DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs

Matthias Springer and Hidehiko Masuhara (Tokyo Institute of Technology)

See our presentation @ ECOOP on Thursday, 15:40 - 16:00!

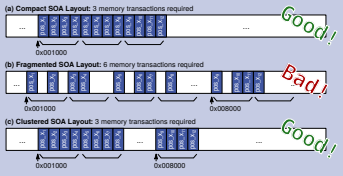
Motivation

- **Object-oriented programming** (OOP) has many applications in high-performance computing (HPC), but it **is too slow**.
- Esp. dynamic memory allocation, which is very useful in OOP.
- Main performance problem on GPUs: **Memory access**.
- DynaSOAr: A framework for efficient OOP on GPUs (3 parts)

1. Data layout DSL: Store objects in a Structure of Arrays (SoA).
2. Dyn. mem. allocator: Allocate objects with low fragmentation.
3. Parallel do-all operation: Expressing parallelism over an obj. set.

Memory Coalescing

Compact/dense data requires fewer vector loads/stores!



For illustration purposes: Vector length 32 byte (4 scalars) instead of 128 byte (32 scalars). N-body sim.

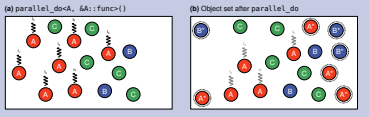
Related Work

- Default CUDA allocator: Slow, unoptimized, unreliable
- Other state-of-the-art GPU allocators:
  - Halloc [1] and ScatterAlloc/mallocMC [2]
  - Lack optimizations for structured data: Fast (de)allocation but **inefficient usage of allocated memory**.
  - Based on hashing techniques: Leads to **high fragmentation**.
- DynaSOAr has detailed **knowledge about the structure of its allocations** (fields of classes/structs). This allows us to apply more optimizations (e.g., SoA layout). Other allocators just allocate raw bytes.
- Data layout DSL based on Ikra-Cpp [3].

[1] A. Adinetz, D. Pleiter. Halloc: A High-Throughput Dynamic Memory Allocator for GPGPU Architectures. GPU Technology Conference, 2004.
[2] M. Steinberger, M. Kenzel, B. Kainz, D. Schmalstieg. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. InPar 2012.
[3] M. Springer, H. Masuhara. Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming with Structure-of-Arrays Layout. WPMVP 2018.

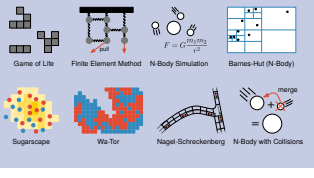
SMMO: Single-Method Multiple-Objects

- Obj.-orient. **programming model** for SIMD architectures.
- OOP-speech for Single-Instruction Multiple-Data (SIMD)
- Run **method** for all objects of a type T: `parallel_do<T, &T::func>`
- Method can **create/delete objects**.
- In parallel: Spawns a **CUDA kernel**.
- Deleting an object of type T other than `this` (bound object) is forbidden.




```
void A::func() {
    if (rand() < 0.5) { new B(); delete this; }
    else { new A(); }
}
```

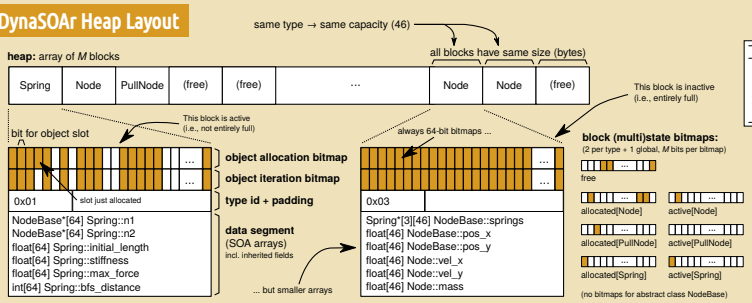
SMMO Examples

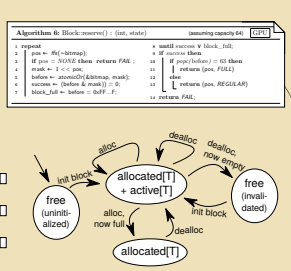



```
#include "dynasoar.h"
class Body;
using AllocatorT = SoaAllocator</*max_num_obj=*/ 167727216, /*T...=*/ Body>;
__device__ DAllocatorHandle<AllocatorT> d_allocator;

class Body : public AllocatorT::Base {
 private:
  declare_field_types(Body, float, float, /*...*/ bool)  // Pre-declare all field types.
  Field<Body, 0> pos_x_;  Field<Body, 1> pos_y_;  /*...*/  Field<Body, 9> was_merged_;

 public:
  __device__ Body(float pos_x, float pos_y) : pos_x_(pos_x), pos_y_(pos_y) {}
  __device__ Body(int idx) : Body(/*...*/) {}  // Constructor for parallel_new.
  __device__ void step_2_move(float dt) { pos_x_ += dt * vel_x_; pos_y += dt * vel_y_; }
  __device__ void step_6_delete_merged() { if (was_merged_) { destroy(d_allocator, this); } }
};

int main() {
  auto* h_allocator = new HAllocatorHandle<AllocatorT>();  // Initialize DynaSOAr heap.
  cudaMemcpyToSymbol(d_allocator, h_allocator->device_handle());
  h_allocator->parallel_new<Body>(65536);
  for (int i = 0; i < 1000; ++i) { h_allocator->parallel_do<Body, &Body::step_2_move(0.5); }
  delete h_allocator;  // Release all GPU memory.
}
```

New object: `new(d_allocator) Body(/*...*/);`
Delete object: `destroy(d_allocator, ptr)`

DynaSOAr Heap Layout





Lock-free Object Allocation (Overview)

If *FAIL*, try up to *r* times before continuing to Line 3.

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

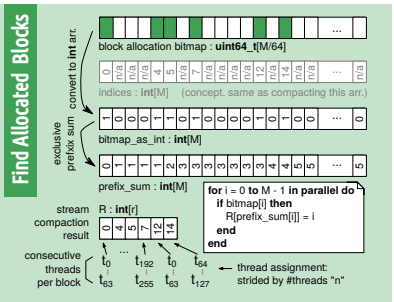
```
1  repeat                                          ➤ Infinite loop if OOM
2    bid ← active[T].try_find_set();               ➤ Find and return the position of any set bit.
3    if bid = FAIL then                            ➤ Slow path
4      bid ← free.clear();                         ➤ Find and clear a set bit atomically, return position.
5      initialize_block<T>(bid);                   ➤ Set type ID, initialize object bitmaps.
6      allocated[T].set(bid);
7      active[T].set(bid);
8    alloc ← heap[bid].reserve();                  ➤ Reserve an object slot.
9    if alloc ≠ FAIL then
10     ptr ← make_pointer(bid, alloc.slot);
11     t ← heap[bid].type;                         ➤ Volatile read
12     if alloc.state = FULL then active[t].clear(bid) ;
13     if t = T then return ptr ;
14     deallocate<t>(ptr);                         ➤ Type of block has changed. Rollback.
15 until false;
```

Problem: How do we know that it is safe to delete a block? A contending thread may be in the middle of a concurrent allocation. **Solution:** Block invalidation, i.e. setting all bits in the obj. alloc. bitmap to 1.

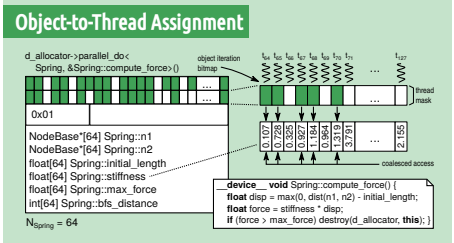
Main Challenges

- **Lock-free implementation:** Based on **atomic operations and retry loops.**
- **Safe memory reclamations:** **All blocks have the same structure** regardless of state, fill level and type. I.e., bitmaps and type ID are always at the same offset.

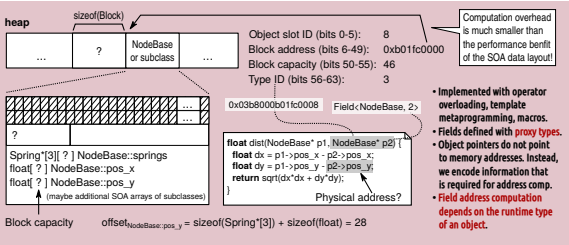
Find Allocated Blocks



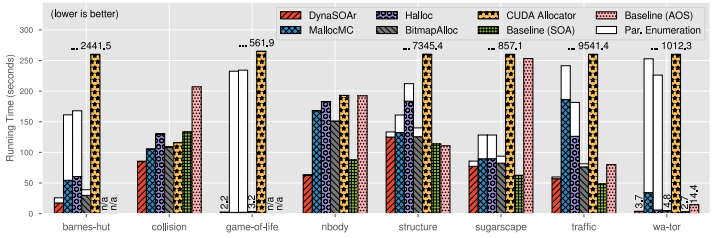
Object-to-Thread Assignment

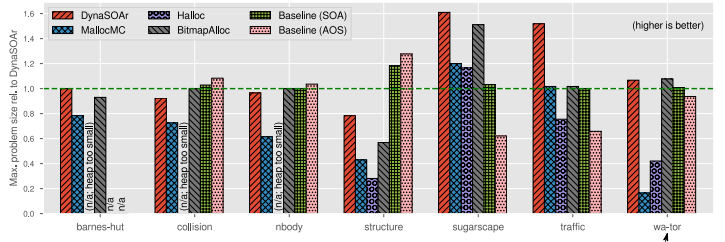


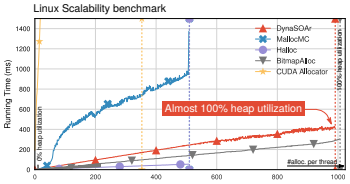
Fake Pointers

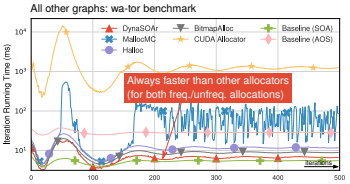


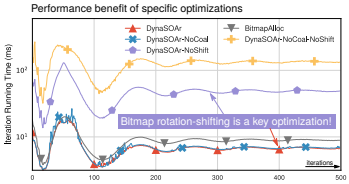
- Implemented with operator overloading, template metaprogramming, macros.
- Fields defined with proxy types.
- Object pointers do not point to memory addresses. Instead, we encode information that is required for address comp.
- Field address computation depends on the runtime type of an object.

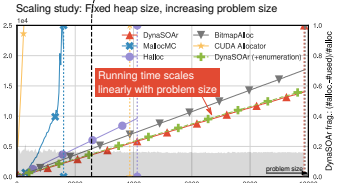












Other Optimizations

[4] X. Huang, C. Rodrigues, S. Jones, I. Buck, W. Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. CIT 2010.

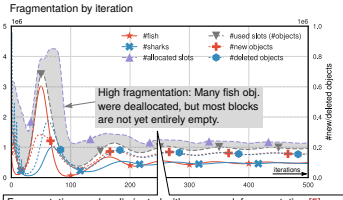
- To find active/free blocks with a logarithmic number of accesses, block states are indexed with **hierarchical, lock-free bitmaps.**
- To reduce allocation contention, simultaneous **allocation requests are coalesced**/combined on a per-warp basis [4].
- To reduce #retries, **bitmaps are rotating-shifted** before ffs.
- To further reduce fragmentation, try Alg. 1 Line 2 up to *r* times.

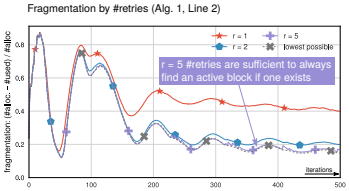
Memory Fragmentation

- Diminishes the benefit of SOA.
- No internal/external fragmentation by design.

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

(only allocated blocks)





Fragmentation can be eliminated with memory defragmentation [5].

[5] M. Springer, H. Masuhara. Massively Parallel GPU Memory Compaction. ISMM 2019.

https://github.com/prg-titech/dynasoar