東京工業大学
Tokyo Institute of Technology

Memory-Efficient Object-Oriented Programming on GPUs

Matthias Springer

Doctoral Thesis Defense, 07/08/2019

Introduction

- *Larger goal:* Making GPU programming easier for developers from other domains (non-GPU experts)
- *In particular:* **Object-oriented programming (OOP) on GPUs**
  - OOP has many benefits: Abstraction, expressiveness, modularity, developer productivity, …
  - But it is **avoided** in high-performance computing (HPC) due to **bad performance.**
- *Goal of this thesis:* Making **fast OOP** available on SIMD arch./**GPUs**
  - Why is OOP slow on GPUs? Focusing on **memory access performance**.
  - Developing a simple object-oriented **programming model** for GPUs: **SMMO**
  - **Optimizing the memory access** of SMMO application with a new CUDA framework.

Thesis Overview and Prototypes

Ikra-Ruby

- Ruby Library with Ruby → CUDA Compiler
- Array-based GPU Programming
- Parallel Array Interface (Sec. 3.1)

  ```
  peach, pmap, pnew, preduce,
  pstencil, pzip, with_index,
  to_command
  ```

- Kernel Fusion through Type Inference (Sec. 4.1)

```
(1..100).pmap do |i| i * i end
```

Background

- GPU Architecture: SIMD (Sec. 2.1)
- Structure of Arrays Data Layout (Sec. 4.2)

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

Ikra-Cpp

- C++/CUDA Framework for OOP on GPUs
- Single-Method Multiple-Objects (Sec. 3.2, Sec. 7)
- Only Two Operations: Parallel Do-all, Parallel New

  ```
  parallel_do<T, &T::func>()
  parallel_new<T>
  ```

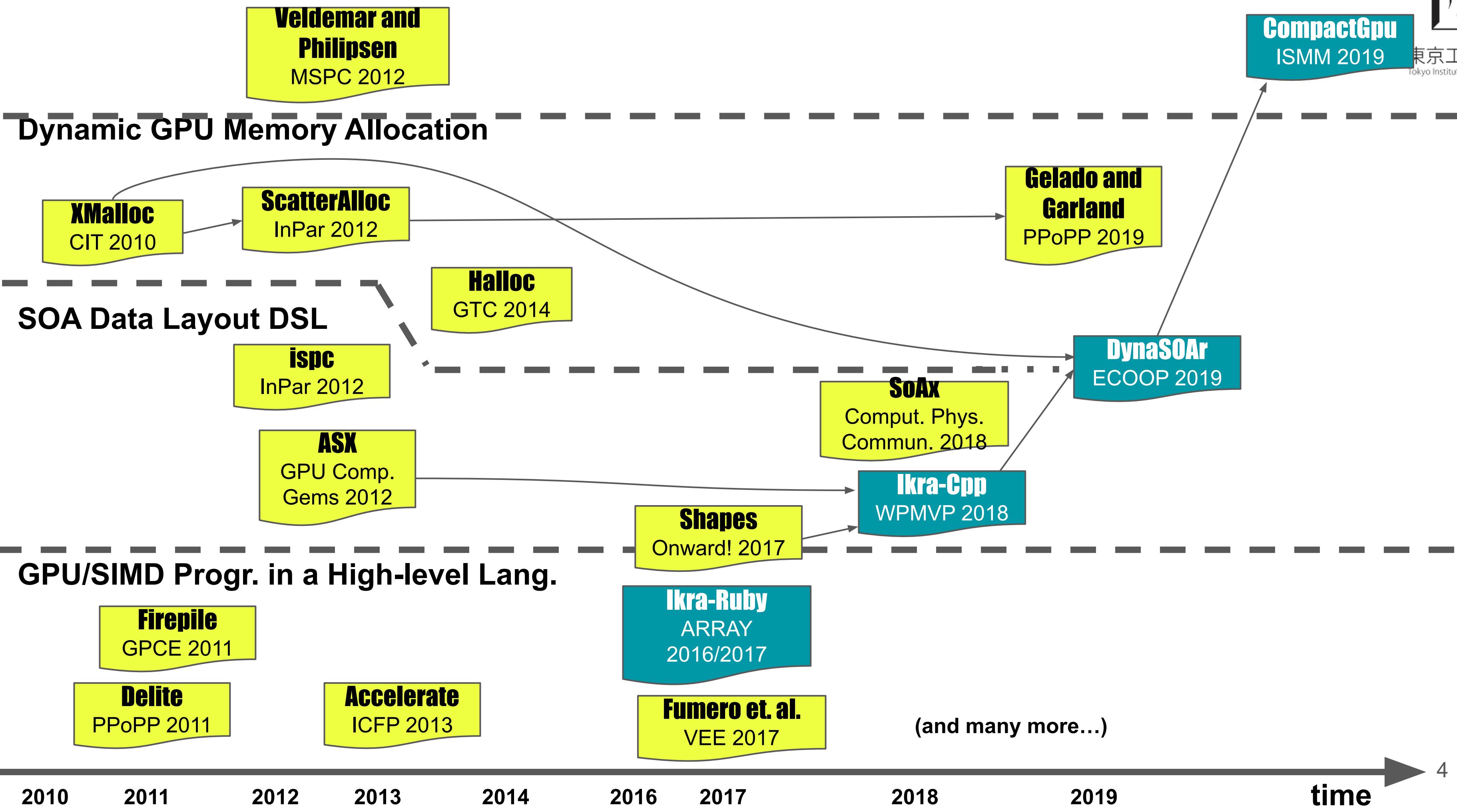
- Structure of Arrays (SOA) Data Layout DSL (Sec. 4.3)
- SOA Extension for Inner Arrays (Sec. 4.4)

DynaSOAr

- Dynamic Memory Allocator for GPUs (Sec. 5)
- Custom Object Layout with SOA Performance
- Uses Lock-free Hierarchical Bitmaps (Sec. 5.3.1)

CompactGpu

- GPU Global Memory Defragmentation (Sec. 6)
- Improving the Efficiency of Vectorized Access



GPU Memory Defragmentation

Veldemar and Philipsen
MSPC 2012

CompactGpu
ISMM 2019

Dynamic GPU Memory Allocation

XMalloc
CIT 2010

ScatterAlloc
InPar 2012

Gelado and Garland
PPoPP 2019

Halloc
GTC 2014

SOA Data Layout DSL

ispc
InPar 2012

ASX
GPU Comp. Gems 2012

SoAx
Comput. Phys. Commun. 2018

DynaSOAr
ECOOP 2019

Ikra-Cpp
WPMVP 2018

Shapes
Onward! 2017

GPU/SIMD Progr. in a High-level Lang.

Firepile
GPCE 2011

Delite
PPoPP 2011

Accelerate
ICFP 2013

Ikra-Ruby
ARRAY 2016/2017

Fumero et. al.
VEE 2017

(and many more…)

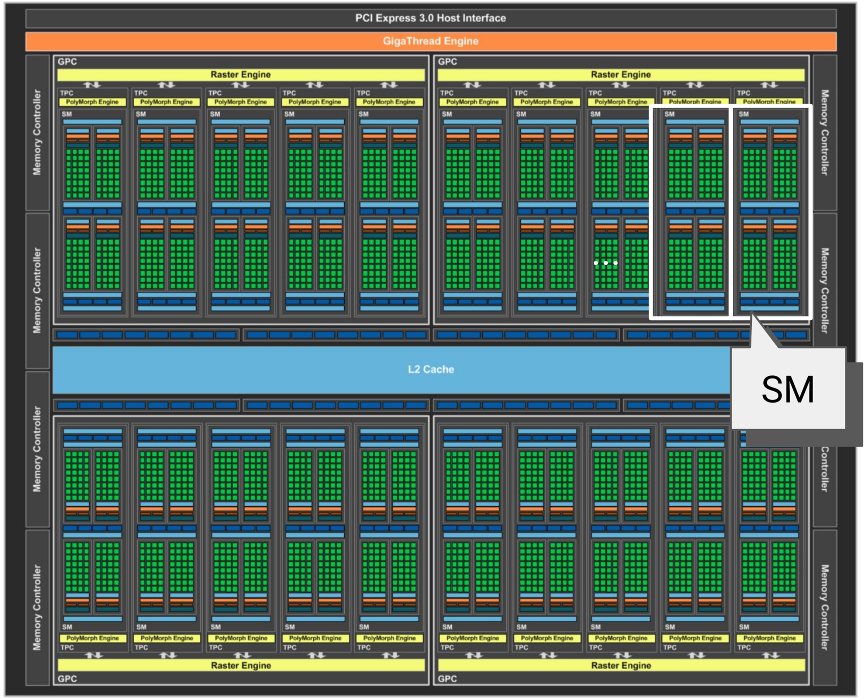
Background

Background: GPU Architecture

- NVIDIA GP104 (GeForce GTX 1080)
- 20 streaming multiprocessors (SMs)
- 128 CUDA cores per SM
- *Total:* 20 * 128 = 2560 CUDA cores

- 8 GB **device memory**
- L1 per SM, shared L2 cache

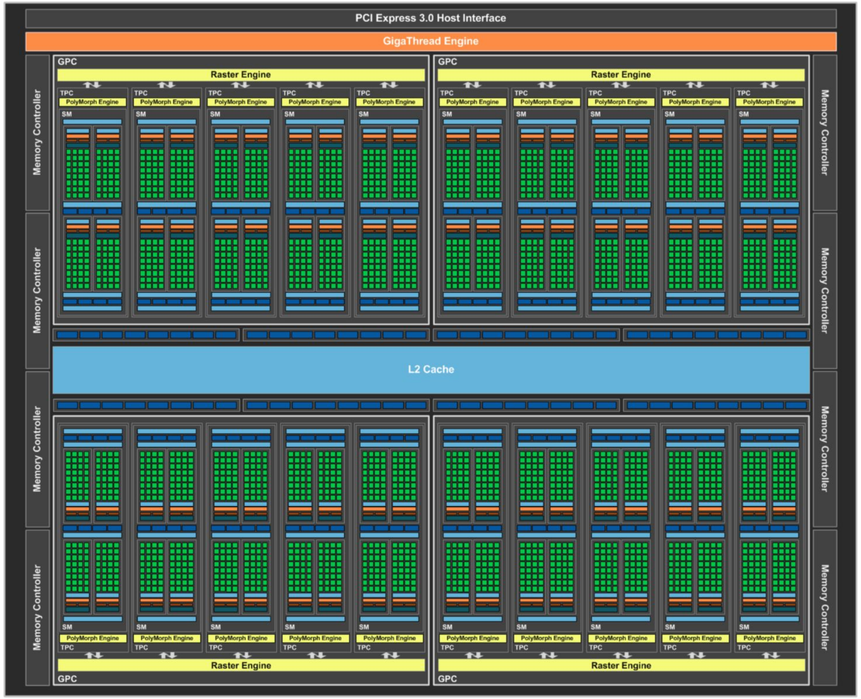
Source: NVIDIA GeForce GTX 1080 whitepaper



Background: GPU Architecture

- NVIDIA GP104 (GeForce GTX 1080)
- 20 streaming multiprocessors (SMs)
- ~~128 CUDA cores per SM~~
- ~~*Total:* 20 * 128 = 2560 CUDA cores~~
- 4 **physical** cores per SM
- *Total:* 20 * 4 = 80 cores
- Each core operates on 128-byte vector registers (32 scalars)
- 8 GB **device memory**
- L1 per SM, shared L2 cache

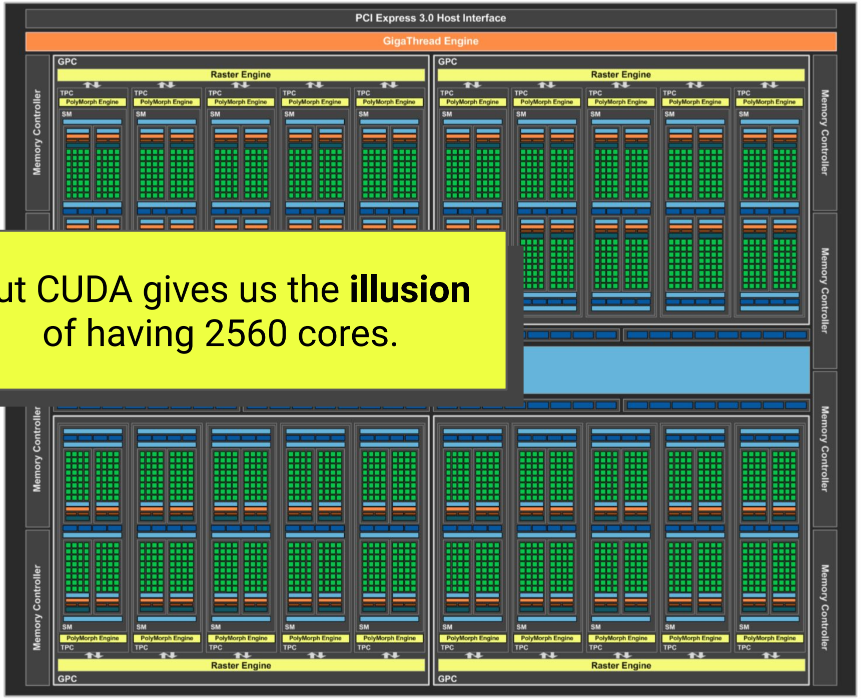
Source: NVIDIA GeForce GTX 1080 whitepaper



Background: GPU Architecture

- NVIDIA GP104 (GeForce GTX 1080)
- 20 streaming multiprocessors (SMs)
- ~~128 CUDA cores per SM~~
- ~~*Total:* 20 * 128 = 2560 CUDA cores~~
- 4 **physical** cores per SM
- *Total:* 20 * 4 = 80 cores
- Each core operates on 128-byte vector registers (32 scalars)
- 8 GB **device memory**
- L1 per SM, shared L2 cache

Source: NVIDIA GeForce GTX 1080 whitepaper



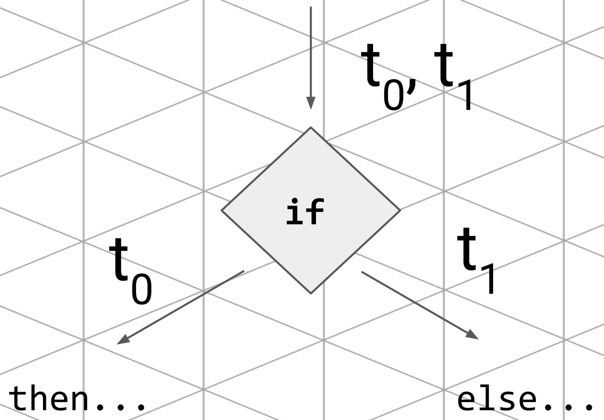
But CUDA gives us the **illusion** of having 2560 cores.

Handout only: Parallelism on GPUs / CUDA

- Thread-level Parallelism: 2560 CUDA cores
  - *SIMD:* Every 32 consecutive cores (**warp**; tid. i*32 … (i+1)*32 - 1) have the same control flow. (Because it is really only one core.)
  - *MIMD:* Every warp has its own control flow.
- Instruction-level Parallelism
  - Sometimes, a core can run more than just one instruction at a time…
  - Not relevant for this work

Handout only: Performance Problems on GPUs

- Non-uniform Control Flow
  - This happens when programmers assume they can program a GPU like a CPU…
  - If the control flow diverges within a warp, both paths are executed sequentially.



Performance Problems on GPUs

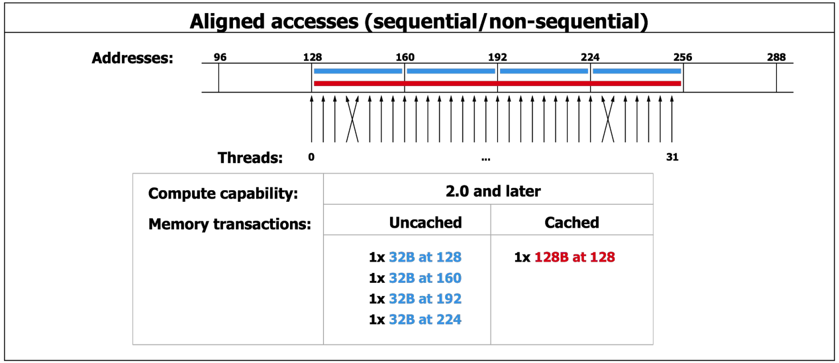
- Device (Global) Memory Access
  - The GPU memory controller is bad at accessing small memory blocks
  - *Simplified view:* The memory controller always accesses **128-byte blocks** (L1/L2 cache line size)

If the programmer **loads 4 bytes**, then the mem. controller loads 128 bytes and **throws 124 bytes away**

  - *Memory coalescing:* The memory controller can **coalesce** (combine) requests that are on the same L1/L2 cache line on a per-warp basis (threads t_{tid} with tid $\in [32*i; 32*(i+1))$).
  - *In different words:* A physical core always accesses memory in aligned, 128-byte blocks.
  - *Rule of thumb:* Threads in a warp should access spatially local memory addresses

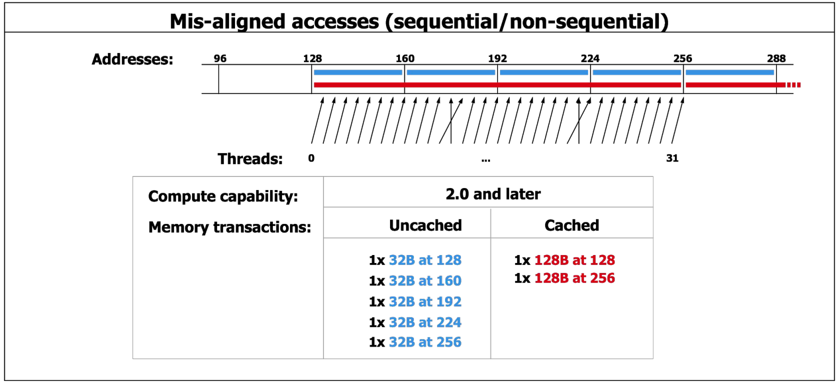
Performance Problems on GPUs

Source:
CUDA C Programming Guide



Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

good coalescing

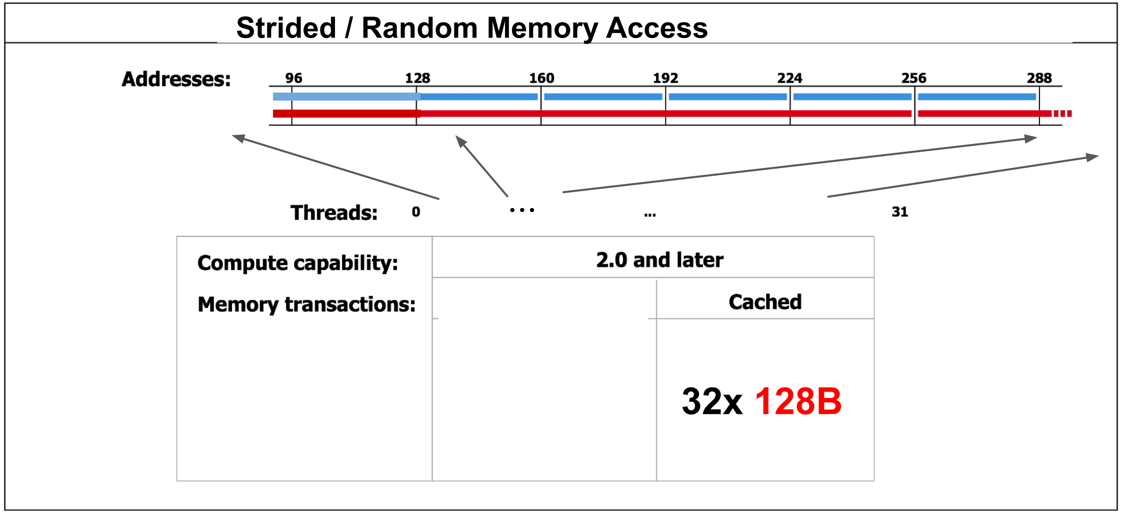


Compute capability:	2.0 and later	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

bad coalescing

Performance Problems on GPUs

Source:
CUDA C Programming Guide



Strided / Random Memory Access

Compute capability:	2.0 and later	
Memory transactions:		Cached
		32x 128B

no coalescing

Problems with OOP on GPUs

Common Belief: OOP is Slow

Object-oriented programming is too slow for high-performance computing.

“ One of the main issues of scientific computing is performance. [...] Object oriented programming is **observed slower** than functional programming. [P. Patel, M.Sc. Thesis, Univ. of Edinburgh, 2006]

The object-oriented programming (OOP) paradigm offers a solution to express reusable algorithms and abstractions through abstract data types and inheritance. However, [...] manipulating abstractions usually results in a run-time overhead. **We cannot afford this loss of performance** since efficiency is a crucial issue in scientific computing. [N. Burrus, et. al. MPOOL 2003]

While object-oriented programming is being embraced in industry [...], its acceptance by the parallel scientific programming community is still tentative. In this latter domain performance is invariably of paramount importance, where even C++ is considered suspect, primarily because of **real or perceived loss of performance**. [K. Davis, et. al. ECOOP 2008 Workshop Reader] ”

Common Belief: OOP is Slow

Object-oriented programming is too slow for high-performance computing.

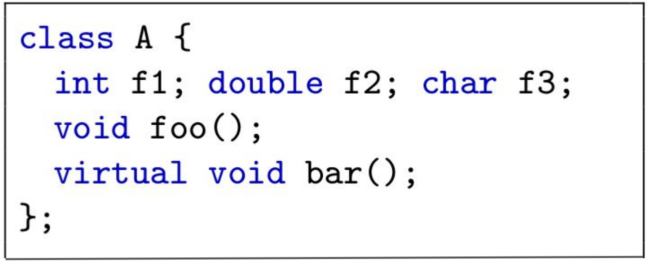
“ One of the main issues of scientific computi is perf [..] riented programming is **observed slower** than func f Edinburgh, 2006]

The object-oriented pr ble algorithms and abstractions thro abstractions usually results in a run- ency is a crucial issue in scientifi

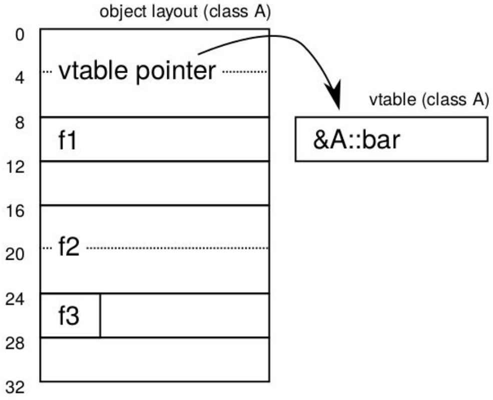
While object-orien ceptance by the parallel scientific programming erformance is invariably of paramount importance, whe ct, primarily because of **real or perceived loss of performance**. [K. Davis, et. al. ECOOP 20 orkshop Reader] ”

Let us identify the reasons **why OOP is slow** in HPC (esp. GPUs) and see if we can **optimize these performance problems**.

Problem with OOP on GPUs



```
class A {
  int f1; double f2; char f3;
  void foo();
  virtual void bar();
};
```

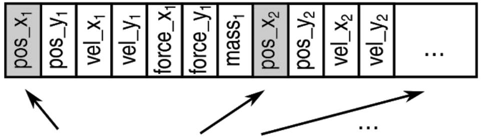


- **Data Layout:** Most languages/compilers (esp. C++/CUDA) do not allow programmers to **customize the layout of objects** in memory.

Structure of Arrays (SOA) Data Layout

(a) Array of Structures (AOS)

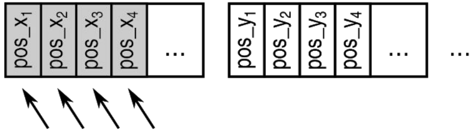
```
struct Body {
  float pos_x, pos_y;
  float vel_x, vel_y;
  float force_x, force_y;
  float mass;
};
Body bodies[32000];
```



strided memory access (slow)

(b) Structure of Arrays (SOA)

```
float Body_pos_x[32000];
float Body_pos_y[32000];
float Body_vel_x[32000];
float Body_vel_y[32000];
float Body_force_x[32000];
float Body_force_y[32000];
float Body_mass[32000];
```



vector load possible (fast)

(c) SOA Code Example

```
__device__ void move(int id) {
    /* Compute force, vel ... */

    pos_x[id] += kDt * vel_x[id];
```

SIMD: All threads (in a warp) perform this load in parallel. Current NVIDIA GPU coalesce these loads into as few 128-byte vector loads as possible. In SOA, fewer vector loads are required to cover all pos_x values than in AOS.

```
    pos_y[id] += kDt * vel_y[id];
}
```

This is no longer OOP.

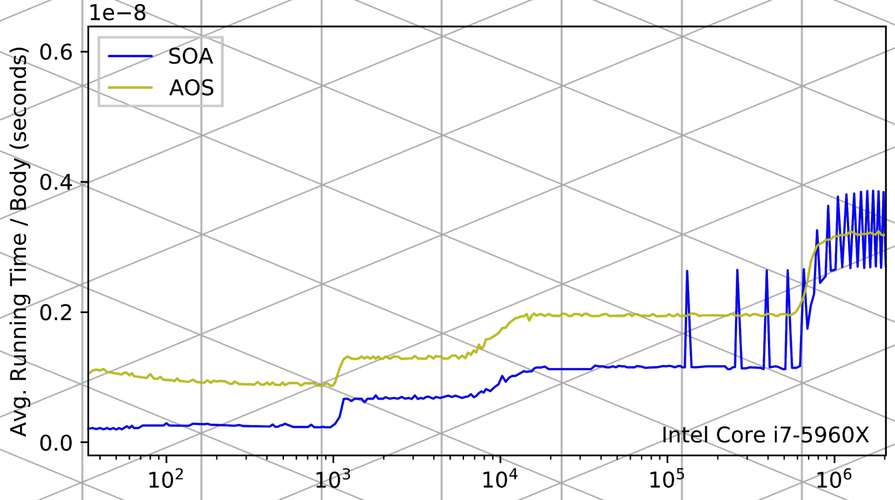
- *AOS:* Standard layout of most compilers/systems
- *SOA:* Best practice for SIMD/GPU programmers
- *[C++] Choose one:* SOA or OOP. **We want to have both!**

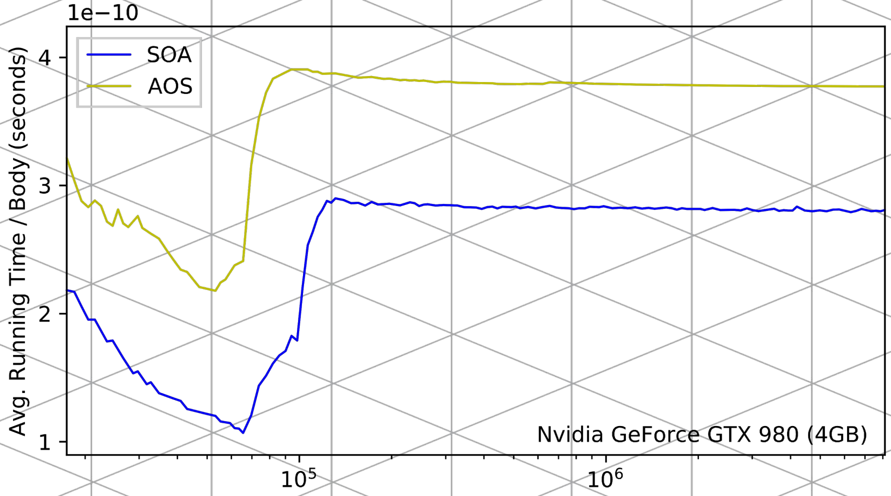
Handout only: Benefits/Disadvantages of SOA

- Benefits of SOA
  - Suitable for vector loads/stores → Good **memory coalescing** on GPUs
    (Only if the program accesses consecutive values at the same time.)
  - Can benefit **L1/L2 cache utilization**: Unused fields do not occupy cache lines.
  - Sometimes **lower memory footprint**: Only SOA arrays must be aligned, not every object.
- Disadvantages of SOA
  - Code is hard to read; **breaking language abstractions** if there is no support for custom object layouts in the programming language (e.g., C++).
- There are experimental languages with customizable data layout, but they have poor GPU support. E.g.: Shapes [1], ispc [2]

[1] J. Franco, et. al. You Can Have It All: Abstraction and Good Cache Performance. Onward! 2017.
[2] M. Pharr, et. al. ispc: A SPMD compiler for high-performance CPU programming. InPar 2012.

Handout only: N-body Perf. with AOS/SOA





(lower is better)

Much better performance with SOA!

Problem with OOP on GPUs

- **Data Layout:** Most languages/compilers (esp. C++/CUDA) do not allow programmers to **customize the layout of objects** in memory.
- **Dynamic Memory Management:** It is supported, but **slow**.

```
Body* b = new Body();
delete b;
```

Problem with OOP on GPUs

- **Data Layout:** Most languages/compilers (esp. C++/CUDA) do not allow programmers to **customize the layout of objects** in memory.
- **Dynamic Memory Management:** It is supported, but **slow**.

```
Body* b = new Body();
delete b;
```

> “Allocating memory dynamically in the kernel can be tempting because it allows GPU code to look more like CPU code. But it can seriously affect performance. [...] The kernel runs in 1500ms when using `__device__ malloc()` and 27ms when using pre-allocated memory. In other words, the test **takes 56x longer to run when memory is allocated dynamically** within the kernel.”
>
> https://stackoverflow.com/questions/13480213/how-to-dynamically-allocate-arrays-inside-a-kernel/13485322#13485322

Problem with OOP on GPUs

- **Data Layout:** Most languages/compilers (esp. C++/CUDA) do not allow programmers to **customize the layout of objects** in memory.
- **Dynamic Memory Management:** It is supported, but **slow**.

THIS THESIS

- **Virtual Function Calls:** Regular calls are by a factor of 10x faster due to inlining. In addition, virt. function calls can cause warp divergence.
- **64-bit Pointers:** Objects are referred to with 64-bit pointers. This can increase the size of objects, compared to 32-bit integers.

Problem with OOP on GPUs

- **Data Layout:** Most languages/compilers (esp. C++/CUDA) do not allow programmers to **customize the layout of objects** in memory.
- **Dynamic Memory Management:**

THIS THESIS

Switch-case statements or instrumentation-based techniques [2]

- **Virtual Function Calls:** Regular calls are by a factor of 10x faster due to inlining. In addition, virt. function calls can cause warp divergence.
- **64-bit Pointers:** Objects are referred to with 64-bit pointers. This can increase the size of objects, compared to 32-bit integers.

Pointer compression [1]

[1] K. Venstermans, et. al. Object-Relative Addressing: Compressed Pointers in 64-Bit Java Virtual Machines. ECOOP 2007.

[2] G. Aigner, et. al. Eliminating virtual function calls in C++ programs. ECOOP 1996.

Expressing GPU Parallelism in Object-oriented Programs

Ikra-Ruby: A Parallel Array Interface for Ruby

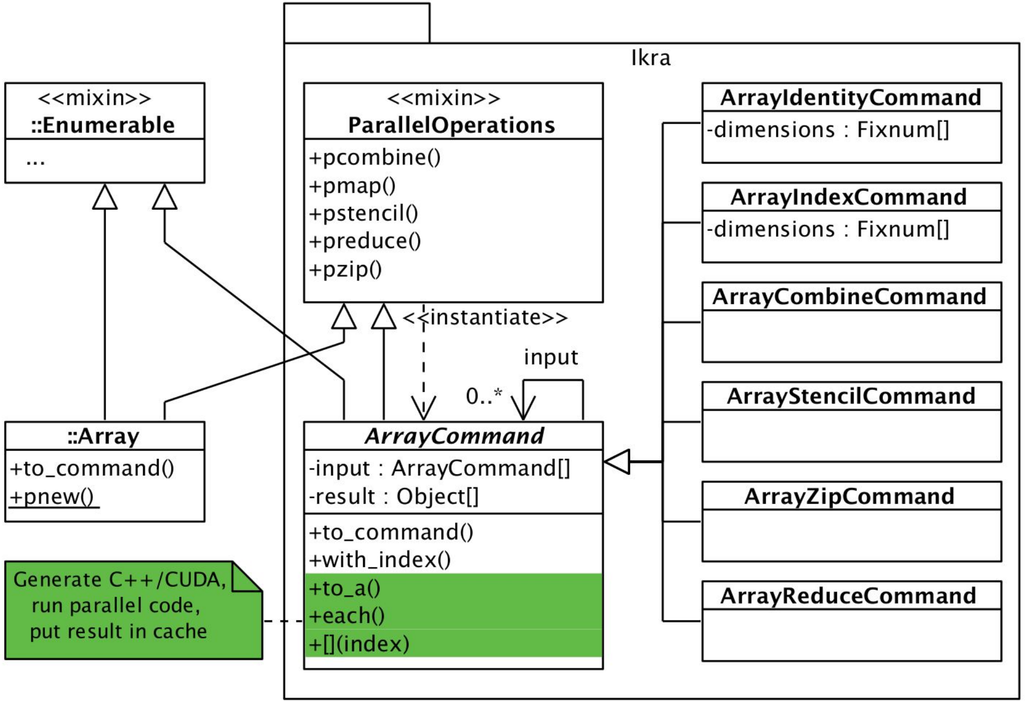
- Parallel array operations [ARRAY16]
  - `Array::pmap(&block)`
  - `Array::pcombine(others…, &block)`
  - `Array class::pnew(n, &block)`
  - `Array::preduce(&block)`
  - `Array::pzip(others…)`
  - `Array::peach(&block)`

Functional array operations are executed **lazily** and can be **chained**, forming a **computing graph**.

only *basic* Ruby features in `block`, no object-oriented programming

- Computation graph is **fused** into a small number of efficient CUDA **kernels**.
- Contribution of Ikra-Ruby:
  - Modular GPU programming style in a dynamically-typed language: Combine multiple small parallel array operations to build a complex program.
  - Kernel fusion of computation graph through type inference [ARRAY17].

Handout only: Ikra-Ruby Architecture



- Parallel operations return an **array command**
- Programmers build a **computation graph** of parallel operations
- Access of result (`to_a`, `[]`, `each`) triggers code generation and GPU execution.

From Ikra-Ruby to Ikra-Cpp

- Ikra-Ruby is suitable for **mathematical computations**.
  E.g.: Computation graph of linear algebra operations in machine learning
- *But:* A **simpler model** is sufficient for many object-oriented HPC applications.
  - `pmap`/`preduce`/`...`: Functional operations → **Immutability of state**
  - Object-oriented programming in mainstream languages: **Imperative state changes**
  - No need for `pmap`/`preduce`/`...`. **peach is sufficient**.
- *Vision:* Develop a limited but more **optimized C++/CUDA backend Ikra-Cpp** and integrate it into Ikra-Ruby (future work).

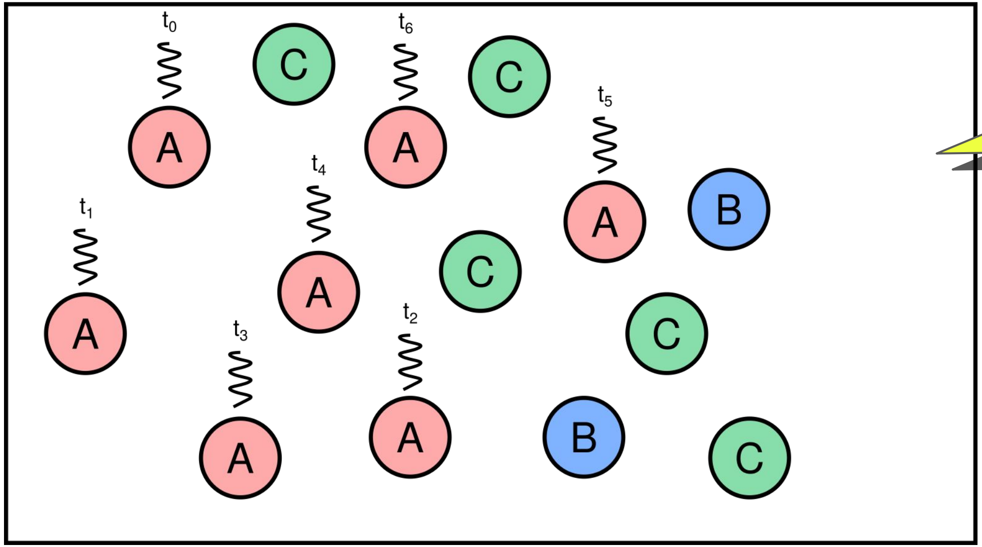
Ikra-Cpp: A CUDA/C++ Framework for SMMO

- A lower-level CUDA/C++ programming interface for SMMO applications.
- SMMO: **Single-Method Multiple-Objects** [WPMVP18, ECOOP19]
- OOP-speech for SIMD (Single-Instruction Multiple-Data)
- Main operation: **`parallel_do<T, &T::func>(args...)`**
  - Run a method `T::func` for all objects of a type `T`.
  - Same as Ikra-Ruby: `objects.peach do |o| o.func(args...) end`
  - New! Objects can be created/deleted inside of a parallel do-all.
- Create many objects at once: **`parallel_new<T>(n, args...)`**
  - Same as Ikra-Ruby: `(0...n).peach do |i| T.new(i, args...) end`
- Sequential do-all: **`device_do<T, &T::func>(args...)`**

arbitrary C++ code allowed, including obj.-orient. programming

SMMO: Single-Method Multiple-Objects (1/3)

```
parallel_do<A, &A::func>()
```

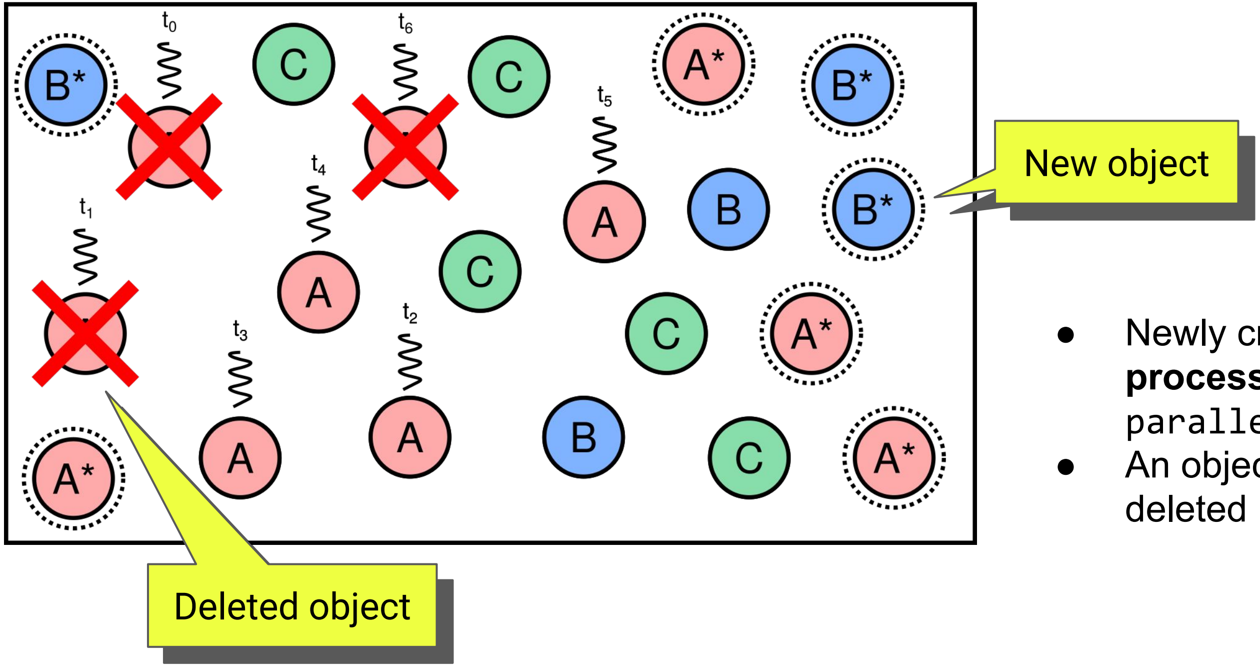


Run `A::func` for all objects of type `A` (in parallel).

- Ikra-Cpp assigns objects to threads.
- Assignment is such that memory coalescing is maximized. (More on that later…)

SMMO: Single-Method Multiple-Objects (2/3)

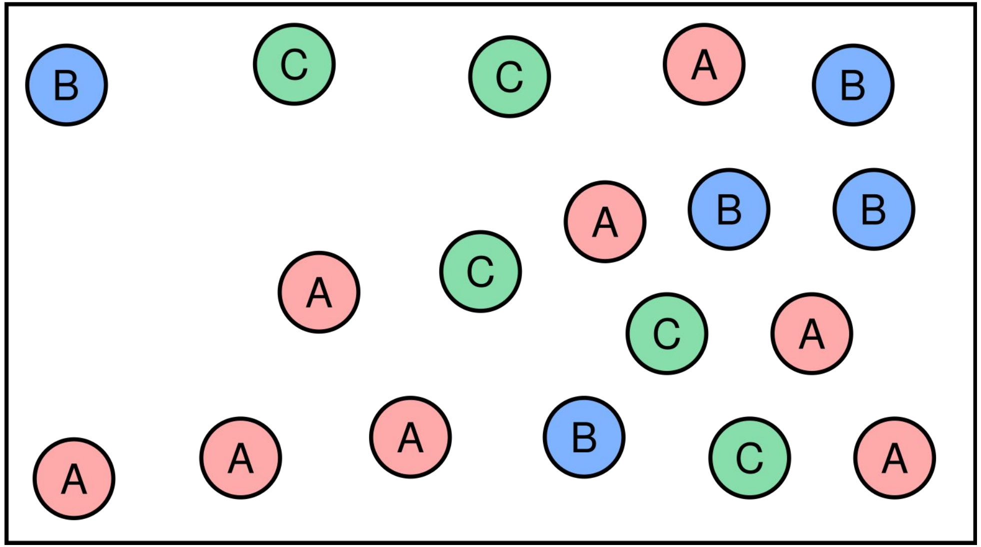
During `parallel_do<A, &A::func>`



- Newly created objects are **not processed** by the same `parallel_do`.
- An object `obj` of type *A* may only be deleted by its assigned thread.

SMMO: Single-Method Multiple-Objects (3/3)

After `parallel_do<A, &A::func>()`



Handout only: Full SMMO Interface

- **`parallel_do<T, &T::func>(args...)`**: Launches a CUDA kernel that runs a member function `T::func` for all objects of type `T` and subtypes (sep. kernel) existing at launch time. `T::func` may allocate new objects but they are not enumerated by this parallel do-all. `T::func` may deallocate any object of different type `U != T`, but this is the only object of type `T` it may deallocate (delete itself).
- **`parallel_new<T>(n, args...)`**: Launches a CUDA kernel that instantiates n objects of type `T`. This operation calls the constructor of `T` in parallel with an object index between `[0; n)` as first argument, followed by `args`....
- **`device_do<T, &T::func>(args...)`**: Runs a member function `T::func` for all object of type `T` in the current CUDA thread. Can only be used inside of a parallel do-all or a manually launched CUDA kernel.
- **`new(d_allocator) T(args...)`**: Allocates a new object of type `T` and returns a pointer to the object. Provided by DynaSOAR.
- **`destroy(d_allocator, ptr)`**: Deletes an object with pointer `ptr`, assuming that the object was allocated with `d_allocator`. Provided by DynaSOAr.
- **`parallel_defrag<T, k1, k2>()`**: Initiates defragmentation of objects of type `T`. Internally, this function may run multiple defragmentation passes depending on parameters `k1` and `k2`. Cannot be used in device code. Provided by CompactGpu.

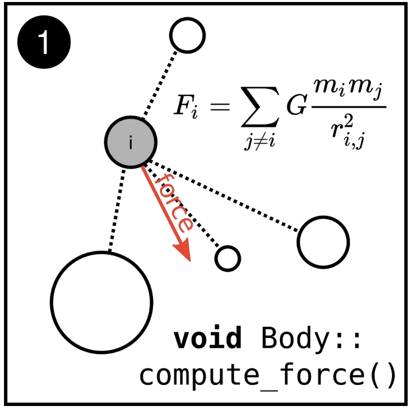
SMMO Examples

[ECOOP-Artifact 2019]

Example: N-Body Simulation

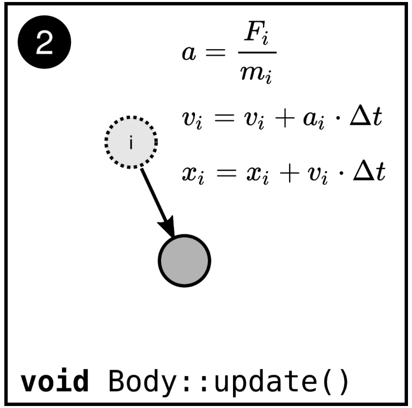
Initialization

```
auto* h_allocator =
    new HAllocatorHandle<AllocatorT>();
h_allocator->parallel_new<Body>(65536);
```



$$F_i = \sum_{j \neq i} G \frac{m_i m_j}{r_{i,j}^2}$$

`void Body::compute_force()`



$$a = \frac{F_i}{m_i}$$

$$v_i = v_i + a_i \cdot \Delta t$$

$$x_i = x_i + v_i \cdot \Delta t$$

`void Body::update()`

Main Loop

```
for (int i = 0; i < kIterations; ++i) {
    h_allocator->parallel_do<Body, &Body::compute_force>();
    h_allocator->parallel_do<Body, &Body::update>();
}

delete h_allocator;
```

Handout only: Example: N-Body Simulation

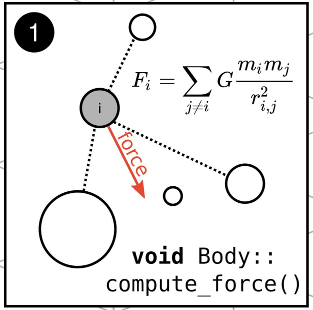
```
#include "dynasoar.h"

// Pre-declare all classes. This simple example has only one class.
class Body;
using AllocatorT = SoaAllocator</*max_num_obj=*/ 16777216, /*T...=*/ Body>;
__device__ DAllocatorHandle<AllocatorT> d_allocator;
```

Example: N-Body Simulation

```
class Body : public AllocatorT::Base {    // Can subclass other user-defined class.
 public:
  // Pre-declare all field types.
  declare_field_types(Body, float, float, float, float, float, float, float)

 private:
  // Declare fields with proxy types but use like normal C++ fields.
  Field<Body, 0> pos_x_;
  Field<Body, 1> pos_y_;
  Field<Body, 2> vel_x_;
  Field<Body, 3> vel_y_;
  Field<Body, 4> force_x_;
  Field<Body, 5> force_y_;
  Field<Body, 6> mass_;
```

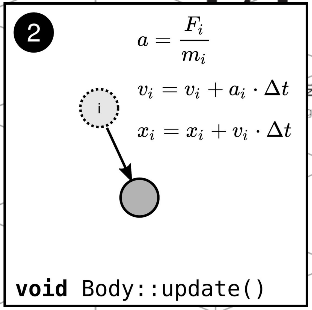
CUDA/C++ embedded **data layout DSL** (for SOA layout)

Example: N-Body Simulation



$$F_i = \sum_{j \neq i} G \frac{m_i m_j}{r_{i,j}^2}$$

void Body::
compute_force()



$$a = \frac{F_i}{m_i}$$

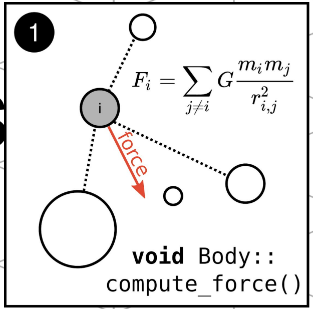
$$v_i = v_i + a_i \cdot \Delta t$$

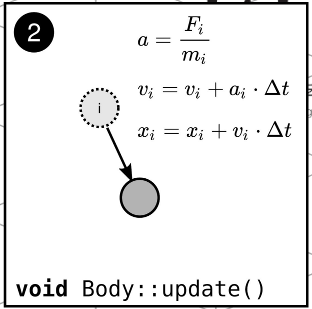
$$x_i = x_i + v_i \cdot \Delta t$$

void Body::update()

```
class Body : public AllocatorT::Base {
  /* ... */

  __device__ Body(float pos_x, float pos_y, float vel_x, float vel_y, float mass)
      : pos_x_(pos_x), pos_y_(pos_y), vel_x_(vel_x), vel_y_(vel_y), mass_(mass) {}

  // This constructor is invoked by parallel_new.
  __device__ Body(int id) : Body(/*pos_x=*/ random_float(0, 1), /*...*/) {}

  __device__ void update(float dt) {
    vel_x_ += force_x_ * dt / mass_;
    vel_y_ += force_y_ * dt / mass_;
    pos_x_ += dt * vel_x_;
    pos_y_ += dt * vel_y_;
  }
```

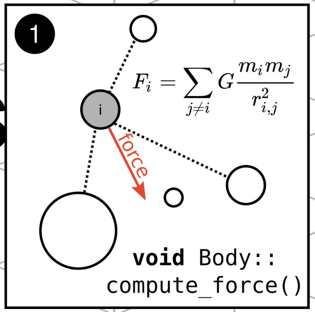
Handout only: Example: N-Body S

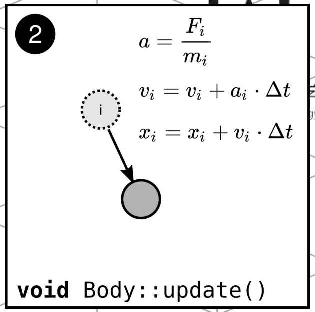



2
$a = \frac{F_i}{m_i}$
$v_i = v_i + a_i \cdot \Delta t$
$x_i = x_i + v_i \cdot \Delta t$
i
void Body::update()

```
class Body : public AllocatorT::Base {
  /* ... */

 public:
  __device__ void apply_force(Body* other) {
    if (other != this) {
      float dx = pos_x_ - other->pos_x_;  float dy = pos_y_ - other->pos_y_;
      float dist = sqrt(dx*dx + dy*dy);
      float F = kGravityConstant * mass_ * other->mass_ / (dist * dist);
      other->force_x_ += F * dx / dist;  other->force_y_ += F * dy / dist;
    }
  }

  __device__ void compute_force() {
    force_x_ = force_y_ = 0.0f;
    d_allocator->device_do<Body, &Body::apply_force>(this);
  }
```

Handout only: Example: N-Body S

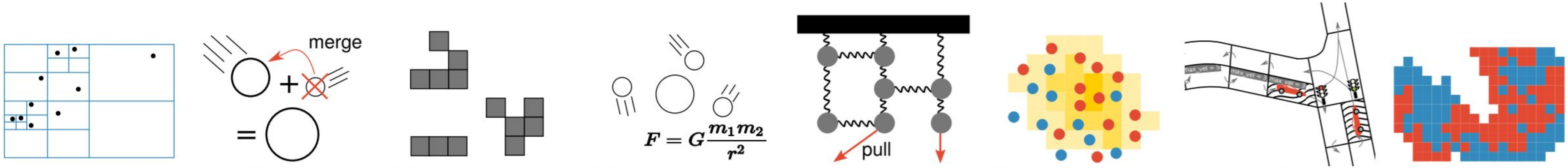



2
$a = \frac{F_i}{m_i}$
$v_i = v_i + a_i \cdot \Delta t$
$x_i = x_i + v_i \cdot \Delta t$
i
void Body::update()

```
class Body : public AllocatorT::Base {
  /* ... */

 public:
  __device__ void apply_force(Body* other) {
    if (other != this) {
      float dx = pos_x_ - other->pos_x_;  float dy = pos_y_ - other->pos_y_;
      float dist = sqrt(dx*dx + dy*dy);
      float F = kGravityConstant * mass_ * other->mass_ / (dist * dist);
      other->force_x_ += F * dx / dist;  other->force_y_ += F * dy / dist;
    }
  }

  __device__ void compute_force() {
    force_x_ = force_y_ = 0.0f;
    d_allocator->device_do<Body, &Body::apply_force>(this);
  }
```

```
for (Body* b : get_objects<Body>) {
  b->apply_force(this);
}
```

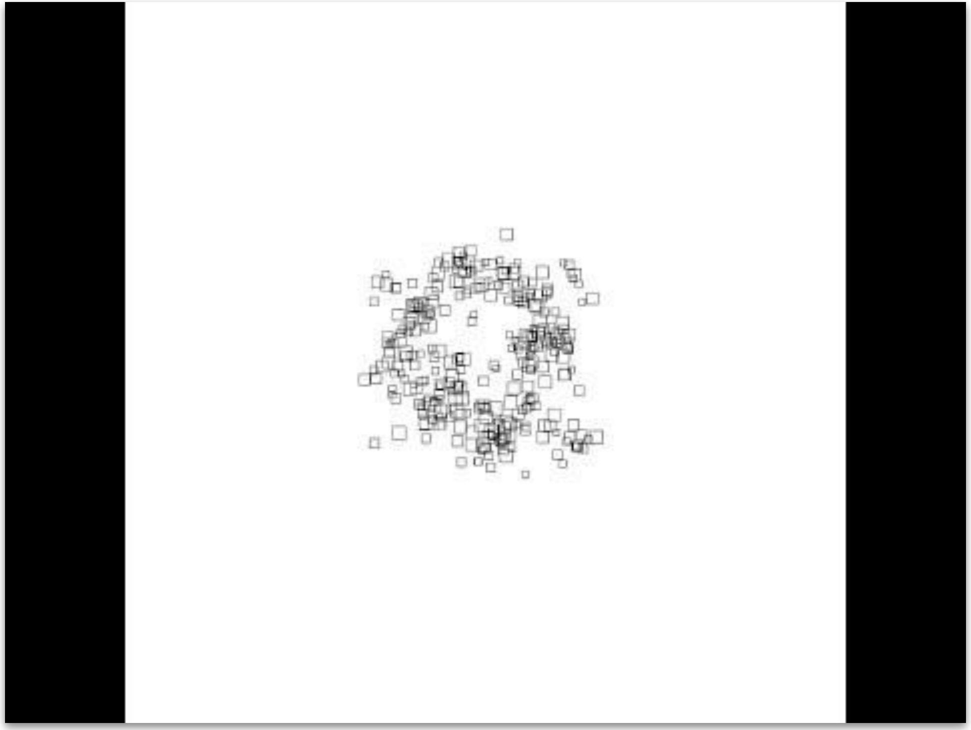
Examples of SMMO Applications



(a) barnes-hut [4]: Parallel Tree Constr. **(b)** collisions: Particle System **(c)** game-of-life: Cellular Automaton **(d)** nbody: Particle System **(e)** structure [14]: Finite Elem. Method **(f)** sugarscape [8]: Agent-based Sim. **(g)** traffic [17]: Nagel-Schr. Model **(h)** wator [6]: Agent-based Sim.

- Implemented and evaluated Ikra-Cpp/DynaSOAr with 8 SMMO applications.
- SMMO can express many different patterns of HPC applications, e.g.:
    - **Cellular automata:** game-of-life, sugarscape, traffic, wa-tor
    - **Agent-based modelling:** sugarscape, traffic, wa-tor
    - Dynamic **tree construction/update:** barnes-hut
    - Applications w/ **graph-structured data:** structure, traffic, breadth-first search

Example: N-Body Simulation



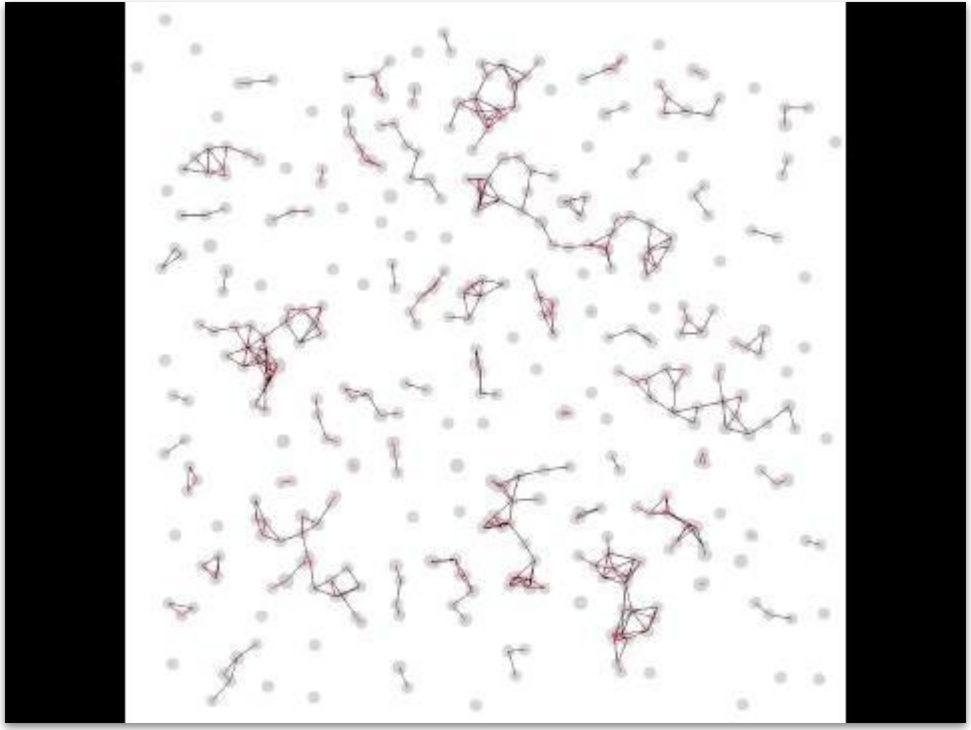
Body

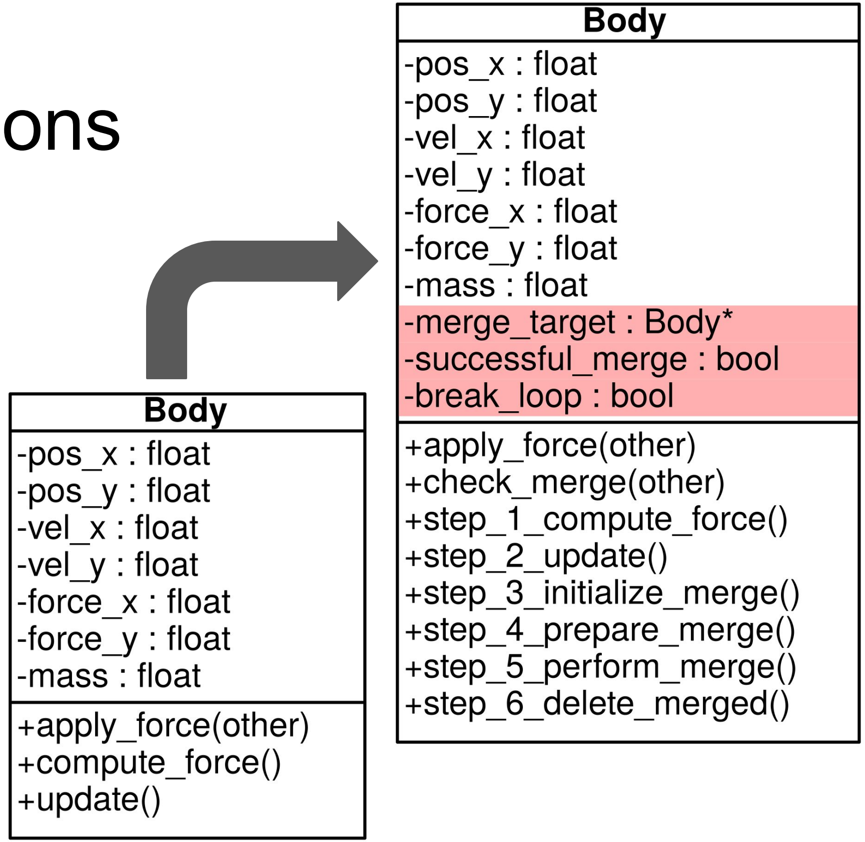
-pos_x : float
-pos_y : float
-vel_x : float
-vel_y : float
-force_x : float
-force_y : float
-mass : float

+apply_force(other)
+compute_force()
+update()

```
parallel_new<Body>(500);

for (int i = 0; i < 1000; ++i) {
  parallel_do<Body, &Body::compute_force>();
  parallel_do<Body, &Body::update>();
}
```

Example: N-Body with Collisions





Body

-pos_x : float
-pos_y : float
-vel_x : float
-vel_y : float
-force_x : float
-force_y : float
-mass : float

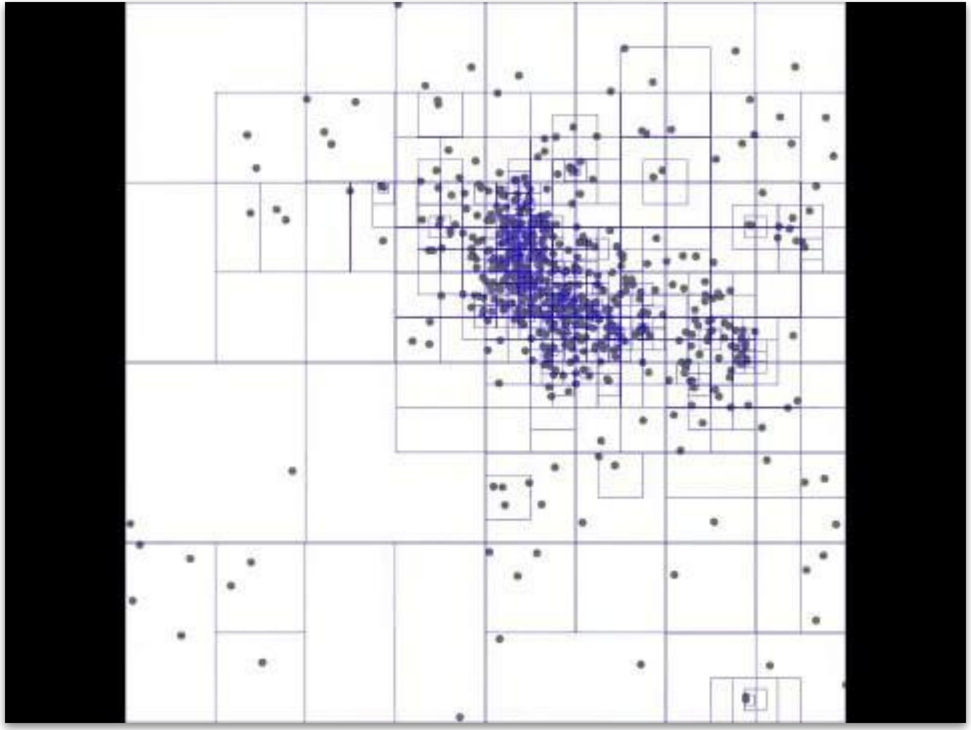
+apply_force(other)
+compute_force()
+update()

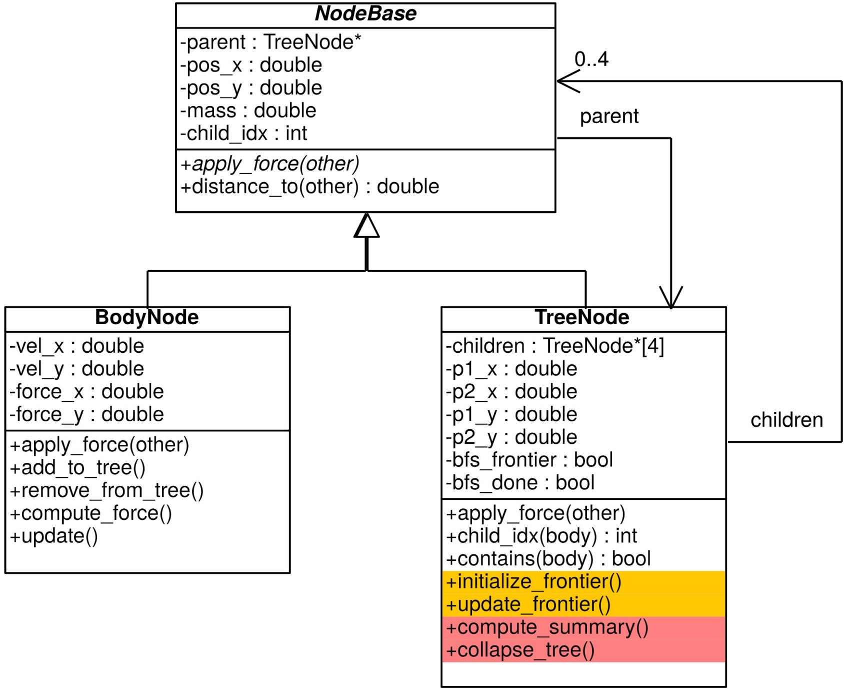
Body

-pos_x : float
-pos_y : float
-vel_x : float
-vel_y : float
-force_x : float
-force_y : float
-mass : float
-merge_target : Body*
-successful_merge : bool
-break_loop : bool

+apply_force(other)
+check_merge(other)
+step_1_compute_force()
+step_2_update()
+step_3_initialize_merge()
+step_4_prepare_merge()
+step_5_perform_merge()
+step_6_delete_merged()

Example: Barnes-Hut N-Body Simulation





NodeBase

-parent : TreeNode*
-pos_x : double
-pos_y : double
-mass : double
-child_idx : int

+*apply_force(other)*
+distance_to(other) : double

BodyNode

-vel_x : double
-vel_y : double
-force_x : double
-force_y : double

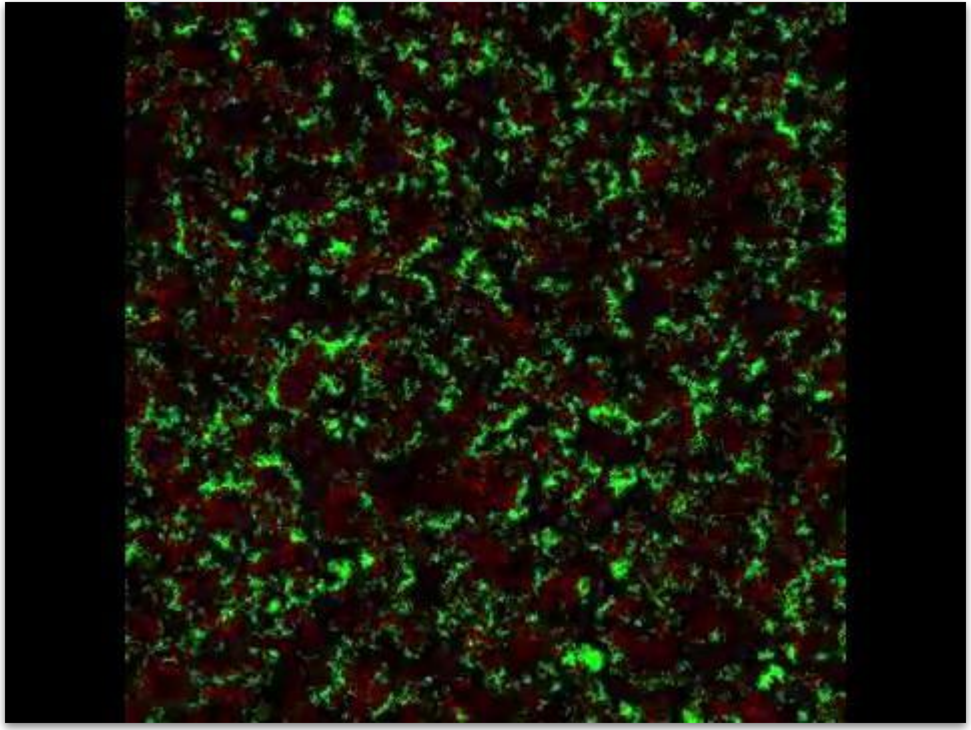
+apply_force(other)
+add_to_tree()
+remove_from_tree()
+compute_force()
+update()

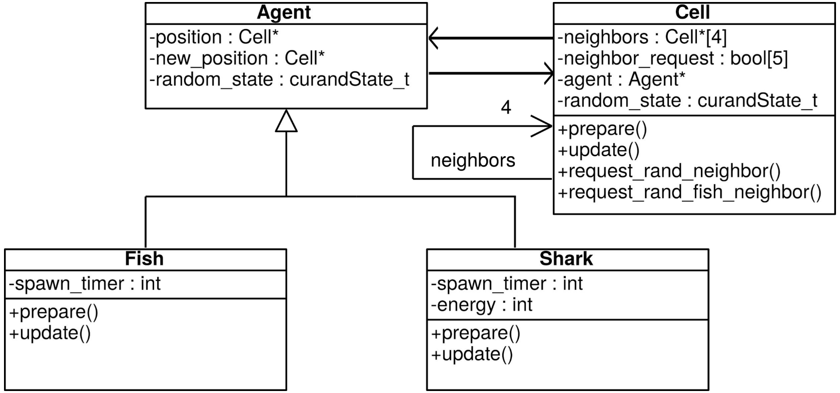
TreeNode

-children : TreeNode*[4]
-p1_x : double
-p2_x : double
-p1_y : double
-p2_y : double
-bfs_frontier : bool
-bfs_done : bool

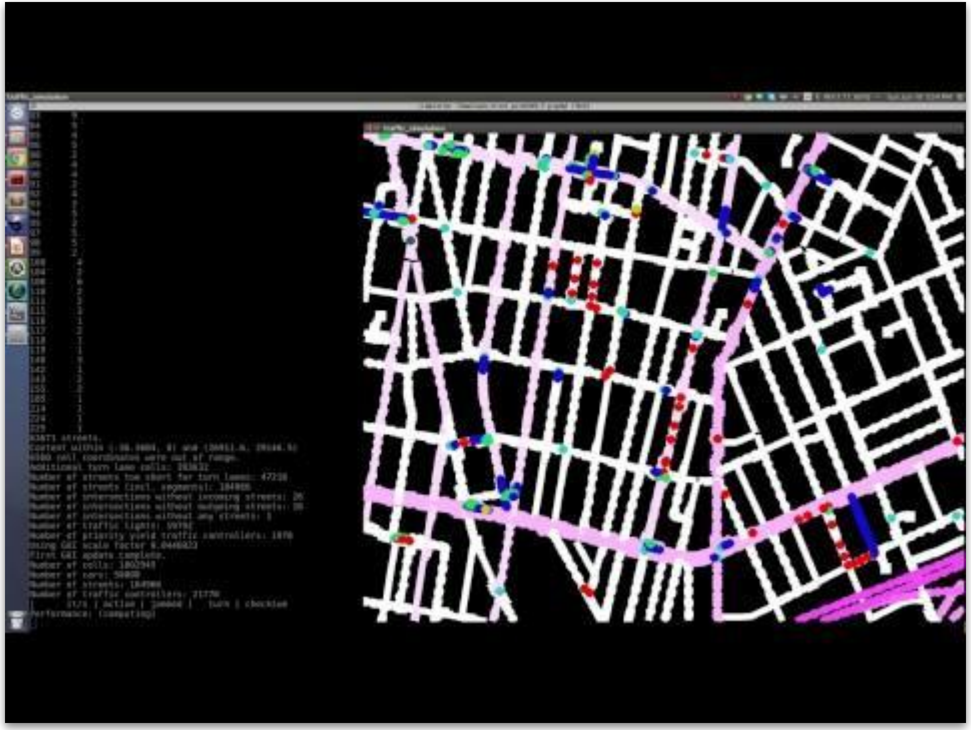
+apply_force(other)
+child_idx(body) : int
+contains(body) : bool
+initialize_frontier()
+update_frontier()
+compute_summary()
+collapse_tree()

Example: Fish-and-Shark (wa-tor)

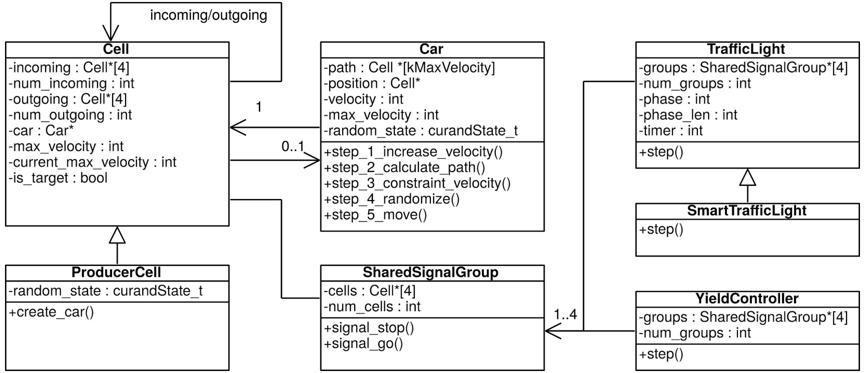




Example: Nagel-Schreckenberg Simulation



```
[illegible] streets.
Content within ( [illegible] ) and ( [illegible] )
[illegible] cell coordinates were out of range.
Additional turn lane cells: [illegible]
Number of streets too short for turn lanes: 47238
Number of streets (incl. segments): 184086
Number of intersections without incoming streets: 26
Number of intersections without outgoing streets: 38
Number of intersections without any streets: 1
Number of traffic lights: 19792
Number of priority yield traffic controllers: 1070
Using GUI scale factor 0.0448023
First GUI update complete.
Number of cells: 1802948
Number of cars: 50000
Number of streets: 184086
Number of traffic controllers: 21770
|      [illegible] | active | jammed |   turn | checksum
Performance: (computing)
```



incoming/outgoing

Cell
-incoming : Cell*[4]
-num_incoming : int
-outgoing : Cell*[4]
-num_outgoing : int
-car : Car*
-max_velocity : int
-current_max_velocity : int
-is_target : bool

1

0..1

Car
-path : Cell *[kMaxVelocity]
-position : Cell*
-velocity : int
-max_velocity : int
-random_state : curandState_t
+step_1_increase_velocity()
+step_2_calculate_path()
+step_3_constraint_velocity()
+step_4_randomize()
+step_5_move()

TrafficLight
-groups : SharedSignalGroup*[4]
-num_groups : int
-phase : int
-phase_len : int
-timer : int
+step()

SmartTrafficLight
+step()

ProducerCell
-random_state : curandState_t
+create_car()

SharedSignalGroup
-cells : Cell*[4]
-num_cells : int
+signal_stop()
+signal_go()

1..4

YieldController
-groups : SharedSignalGroup*[4]
-num_groups : int
+step()

An SOA Data Layout DSL for Ikra-Cpp [WPMVP18]

- Ikra-Cpp provides two ways of memory allocation:
  `new` `T()`, `parallel_new<T>(n)`
- Objects are not allocated in one block of memory, but in a **custom layout**.
- To allow for OOP abstractions: Embedded C++/CUDA data layout DSL

```
class Body : public AllocatorT::Base {
 public:
  declare_field_types(Body, float, float, float, float, float, float, float)

 private:
  Field<Body, 0> pos_x_;
  Field<Body, 1> pos_y_;
  Field<Body, 2> vel_x_;
  Field<Body, 3> vel_y_;
  Field<Body, 4> force_x_;
  Field<Body, 5> force_y_;
  Field<Body, 6> mass_;
```

Proxy types are *implicitly converted* to base types.

Handout only: Implicit Conversion of Proxy Types

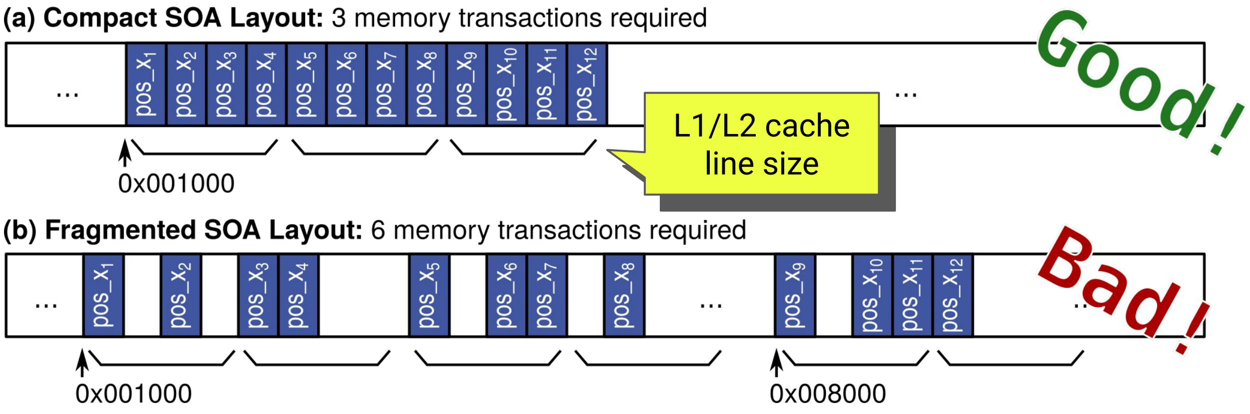
- Objects are referred to with **fake pointers**: Encoding all information required to compute the physical memory location of each field value.
- Objects and proxy type values always appear as **lvalues**.
- Embedded DSL is implemented with advanced C++ features: template metaprogramming, operator overloading, type punning

```
template<int Index>
class Field {
  using BaseT = /* Index-th predeclared type */;
  operator T&() const { return *data_ptr(); }

  T* data_ptr() const {
    uint64_t ptr = reinterpret_cast<uint64_t>(this);
    // Compute physical memory location of value based on ptr. We could implement an arbitrary object
    // layout here (not just SOA). See thesis for details.
  }
}
```

DynaSOAr: A Dynamic Memory Allocator with SOA Performance [ECOOP 2019]

Design Requirements

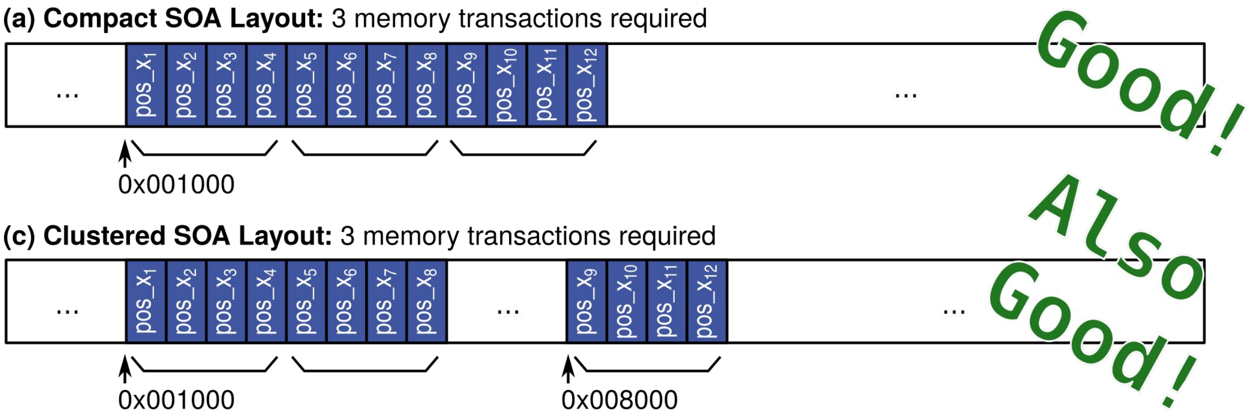
- *Programming Interface:* **`new`** / **`delete`** operations
- *Memory Layout:* Efficient memory access **in `parallel_do` operations**
  - *Goal:* Achieve **coalesced** (vectorized) memory access with SOA-style allocation.
  - Trading **faster data access** for slower memory (de)allocation time.
  - **Low fragmentation** is key: Fragmented data requires more vector transactions.



- *Lock-free Implementation:* Locking can easily lead to deadlocks on GPUs

Design Requirements

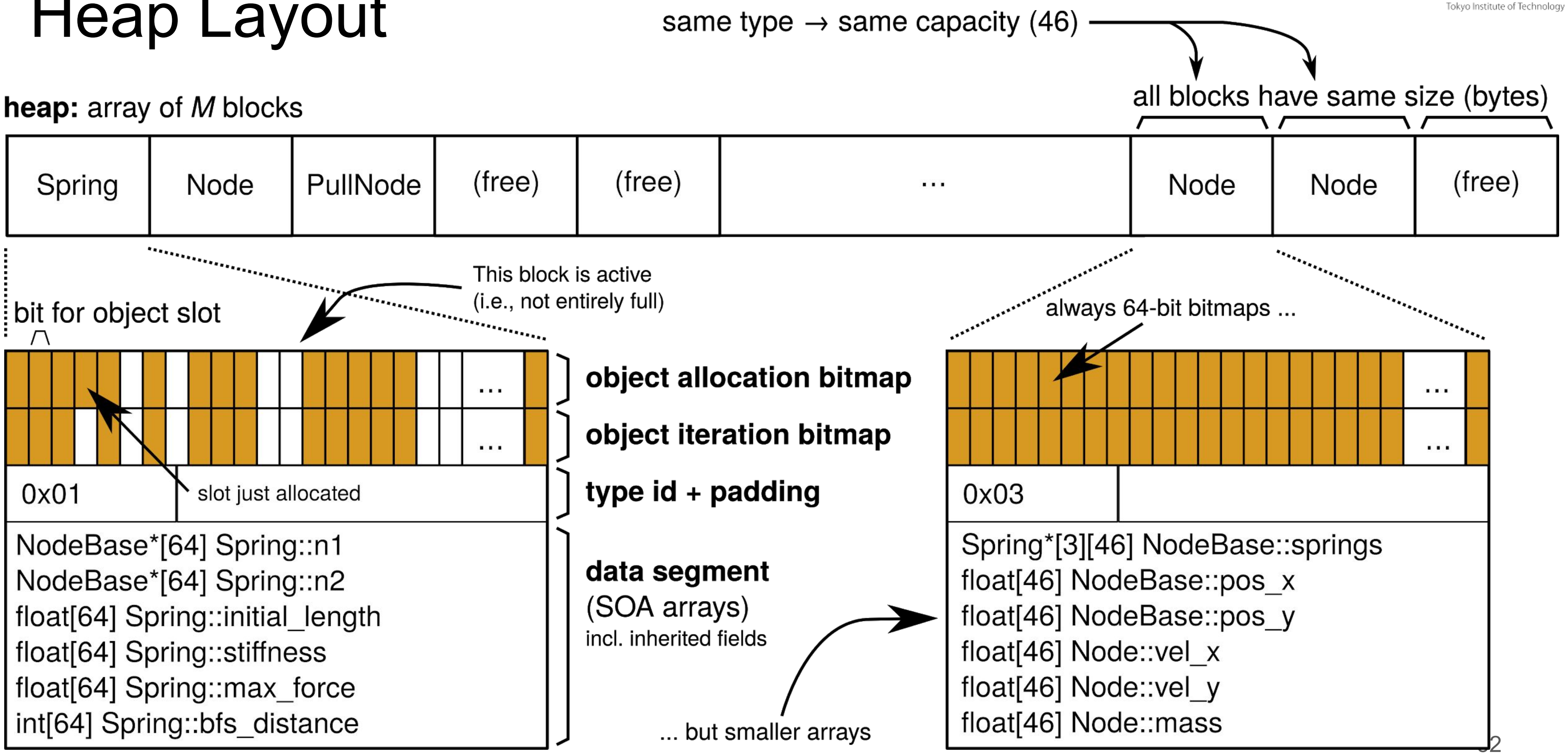
- *Programming Interface:* **`new`** / **`delete`** operations
- *Memory Layout:* Efficient memory access **in `parallel_do` operations**
  - *Goal:* Achieve **coalesced** (vectorized) memory access with SOA-style allocation.
  - Trading **faster data access** for slower memory (de)allocation time.
  - **Low fragmentation** is key: Fragmented data requires more vector transactions.


(a) Compact SOA Layout: 3 memory transactions required

(c) Clustered SOA Layout: 3 memory transactions required

- *Lock-free Implementation:* Locking can easily lead to deadlocks on GPUs

Heap Layout



same type → same capacity (46)

all blocks have same size (bytes)

heap: array of *M* blocks

Spring	Node	PullNode	(free)	(free)	…	Node	Node	(free)

bit for object slot

This block is active
(i.e., not entirely full)

always 64-bit bitmaps …

object allocation bitmap

object iteration bitmap

type id + padding

data segment
(SOA arrays)
incl. inherited fields

... but smaller arrays

0x01

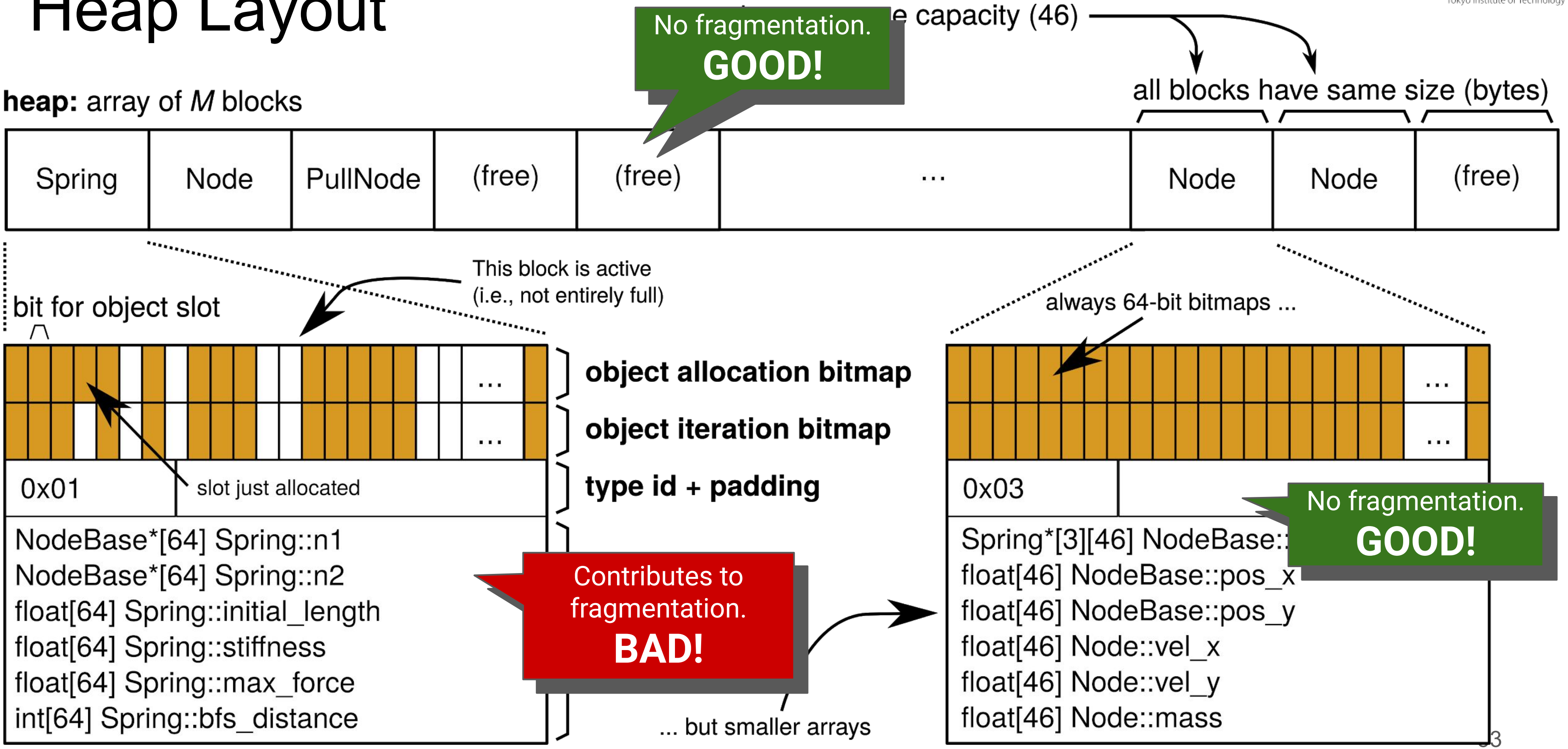
slot just allocated

NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

0x03

Spring*[3][46] NodeBase::springs
float[46] NodeBase::pos_x
float[46] NodeBase::pos_y
float[46] Node::vel_x
float[46] Node::vel_y
float[46] Node::mass

Heap Layout



e capacity (46)

all blocks have same size (bytes)

heap: array of *M* blocks

Spring | Node | PullNode | (free) | (free) | … | Node | Node | (free)

No fragmentation.
GOOD!

bit for object slot

This block is active
(i.e., not entirely full)

always 64-bit bitmaps ...

object allocation bitmap

object iteration bitmap

type id + padding

0x01

slot just allocated

NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

Contributes to
fragmentation.
BAD!

... but smaller arrays

0x03

Spring*[3][46] NodeBase::
float[46] NodeBase::pos_x
float[46] NodeBase::pos_y
float[46] Node::vel_x
float[46] Node::vel_y
float[46] Node::mass

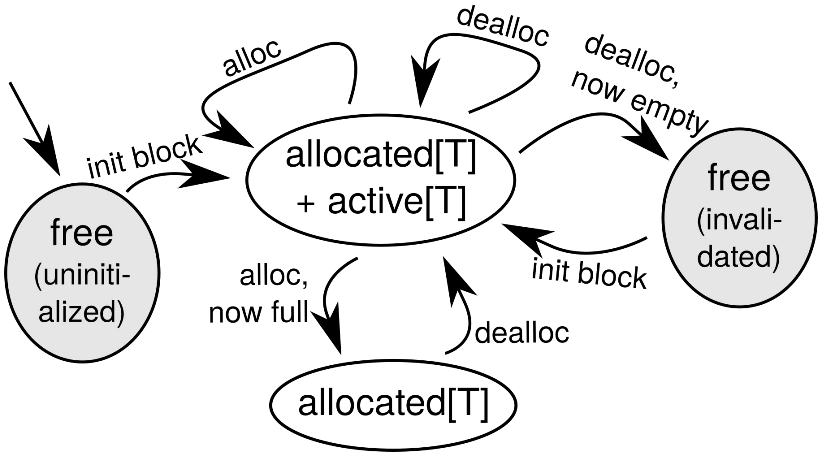
No fragmentation.
GOOD!

Handout only: Heap Layout

- Objects are allocated in **blocks** in SOA layout.
- Blocks contain objects of only one C++ class/struct type.
- All blocks have the **same size in bytes** but their capacity (max. #objects) depends on the size of their objects.
- **Object allocation bitmaps** keep track of free/occupied object slots.
  - (De)allocation: Changed with atomic bitwise operations (e.g., `atomicAnd`).
  - Always 64 bit in size (maximum capacity).

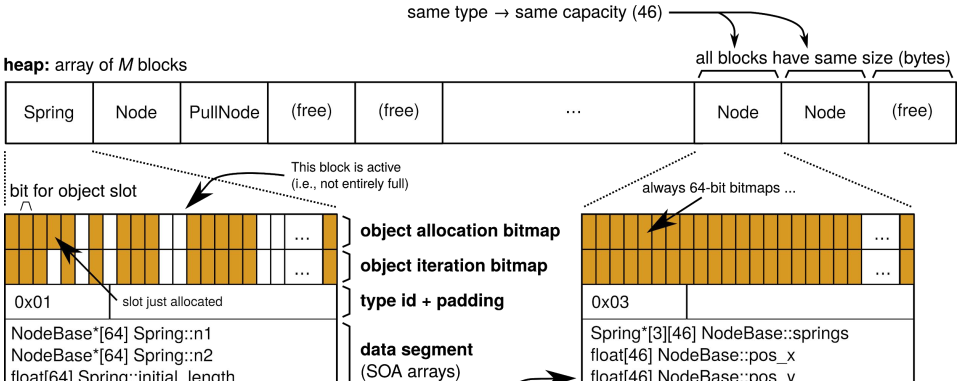
Block (Multi)States

- **free:** Contains no objects.
- **allocated[*T*]:** Contains only objects of C++ class/struct *T*.
- **active[T]:** Is **allocated[*T*]** and not full. (Space for at least 1 more object)

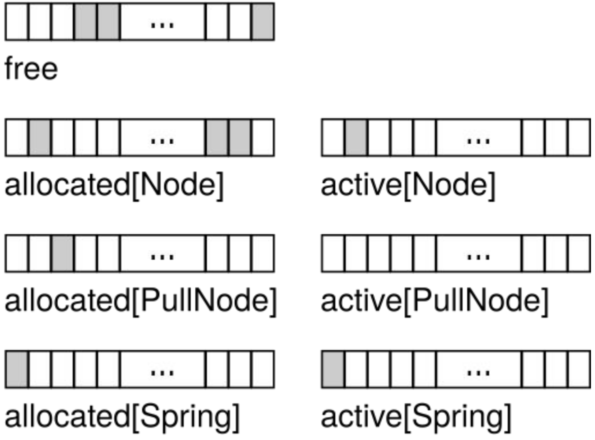


Block State Bitmaps

- Block states are **indexed** by bitmaps.
- Indices may be temporarily inconsistent with actual block states, but they are **eventually consistent**.
- *Main challenge:* Algorithms must be able to handle such inconsistencies.

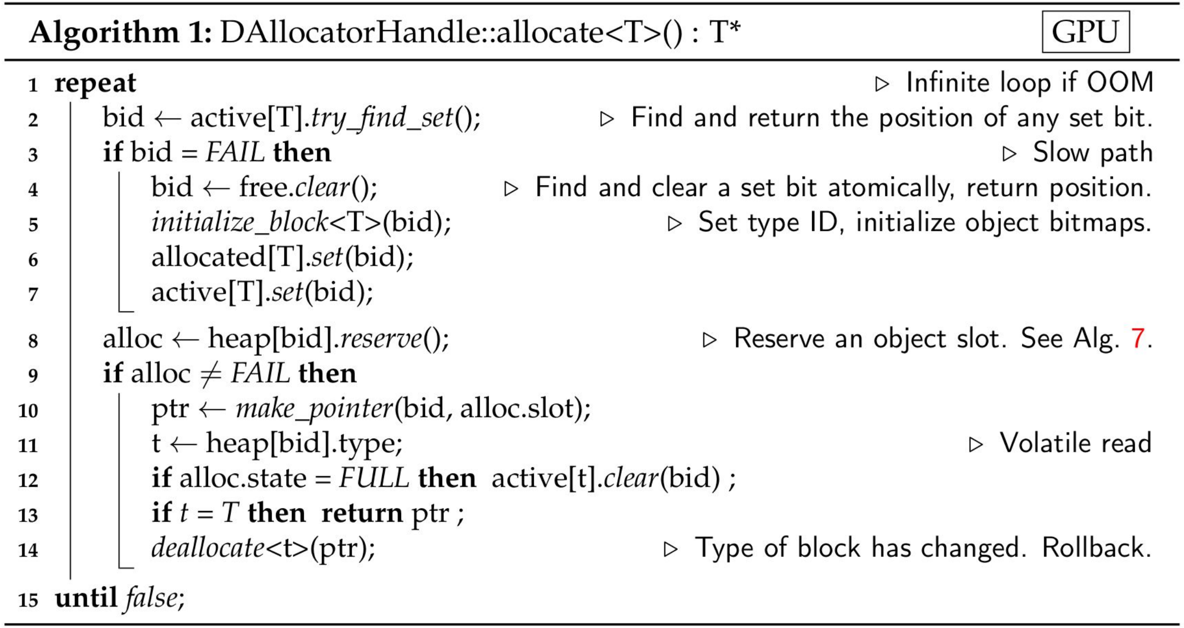


block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)



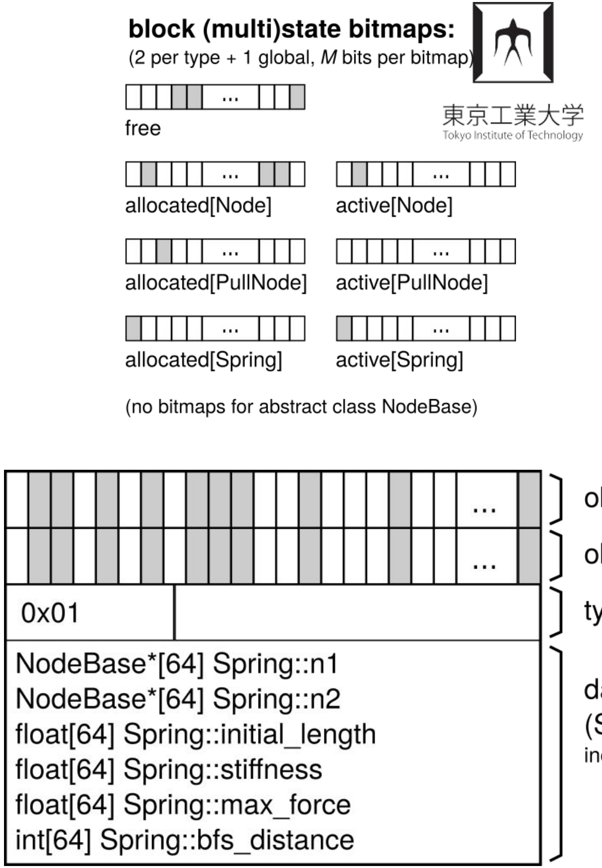
(no bitmaps for abstract class NodeBase)

Algorithm: Object Allocation

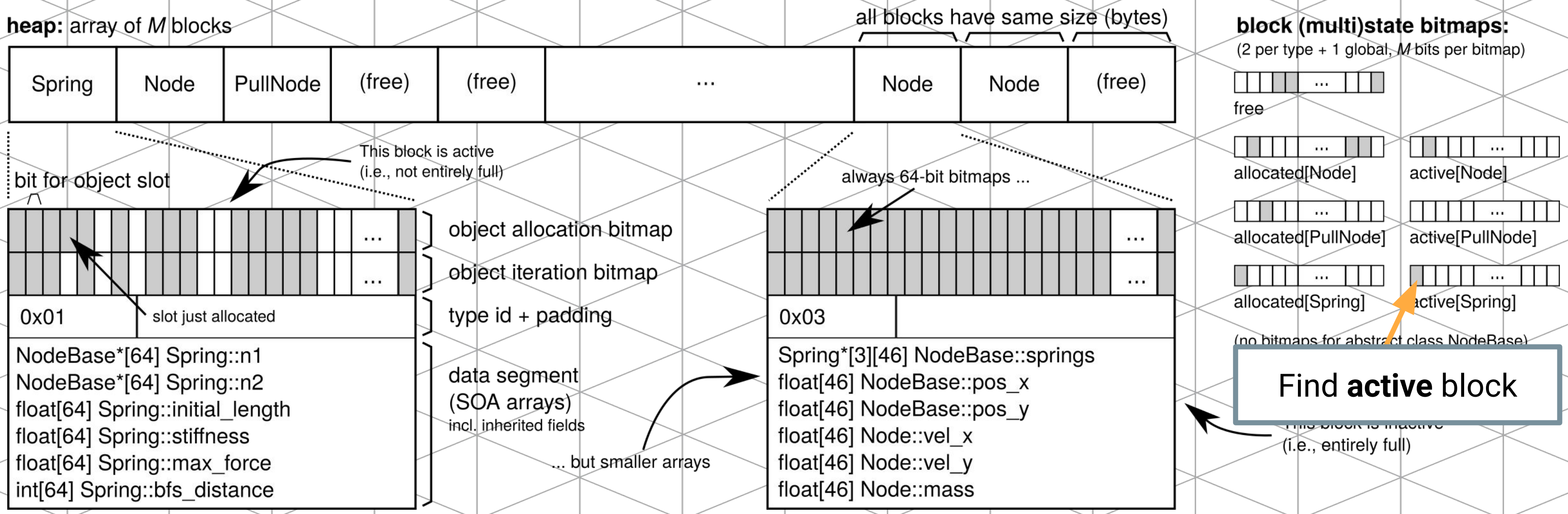


Algorithm 1: DAllocatorHandle::allocate<T>() : T* GPU

```
repeat                                                      ▷ Infinite loop if OOM
    bid ← active[T].try_find_set();       ▷ Find and return the position of any set bit.
    if bid = FAIL then                                              ▷ Slow path
        bid ← free.clear();        ▷ Find and clear a set bit atomically, return position.
        initialize_block<T>(bid);          ▷ Set type ID, initialize object bitmaps.
        allocated[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve();                ▷ Reserve an object slot. See Alg. 7.
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;                                         ▷ Volatile read
        if alloc.state = FULL then active[t].clear(bid) ;
        if t = T then return ptr ;
        deallocate<t>(ptr);                   ▷ Type of block has changed. Rollback.
until false;
```


block (multi)state bitmaps:
(2 per type + 1 global, M bits per bitmap)
...
free
allocated[Node]
active[Node]
allocated[PullNode]
active[PullNode]
allocated[Spring]
active[Spring]
(no bitmaps for abstract class NodeBase)
...
ol
ol
0x01
ty
NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance
da
(S
in

Example: **`new Spring()`**, Fast path



heap: array of *M* blocks

all blocks have same size (bytes)

Spring | Node | PullNode | (free) | (free) | … | Node | Node | (free)

bit for object slot

This block is active
(i.e., not entirely full)

always 64-bit bitmaps …

object allocation bitmap

object iteration bitmap

type id + padding

data segment
(SOA arrays)
incl. inherited fields

… but smaller arrays

0x01

slot just allocated

NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

0x03

Spring*[3][46] NodeBase::springs
float[46] NodeBase::pos_x
float[46] NodeBase::pos_y
float[46] Node::vel_x
float[46] Node::vel_y
float[46] Node::mass

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)

free

allocated[Node] | active[Node]

allocated[PullNode] | active[PullNode]

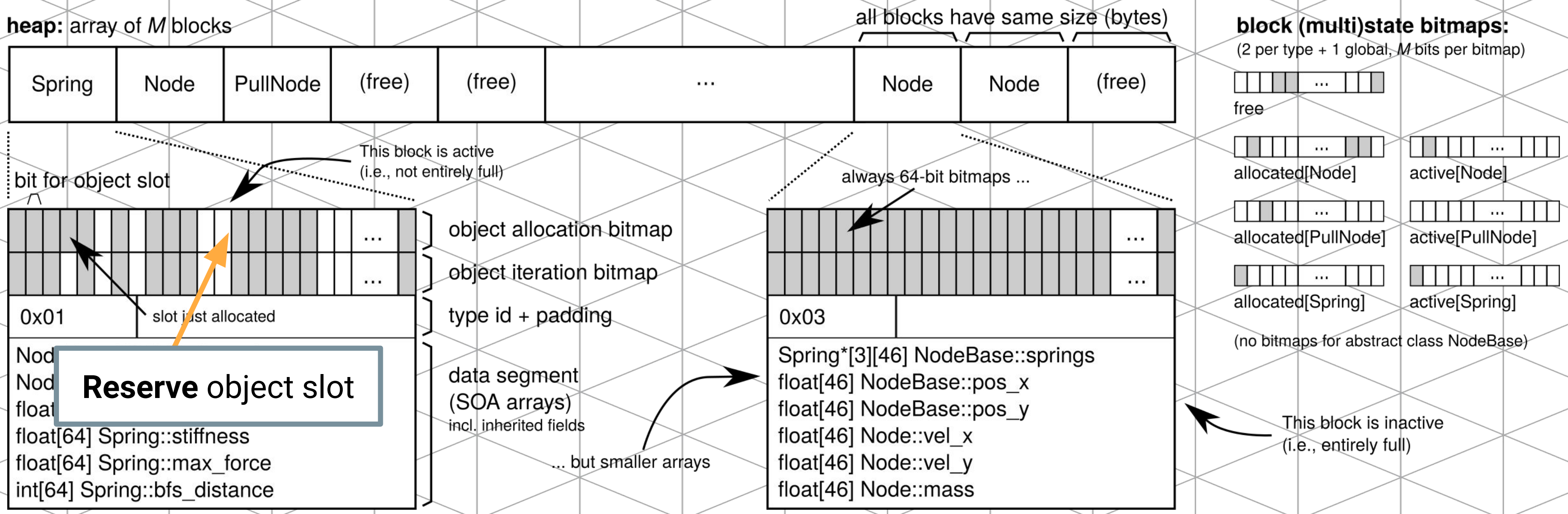
allocated[Spring] | active[Spring]

(no bitmaps for abstract class NodeBase)

This block is inactive
(i.e., entirely full)

Find **active** block

Example: **`new`** `Spring()`, Fast path



heap: array of *M* blocks

all blocks have same size (bytes)

Spring | Node | PullNode | (free) | (free) | … | Node | Node | (free)

bit for object slot

This block is active
(i.e., not entirely full)

object allocation bitmap

object iteration bitmap

type id + padding

data segment
(SOA arrays)
incl. inherited fields

0x01

slot just allocated

Nod
Nod
float
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

Reserve object slot

always 64-bit bitmaps …

... but smaller arrays

0x03

Spring*[3][46] NodeBase::springs
float[46] NodeBase::pos_x
float[46] NodeBase::pos_y
float[46] Node::vel_x
float[46] Node::vel_y
float[46] Node::mass

This block is inactive
(i.e., entirely full)

block (multi)state bitmaps:
(2 per type + 1 global, *M* bits per bitmap)

free

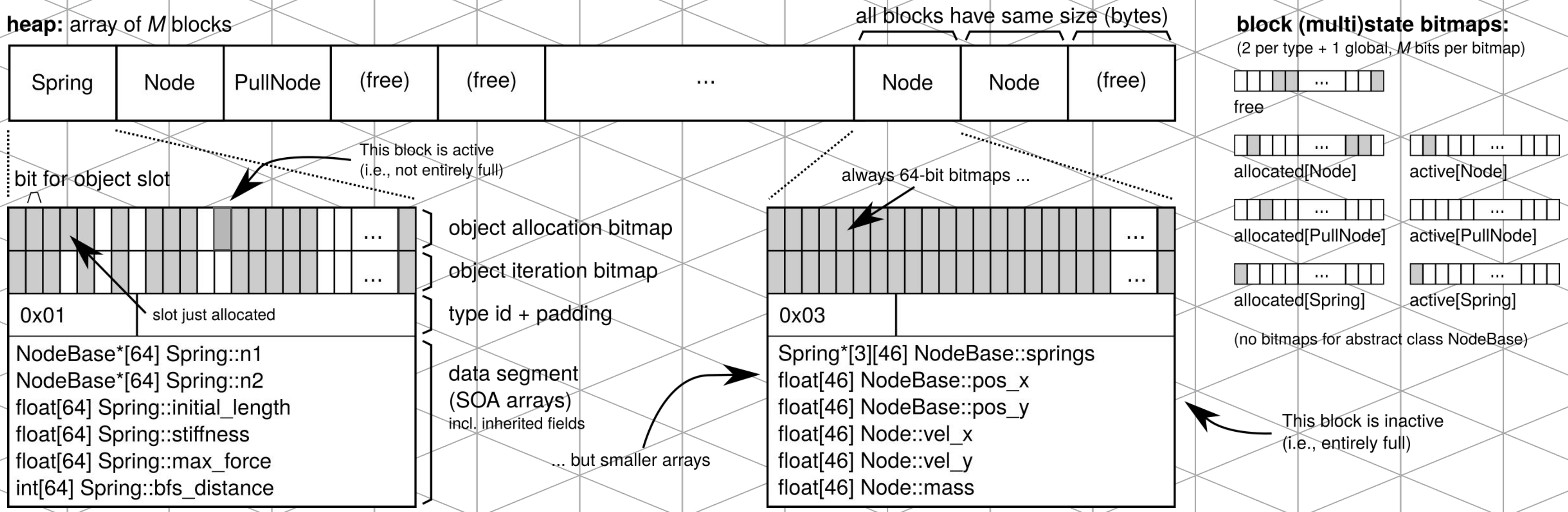
allocated[Node] active[Node]

allocated[PullNode] active[PullNode]

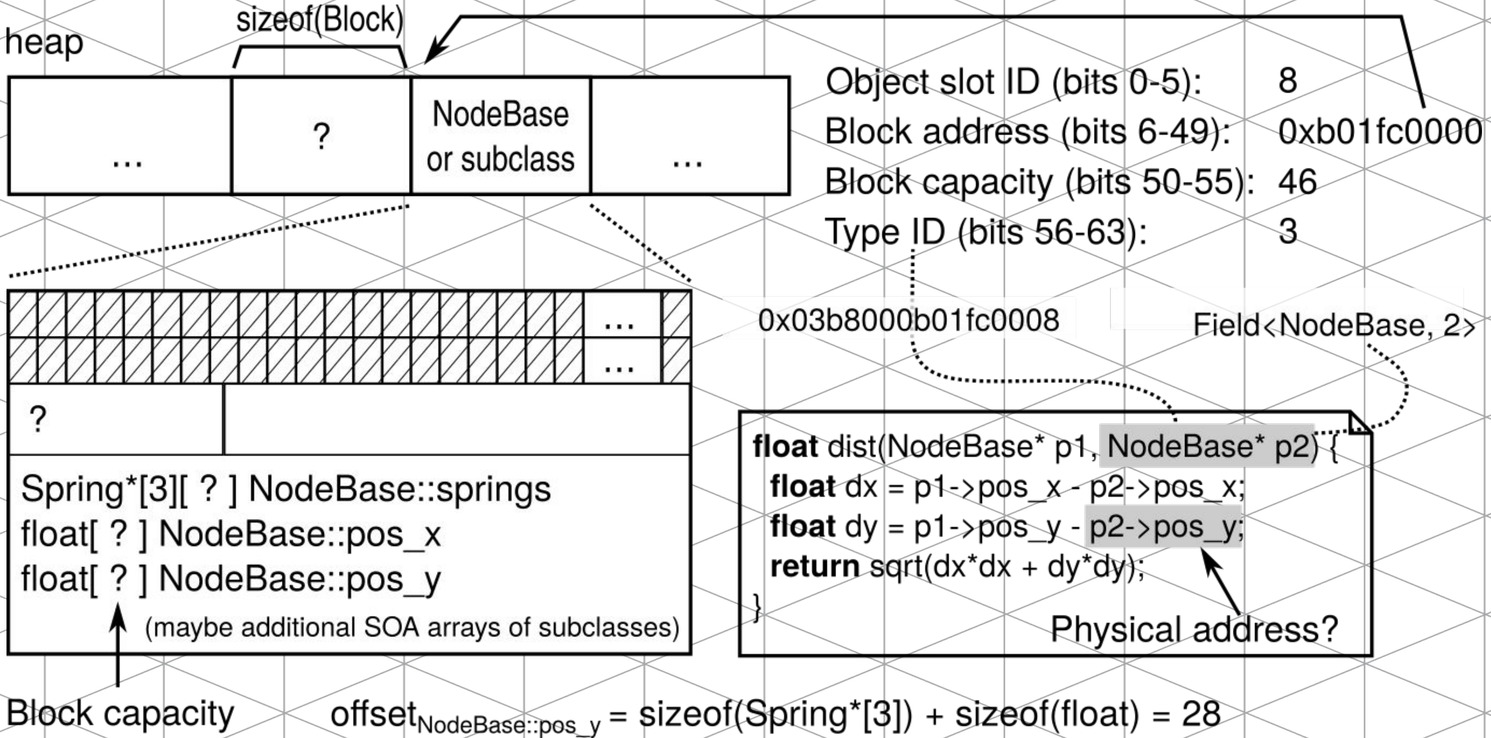
allocated[Spring] active[Spring]

(no bitmaps for abstract class NodeBase)

Example: **`new`** `Spring()`, Fast path



Handout only: Fake Pointers



heap

sizeof(Block)

… | ? | NodeBase or subclass | …

Object slot ID (bits 0-5): 8
Block address (bits 6-49): 0xb01fc0000
Block capacity (bits 50-55): 46
Type ID (bits 56-63): 3

0x03b8000b01fc0008

Field<NodeBase, 2>

…
…

?

Spring*[3][?] NodeBase::springs
float[?] NodeBase::pos_x
float[?] NodeBase::pos_y
(maybe additional SOA arrays of subclasses)

```
float dist(NodeBase* p1, NodeBase* p2) {
  float dx = p1->pos_x - p2->pos_x;
  float dy = p1->pos_y - p2->pos_y;
  return sqrt(dx*dx + dy*dy);
}
```

Physical address?

Block capacity

$\text{offset}_{\text{NodeBase::pos\_y}} = \text{sizeof(Spring*[3])} + \text{sizeof(float)} = 28$

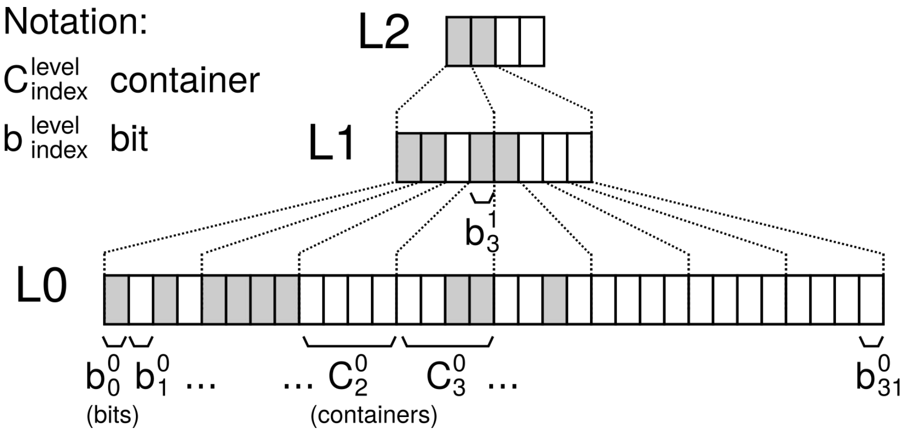
Handout only: Fake pointers

- *Problem:* Objects are not stored in one block of memory. How to refer to them with an object pointer?
- *Solution:* Object pointers are **not memory locations** but encode all information required to compute the physical location of each field (fake pointer).
- Pointers are 64 bit in CUDA, but only a few bits are actually utilized because GPUs have less than 32 GB memory. We can **store additional information in unused bits**.
- Fake pointer = Address of DynaSOAr block + additional information encoded in unused bits

Additional Optimizations

Block state bitmaps

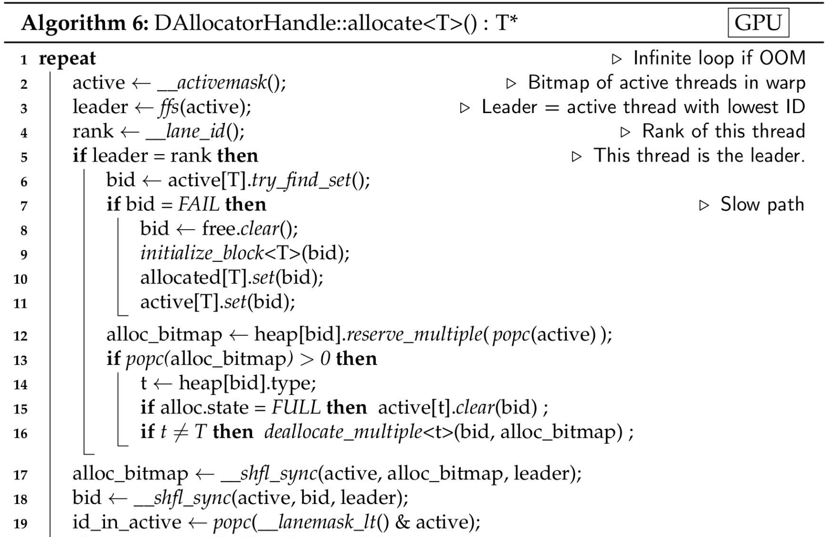
- **Hierarchical Bitmaps:** Finding set bits in a large bitmap is slow. We can find bits in a hierarchical bitmap with a logarithmic number of accesses.


Notation:
C^{level}_{index} container
b^{level}_{index} bit
L2
L1
b^{1}_{3}
L0
b^{0}_{0} b^{0}_{1} …
(bits)
… C^{0}_{2} C^{0}_{3} …
(containers)
b^{0}_{31}

```
template<int N, bool HasNested>
struct Bitmap;

template<int N>
struct Bitmap<N, /*HasNested=*/ false> {
  static const int kNumContainers = (N + 64 - 1) / 64; // ceil(N / 64)
  uint64_t containers[kNumContainers];
};

template<int N>
struct Bitmap<N, /*HasNested=*/ true> {
  static const int kNumContainers = (N + 64 - 1) / 64; // ceil(N / 64)
  static const bool kContinueHierarchy = kNumContainers > 1;

  uint64_t containers[kNumContainers];
  Bitmap<kNumContainers, kContinueHierarchy> nested;
};
```

Additional Optimizations

- **Hierarchical Bitmaps:** Finding set bits in a large bitmap is slow. We can find bits in a hierarchical bitmap with a logarithmic number of accesses.
- **Allocation Request Coalescing:** A **leader** thread reserves object slots **on behalf of all allocating threads** in the warp [1].



Algorithm 6: DAllocatorHandle::allocate<T>() : T* GPU

```
repeat                                                     ▷ Infinite loop if OOM
    active ← __activemask();                               ▷ Bitmap of active threads in warp
    leader ← ffs(active);                                  ▷ Leader = active thread with lowest ID
    rank ← __lane_id();                                    ▷ Rank of this thread
    if leader = rank then                                  ▷ This thread is the leader.
        bid ← active[T].try_find_set();
        if bid = FAIL then                                 ▷ Slow path
            bid ← free.clear();
            initialize_block<T>(bid);
            allocated[T].set(bid);
            active[T].set(bid);
        alloc_bitmap ← heap[bid].reserve_multiple( popc(active) );
        if popc(alloc_bitmap) > 0 then
            t ← heap[bid].type;
            if alloc.state = FULL then  active[t].clear(bid) ;
            if t ≠ T then  deallocate_multiple<t>(bid, alloc_bitmap) ;
    alloc_bitmap ← __shfl_sync(active, alloc_bitmap, leader);
    bid ← __shfl_sync(active, bid, leader);
    id_in_active ← popc(__lanemask_lt() & active);
```

Extended version of Alg. 1.
Implemented with CUDA
warp-level primitives.

[1] X. Huang, et. al. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. CIT 2010.

Additional Optimizations

- **Hierarchical Bitmaps:** Finding set bits in a large bitmap is slow. We can find bits in a hierarchical bitmap with a logarithmic number of accesses.
- **Allocation Request Coalescing:** A **leader** thread reserves object slots **on behalf of all allocating threads** in the warp.
- **Efficient Bit Operations:** Utilize bit-level **integer intrinsics** (e.g., *ffs*).

Find first set: Return index of first set bit in integer.

Additional Optimizations

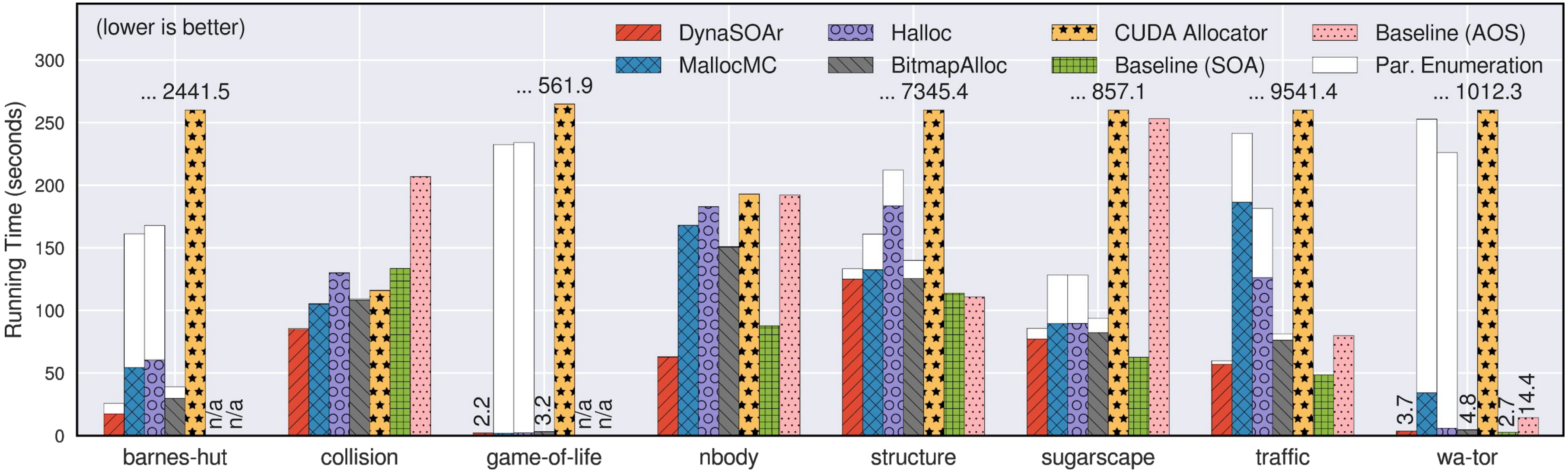
- **Hierarchical Bitmaps:** Finding set bits in a large bitmap is slow. We can find bits in a hierarchical bitmap with a logarithmic number of accesses.
- **Allocation Request Coalescing:** A **leader** thread reserves object slots **on behalf of all allocating threads** in the warp.
- **Efficient Bit Operations:** Utilize bit-level **integer intrinsics** (e.g., *ffs*).
- **Bitmap Rotation:** To reduce the probability of threads choosing the same bit, **rotate-shift bitmaps** before selecting a bit (i.e., before *ffs* etc.).

Related Work and Challenges

- DynaSOAr is an object allocator. **Other allocators request X number of bytes. We allocate structured data** (objects).
  - DynaSOAr is aware of the structure of its allocations → Better optimizations (SOA data layout)
- Main challenges
  - Low fragmentation through blocks states: Always allocate in active[T] blocks. This is **less efficient than hashing** (what other allocators do [1, 2]). Algorithms must be optimized!
  - **Safe memory reclamation:** When is it safe to delete a block?
    (We have may have many concurrent allocate/deallocate operations.)
  - (Eventual) **consistency between various internal data structures.**
    (e.g.: block states and block state bitmaps)

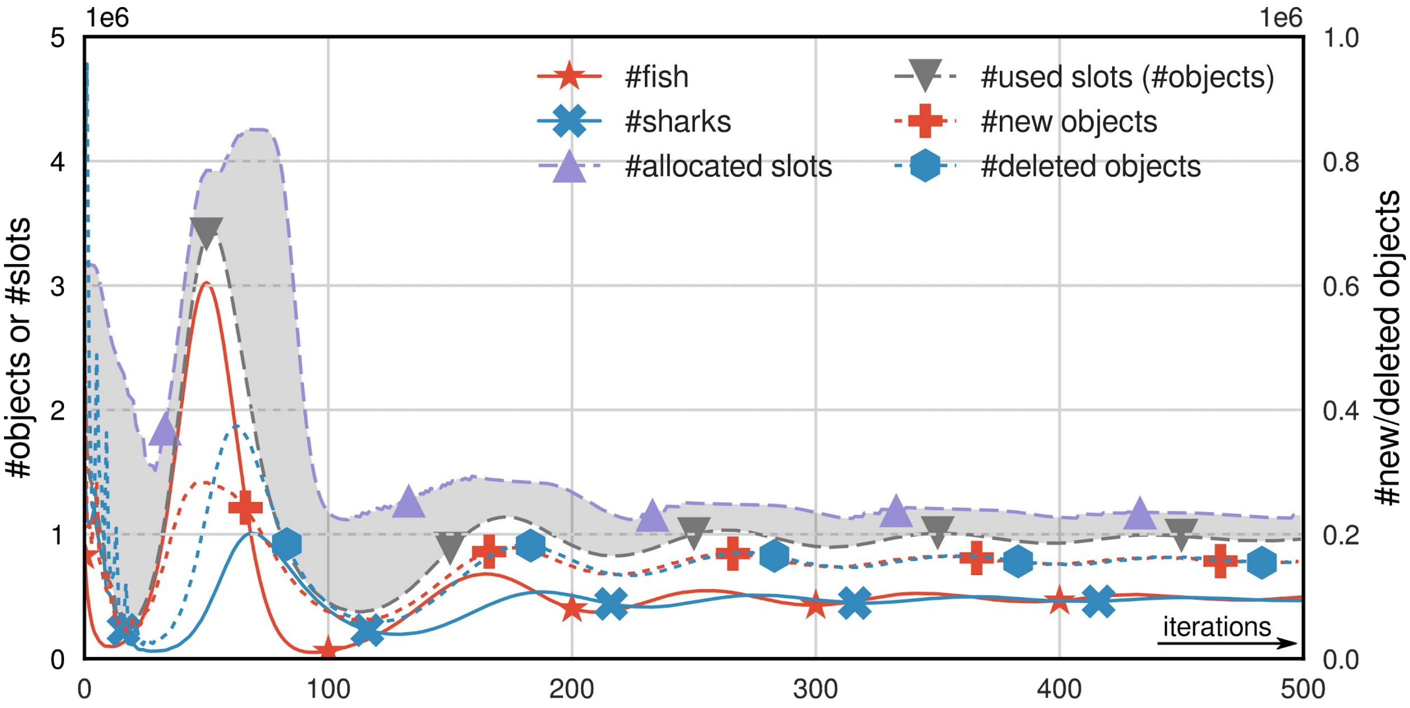
[1] A. V. Adinetz, D. Pleiter. Halloc: A High-Throughput Dynamic Memory Allocator for GPGPU Architectures. GPU Technology Conference 2014.
[2] M. Steinberger, M. Kenzel, B. Kainz, D. Schmalstieg. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. InPar 2012.

Benchmarks: Running Time



- *Baseline:* Without dynamic memory allocation

wa-tor Fragmentation

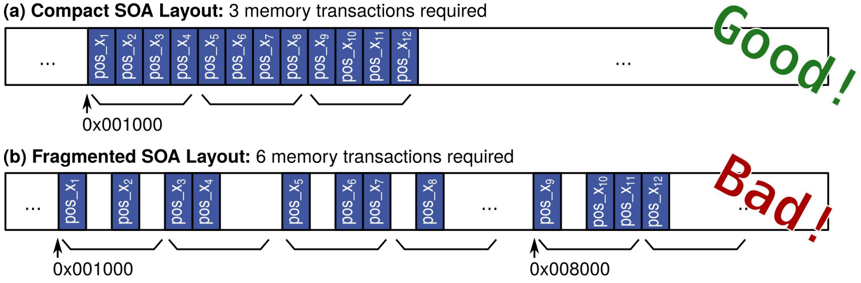


CompactGpu: GPU Memory Defragmentation [ISMM 2019]

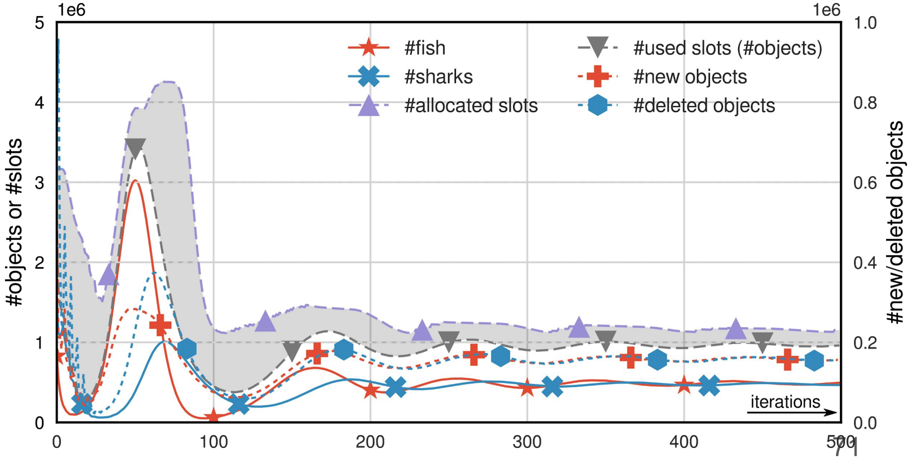
Why Memory Defragmentation?

- *Space Efficiency:* Lower overall memory consumption.
- *Performance:* Reading/writing compact, less fragmented data requires fewer memory access transactions.

$$F = \frac{\sum_{b \in Blocks} (N_{type(b)} - used(b))}{\sum_{b \in Blocks} N_{type(b)}} \approx \frac{1}{\#\text{blocks}} \sum_{b \in Blocks} \frac{\#\text{free slots}(b)}{\#\text{slots}(b)}$$


(a) Compact SOA Layout: 3 memory transactions required

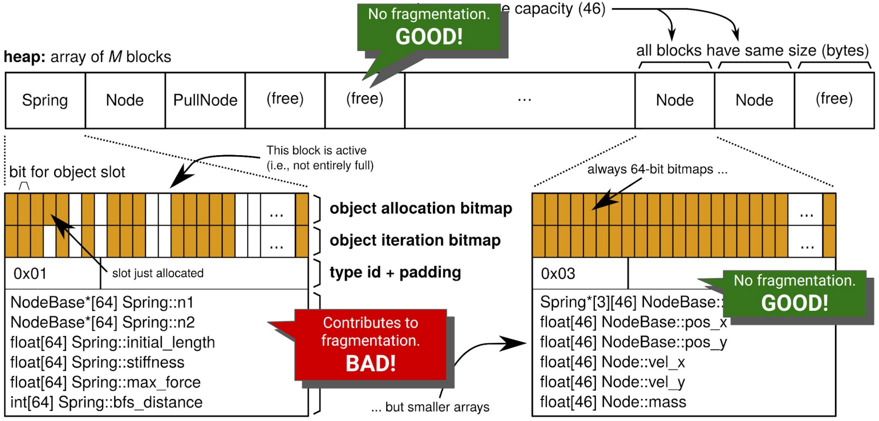
(b) Fragmented SOA Layout: 6 memory transactions required



Why Memory Defragmentation?

- *Space Efficiency:* Lower overall memory consumption.
- *Performance:* Reading/writing compact, less fragmented data requires fewer memory access transactions.

$$F = \frac{\sum_{b \in Blocks} (N_{type(b)} - used(b))}{\sum_{b \in Blocks} N_{type(b)}} \approx \frac{1}{\#\text{blocks}} \sum_{b \in Blocks} \frac{\#\text{free slots}(b)}{\#\text{slots}(b)}$$



No fragmentation. GOOD!

e capacity (46)

all blocks have same size (bytes)

heap: array of *M* blocks

Spring | Node | PullNode | (free) | (free) | ... | Node | Node | (free)

bit for object slot

This block is active (i.e., not entirely full)

always 64-bit bitmaps ...

object allocation bitmap

object iteration bitmap

type id + padding

0x01

slot just allocated

NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

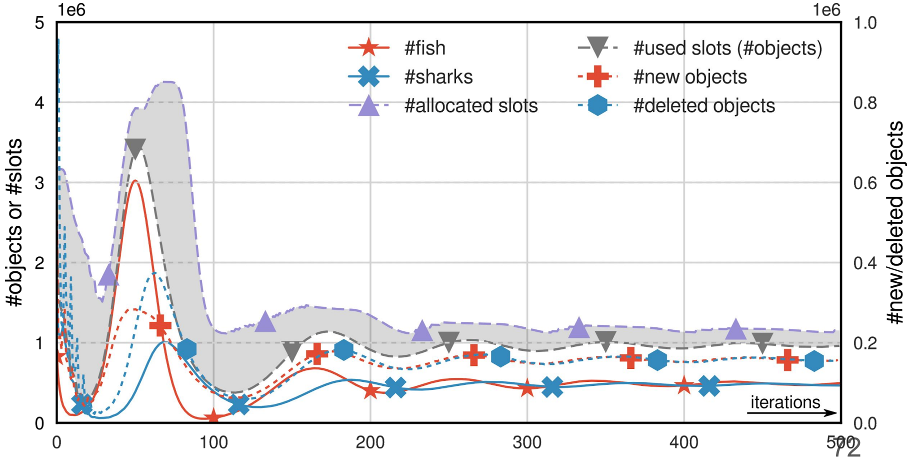
Contributes to fragmentation. BAD!

... but smaller arrays

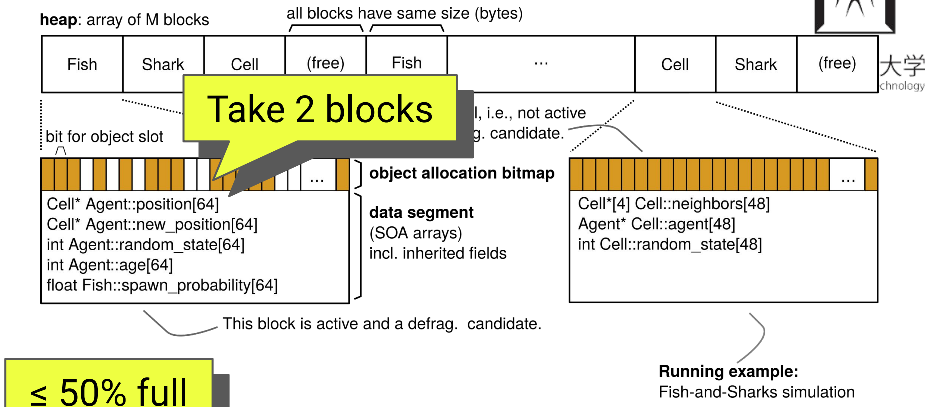
0x03

Spring*[3][46] NodeBase::
float[46] NodeBase::pos_x
float[46] NodeBase::pos_y
float[46] Node::vel_x
float[46] Node::vel_y
float[46] Node::mass

No fragmentation. GOOD!



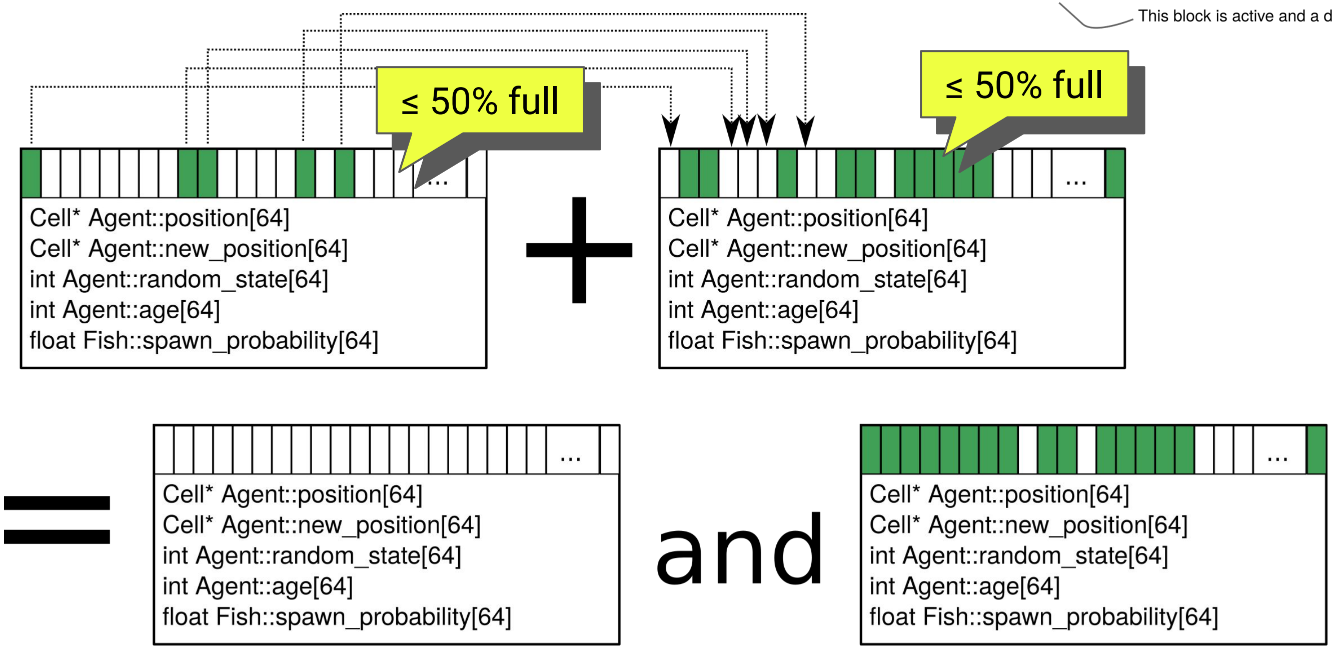
Block Merging: 1 + 1 = 1



Cell* Agent::position[64]
Cell* Agent::new_position[64]
int Agent::random_state[64]
int Agent::age[64]
float Fish::spawn_probability[64]

Cell*[4] Cell::neighbors[48]
Agent* Cell::agent[48]
int Cell::random_state[48]

Do this in parallel for all *eligible* blocks:



Cell* Agent::position[64]
Cell* Agent::new_position[64]
int Agent::random_state[64]
int Agent::age[64]
float Fish::spawn_probability[64]

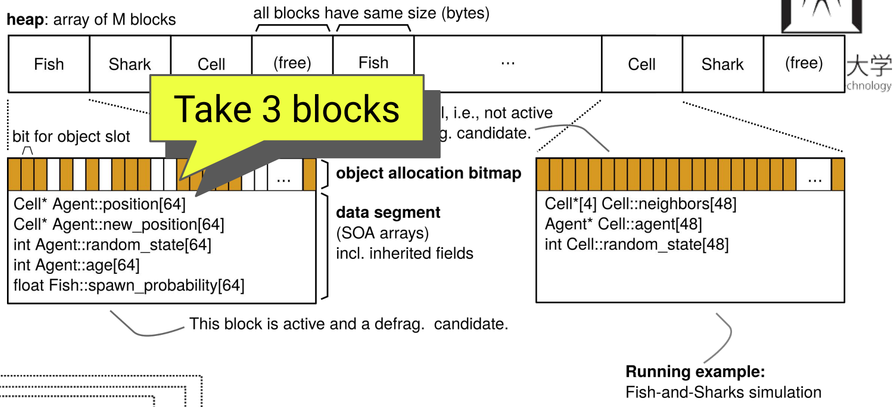
Cell* Agent::position[64]
Cell* Agent::new_position[64]
int Agent::random_state[64]
int Agent::age[64]
float Fish::spawn_probability[64]

Cell* Agent::position[64]
Cell* Agent::new_position[64]
int Agent::random_state[64]
int Agent::age[64]
float Fish::spawn_probability[64]

Cell* Agent::position[64]
Cell* Agent::new_position[64]
int Agent::random_state[64]
int Agent::age[64]
float Fish::spawn_probability[64]

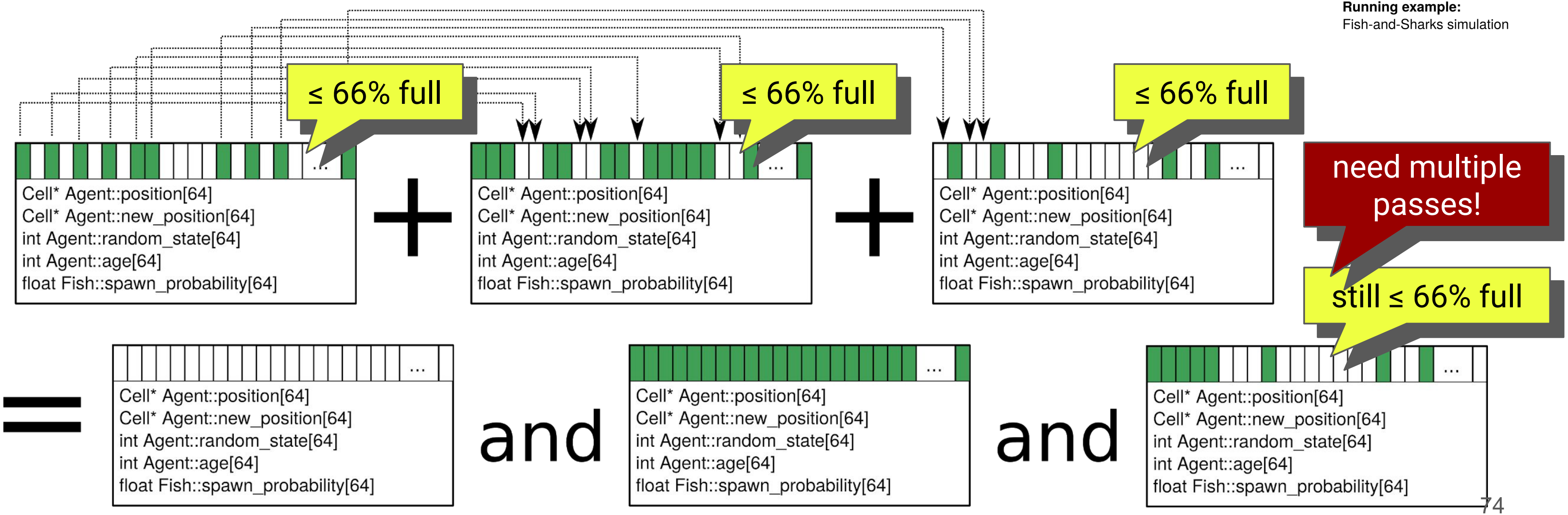
Block Merging: 1 + 2 = 2

Do this in parallel for all *eligible* blocks:



Cell* Agent::position[64]
Cell* Agent::new_position[64]
int Agent::random_state[64]
int Agent::age[64]
float Fish::spawn_probability[64]

Cell*[4] Cell::neighbors[48]
Agent* Cell::agent[48]
int Cell::random_state[48]

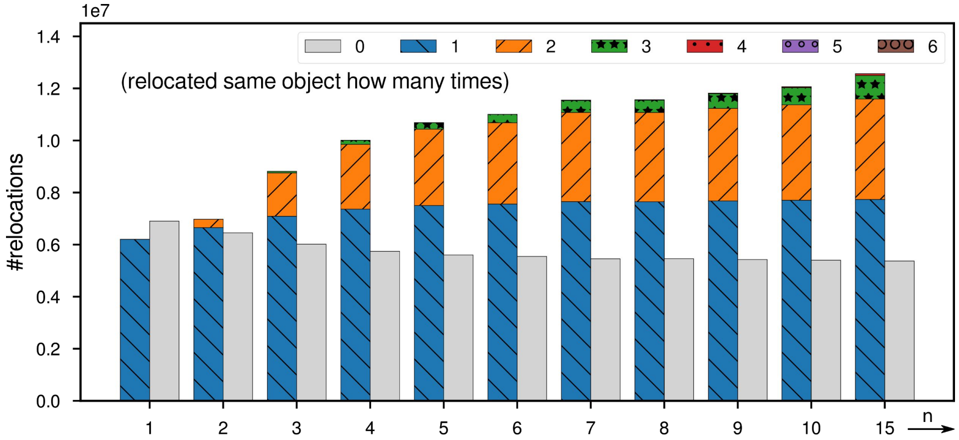


Cell* Agent::position[64]
Cell* Agent::new_position[64]
int Agent::random_state[64]
int Agent::age[64]
float Fish::spawn_probability[64]

Block Merging: 1 + n = n

- S_1 can be merged into T_1 if S_1 and T_1 are ≤ 50% full.
- S_1 can be merged into T_1, T_2 if S_1, T_1, T_2 are ≤ 66.6% full.
- S_1 can be merged into T_1, …, T_n if S_1, T_1, …, T_n are ≤ n/(n+1) full.

- **Defragmentation factor *n*** can be configured.
  - Higher *n*: Better defrag. guarantees.
  - Lower *n*: A bit faster, fewer passes.
- Blocks that are ≤ n/(n+1) full are **defrag. candidates** (*eligible*).



Handout only: Defragmentation by Block Merging

- *After defragmentation:*
  - All blocks with fill level ≤ $n/(n+1)$ are gone.
  - Only blocks with fill level > $n/(n+1)$ are left over.
  - Therefore, fragmentation is ≤ $1 - n/(n+1) = 1/(n+1)$.
- One defragmentation pass eliminates all source blocks: $1/(n+1)$ of all defragmentation candidates.
  - To eliminate all defragmentation candidates, we need $\log_{(n+1)/n}$ *#candidates* many passes.

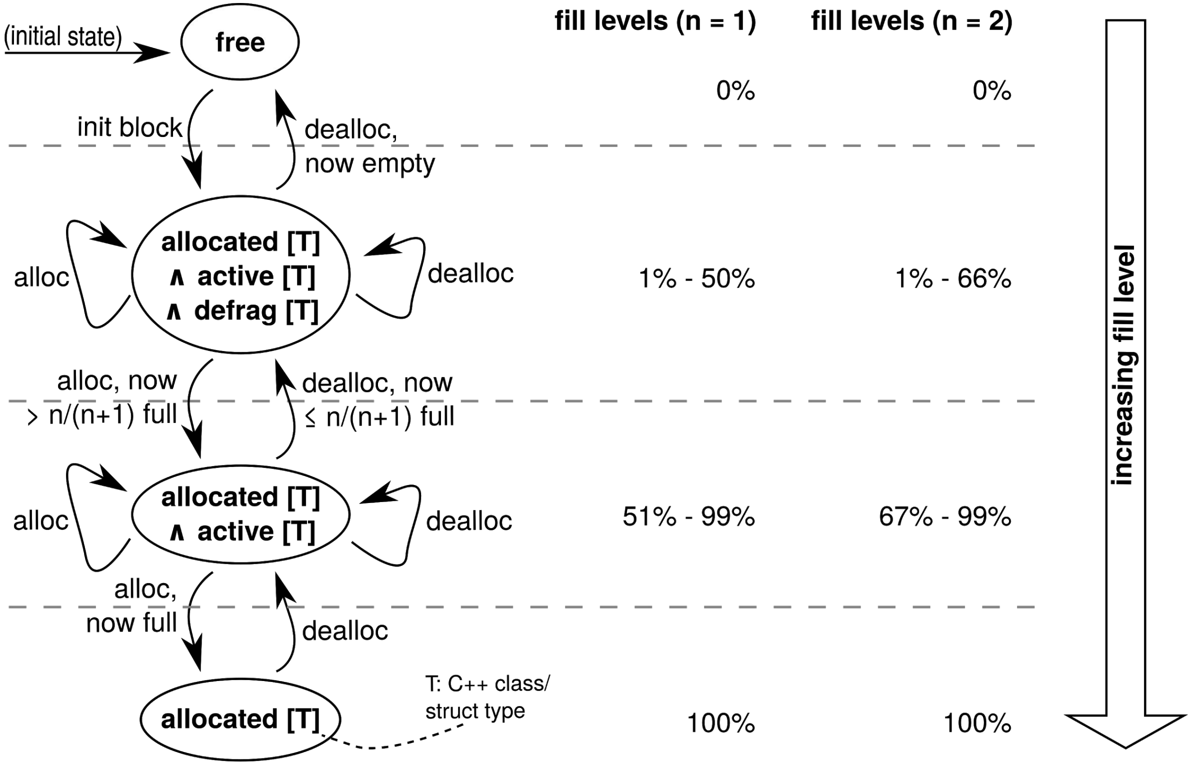
Handout only: Defragmentation by Block Merging

- Why do we require that all *n* blocks are ≤ *n/(n+1)* full instead of all blocks together ≤ 100% full?
  - Makes it easier to identifier blocks that contribute to defragmentation.
  - More uniform control flow (similar number of object relocations).
- Is there a better way to choose source/target blocks?
  - Defragmentation candidate state is encoded in **only 1 bit**, so no, unless we use more than 1 bit.
  - Even then, unlikely to result in faster defragmentation because there would be **more control flow divergence**.
  - See discussion in thesis.

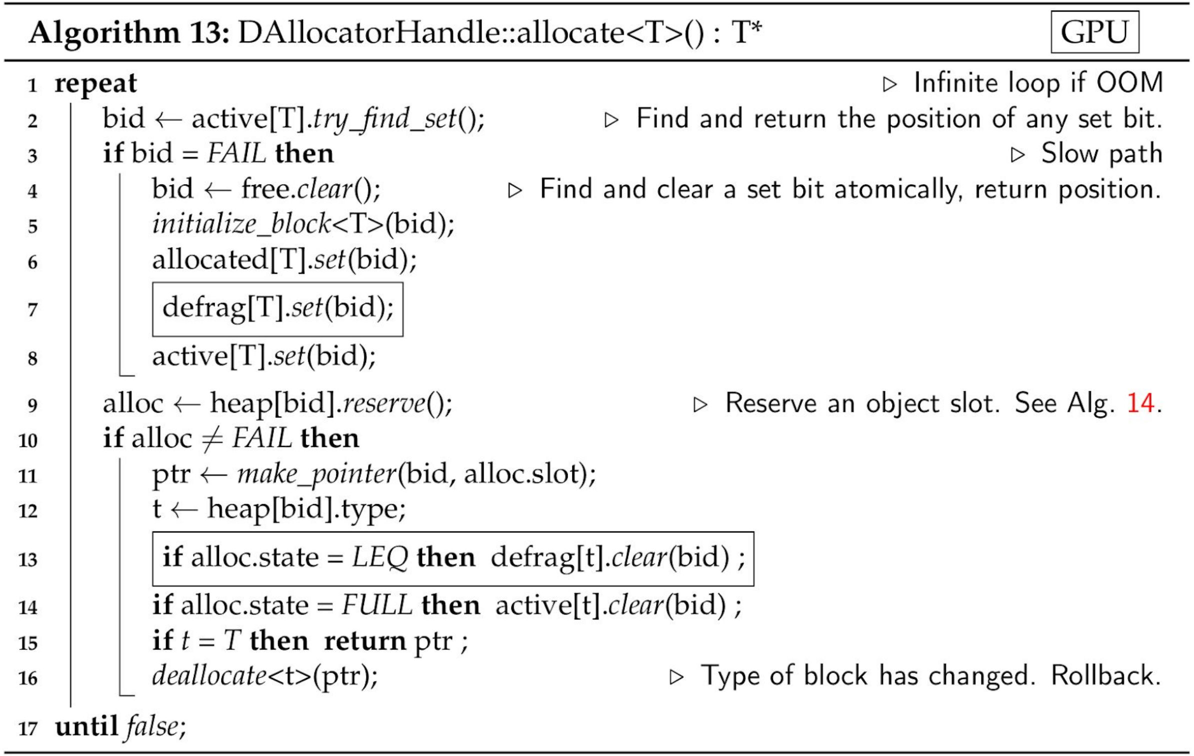
Handout only: CompactGpu is...

- **configurable:** Target fragmentation rate can be tuned.
- **in-place:** No auxiliary storage necessary. Entire heap remains useable.
- **incremental:** A single defragmentation pass is fast and compacts only a fraction of the heap. Multiple passes are required for full defragmentation.
- **a stop-the-world approach**
- **fully parallel:** Every step is a perfectly parallel CUDA kernel.
- **no order-preserving:** Objects may be arranged in a different order.

Extension of DynaSOAr Block States



Keeping Track of Defragmentation Candidates



Algorithm 13: DAllocatorHandle::allocate<T>() : T* GPU

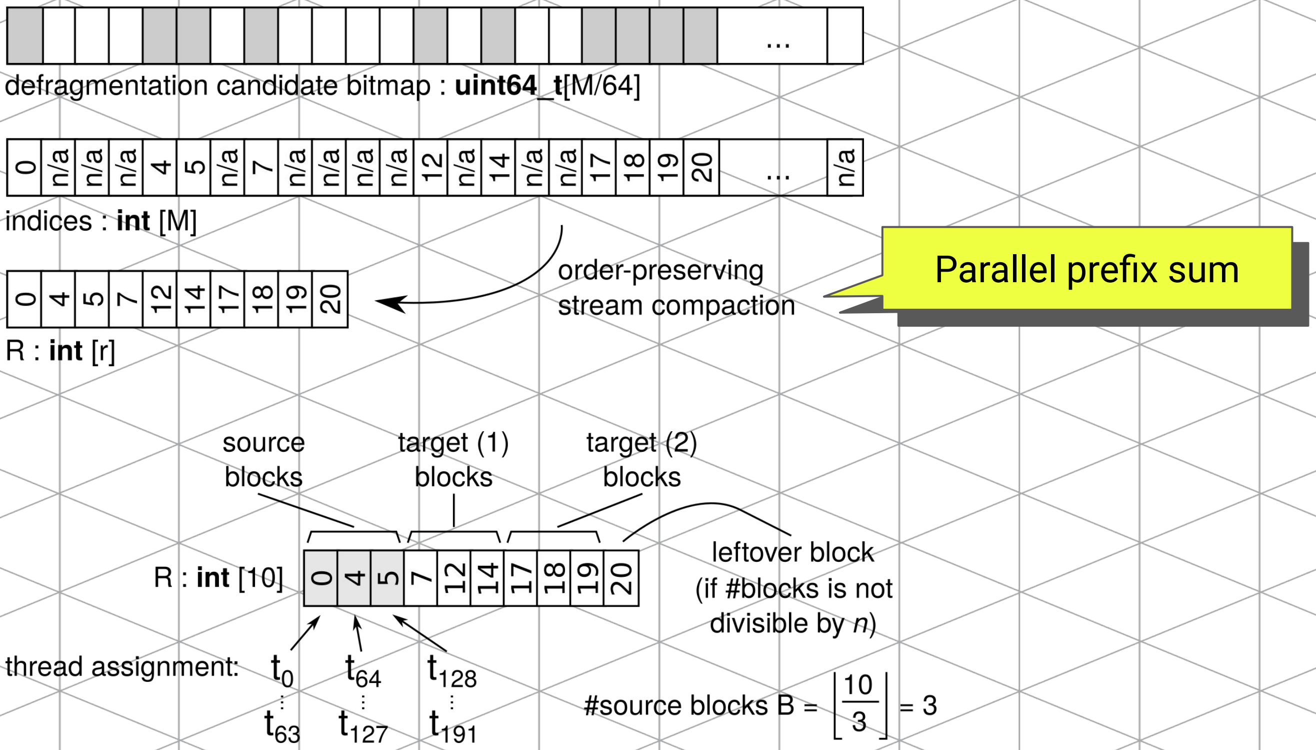
```
repeat                                                      ▷ Infinite loop if OOM
    bid ← active[T].try_find_set();          ▷ Find and return the position of any set bit.
    if bid = FAIL then                                              ▷ Slow path
        bid ← free.clear();          ▷ Find and clear a set bit atomically, return position.
        initialize_block<T>(bid);
        allocated[T].set(bid);
        defrag[T].set(bid);
        active[T].set(bid);
    alloc ← heap[bid].reserve();                   ▷ Reserve an object slot. See Alg. 14.
    if alloc ≠ FAIL then
        ptr ← make_pointer(bid, alloc.slot);
        t ← heap[bid].type;
        if alloc.state = LEQ then  defrag[t].clear(bid) ;
        if alloc.state = FULL then  active[t].clear(bid) ;
        if t = T then  return ptr ;
        deallocate<t>(ptr);                       ▷ Type of block has changed. Rollback.
until false;
```

Defragmentation Overview

- Defragmentation is **manual**: Programmer has to initiate defragmentation.
- Programmer **specifies the C++ type** that should be defragmented.

1. Choose source/target blocks (parallel prefix sum).
2. Copy objects from source to target blocks (very efficient due to SOA layout).
3. Store **forwarding pointer**s in old locations.
4. Scan the heap and rewrite pointers to old locations.
   (Fast due to optimizations that reduce #memory accesses.)
5. Update block state bitmaps.

Step 1: Choose Source/Target Blocks



▨ □ □ □ ▨ ▨ □ ▨ □ □ □ □ ▨ □ ▨ □ □ ▨ ▨ ▨ ▨ ... □

defragmentation candidate bitmap : **uint64_t**[M/64]

0 n/a n/a n/a 4 5 n/a 7 n/a n/a n/a n/a 12 n/a 14 n/a n/a 17 18 19 20 ... n/a

indices : **int** [M]

order-preserving stream compaction

Parallel prefix sum

0 4 5 7 12 14 17 18 19 20

R : **int** [r]

source blocks

target (1) blocks

target (2) blocks

R : **int** [10]

0 4 5 7 12 14 17 18 19 20

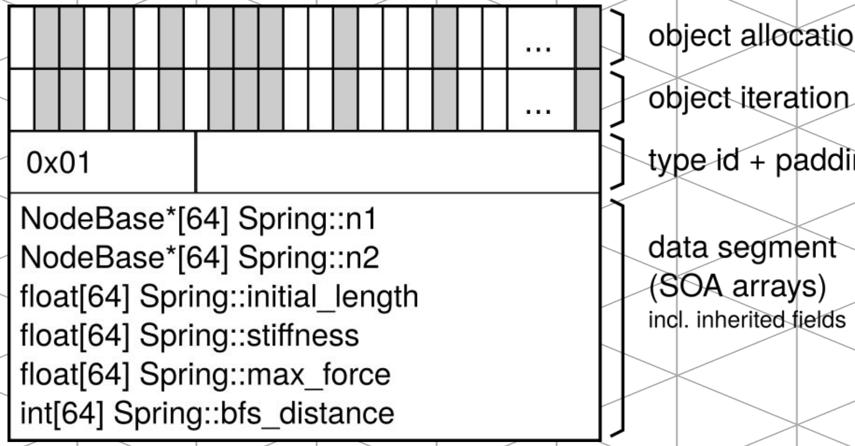
leftover block (if #blocks is not divisible by *n*)

thread assignment: $t_0 \vdots t_{63}$ $t_{64} \vdots t_{127}$ $t_{128} \vdots t_{191}$

#source blocks B = $\left\lfloor \frac{10}{3} \right\rfloor$ = 3

Step 2: Copy Objects

- *Fully parallel:* One thread per source object slot
- *No synchronization necessary:* Every thread can compute its source/target object slot/block index based on ***R*****, thread ID and object allocation bitmaps**.



0x01

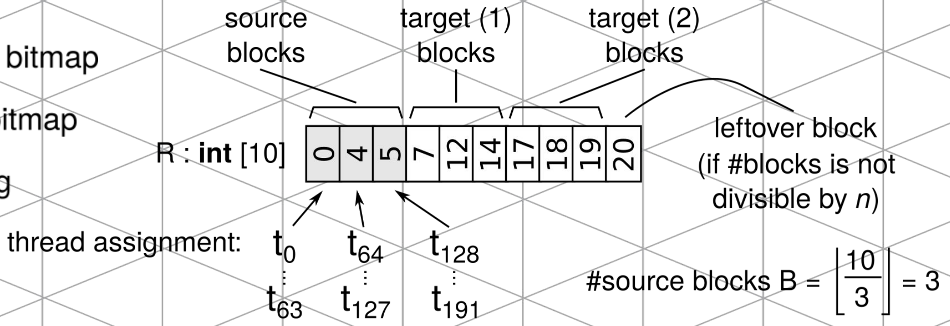
NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance

object allocation bitmap

object iteration bitmap

type id + padding

data segment
(SOA arrays)
incl. inherited fields



source blocks

target (1) blocks

target (2) blocks

R : **int** [10]

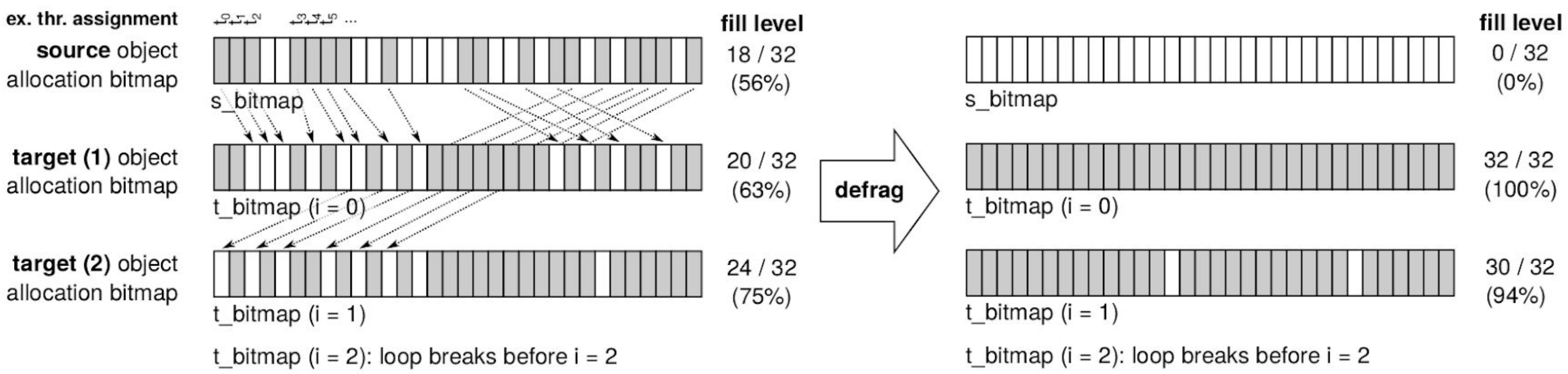
0	4	5	7	12	14	17	18	19	20

leftover block
(if #blocks is not
divisible by *n*)

thread assignment: $t_0 \vdots t_{63}$ $t_{64} \vdots t_{127}$ $t_{128} \vdots t_{191}$

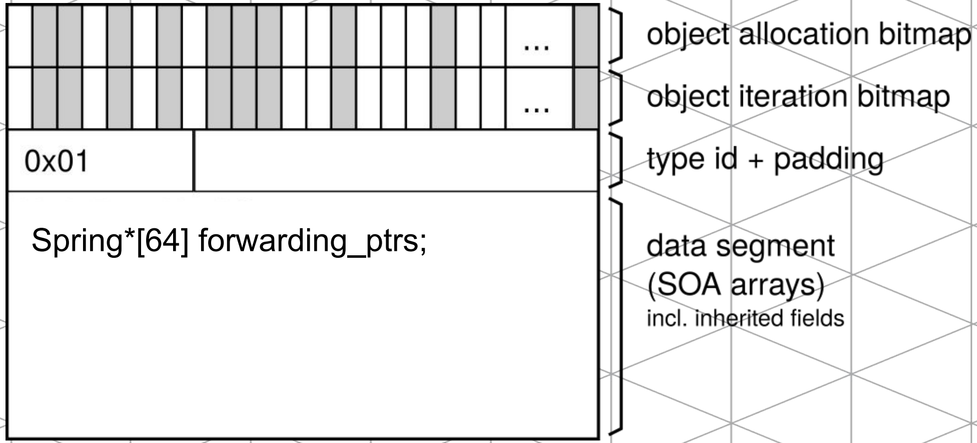
#source blocks $B = \left\lfloor \frac{10}{3} \right\rfloor = 3$

Step 2: Copy Objects


ex. thr. assignment
t_0 t_1 t_2 t_3 t_4 t_5 ...
source object allocation bitmap
s_bitmap
target (1) object allocation bitmap
t_bitmap (i = 0)
target (2) object allocation bitmap
t_bitmap (i = 1)
t_bitmap (i = 2): loop breaks before i = 2
fill level
18 / 32 (56%)
20 / 32 (63%)
24 / 32 (75%)
defrag
s_bitmap
t_bitmap (i = 0)
t_bitmap (i = 1)
t_bitmap (i = 2): loop breaks before i = 2
fill level
0 / 32 (0%)
32 / 32 (100%)
30 / 32 (94%)

Step 3: Store Forwarding Ptrs. in Source Blocks

- Overwrite data segment of source blocks with forwarding pointers.

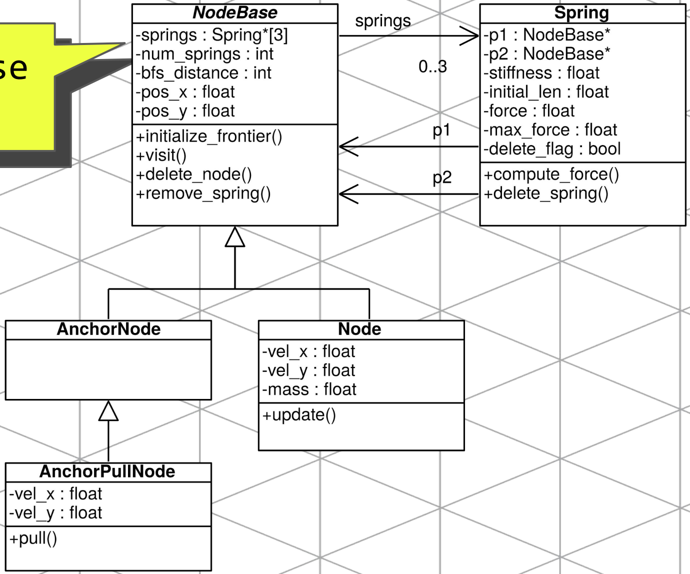


Step 4: Rewrite Pointers to Relocated Objects

- *Conceptually:* A parallel do-all operation

```
parallel_do<NodeBase, &AllocatorT::Base::rewrite_field<NodeBase, 0>>()
```

First field (idx. 0) of `NodeBase` has type `Spring*[3]`.



NodeBase
-springs : Spring*[3]
-num_springs : int
-bfs_distance : int
-pos_x : float
-pos_y : float
+initialize_frontier()
+visit()
+delete_node()
+remove_spring()

Spring
-p1 : NodeBase*
-p2 : NodeBase*
-stiffness : float
-initial_len : float
-force : float
-max_force : float
-delete_flag : bool
+compute_force()
+delete_spring()

AnchorNode

Node
-vel_x : float
-vel_y : float
-mass : float
+update()

AnchorPullNode
-vel_x : float
-vel_y : float
+pull()

- We are rewriting every field that could potentially have a pointer to a relocated object.
- *Discussion:* C++ Boehm GC [1]

[1] H. J. Boehm. Space Efficient Conservative Garbage Collection. PLDI 1993.

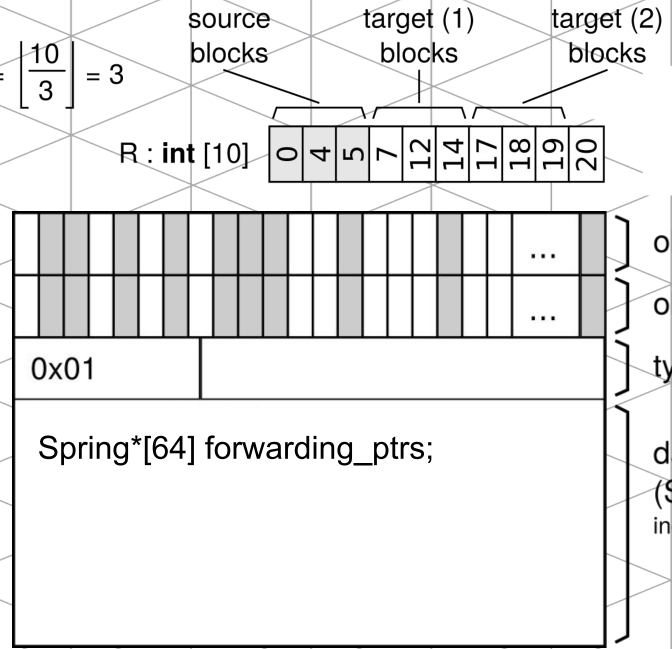
Step 4: Rewrite Pointers to Relocated Objects

- *Conceptually:* A parallel do-all operation

```
parallel_do<NodeBase, &AllocatorT::Base::rewrite_field<NodeBase, 0>>()
```

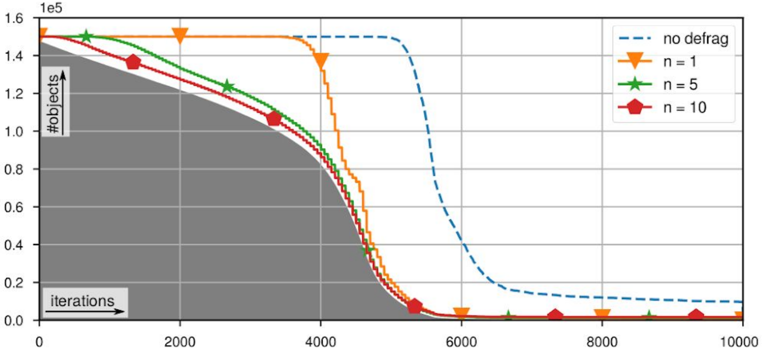
```
template<typename T, int Idx>
void AllocatorT_Base::rewrite_field {
    void** addr = &get_field<Idx>();
    int s_bid = extract_bid(*ptr);

    if (s_bid < R[B] && defrag[T][s_bid]) {
        int s_oid = extract_oid(*ptr);
        *ptr = heap[s_bid].data.forwarding_ptrs[s_oid];
    }
}
```

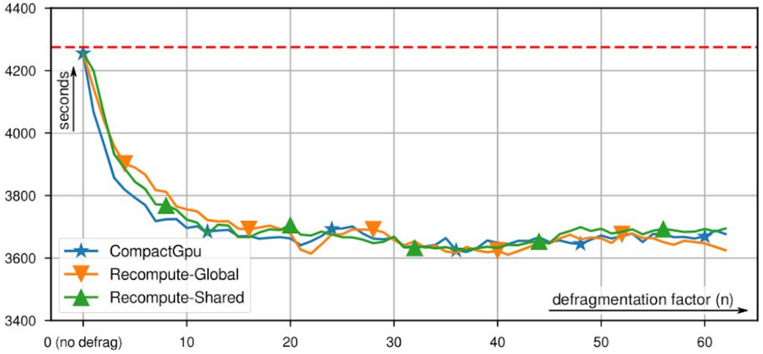
$$\#\text{source blocks } B = \left\lfloor \frac{10}{3} \right\rfloor = 3$$



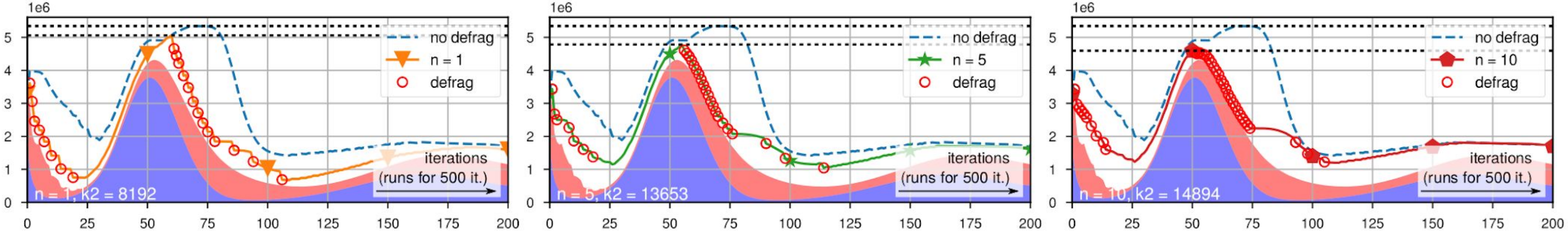
Experimental Results

collision





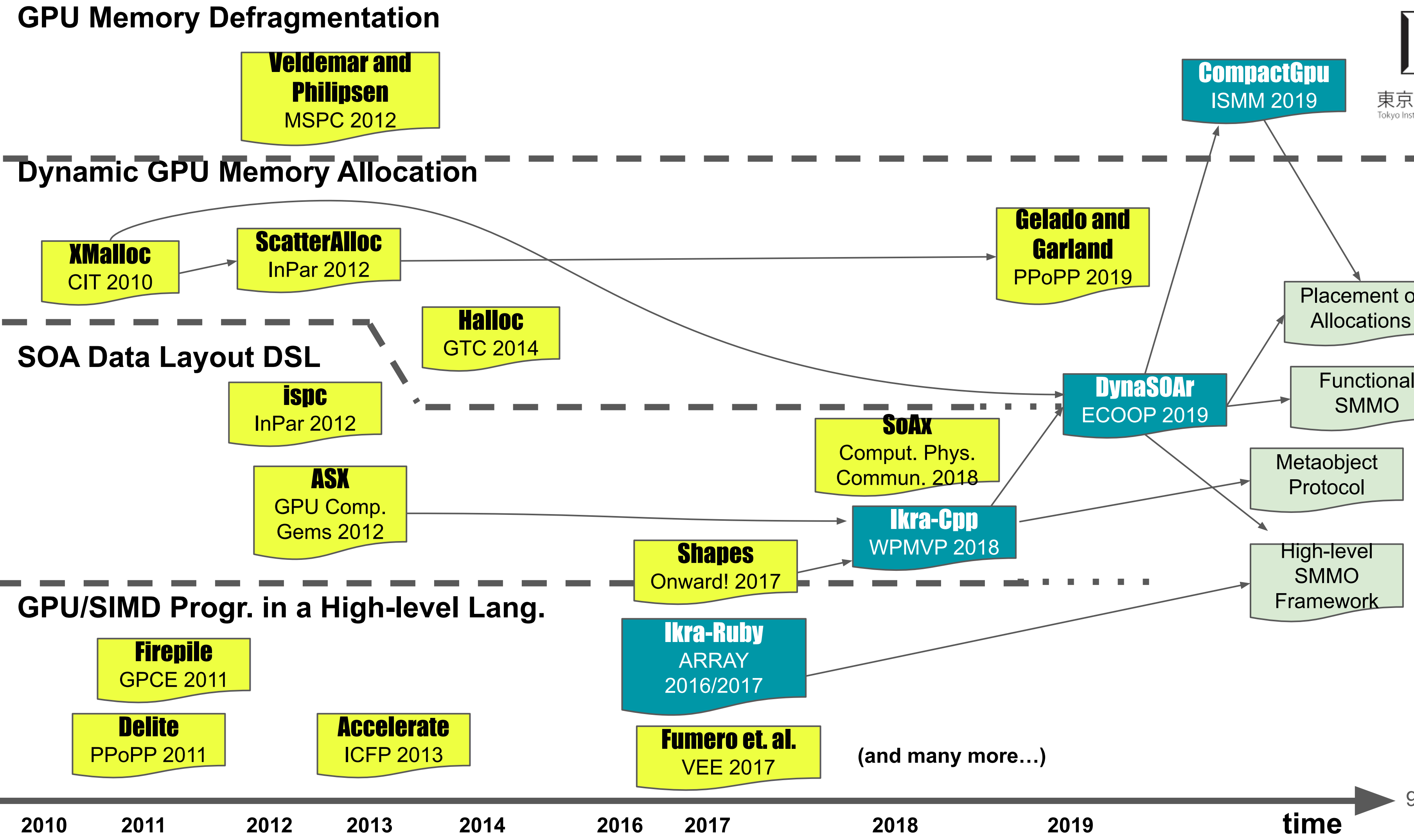
wa-tor



Conclusion

Conclusion

- Object-oriented programming is **not slow if properly optimized**.
- This thesis: 3 memory access optimizations, eliminating OOP overhead.
  - An embedded **SOA data layout DSL** for C++/CUDA.
  - *DynaSOAr:* A **dynamic memory allocator** with efficient memory access.
  - *CompactGpu:* A **memory defragmentation system** for GPUs, bringing performance of dynamically allocated memory accesses closer to SOA layout performance.
- Potential future work
  - Integrate Ikra-Cpp into a **high-level language** (e.g., as part of Ikra-Ruby). (*Note:* Many high-level language have a garbage collector!)
  - Explore if/how SMMO can be extended to a **functional OOP** style.
  - Give programmers more **control over data placement** of dynamic allocations.
  - Develop a **metaobject protocol** based on Ikra-Cpp's data layout DSL.



GPU Memory Defragmentation

Veldemar and Philipsen
MSPC 2012

CompactGpu
ISMM 2019

Dynamic GPU Memory Allocation

XMalloc
CIT 2010

ScatterAlloc
InPar 2012

Gelado and Garland
PPoPP 2019

Halloc
GTC 2014

SOA Data Layout DSL

ispc
InPar 2012

ASX
GPU Comp. Gems 2012

SoAx
Comput. Phys. Commun. 2018

Shapes
Onward! 2017

Ikra-Cpp
WPMVP 2018

DynaSOAr
ECOOP 2019

Placement of Allocations

Functional SMMO

Metaobject Protocol

High-level SMMO Framework

GPU/SIMD Progr. in a High-level Lang.

Firepile
GPCE 2011

Delite
PPoPP 2011

Accelerate
ICFP 2013

Ikra-Ruby
ARRAY 2016/2017

Fumero et. al.
VEE 2017

(and many more…)

Main Contributions of this Thesis

- The SMMO (**Single-Methods Multiple-Objects**) programming model and eight SMMO example applications.
- An embedded SOA **data layout DSL** in C++/CUDA.
- An extension of the SOA data layout to **dynamic object set sizes**.
  Technically, this is no longer an SOA layout, but it has the same performance characteristics.
- **DynaSOAr:** A lock-free, hierarchical GPU memory allocator; the first one with a custom object layout.
- A lock-free, hierarchical **bitmap data structure**.
- **CompactGpu:** An efficient memory defragmentation system for GPUs.

Future Research Directions

- Is SMMO suitable for **garbage collected languages**?

  In SMMO, we run a method for all heap-allocated objects. These objects are not necessarily reachable from other objects and a GC may delete them.
- Can SMMO be generalized to **functional OOP** [1, 2]?

  In functional OOP, the state of objects is immutable. Changing a field of an object results in a new object. We would require a `parallel_map` instead of a `parallel_do`. How does this affect object allocation? Furthermore, how easy/intuitive will such a programming model be for programmers?
- Can we give programmers more control over the **placement of allocations**?

  This could improve memory coalescing and cache utilization but it is a tedious job.

  Possible direction: Let programmers provide a comparator function (as used in sorting) and use it to select active blocks. We would need to keep more blocks active than before, thus increasing fragmentation.
- Can Ikra-Cpp's DSL be extended to a fully-fledged **metaobject protocol** [3]?

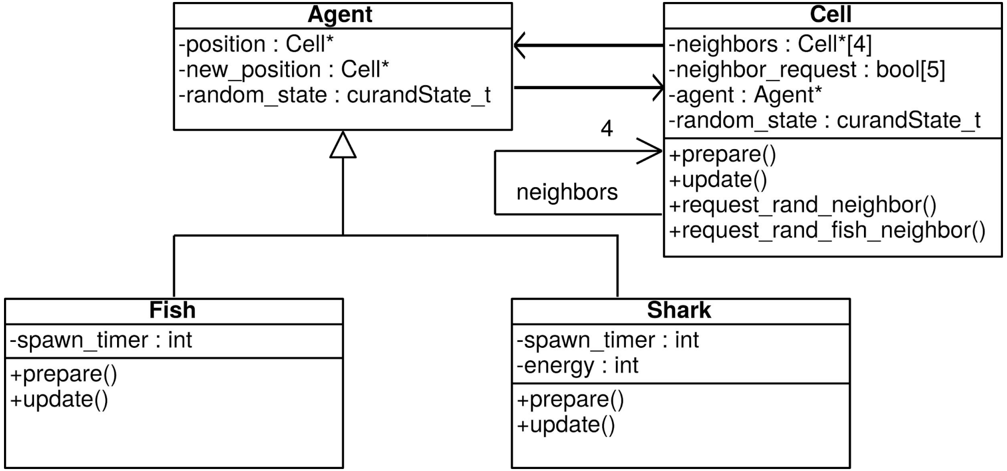
[1] M. Felleisen. Functional Objects. In: ECOOP 2004.
[2] K. Emoto, K. Matsuzaki, Z. Hu, A. Morihata, H. Iwasaki. Think Like a Vertex, Behave Like a Function! A Functional DSL for Vertex-Centric Big Graph Processing. In: ICFP 2016.
[3] S. Chiba. A Metaobject Protocol for C++. In: OOPSLA 1995.

Backup Slides

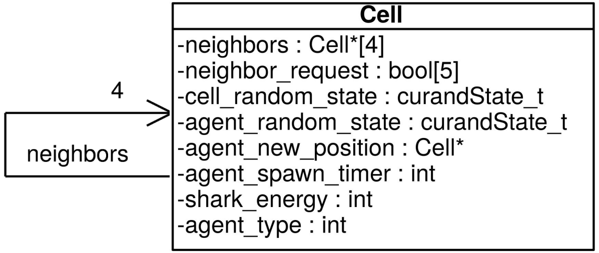
What are the Benefits of OOP?

- Many applications have an **inherent object structure** (e.g., in agent-based modelling). We want the code to reflect this structure.
  Benefits: abstraction, encapsulation, inheritance, …
- Code is **more readable** compared to a hand-written SOA layout, e.g.:
  - OOP: `parent_->children_[child_index_] = single_child;`
  - SOA: `TreeNode_children[TreeNode_child_idx[id]][TreeNode_parent[id]] = single_child;`
- Without **dynamic memory allocation**, programmers must maintain an `inactive` bit for deleted object or entirely rewrite the application (or implement their own allocator). See wa-tor example in the thesis.
- Richer **type information**: Type checker can **detect programming mistakes** earlier and programmers do not have to maintain type IDs (see barnes-hut).

wa-tor with/without OOP/Dyn. Mem. Allocation



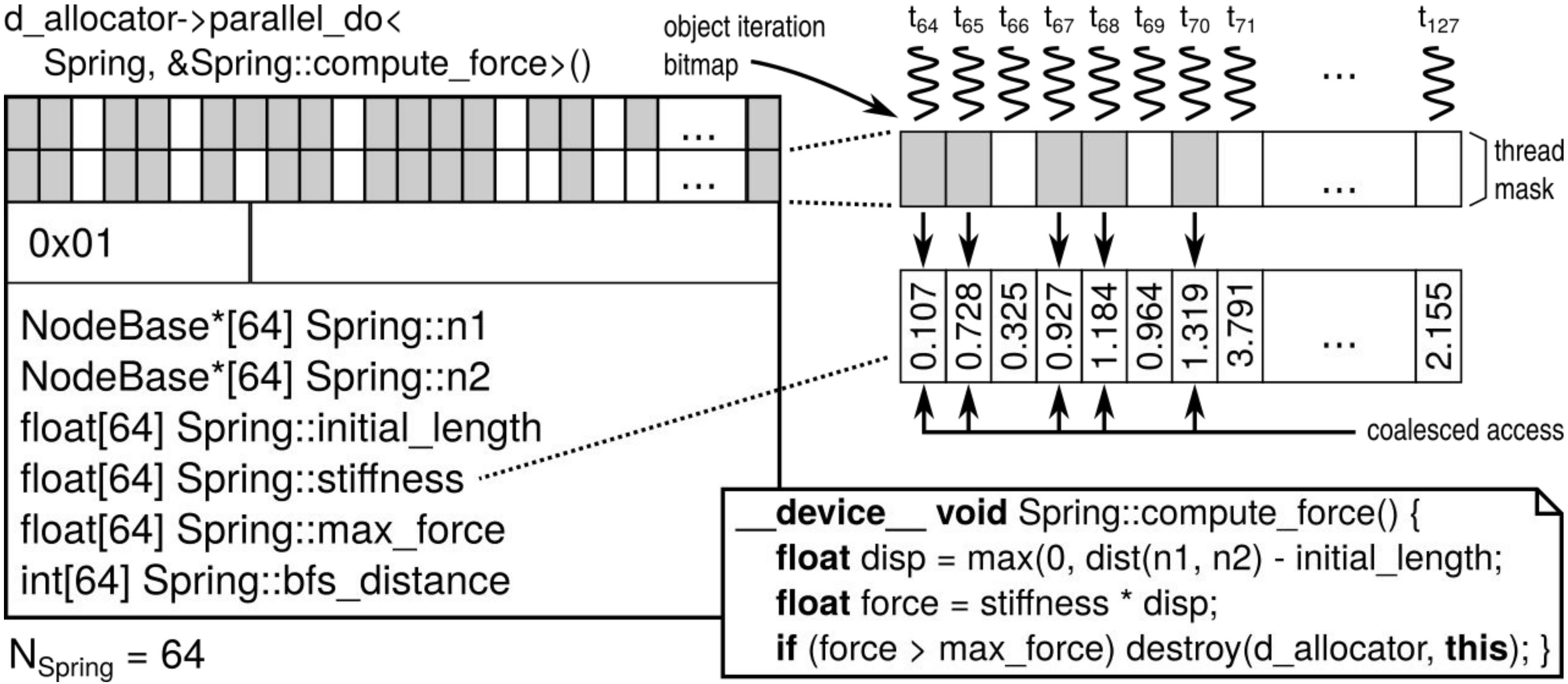
(a) with dyn. alloc.


Cell
-neighbors : Cell*[4]
-neighbor_request : bool[5]
-cell_random_state : curandState_t
-agent_random_state : curandState_t
-agent_new_position : Cell*
-agent_spawn_timer : int
-shark_energy : int
-agent_type : int
4
neighbors

(b) without dyn. alloc. (methods omitted)

- All fields are merged into a **single structure** in (b).
- The structure/network of cells is fixed, so they can be **statically allocated**.

Thread Assignment during `parallel_do`



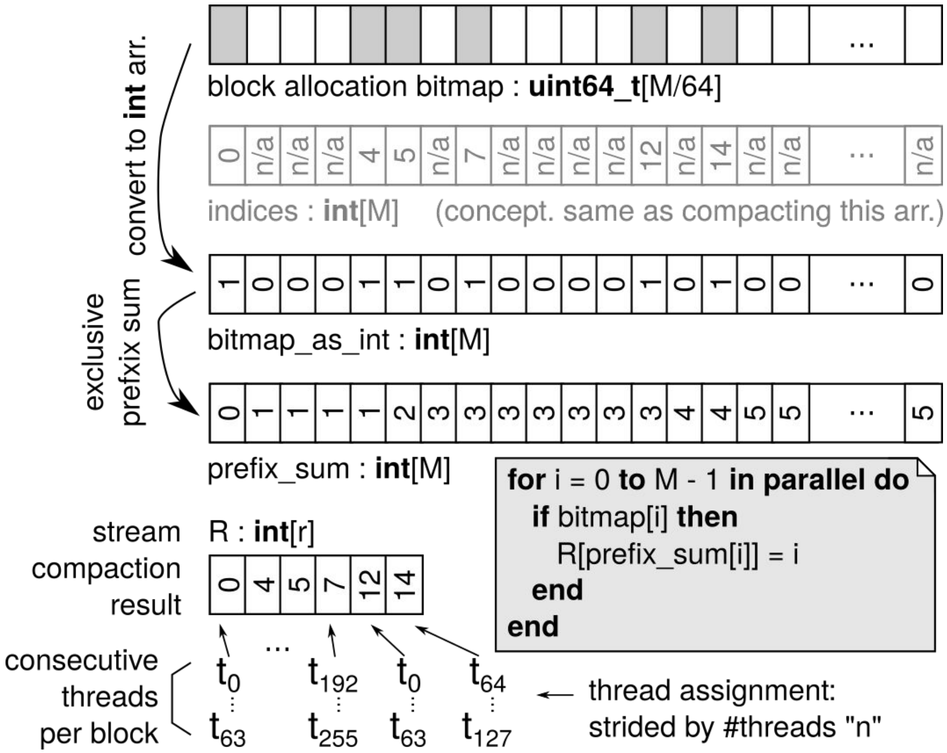
```
d_allocator->parallel_do<
  Spring, &Spring::compute_force>()
```

```
NodeBase*[64] Spring::n1
NodeBase*[64] Spring::n2
float[64] Spring::initial_length
float[64] Spring::stiffness
float[64] Spring::max_force
int[64] Spring::bfs_distance
```

$N_{Spring} = 64$

```
__device__ void Spring::compute_force() {
  float disp = max(0, dist(n1, n2) - initial_length;
  float force = stiffness * disp;
  if (force > max_force) destroy(d_allocator, this); }
```

Thread Assignment during `parallel_do`



indices	0	n/a	n/a	n/a	4	5	n/a	7	n/a	n/a	n/a	n/a	12	n/a	14	n/a	n/a	…	n/a
bitmap_as_int	1	0	0	0	1	1	0	1	0	0	0	0	1	0	1	0	0	…	0
prefix_sum	0	1	1	1	1	2	3	3	3	3	3	3	3	4	4	5	5	…	5
R	0	4	5	7	12	14													

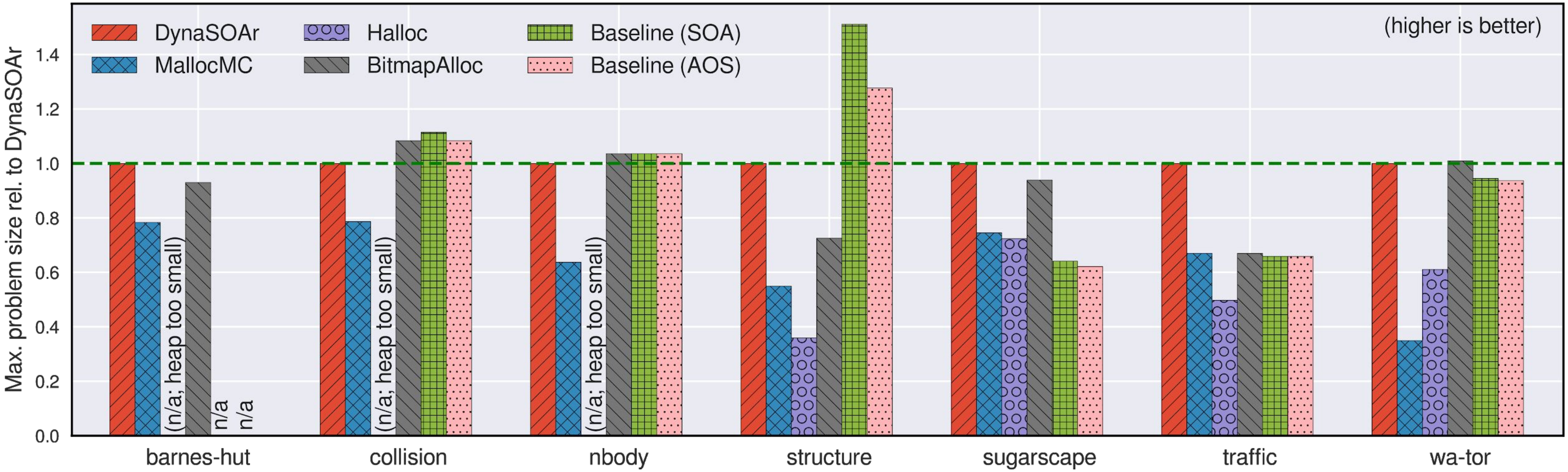
```
for i = 0 to M - 1 in parallel do
  if bitmap[i] then
    R[prefix_sum[i]] = i
  end
end
```

- Same algorithm is used for selecting source blocks in CompactGpu.

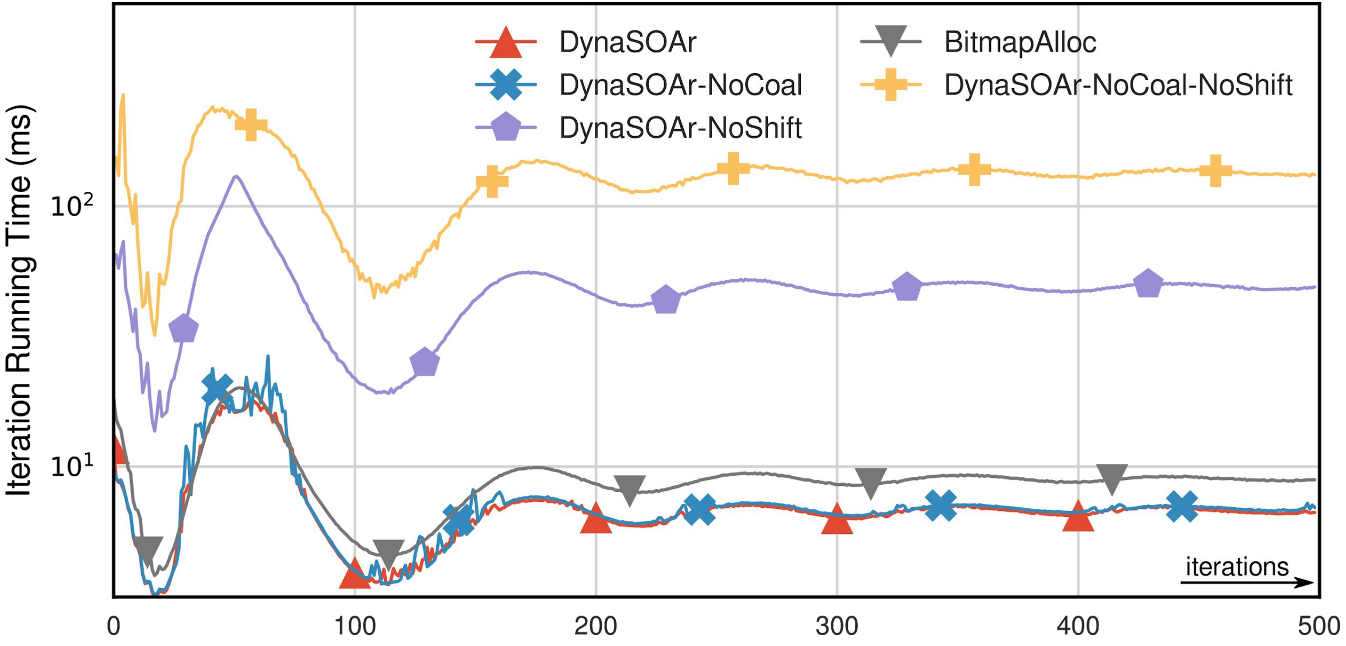
Additional DynaSOAr Optimizations

- **Hierarchical Bitmaps:** Finding set bits in a large bitmap is slow. We can find bits in a hierarchical bitmap with a logarithmic number of accesses.
- **Allocation Request Coalescing:** A **leader** thread reserves object slots **on behalf of all allocating threads** in the warp.
- **Efficient Bit Operations:** Utilize bit-level **integer intrinsics** (e.g., *ffs*).
- **Bitmap Rotation:** To reduce the probability of threads choosing the same bit, **rotate-shift bitmaps** before selecting a bit (i.e., before *ffs* etc.).
- **Retry Active Block Lookups:** If no active block could be found (e.g., due to bitmap inconsistencies), **retry** for a constant number of times.

Benchmarks: Space Efficiency



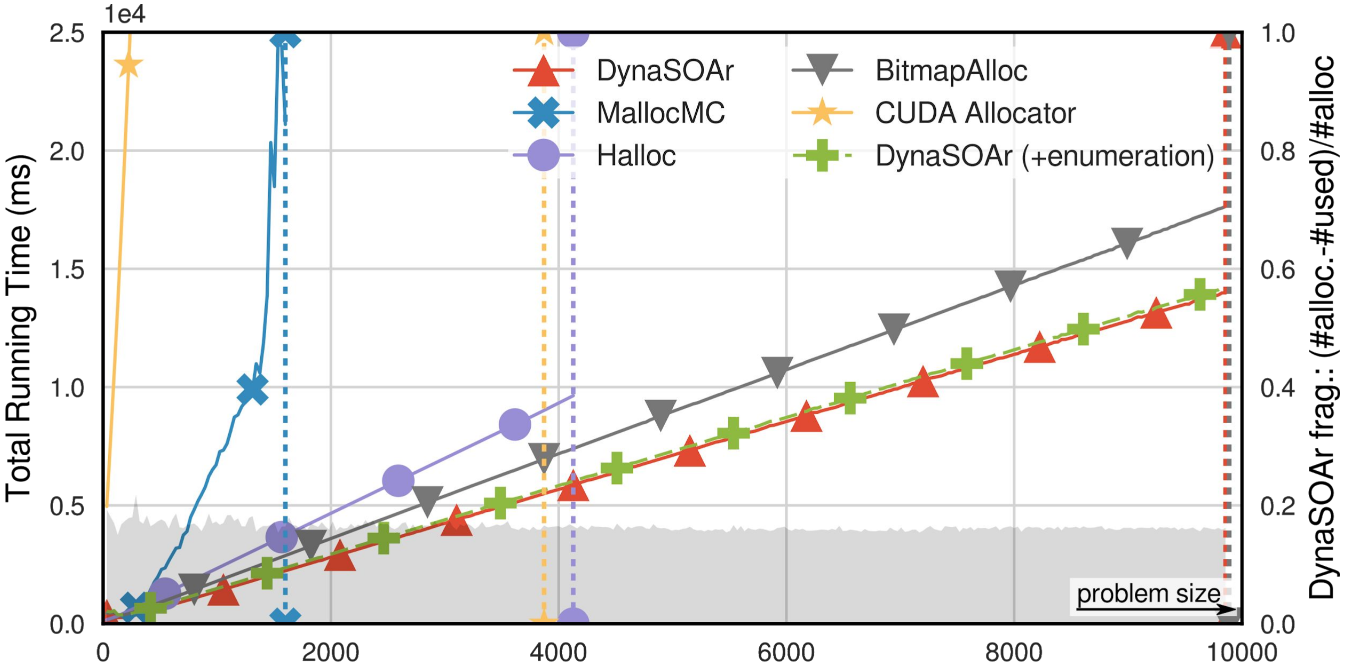
wa-tor: Pinpointing DynaSOAr’s Speedup



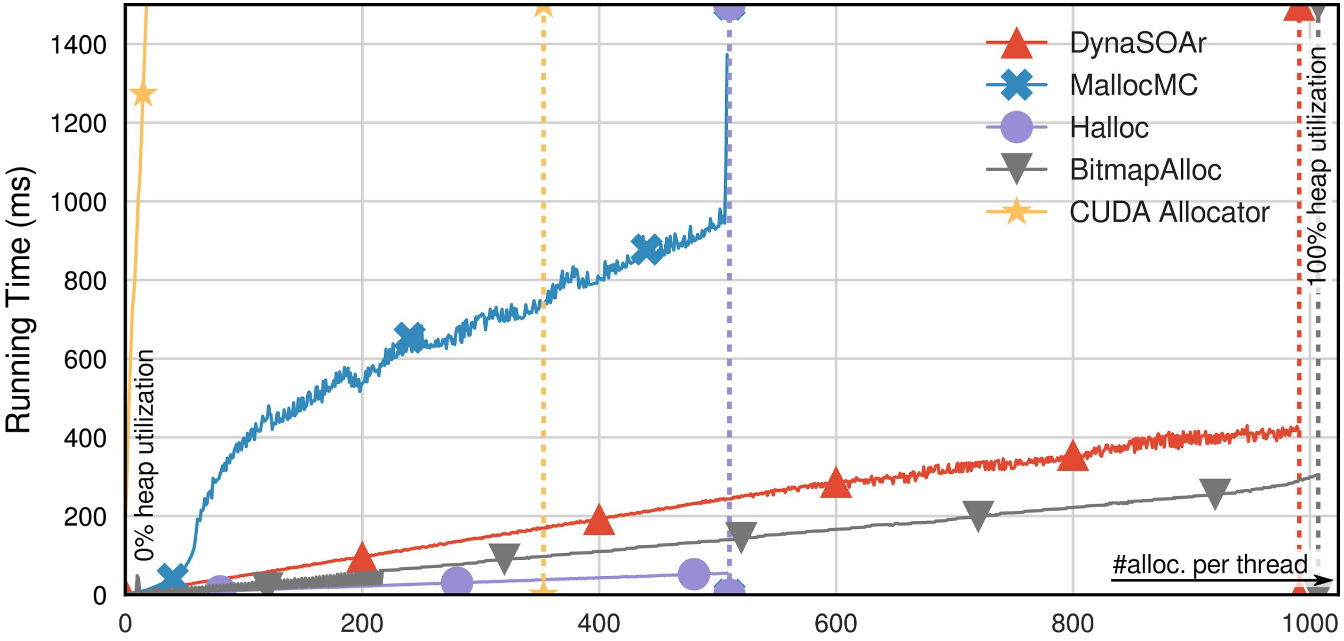
wa-tor Scaling Benchmark

$$F = \frac{1}{\#\text{blocks}} \sum_{b \in Blocks} \frac{\#\text{free slots}(b)}{\#\text{slots}(b)}$$

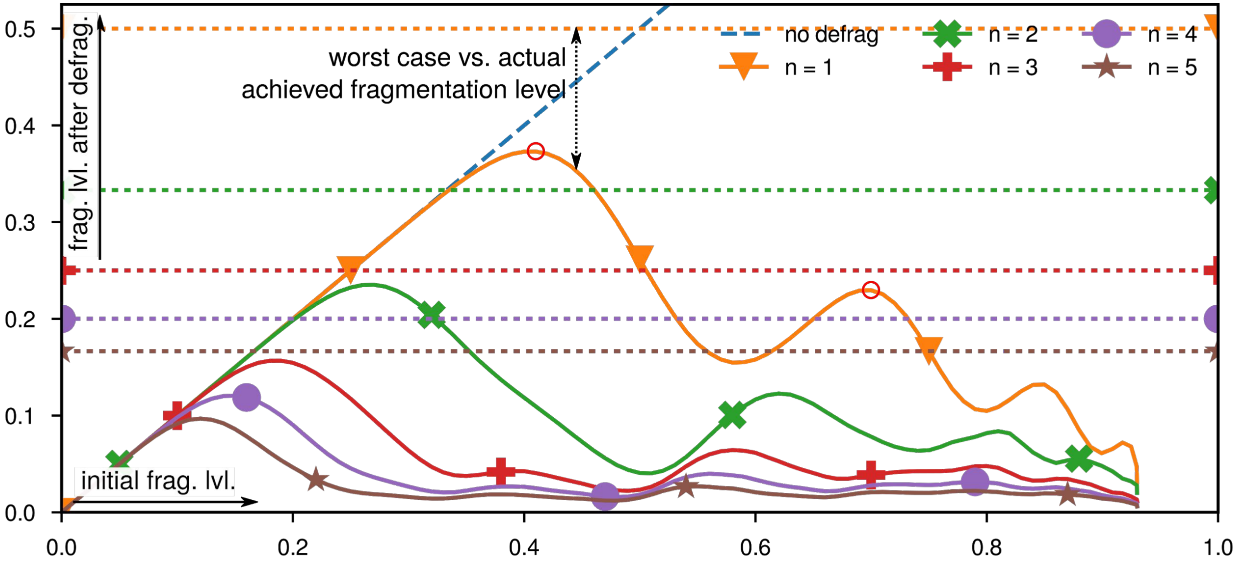
東京工業大学
Tokyo Institute of Technology



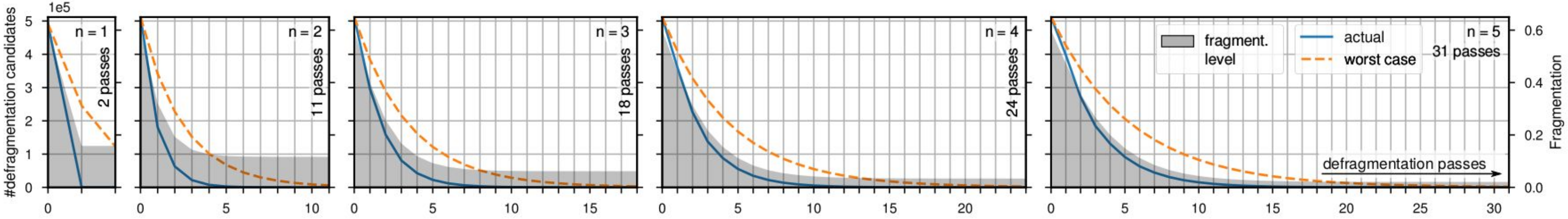
Linux Scalability Benchmark: Pure (de)alloc



CompactGpu Microbenchmark Results

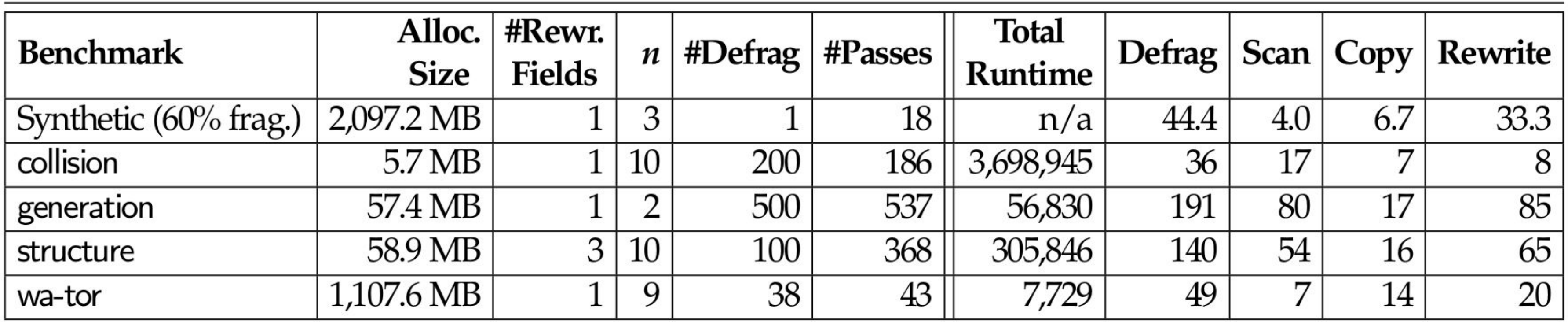


CompactGpu Microbenchmark Results



- In reality, we need fewer defragmentation passes to eliminate all defragmentation candidates.
    - Fewer than the theoretical worst-case #passes: $\mathit{log}_{(n+1)/n}$ *#candidates*

CompactGpu Benchmark Characteristics



Benchmark	Alloc. Size	#Rewr. Fields	*n*	#Defrag	#Passes	Total Runtime	Defrag	Scan	Copy	Rewrite
Synthetic (60% frag.)	2,097.2 MB	1	3	1	18	n/a	44.4	4.0	6.7	33.3
collision	5.7 MB	1	10	200	186	3,698,945	36	17	7	8
generation	57.4 MB	1	2	500	537	56,830	191	80	17	85
structure	58.9 MB	3	10	100	368	305,846	140	54	16	65
wa-tor	1,107.6 MB	1	9	38	43	7,729	49	7	14	20