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
```

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

Background

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

Resources:
- https://github.com/prg-titech/ikra-ruby
- https://github.com/prg-titech/ikra-cpp
- https://github.com/prg-titech/dynasoar
Background
Background: GPU Architecture

- NVIDIA GP104 (GeForce GTX 1080)
- 20 streaming multiprocessors (SMs)
- 128 CUDA cores per SM
- **Total:** $20 \times 128 = 2560$ 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
  - \textit{Total}: \(20 \times 128 = 2560\) CUDA cores
- 4 \textit{physical} cores per SM
  - \textit{Total}: \(20 \times 4 = 80\) 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
  - \( \text{Total: } 20 \times 128 = 2560 \text{ CUDA cores} \)
- 4 physical cores per SM
  - \( \text{Total: } 20 \times 4 = 80 \text{ cores} \)
- 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.
Thread-level Parallelism: 2560 CUDA cores
  - **SIMD**: Every 32 consecutive cores (warp; tid. $i\times32 \ldots (i+1)\times32 - 1$) 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)
  - *Memory coalescing:* The memory controller can *coalesce* (combine) requests that are on the same L1/L2 cache line on a per-warp basis (threads \( t_{\text{tid}} \) with \( \text{tid} \in [32*i; 32*(i+1)) \)).
  - *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

If the programmer loads 4 bytes, then the mem. controller loads 128 bytes and throws 124 bytes away
Performance Problems on GPUs

Source: CUDA C Programming Guide

**Aligned accesses (sequential/non-sequential)**

- **Addresses:** 0, 128, 160, 192, 224, 256, 288
- **Threads:** 0, ..., 31

- **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

**Mis-aligned accesses (sequential/non-sequential)**

- **Addresses:** 0, 128, 160, 192, 224, 256, 288
- **Threads:** 0, ..., 31

- **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

**good coalescing**

**bad coalescing**
Performance Problems on GPUs

Strided / Random Memory Access

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
</table>

| Threads:   | 0  | ... | ... | 31  |

<table>
<thead>
<tr>
<th>Compute capability:</th>
<th>2.0 and later</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transactions:</td>
<td>Cached</td>
</tr>
<tr>
<td></td>
<td>32x 128B</td>
</tr>
</tbody>
</table>

no coalescing

Source: CUDA C Programming Guide
Problems with OOP on GPUs
Common Belief: OOP is Slow

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

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

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

```c
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)

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

(b) Structure of Arrays (SOA)

```c
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];
```

(c) SOA Code Example

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

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

- **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]

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

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*.

```c++
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.

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**.

- **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:**

- **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.


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)

- 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].

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

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

$\text{parallel\_do}<A, \&A::\text{func}>().$
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>()

Diagram with nodes labeled A, B, and C.
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 (separate 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

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

Main Loop

```cpp
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, 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

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 Simulation**

```cpp
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);
    }
};
```
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);
    }
}
Examples of SMMO Applications

- 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

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
- 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()
```

+ apply_force(other)
+ compute_force()
+ update()
Example: Barnes-Hut N-Body Simulation
Example: Fish-and-Shark (wa-tor)
Example: Nagel-Schreckenberg Simulation
An SOA Data Layout DSL for Ikra-Cpp [WPMVP18]

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

```cpp
class Body : public AllocatorT::Base {
    public:
        declare_field_types(Body, float, 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_; 
}
```

\textbf{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

```cpp
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.

![Diagram of memory layout]

- **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

(b) Clustered SOA Layout: 3 memory transactions required

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

heap: array of $M$ blocks

<table>
<thead>
<tr>
<th>Spring</th>
<th>Node</th>
<th>PullNode</th>
<th>(free)</th>
<th>(free)</th>
<th>...</th>
<th>Node</th>
<th>Node</th>
<th>(free)</th>
</tr>
</thead>
</table>

- 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
  - ... but smaller arrays

- same type $\rightarrow$ same capacity (46)
- all blocks have same size (bytes)
- always 64-bit bitmaps ...

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

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

heap: array of \( M \) blocks

<table>
<thead>
<tr>
<th>Spring</th>
<th>Node</th>
<th>PullNode</th>
<th>(free)</th>
<th>(free)</th>
<th>...</th>
<th>Node</th>
<th>Node</th>
<th>(free)</th>
</tr>
</thead>
</table>

- **No fragmentation. GOOD!**
- All blocks have the same size (bytes)
- Remaining capacity (46)

**Bit for object slot**
- This block is active (i.e., not entirely full)

**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!**

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!**

... but smaller arrays
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)

- **free**
- **allocated** node
- **active** node
- **allocated** pull node
- **active** pull node
- **allocated** spring
- **active** spring

(no bitmaps for abstract class NodeBase)
Algorithm: Object Allocation

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

1 repeat  
2   bid ← active[T].try_find_set();  
3   if bid = FAIL then  
4     bid ← free.clear();  
5     initialize_block<T>(bid);  
6     allocated[T].set(bid);  
7     active[T].set(bid);  
8   alloc ← heap[bid].reserve();  
9   if alloc ≠ FAIL then  
10      ptr ← make_pointer(bid, alloc.slot);  
11      t ← heap[bid].type;  
12      if alloc.state = FULL then active[t].clear(bid);  
13      if t = T then return ptr;  
14      deallocate<T>(ptr);  
15 until false;  

▶ Infinite loop if OOM
▶ Find and return the position of any set bit.
▶ Slow path
▶ Find and clear a set bit atomically, return position.
▶ Set type ID, initialize object bitmaps.
▶ Reserve an object slot. See Alg. 7.
▶ Volatile read
▶ Type of block has changed. Rollback.
Example: `new Spring()`, Fast path

**heap**: array of $M$ blocks

- `Spring`
- `Node`
- `PullNode`
- (free)
- (free)
- ...
- Node
- Node
- (free)

**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]`

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

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

**Find active block**

- `bit for object slot`
- `object allocation bitmap`
- `object iteration bitmap`
- `type id + padding`
- `data segment (SOA arrays)`
- `incl. inherited fields`
- `... but smaller arrays`

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

**heap:** array of $M$ blocks

<table>
<thead>
<tr>
<th>Spring</th>
<th>Node</th>
<th>PullNode</th>
<th>(free)</th>
<th>(free)</th>
<th>...</th>
<th>Node</th>
<th>Node</th>
<th>(free)</th>
</tr>
</thead>
</table>

- **Reserve object slot**
  - 0x01
  - slot last allocated
  - 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
- ... 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

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

**heap:** array of $M$ blocks

<table>
<thead>
<tr>
<th>Spring</th>
<th>Node</th>
<th>PullNode</th>
<th>(free)</th>
<th>(free)</th>
<th>...</th>
<th>Node</th>
<th>Node</th>
<th>(free)</th>
</tr>
</thead>
</table>

- **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
  - but smaller arrays

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

- **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)
Handout only: Fake Pointers

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

Block capacity offset_{NodeBase::pos_y} = sizeof(Spring*[3]) + 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

- **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.

```
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:**

1. repeat
2. active ← __activemask(); ▶ Infinite loop if OOM
3. leader ← fs(active); ▶ Bitmap of active threads in warp
4. rank ← __lane_id(); ▶ Leader = active thread with lowest ID
5. if leader = rank then
6.   bid ← active[T].try_find_set(); ▶ Rank of this thread
7. if bid = FAIL then
8.   bid ← free.clear(); ▶ Slow path
9.   initialize_block<T>(bid);
10. allocated[T].set(bid);
11. active[T].set(bid);
12. alloc_bitmap ← heap[bid].reserve_multiple(popc(active));
13. if popc(alloc_bitmap) > 0 then
14.   t ← heap[bid].type;
15.   if alloc.state = FULL then active[t].clear(bid);
16.   if t ≠ T then deallocate_multiple<T>({bid, alloc_bitmap});
17. alloc_bitmap ← __shfl_sync(active, alloc_bitmap, leader);
18. bid ← __shfl_sync(active, bid, leader);
19. id_in_active ← popc(__lanemask_H()) & active;

---

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)

Benchmarks: Running Time

- **Baseline**: Without dynamic memory allocation
wa-tor Fragmentation

![Graph showing wa-tor fragmentation](image)
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.
Why Memory Defragmentation?

- **Space Efficiency**: Lower overall memory consumption.
- **Performance**: Reading/writing compact, less fragmented data requires fewer memory access transactions.
Block Merging: $1 + 1 = 1$

Do this in parallel for all *eligible* blocks:

- Take 2 blocks
- ≤ 50% full
- ≤ 50% full

$$\begin{align*}
\text{Cell\* Agent::position[64]} \\
\text{Cell\* Agent::new\_position[64]} \\
\text{int Agent::random\_state[64]} \\
\text{int Agent::age[64]} \\
\text{float Fish::spawn\_probability[64]} \\
\end{align*} + \\
\begin{align*}
\text{Cell\* Agent::position[64]} \\
\text{Cell\* Agent::new\_position[64]} \\
\text{int Agent::random\_state[64]} \\
\text{int Agent::age[64]} \\
\text{float Fish::spawn\_probability[64]} \\
\end{align*} = \\
\begin{align*}
\text{Cell\* Agent::position[64]} \\
\text{Cell\* Agent::new\_position[64]} \\
\text{int Agent::random\_state[64]} \\
\text{int Agent::age[64]} \\
\text{float Fish::spawn\_probability[64]} \\
\end{align*} \text{and} \\
\begin{align*}
\text{Cell\* Agent::position[64]} \\
\text{Cell\* Agent::new\_position[64]} \\
\text{int Agent::random\_state[64]} \\
\text{int Agent::age[64]} \\
\text{float Fish::spawn\_probability[64]} \\
\end{align*}
Block Merging: 1 + 2 = 2

Do this in parallel for all eligible blocks:

\[ \text{Cell* Agent::position[64] Cell* Agent::new_position[64] int Agent::random_state[64] int Agent::age[64] float Fish::spawn\_probability[64]} \]

\[ \text{Cell* Agent::position[64] Cell* Agent::new_position[64] int Agent::random_state[64] int Agent::age[64] float Fish::spawn\_probability[64]} \]

\[ \text{Cell* Agent::position[64] Cell* Agent::new_position[64] int Agent::random_state[64] int Agent::age[64] float Fish::spawn\_probability[64]} \]

\[ \text{Cell* Agent::position[64] Cell* Agent::new_position[64] int Agent::random_state[64] int Agent::age[64] float Fish::spawn\_probability[64]} \]

\[ \text{Cell* Agent::position[64] Cell* Agent::new_position[64] int Agent::random_state[64] int Agent::age[64] float Fish::spawn\_probability[64]} \]

\[ \text{Cell* Agent::position[64] Cell* Agent::new_position[64] int Agent::random_state[64] int Agent::age[64] float Fish::spawn\_probability[64]} \]

\[ \text{Cell* Agent::position[64] Cell* Agent::new_position[64] int Agent::random_state[64] int Agent::age[64] float Fish::spawn\_probability[64]} \]

\[ \text{Cell* Agent::position[64] Cell* Agent::new_position[64] int Agent::random_state[64] int Agent::age[64] float Fish::spawn\_probability[64]} \]
Block Merging: $1 + n = n$

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

- **Defragmentation factor $n$**
  - can be configured.
    - Higher $n$: Better defrag. guarantees.
    - Lower $n$: A bit faster, fewer passes.

- Blocks that are $\leq n/(n+1)$ full are defrag. candidates *(eligible).*
Handout only: Defragmentation by Block Merging

- After defragmentation:
  - All blocks with fill level $\leq n/(n+1)$ are gone.
  - Only blocks with fill level $> n/(n+1)$ are left over.
  - Therefore, fragmentation is $\leq 1 - n/(n+1) = 1/(n+1)$.

- One defragmentation pass eliminates all source blocks: $1/(n+1)$ of all defragmentation candidates.
  - To eliminate all defragmentation candidates, we need $\log_{(n+1)/n} \#\text{candidates}$ many passes.
Why do we require that all $n$ blocks are $\leq n/(n+1)$ full instead of all blocks together $\leq 100\%$ full?

- Makes it easier to identify 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

(initial state) → free

init block → dealloc, now empty

allocated [T], active [T], defrag [T]

alloc → dealloc

allocated [T], active [T]
alloc, now full → dealloc

allocated [T]
alloc, now full → dealloc

t: C++ class/struct type

fill levels (n = 1) fill levels (n = 2)
0% 0%
1% - 50% 1% - 66%
51% - 99% 67% - 99%
100% 100%
Keeping Track of Defragmentation Candidates

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

```plaintext
repeat
  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);
  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;
```

[GPU]
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 pointers** 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 : \texttt{uint64_t}[M/64]

indices : \texttt{int}[M]

order-preserving stream compaction

source blocks

R : \texttt{int}[r]

target (1) blocks

R : \texttt{int}[10]

target (2) blocks

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

thread assignment:

\[ t_0 \to t_{64} \to t_{128} \]

\[ t_{63} \to t_{127} \to t_{191} \]

\#source blocks \(B = \left\lceil \frac{10}{3} \right\rceil = 3\)
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.

<table>
<thead>
<tr>
<th>0x01</th>
<th>NodeBase*[64] Spring::n1</th>
<th>NodeBase*[64] Spring::n2</th>
<th>float[64] Spring::initial_length</th>
<th>float[64] Spring::stiffness</th>
<th>float[64] Spring::max_force</th>
<th>int[64] Spring::bfs_distance</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>12</th>
<th>14</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>R:</td>
<td>int [10]</td>
<td>target (1) blocks</td>
<td>target (2) blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Source blocks
- Target blocks

**Data segment (SOA arrays)**
- Incl. inherited fields

**Object allocation bitmap**

**Object iteration bitmap**

**Type id + padding**

- Thread assignment:
  - $t_0$, $t_{64}$, $t_{128}$
  - $t_{63}$, $t_{127}$, $t_{191}$

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

#source blocks $B = \left\lfloor \frac{10}{3} \right\rfloor = 3$
Step 2: Copy Objects

**ex. thr. assignment**

- **source object allocation bitmap**
  - s_bitmap
  - fill level: 18 / 32 (56%)

- **target (1) object allocation bitmap**
  - t_bitmap (i = 0)
  - fill level: 20 / 32 (63%)

- **target (2) object allocation bitmap**
  - t_bitmap (i = 1)
  - fill level: 24 / 32 (75%)

  
  defrag

- **target object allocation bitmap**
  - t_bitmap (i = 2): loop breaks before i = 2
  - fill level: 30 / 32 (94%)

- **source object allocation bitmap**
  - s_bitmap
  - fill level: 0 / 32 (0%)
Step 3: Store Forwarding Ptrs. in Source Blocks

- Overwrite data segment of source blocks with forwarding pointers.

```c
Spring* [64] forwarding_ptrs;
```

 ![Diagram showing object allocation bitmap, object iteration bitmap, type id + padding, data segment (SOA arrays) including inherited fields]
Step 4: Rewrite Pointers to Relocated Objects

- **Conceptually:** A parallel do-all operation
  
  \[
  \text{parallel\_do<\text{NodeBase}, \&\text{AllocatorT::Base::rewrite\_field<\text{NodeBase}, 0>>()}}
  \]

- We are rewriting every field that could potentially have a pointer to a relocated object.

- **Discussion:** C++ Boehm GC [1]

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];
    }
  }
  ```
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.
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]?”**


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.

(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

```c
_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
```

\( N_{Spring} = 64 \)

```c
__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

- 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

Max. problem size rel. to DynaSOAr

- DynaSOAr
- Halloc
- Baseline (SOA)
- Baseline (AOS)
- mallocMC
- BitmapAlloc

(higher is better)
wa-tor: Pinpointing DynaSOAr’s Speedup
wa-tor Scaling Benchmark

\[
F = \frac{1}{\# \text{blocks}} \sum_{b \in \text{Blocks}} \frac{\# \text{free slots}(b)}{\# \text{slots}(b)}
\]
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: \( \log_{(n+1)/n} \#candidates \)
# CompactGpu Benchmark Characteristics

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