start experimenting with a partial inlining size that ensures that 80% of all inner array elements are inlined.

[1] Part of the Abseil library. See https://github.com/abseil/abseil-cpp.

3 Design and Usage of Ikra-Cpp

To make the development of object-oriented GPU programs easier and more productive, we developed Ikra-Cpp, an embedded C++/CUDA DSL [22]. It focuses on data layout optimizations, including strategies for inner array inlining. In the next paragraph, we describe the frontier-based BFS algorithm for GPUs. This algorithm serves as an example when describing the API of Ikra-Cpp in Section 3.1. Finally, Section 3.2 describes the main implementation ideas behind Ikra-Cpp; for details, refer to our implementation paper [22].

Frontier-based Breadth-first Search BFS is a fundamental algorithm in graph processing. A variety of implementation strategies have been proposed for GPUs, some based on advanced techniques such as hierarchical queues [13] or virtual warp-centric programming [8]. The frontier-based BFS algorithm [16] is among the simplest ones and provides a reasonable speedup compared to CPU execution.

BFS computes the distance of every vertex from a designated start vertex. At first, the start vertex has distance zero and all other vertices have distance infinity. The algorithm now proceeds iteratively. In iteration i, all vertices with distance i (i.e., the *frontier*) are processed in parallel: For every vertex in the frontier, all of its neighbors are updated with distance $i + 1$, unless they already have a smaller distance value. The algorithm terminates if no updates are performed or, in this example[2], after a fixed number of iterations. BFS is an interesting example for Ikra-Cpp because the adjacency lists are arrays of different sizes.

3.1 Programming Interface

C++ classes with SOA layout must inherit from the special class `SoaLayout` (Fig. 6(a), Line 1). The first template argument is the class itself and the second one is the maximum number of objects of that class, a compile-time constant. Objects must be created with the `new` keyword. They cannot be stack-allocated or stored in local variables. They are always referred to with pointers or references. A SOA class must be initialized with two macros (Fig. 6(a), Lines 4, 17).

Field Types Fields of primitive type are declared with special *SOA field types* ending with an underscore (Fig. 6(a), Lines 15, 16). For example, an `int` variable is declared with `int_`. All other types are declared with `field_(type)`, where *type* is any C++ type. Finally, arrays are declared as follows.

- `fully_inlined_array_(`T`,` N`)`: Stores N elements of base type T fully inlined.
- `inlined_array_(`T`,` N`)`: Stores N elements of base type T partially inlined. The full size of the array must be specified during field initialization (Fig. 6(a), Line 5).

The ideal strategy for a given application depends heavily on its runtime data access patterns. Section 5 discusses guidelines for choosing a good strategy.

[2]Such a termination criteria can be implemented with Ikra-Cpp.

```
1  class Vertex : public SoaLayout<Vertex, kMaxNumVertices,
2      /* This arena can store 2 arr. elements per vertex on avg. */
3      StorageWithArena<kMaxNumVertices*2*sizeof(Vertex*)>> {
4   public: IKRA_INITIALIZE_CLASS
5    __host__ Vertex(int num_n) : neighbors_(num_neighbors),
6                                 num_neighbors_(num_neighbors) {}
7
8    __device__ void visit(int frontier) {
9      if (distance_ == frontier)
10       for (int i = 0; i < num_neighbors_; ++i)
11         neighbors_[i]->update(distance_ + 1);
12   }
13
14   __device__ void update(int new_distance) {
15     if (new_distance < distance_) distance_ = new_distance;
16   }
17
18   int_ distance_ = std::numeric_limits<int>::max();
19   int_ num_neighbors_; inlined_array_(Vertex*, 3) neighbors_;
20 }; IKRA_DEVICE_STORAGE(Vertex);
```

(a) `Vertex` class for BFS with frontiers. Regardless of the runtime size of a `neighbors_` array, 3 array slots are allocated (inlined) in the object. This number (*partial inlining size*) was chosen based on dataset characteristics, such that more than 80% of inner array elements are inlined. Additional elements are stored in an arena, whose size was calculated from dataset characteristics and the partial inlining size. The constructor has no `__device__` annotation because all objects are created from host code in this example.

```
1  void load_vertices_from_stream(std::fstream str) {
2    int num_neighbors, next;
3    while (str >> num_neighbors) { // read integer into variable
4      Vertex* vertex = new Vertex(num_neighbors);
5      for (int i = 0; i < num_neighbors; ++i) {
6        str >> next; Vertex* n = Vertex::get_uninitialized(next);
7        vertex->neighbors_[i] = n;
8      } } }
9  void run_bfs(int start_vertex) {
10   Vertex::get(start_vertex)->update(0);
11   for (int i = 0; i < 100; ++i) cuda_execute(&Vertex::visit, i);
12 }
```

(b) Creating new vertices on the host and running BFS on the device, assuming the input stream contains of a concatenation of adjacency lists (vertex IDs), preceeded by their lengths. For performance reasons, array elements should not be set one by one (Line 7), but transferred as an entire array.

Figure 6. Example: Breadth-first search on a directed graph in Ikra-Cpp.

Ikra-Cpp has *storage strategies* for allocating memory. A strategy is specified as the third template argument to `SoaLayout` (Fig. 6(a), Line 3). The default strategy (if no third argument is provided) is the *static storage strategy* which statically allocates a chunk of data. The benefit of this approach is that many address computations can be constant folded and are more efficient [11]. However, this also means that the maximum number of objects of a class is now a compile-time constant. If a partially inlined array is used, programmers must use the *static storage strategy with fixed-size arena*. This strategy allocates an arena of given size, within which external storage of additional inner array elements is allocated efficiently with bump pointer allocation.

Creating Objects In Ikra-Cpp, every object has an implicit ID, accessible with the function `id()` and starting from zero. New objects are created with the `new` keyword both on the host and on the device, regardless of where data is

stored physically, as long as the constructor has the required `__device__` or `__host__` qualifiers (Fig. 6(a), Line 5). In general, performance-critical code should run where the data is located; however, in our experiments it was often more convenient to run setup code (e.g., loading and parsing data from an external source, and creating the necessary objects) on the host. Ikra-Cpp takes care of memory transfers automatically.

The static function `get()` (Fig. 6(b), Line 10) returns a pointer to an object by ID. `get_uninitialized()` can be used to get a pointer to a not-yet-created object, which is useful during data imports (Fig. 6(b), Line 6).

Parallel Execution The Ikra-Cpp *executor API* can run member functions or create multiple objects in parallel on the GPU, without having to define or launch CUDA kernels explicitly. `cuda_execute` (Fig. 6(b), Line 11) runs a `__device__` member function in parallel for all objects of a class. Programmers can also restrict execution to an ID range or an array/vector of pointers. Multiple objects can be created in parallel with `cuda_construct`, Ikra-Cpp's parallel version of the C++ operator `new[]`. Since only one set of arguments can be provided for all new objects, constructors must use the `id()` function if objects should be initialized differently.

3.2 Implementation

Ikra-Cpp is implemented with advanced C++ techniques such as template metaprogramming and operator overloading. However, the fundamental idea behind its implementation is simple and centered around *fake pointers*: Pointers to SOA objects do not point to valid, allocated memory but encode object IDs. Fields such as `distance_` (Fig. 6(a), Line 15) are declared with special types like `int_`. When they are used in a context that requires an integer (e.g., Fig. 6(a), Line 10), C++ will convert the value from `int_` to `int`. This conversion procedure is overloaded statically by Ikra-Cpp (via the *implicit conversion operator*): The object ID is extracted from the `this` pointer and the actual memory location of that field value is calculated and accessed. After constant folding and function inlining, the generated binary code is as efficient and often identical to a hand-written SOA layout [22].

4 Example: Traffic Flow Simulation

Traffic flow simulations are important tools in transportation planning [14] and used to guide the design of city street networks. In this section, we present an agent-based microsimulation of single vehicles (*agents*) which move on a street network. It is based on the Nagel-Schreckenberg model [17], a simple cellular automata, which can reproduce real-world traffic phenomena [26, 27] such as traffic jams.

4.1 Simulation Model

In the Nagel-Schreckenberg model, a street (*link*) is divided into equally-sized cells, each of which can contain up to one agent. An agent can move onto a neighboring cell only if

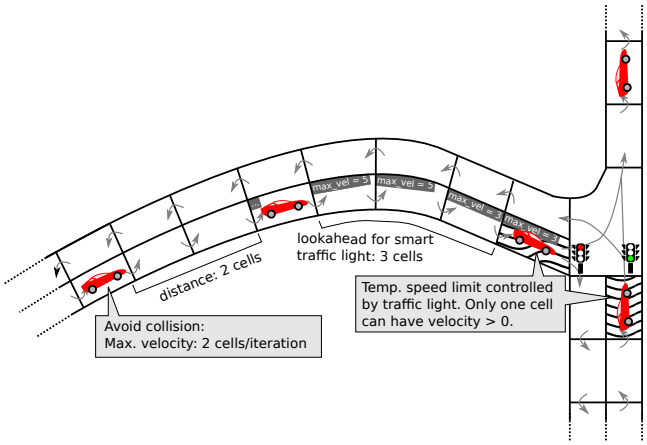


Figure 7. Example: Cells and their connections on a small street network with a single intersection. The intersection itself has no cell.

it is free. Every agent has a velocity, measured in cells per iteration. Both agents and cells have a maximum velocity; the latter one can be used to model speed limits on links. An iteration in our simulation consists of the following steps, each of which is executed for every agent a.

1. Increase the agent's velocity unless it is already driving at its maximum velocity: $v_a \leftarrow min(v_a + 1, v\_max_a)$
2. Determine the agent's path of movement of length v_a, i.e., the next v_a many cells it will pass through. A *navigation strategy* determines the next cell at an intersection with multiple outgoing cells. This simulation uses random walk, biased towards large streets.
3. Avoid collisions with other agents and enforce speed limits. To avoid collisions, reduce the speed v_a to the largest possible value such that the first v_a many cells on the calculated path are free. To follow speed limits, reduce v_a to the largest possible value, such that $v_a \leq v\_max_c$ for each cell c among the first v_a many cells on the calculated path.
4. Reduce the agent's velocity by one unit with a probability of 20% (*randomization*).
5. Move the agent from its current location by v_a many cells according to the calculated path.

Intersections and Traffic Lights Every cell has zero, one, or multiple outgoing cells, forming a directed graph. In the first case, the cell is a *sink*, i.e., a link leaves the simulation area. Agents entering a sink will be randomly redistributed. The second case is most common and represents a regular link cell. The third case appears at intersections, where a cell is connected to the first cell of every outgoing[3] link.

It is important to ensure that only one agent enters an outgoing link (i.e., its first cell) at an intersection in one iteration, even if multiple agents from different incoming links are waiting. To that end, a *traffic controller* can impose

[3]Two-way links consists of one incoming and one outgoing link.

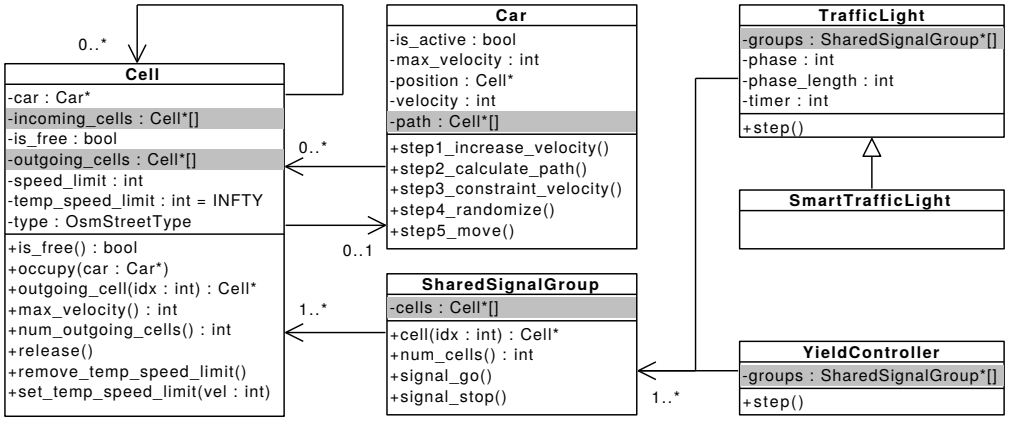


Figure 8. Simplified Architecture and Classes of Traffic Flow Simulation. For every array (highlighted in gray), an additional *array size* field is allocated.

```
for (int i = 0; i < 100; ++i) {
  cuda_execute(&SmartTrafficLight::step);
  cuda_execute(&YieldController::step);

  cuda_execute(&Car::step1_increase_velocity);
  cuda_execute(&Car::step2_extend_path);
  cuda_execute(&Car::step3_constraint_velocity);
  cuda_execute(&Car::step4_randomize);
  cuda_execute(&Car::step5_move);
}
```

Figure 9. Running the simulation for 100 iterations.

temporary speed limits on cells, e.g., a speed limit of zero, corresponding to a red light [3]. Traffic controllers set and remove speed limits for the last cells of all incoming links such that only one incoming link has a green light at a time[4]. We implemented three kinds of controllers.

- *Traffic lights* impose a temporary speed limit of zero on all but one incoming link. A green *phase* is scheduled round-robin for *phase length* many iterations among all incoming links.
- *Smart traffic lights* work like normal traffic lights but assign a green phase to an incoming link immediately if it is the only one with a waiting agent. Real traffic lights have sensors/cameras to provide such behavior. All traffic lights in the benchmark section are smart.
- *Yield controllers* model yield traffic signs, which are often found at the end of merge lanes of highway entrances. Given n incoming links, a temporary speed limit of zero is assigned to all links $i > s$ if link s has an agent, i.e., incoming links/cells in the controller should be ordered by priority.

The last two controllers are implemented such that traffic does not have to stop/slow down in front of an intersection, by checking all cells from which an agent could cross an intersection in one iteration (not only the closest incoming cell), as indicated by the maximum allowed speed limit on a link (*lookahead*). This is done with graph traversal on back edges (*incoming cells*), which terminates when an agent was found within lookahead range.

Parallelization The Nagel-Schreckenberg model is suitable for parallel execution on GPUs [10] because it does not require any special parallel data structures such as queues or arrays with concurrent access. Parallelism is expressed in terms of agents and traffic controllers. The first four steps are read-only on cells and serve as a *preparation phase*, building up a temporary data structure (path, speed limit) within agents. The final step starts when the previous ones were completed for all agents and writes updates to cells. A cell is never updated by multiple agents within an iteration.

4.2 Implementation in Ikra-Cpp

This simulation has a class for every real-world entity (Fig. 8). *Shared signal groups* impose temporary speed limits (zero = *stop signal*/red light or infinity = *go signal*/green light) on a group of cells. They are used to implement turn lanes at intersections; all turn lanes of an incoming link have the same signal. Furthermore, they are useful for more realistic setups, where more than one link has a go signal. Real-world street networks can be imported from OpenStreetMap (OSM) dumps in GraphML file format. Such dumps include link properties such as position, shape, connections to other streets, speed limits and OSM street type (e.g., "highway").

Arrays The classes in this simulation contain 6 arrays and they are used as follows.

- `Car::path`: Cleared at the beginning of an iteration, filled sequentially in Step 2, read sequentially in Step 3 and Step 5 (loops may *break* and not read/write the entire array).
- `Cell::outgoing_cells`: Read sequentially in Step 2 to determine the next `Car::velocity` many cells. The entire array and pointed-to cells are read to calculate

[4]This simple model can easily be extended to a more realistic one where agents can enter an intersection from multiple incoming links.

probabilities of biased random walk, favoring large links according to OSM link type.
- `Cell::incoming_cells`: Read during graph traversal of yield controllers and smart traffic lights, for every cell (and reachable cells up to a max. distance/lookahead) within every signal group of that controller. Traversal stops when an agent was found. Traffic controllers traverse the graph with recursive DFS[5].
- `TrafficLight/YieldController::groups`: Read sequentially in the respective class's `step` function, but the loop may break early. See `Cell::incoming_cells`.
- `SharedSignalGroup::cells`: Read sequentially during graph traversal. See `Cell::incoming_cells`.

5 Performance Evaluation

All experiments were performed on a desktop machine with an Intel i7-5960X CPU (8x 3.00 GHz), 32 GB main memory, an Nvidia GeForce GTX 980 GPU (4 GB memory), Ubuntu 16.04, and the *nvcc* compiler from the CUDA Toolkit 9.1. We focus on GPU execution, but similar effects can be observed on CPUs with a auto-vectorizing compiler. Our previous work has shown that the performance overhead of Ikra-Cpp compared to a handwritten SOA layout is negligible [22].

5.1 Synthetic Benchmark

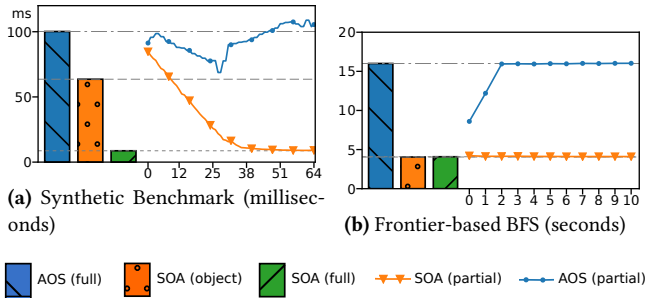
To isolate the effect of array inlining, we created a synthetic benchmark with a test class containing one `int` data field, one `int` array and one `int` field storing the array size. The array has between 32 and 64 elements (chosen randomly). The benchmark adds the data field value to all array elements in a loop, i.e., all inner array elements are read and written.

Fig. 10(a) shows the running time for 262,144 test objects. The "Array as Object" SOA version is more than 30% faster than the AOS version. The fully inlined SOA version is an order of magnitude faster than the AOS version. Partial array inlining sizes larger than 32 do not improve the performance in SOA mode anymore, because inner array accesses are unlikely to be coalesced from that point.

5.2 Breadth-first Search

Fig. 10(b) shows the running time for the frontier-based BFS algorithm with different layout strategies on the Pennsylvania road network (1,088,092 vertices, 3,083,796 directed edges, avg. degree 2.83) [12]. The graph clearly shows the benefit of SOA over AOS. The performance of AOS degrades with a growing inlining size, because more cache entries for non-existing *neighbors* array slots are wasted. BFS does not benefit from memory coalescing when inlining inner arrays in SOA mode (compare "SOA (object)" and "SOA (full)"), because the neighbors accessed in Fig. 6(a), Line 10 have "random" IDs; for memory coalescing, consecutive ID ranges for any given array index are required among vertices within a warp. Nested parallelism can speed up such algorithms [8] and would benefit from an *Array as Object* layout.

[5]BFS with nested parallelism should be used for better performance.



(a) Synthetic Benchmark (milliseconds) **(b)** Frontier-based BFS (seconds)

Figure 10. Running time for synthetic benchmark and BFS. The x-axis shows different inner array layout strategies and inlining sizes.

5.3 Traffic Flow Simulation

We ran the traffic flow simulation with 229,376 agents on a real-world OSM street networks of the Denver-Aurora urbanized area [1] (8,339,802 cells, 62,819 smart traffic lights, 4,792 yield controllers, 205,129 shared signal groups). Maximum cell velocities are $\mu = 6.16$ on average ($\sigma = 1.06$). After 1000 iterations, agents had an average velocity of $\mu = 1.57$ ($\sigma = 2.56$). Fig. 11 shows the running times and device memory requirements for AOS/SOA mode and various inner array inlining strategies. In every subfigure, various inner array layout strategies for a single field are evaluated; all other arrays are fully inlined.

AOS vs. SOA The benefit of a SOA layout over an AOS layout is clearly visible in this benchmark. Regardless of which inner array inlining strategy is chosen, the running time spent on processing agents is always significantly smaller in SOA (any subfigure, upper chart, e.g., first bar vs. second bar). We can observe a similar behavior for traffic controllers.

Inner Array Inlining in SOA The benefit of array inlining can be observed best in SubFig. (c) when comparing the running time of a fully inlined `Car::path` (third bar) with one that is stored "as object". The fully inlined version is 15% faster. This is because our implementation of the Nagel-Schreckenberg algorithm iterates over the path array in every thread; first to precompute a path, then to adjust the agent's velocity. Because all agents are processed in order (i.e., thread i processes agent with ID i), memory accesses are coalesced. Furthermore, a slightly better speedup can be achieved with partial array inlining, starting from an inlining size of 6, because few agents (<20%) have a velocity higher than 6. In that case, accesses to `Car::path` at indices higher than 6 are unlikely to be coalesced because very few threads access these array slots; we believe that the additional speedup is due to better cache utilization and prefetching: A cache line in the external arena storage contains multiple elements of the same inner array.

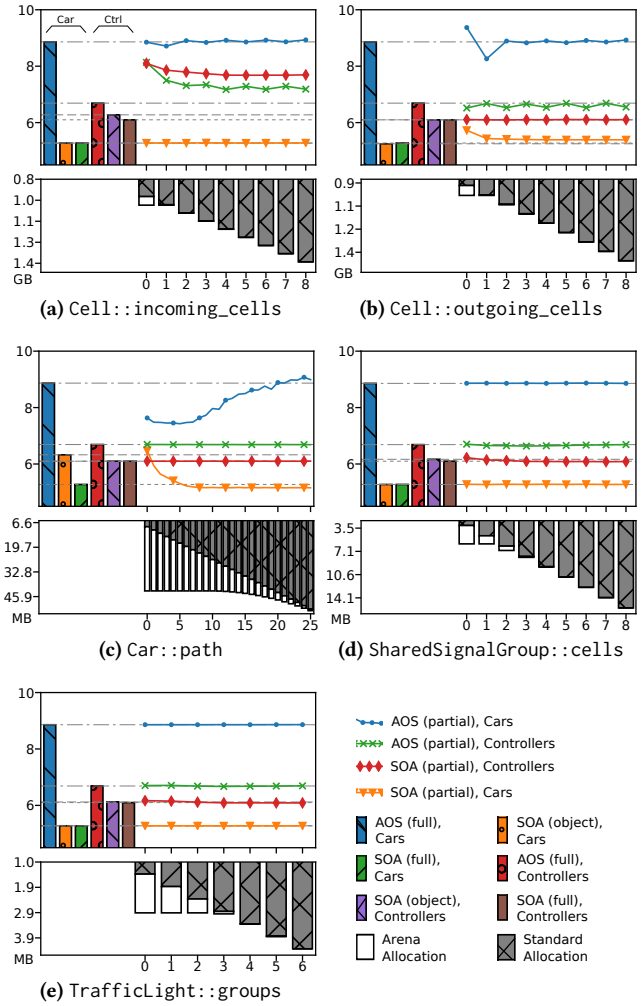


Figure 11. Running Time and Memory Requirement for Traffic Simulation. The upper part of every subfigure shows the running time (seconds, y-axis) in AOS/SOA data layout with various inner array inlining strategies and inlining sizes (x-axis). The lower part shows the memory requirement in AOS data layout (GB or MB, y-axis), grouped by arena/standard allocation.

Cells (Subfig. (a), (b)) do not benefit much from inner array inlining when comparing running times. This is because parallelization in this program is never expressed in terms of cells, but always in terms of agents and traffic controllers. Those entities access fields and inner array elements of cells at random, because they contain pointers to "random" `Cell` objects. Consequently, the CUDA threads within a warp access data from cells with totally different IDs, i.e., the data locality required for memory coalescing is missing.

Memory Footprint The lower part of every subgraph shows the memory requirement for all objects of the respective class. The white part represents inlined allocation (e.g., regular fields or inlined data of arrays). The darker part represents external storage (arena allocation). Subfig. (a) and (b) show that the vast amount of cell data is used on regular fields (e.g., speed limits, array sizes, etc.) and only around 60 MB each are required for incoming and outgoing cell arrays (see arena allocation for inlining size 0). If more than one array element is inlined, the memory footprint increases gradually, because 95% of all cells have only one neighbor.

The memory footprint for agents increases for inlining sizes above 15, because all agents have a maximum speed limit of at least 15; an array of size at least 15 must be allocated for every agent. An inlining size of 15 results in a good tradeoff between memory footprint and performance; no performance is lost compared to a fully inlined inner array.

Other Observations Compared to fully inlined inner arrays, accessing elements on partially inlined inner arrays requires an additional `if` check for checking if an element is located within the external storage. This overhead is negligible on GPUs (e.g., compare "SOA (partial), Cars" at a high inlining size with "SOA (full), Cars" (third bar)). We attribute this to the fact that GPUs do not execute speculatively, i.e., no performance can be lost by mispredicting a branch target.

In AOS mode, the performance of agent computation decreases with a higher inlining size (SubFig. (c), "AOS (partial), Cars"). This is because the likelihood of a path element being brought into the cache together with another field/element (prefetching/cache line) increases, even though that element might never be accessed (few cars have a velocity > 6).

6 Related Work

SOA is a well-studied data layout technique for CPUs and GPUs and a best practice for GPU programmers [6]. C++ libraries like SoAx [7] and ASX [25] allow programmers to use SOA as a data layout strategy while maintaining an AOS-like programming style. The Intel SPMD Program Compiler [20] extends the C programming language with additional keywords for SOA and minimizes notation/API overhead. Moreover, previous work has investigated how to apply layouts like SOA or AoSoA (*Array of Structure of Arrays*) automatically, without programmer intervention [9]. Furthermore, there has been work on applying such layouts to complex structures with multi-dimensional arrays [28]. The focus of Ikra-Cpp is on object-oriented programming (i.e., classes, member functions, constructors, etc.); C++ features that are not supported by such libraries/compilers. Complex structures like AoSoA or multi-dimensional arrays could be supported in Ikra-Cpp in the future.

Object inlining has been proposed for Java-like object-oriented languages for better cache performance and reducing overheads due to allocation and pointer indirections [2]. Later work applied the idea of data inlining to arrays, which simplifies address arithmetics of array accesses and eliminates load instructions in the assembly [29]. Ikra-Cpp applies the same idea to arrays in SOA layout, under simplified assumptions: Arrays are of fixed size and inlining is controlled manually by the programmer.

7 Summary

In this work, we presented an overview of various data layout strategies for inner arrays in a Structure of Arrays layout. Such arrays can be split and regrouped by array index (*inlined* into the SOA layout) to take advantage of SIMD speedups through vectorized instructions or memory coalescing. Since writing and maintaining such low-level code is tedious, we extended Ikra-Cpp, a C++/CUDA DSL for high-performance object-oriented programming on CPUs and GPUs, with additional types that lay out inner arrays in a more SIMD-friendly format. In the future, we will focus on control flow optimizations for SIMD programs such as nested parallelism or control flow divergence avoidance mechanisms.

Acknowledgments

We would like to thank Toyotaro Suzumura from IBM Research for valuable discussions about traffic simulations and our implementation. This work was supported by a JSPS Research Fellowship for Young Scientists and JSPS KAKENHI Grant Number 18J14726.

References

[1] G. Boeing. 2017. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems* 65 (2017), 126–139.

[2] J. Dolby and A. Chien. 2000. An Automatic Object Inlining Optimization and Its Evaluation *(PLDI '00)*. ACM, 345–357.

[3] J. Esser and M. Schreckenberg. 1997. Microscopic Simulation of Urban Traffic Based on Cellular Automata. *Int. J. Mod. Phys. C* 08, 05 (1997), 1025–1036.

[4] N. Faria, R. Silva, and J. L. Sobral. 2013. Impact of Data Structure Layout on Performance. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 116–120.

[5] P. Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing – HiPC 2007*. Springer Berlin Heidelberg, 197–208.

[6] M. Harris. 2007. Optimizing CUDA. (2007). Supercomputing Tutorial.

[7] H. Homann and F. Laenen. 2018. SoAx: A generic C++ Structure of Arrays for handling particles in HPC codes. *Comput. Phys. Commun.* 224 (2018), 325–332.

[8] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. 2011. Accelerating CUDA Graph Algorithms at Maximum Warp *(PPoPP '11)*. ACM, 267–276.

[9] K. Kofler, B. Cosenza, and T. Fahringer. 2015. Automatic Data Layout Optimizations for GPUs *(Euro-Par 2015)*. Springer, 263–274.

[10] P. Korček, L. Sekanina, and O. Fučík. 2011. Cellular automata based traffic simulation accelerated on GPU *(MENDEL2011)*. Institute of Automation and Computer Science FME BUT, 395–402.

[11] A. S. D. Lee and T. S. Abdelrahman. 2017. Launch-Time Optimization of OpenCL GPU Kernels. In *Proceedings of the General Purpose GPUs (GPGPU-10)*. ACM, 32–41.

[12] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. 2008. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *CoRR* (Oct. 2008).

[13] L. Luo, M. Wong, and W. Hwu. 2010. An Effective GPU Implementation of Breadth-first Search *(DAC '10)*. ACM, 52–55.

[14] S. Maerivoet and B. De Moor. 2005. Transportation Planning and Traffic Flow Models. *ArXiv Physics e-prints* (July 2005).

[15] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing *(SIGMOD '10)*. ACM, 135–146.

[16] D. Merrill, M. Garland, and A. Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *ACM Trans. Parallel Comput.* 1, 2, Article 14 (Feb. 2015), 30 pages.

[17] K. Nagel and M. Schreckenberg. 1992. A cellular automaton model for freeway traffic. *J. Phys. I France* 2, 12 (1992), 2221–2229.

[18] J. Nickolls, I. Buck, M. Garland, and K. Skadron. 2008. Scalable Parallel Programming with CUDA. *Queue* 6, 2 (March 2008), 40–53.

[19] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. 2008. GPU Computing. *Proc. IEEE* 96, 5 (May 2008), 879–899.

[20] M. Pharr and W. R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing*. IEEE, 1–13.

[21] V. K. Proulx. 1998. Traffic Simulation: A Case Study for Teaching Object Oriented Design *(SIGCSE '98)*. ACM, 48–52.

[22] M. Springer and H. Masuhara. 2018. Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming with Structure-of-Arrays Layout *(WPMVP '18)*. ACM, Article 6, 9 pages.

[23] D. Strippgen and K. Nagel. 2009. Multi-agent traffic simulation with CUDA *(HPCS '09)*. IEEE, 106–114.

[24] D. Strippgen and K. Nagel. 2009. Using Common Graphics Hardware for Multi-agent Traffic Simulation with CUDA *(Simutools '09)*. ICST, Article 62, 8 pages.

[25] R. Strzodka. 2012. Ch. 31 - Abstraction for AoS and SoA Layout in C++. In *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 429–441.

[26] J. Wahle, J. Esser, L. Neubert, and M. Schreckenberg. 1998. A Cellular Automaton Traffic Flow Model for Online-Simulation of Urban Traffic. In *Cellular Automata: Research Towards Industry*. Springer, 185–193.

[27] J. Wahle, L. Neubert, J. Esser, and M. Schreckenberg. 2001. A cellular automaton traffic flow model for online simulation of traffic. *Parallel Comput.* 27, 5 (2001), 719–735. Cellular automata: From modeling to applications.

[28] N. Weber and M. Goesele. 2014. Auto-tuning Complex Array Layouts for GPUs *(PGV '14)*. Eurographics Association, 57–64.

[29] C. Wimmer and H. Mössenböck. 2008. Automatic Array Inlining in Java Virtual Machines *(CGO '08)*. ACM, 14–23.