SoaAlloc: Accelerating Single-Method Multiple-Objects Applications on GPUs

Student Research Competition @ SPLASH 2018

Matthias Springer
Tokyo Institute of Technology
Research Goal: OOP for GPUs

- Fast Object-oriented Programming (OOP) on GPUs
- SIMD-friendly class of OOP applications: Single-Method Multiple-Objects (SMMO)
- Many practical SMMO applications in HPC, e.g.:

Traffic Flow Simulation [1]
Evacuation Simulation [2]
Predator-Prey

Animation: https://en.wikipedia.org/wiki/Wa-Tor
Single-Method Multiple-Objects

- Run same method for all objects of a type
- Running Example: Fish-and-Shark Simulation

- Creating and deleting objects (fish, sharks) all the time!
- Run move() method for all fish and shark objects in parallel
For Good Performance: SOA Data Layout

• Standard optimization on GPUs for good memory bandwidth utilization and better cache performance

  • `class Shark {
    float health;
    Cell* position;
    /* ... */

    void step_health() {
      health = health - 1;
      if (health == 0)
        delete this;
    }
  };

  Shark sharks[1000];

  Array of Structures (AOS)

• `float S_health[1000];
  Cell* S_position[1000];`

  vector load possible

  `void S_step_health(int id) {
    S_health[id] =
    S_health[id] - 1;
    if (S_health[id] == 0)
      S_destruct(id);
  }`

  `Structure of Arrays (SOA)`
Main Challenges

• How to combine dynamic memory allocations with SOA?
• How to keep fragmentation low?
• With thousands of parallel threads, how to implement all of this in a lock-free fashion? (Memory allocator runs entirely on the GPU!)

• Allocator Interface:
  new<T>(), delete<T>(), do_all<T>(func*)
Based on Ideas from Related Work

- Other GPU memory allocators (e.g., [3]): Fast allocations, but **slow memory access**
- How to represent pointers? E.g.: global references [4]
- C++/CUDA **DSLs** for SOA data layout [5, 6]

---

## Allocation Data Structure

### Heap: Array of M Blocks

<table>
<thead>
<tr>
<th>alloc. + active (Fish)</th>
<th>alloc. + active (Shark)</th>
<th>alloc. (Fish)</th>
<th>uninit.</th>
<th>alloc. (Fish)</th>
<th>...</th>
<th>alloc. (Shark)</th>
</tr>
</thead>
</table>

### Active Block

- `Cell* Fish::position[64]`
- `Cell* Fish::new_position[64]`
- `int Fish::random_state[64]`
- `int Fish::egg_timer[64]`

### Object Allocation Bitmap (64 bits)

### Data Segment (SOA arrays)
Allocation Data Structure

heap: array of M blocks

alloc. + active (Fish) | alloc. + active (Shark) | alloc. (Fish) | uninit. | alloc. (Fish) | ... | alloc. (Shark)

Object Allocation Bitmap (64 bits)

Cell* Fish::position[64]
Cell* Fish::new_position[64]
int Fish::random_state[64]
int Fish::egg_timer[64]

Data Segment (SOA arrays)

Cell* Shark::position[56]
Cell* Shark::new_position[56]
int Shark::random_state[56]
int Shark::egg_timer[56]
int Shark::health[56]

inactive block
Fragmentation: Lower is Better

Cell* Fish::position[64]
Cell* Fish::new_position[64]
int Fish::random_step[64]
int Fish::egg_counter[64]
Fragmentation: Lower is Better

Cell* Fish::position[64]
Cell* Fish::new_position[64]
int Fish::random_state[64]
int Fish::egg_timer[64]

...
Object Allocation

new T()

Find active block of type T

success? no yes

Find free block

Initialize block

Allocate object in block

Run object constructor

yes no

success?
How to find blocks?

1. new T()
2. Find active block of type T
3. success?
   - no: Find free block
   - yes: Initialize block
4. Allocate object in block
5. Run object constructor
   - yes: success?
   - no: Run object constructor
How to find blocks?

Instead of scanning the entire heap:
Scan a (large) bitmap
How to find blocks?

Instead of scanning the entire heap:
Traverse a hierarchical bitmap
Preliminary Benchmark Results

More than 2x speedup compared to MallocMC/ScatterAlloc

Fish-and-Sharks Simulation

Iteration Running Time (ms)

Iterations (time)

SoaAlloc
MallocMC
Hallocc
CUDA Alloc.
Preliminary Benchmark Results

Gray area: Fragmentation overhead
(allocated but unused memory)
Future Work

• Evaluate SoaAlloc with more benchmarks
• Explicit memory defragmentation may lead to further speedups
• Refine implementation: e.g., per-warp private blocks (similar to private heaps)
Preliminary Benchmark Results

![Graph showing iteration running time vs iterations (time) for different allocation methods. The x-axis represents iterations (time) ranging from 0 to 500, and the y-axis represents iteration running time in milliseconds (ms), with a logarithmic scale ranging from $10^0$ to $10^2$. The graph compares five allocation methods: SoaAlloc, SoaAlloc-Aos, SoaAlloc-NoCoal, MallocMC, and AosAlloc. The graph illustrates the performance differences among these methods over time.]