SPLASH: G: DynaSOAr: Accelerating Single-Method Multiple-Objects Applications on GPUs*

Matthias Springer[†]
Tokyo Institute of Technology, Japan
matthias.springer@acm.org

Abstract

Object-oriented programming (OOP) has long been regarded as too inefficient for SIMD high-performance computing, despite the fact that many important HPC applications have an inherent object structure. We discovered a broad subset of OOP that can be implemented efficiently on massively parallel SIMD accelerators. We call it *Single-Method Multiple-Objects* (SMMO), because parallelism is expressed by running a method on all objects of a type.

To make fast GPU programming available to domain experts who are less experienced in GPU programming, we developed DynaSOAr, a CUDA framework for SMMO applications. DynaSOAr improves the usage of allocated memory with an SOA data layout and achieves low memory fragmentation through efficient management of free and allocated memory blocks with lock-free, hierarchical bitmaps.

Keywords   Hier. Bitmaps, GPUs, Memory Allocation, OOP

1   Introduction

General-purpose GPU computing has long been a tedious job, requiring programmers to write hand-optimized, low-level programs. In an attempt to make GPU computing available to a broader range of developers, our efforts are centered around bringing fast object-oriented programming (OOP) to low-level languages such as CUDA and C++.

OOP has a wide range of applications in high-performance computing (HPC) but is often avoided due to bad performance [18]. Dynamic memory management (DMM), i.e., the ability/flexibility of creating/deleting objects at any time, is one of the corner stones of OOP. Programmers often have to go to great lengths to build their own allocators or write their applications is a more convoluted way due to slow DMM (e.g., [13]).

In recent years, fast, dynamic memory allocators have been developed for GPUs [1, 9, 22]. However, while these allocators often provide good raw (de)allocation performance, they miss key optimizations for structured data, leading to poor data locality and memory bandwidth utilization when accessing allocated memory.

1.1   Single-Method Multiple-Objects (SMMO)

We identified a class of HPC applications that can be expressed as object-oriented programs and implemented efficiently on SIMD architectures. We call this class *Single-Method Multiple-Objects* (SMMO) because parallelism is expressed by running a method on

all objects of a type (*parallel do-all*). Objects may be created/deleted at any time within a parallel do-all operation.

SMMO is a broad class of problems[1] with many real-world applications [21], such as simulations for population dynamics, (e.g., Sugarscape [8]), evacuations [13], wildfire spreading [19], finite element methods or particle systems, to name just a few. SMMO can also express BFS graph traversals and dynamic tree updates/constructions such as in Barnes-Hut [4].

1.2   Object Allocation with Efficient Memory Access

We developed DynaSOAr [21], a CUDA framework for SMMO applications. DynaSOAr consists of three parts:

1. A **dynamic memory allocator** based on concurrent, lock-free, hierarchical bitmaps. Our allocator stores objects in a Structure of Arrays (SOA) data layout.
2. An efficient **parallel do-all operation**. Such operations spawn a CUDA kernel and control the assignment of objects to GPU threads.
3. An embedded C++ **data layout DSL** for object-oriented programming with SOA layout, inspired by Ikra-Cpp [20]. This DSL allows us to implement DynaSOAr entirely in C++/CUDA without a custom code generator/preprocessor.

DynaSOAr achieves significant speedups over state-of-the-art GPU memory allocators by controlling both (1) data layout and (2) data access.

Structure of Arrays (SOA) Layout   SOA is a well-studied best practice of SIMD programming. Compared to a traditional AOS layout (Fig. 1a), an SOA layout (Fig. 1b) can increase memory bandwidth and cache utilization when the same fields of multiple objects are simultaneously accessed [11].

SIMD architectures achieve parallelism by running the same processor instruction on a vector register. Even though recent GPUs appear to have thousands of *CUDA cores*, the hardware actually only has a few hundred physical cores, each operating on 128-byte vector registers containing 32 `float`/`int` scalars[2]. Memory accesses of a physical core (i.e., 32 consecutive GPU threads; *warp* in CUDA) that fall into an aligned 128-byte address window are serviced with efficient vector loads (*memory coalescing*). Getting data into and out of vector registers is often the biggest bottleneck and peak memory bandwidth utilization can be achieved only with memory coalescing.

SOA can increase memory coalescing when accessing structured data because values of the same field are stored together. We improved the SOA layout in three ways:

- We built an embedded **C++/CUDA DSL** such that programmers can write code like in Fig. 1d instead of Fig. 1c.

---

[1]We implemented a few SMMO applications from different domains: https://github.com/prg-titech/dynasoar/wiki/Benchmark-Applications
[2]We are focusing on recent NVIDIA architectures, but other architectures are similar.

(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) Hand-written SOA Code Example

```
__device__ void move(int id) {
  /* Compute force, vel ... */
  pos_x[id] += Δt * vel_x[id];
```

SIMD: All threads perform this load in parallel.
Current NVIDIA GPUs *coalesce* 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] += Δt * vel_y[id];
}

int main() { /* Run move() in par. on GPU */ }
```

(d) DynaSOAr: (b) and (c) + DSL + Dyn. Object Allocator

```
class Body; using AllocT = Allocator<Body, 32000>;
__device__ AllocT allocator;                    [Inheritance OK]

class Body : public AllocT::Base {
  declare_types(Body, float, float /*, ... */)
  Field<Body, 0> pos_x;
  Field<Body, 1> pos_y; /* more fields */        [No object IDs]
  __device__ void move() {                       [(... but pointers)]
    pos_x += Δt * vel_x;   pos_y += Δt * vel_y; }
};

int main() {
  allocator.parallel_do<&Body::move>(); }
```
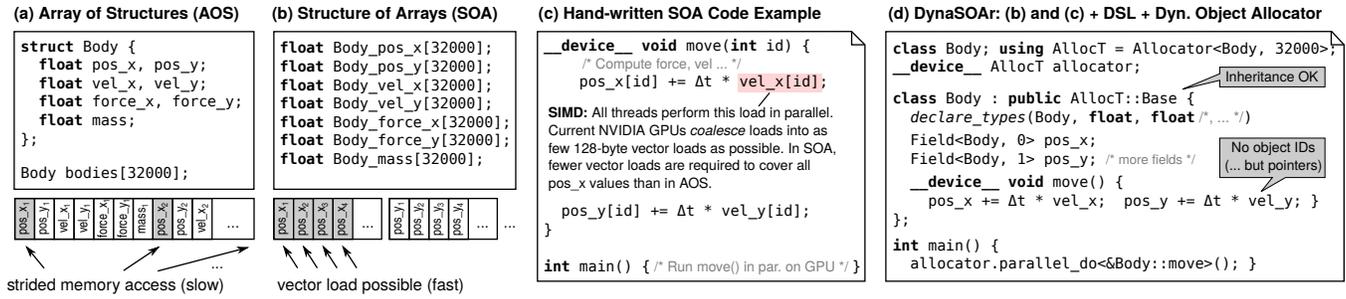
Figure 1. N-Body example: Excerpt of structure definition and Euler method time step (AOS/SOA/DynaSOAr).
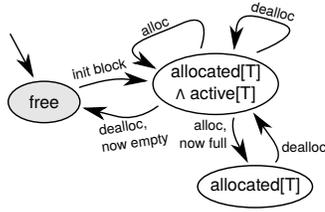

Figure 2. Block states. Initially, every block is *free*. *Allocated* blocks contain only objects of a specific type T. To reduce fragmentation, new objects are allocated in *active* blocks of the corresponding type.

- We developed the first **GPU dynamic memory allocator** (C++ new/delete interface) with SOA performance characteristics. **Low fragmentation** is key: If data is fragmented, more vector accesses are needed for the same number of bytes, reducing the benefit of SOA.
- Contrary to other systems such as *Columnar Objects* [15], we support class inheritance without wasting memory.

1.3 Contributions

Our work contributes to the state-of-the-art as follows:

- The concept of and examples of **SMMO** applications.
- **SOA Improvements**: Embedded C++ DSL and subclassing.
- A **dynamic memory allocator** for GPUs with efficient memory access and low fragmentation.
- A concurrent, lock-free, hierarchical **bitmap data structure**, based on atomic operations and retry loops.

2 The DynaSOAr System

DynaSOAr divides the heap into blocks of equal size in bytes (Fig. 3). Every block contains only objects of the same C++ class/struct type in SOA layout: one *SOA array* per field. Blocks of the smallest type in the system[3] always have a *capacity* (#object slots) of 64 and determine the block size in bytes, and thus the number of blocks M on the heap. Blocks of other types may have a smaller capacity depending on the size of their type in relation to the smallest type.

Coalesced Access   Although this layout does not constitute a plain structure of arrays as in Fig. 1b but *multiple structures* of arrays (one per block), it has the same performance characteristics as long as every SOA array is at least 128 bytes big (cache line and vector register size). This is the case for block capacities of at least 32 (assuming 4 byte fields).

Block States   A 64-bit *object allocation bitmap* keeps track of allocations. A block can be in one or more *multistates* (Fig. 2).

- *free:* The block is empty and does not contain any objects.
- *allocated[T]:* The block contains at least 1 object of type T. No other types can be stored in this block.
- *active[T]:* The block contains objects of type T. It is not full yet, i.e., it has space for at least one more object. $active[T] \Rightarrow allocated[T]$.

Allocation and deallocation routines frequently have to lookup blocks by state. For that reason, block states are indexed by bitmaps of size M; one bitmap per state.

Concurrency   Due to concurrent operations of other threads, these bitmaps can be temporarily inconsistent with the actual state of a block. Designing lock-free algorithms is notoriously difficult [16] and we carefully designed our algorithms such that they can detect inconsistencies and retry if necessary.

Our allocator design has one key feature that makes this problem more tractable: Apart from the data segment, every block has the same structure. Object allocation bitmaps are always located at the same offset within a block and the memory locations of blocks are fixed. Therefore, our algorithms can use block state bitmaps to quickly lookup blocks but then update object allocation bitmaps, which are the *single source of truth*, with atomic operations. At this *linearization point* [10], our algorithms can detect inconsistencies[4].

2.1 Bitmap Operations

DynaSOAr uses large bitmaps for indexing block states. These bitmaps provide a few basic operations.

- try_clear(pos): Atomically clear the bit at pos. If the bit was already cleared, return false, otherwise true.
- clear(pos): Clear the bit at pos. This is identical to: **while** (!try_clear(pos)) {}.
- *Similarly:* try_set(pos) and set(pos).
- try_find_set(): Find and return the position of a set bit or FAIL if none was found (e.g., because the bitmap is inconsistent).
- clear(): Find a set bit and clear it. This is identical to: **while** ((i = try_find_set()) != FAIL && try_clear(i)) {}; **return** i;

All operations are thread-safe and can be invoked by threads concurrently. try_clear(pos) and try_set(pos) are implemented with atomic bit-wise OR/AND operations[5]. Atomic operations are generally slow, but became much faster with recent GPUs [5].

---

[3]All types must be pre-declared with the allocator (template arg. in 1st line of Fig. 1d).

[4]Due to space limitations, we cannot give an exhaustive description of our algorithms here, but we invite the interested reader to take a look at our full paper [21].

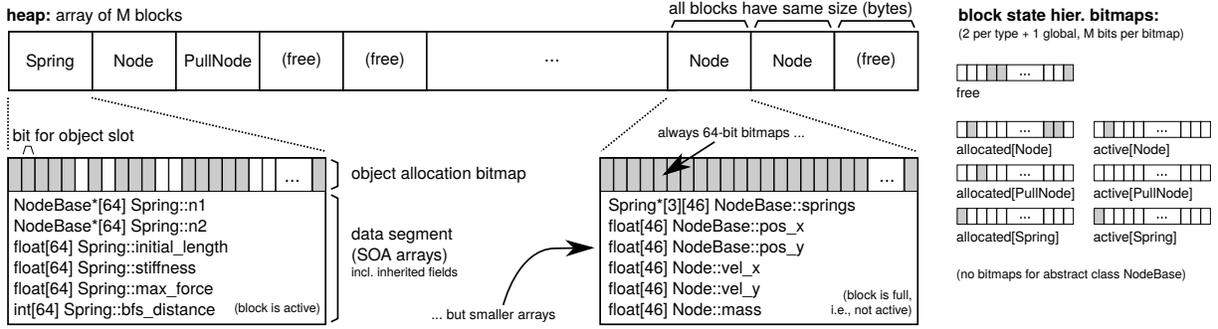[5]Atomic operations return the original value in memory.

Figure 3. Heap layout of a finite element method. DYNASOAR is a *slab allocator* [3]: Only objects of the same C++ type are stored in a block.

Algorithm 1: Allocator::allocate<T>() : T* | GPU |

```
1  repeat                                    ▷ infinite loop if OOM
2  │  bid ← active[T].try_find_set();
3  │  if bid = FAIL then                               ▷ slow path
4  │  │   bid ← free.clear();
5  │  │   initialize_block<T>(bid);
6  │  │   allocated[T].set(bid);
7  │  └   active[T].set(bid);
8  │  alloc ← get_block<T>(bid).reserve();
9  │  if alloc ≠ FAIL then        ▷ block filled up after line 2?
10 │  │   if alloc.state = FULL then      ▷ allocated last obj. slot
11 │  │   │   active[t].clear(bid);
12 │  └   return make_pointer(bid, alloc.slot);
13 until false;                          ▷ select new block + retry;
```
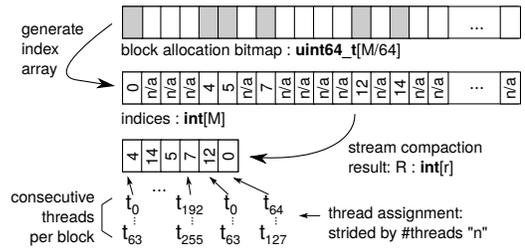
2.2 Object Allocation

Alg. 1 gives a simplified overview of object allocation. This is a thread-safe and entirely lock-free algorithm.

To keep memory fragmentation low, DYNASOAR allocates new objects always in active blocks. These are blocks that already contain some objects of the same type, but still have some vacancy. Only if no active block could be found, a new block is initialized (*slow path*).

This technique is in contrast to state-of-the-art GPU allocators, which scatter new allocations in the heap (e.g., using hashing), to reduce collisions and synchronization among threads [1, 22]. DYNASOAR requires more synchronization between threads, in the form of atomic operations. This makes (de)allocations more expensive. However, DYNASOAR's allocation policy leads to much lower fragmentation and denser allocations. This results in better overall performance of SMMO applications on SIMD architectures.

2.3 Parallel Do-all

In DYNASOAR, we express parallelism of SMMO applications with parallel do-all operations. Such operations spawn a CUDA kernel and run a method in parallel for all objects of a type *T*. Objects are assigned to GPU threads in such a way that the GPU can coalesce field reads/writes of the **this** object[6]. This increases memory bandwidth utilization and overall application performance.

---

[6]Field reads/writes of other objects do not benefit from additional memory coalescing.

Figure 4. Compaction of *allocated[T]* array and assigning objects to GPU threads. Assuming block capacity *n* = 64.

Before a CUDA kernel is launched, we precompute which objects are processed by each GPU thread (Fig. 4) in two steps that can be implemented efficiently on GPUs.

1. Generate an index array of set bits in *allocated[T]*.
2. Filter out unused index values (*stream compaction*). Can be implemented with a prefix sum algorithm [2].

If *c* is the capacity of blocks of type *T*, we assign *c* consecutive threads to each allocated block of type *T*. Since these *c* threads have consecutive IDs, they will run on the same 1–3 physical cores, so accesses are coalesced; not necessarily perfectly coalesced but better than in AOS. Not all blocks are entirely full, so some threads will have no objects to process; this mechanism is nevertheless fast, because we expect blocks to have a high average fill level since new allocations are always performed in active blocks.

Inside a CUDA kernel, a thread can now quickly read its assigned blocks from the filtered index array *R* and process the objects at slot *tid* % *c* in these blocks, if they have an object allocated there.

2.4 Additional Optimizations

DYNASOAR employs optimizations in addition to SOA to speed up allocations.

- Block bitmaps are **hierarchical** and updated with **eventually consistent, lock-free algorithms**, so that we can find *free* blocks (try_find_set()) with a log. number of accesses.
- To reduce thread contention, we borrow **allocation request coalescing** from XMalloc [12]: A leader thread allocates objects on behalf of all threads in a warp.
- Efficient implementation and engineering efforts: We utilize **low-level optimizations** such as int. intrinsics (e.g., Alg 2).

Algorithm 2: Bitmap::*try_clear*(pos) : void | GPU |

---

1 cid ← pos / 64;

2 offset ← pos % 64;

3 mask ← 1 << offset;

4 prev ← *atomicAnd*(&container[cid], ~mask);

5 success ← (prev & mask) ≠ 0;

6 **if** *success ∧ has_nested ∧ popc(prev) = 1* **then**

7 | nested.*clear*(cid);
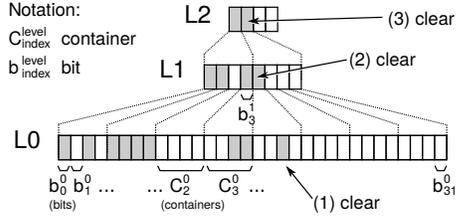
   > population cnt.:
   > number of set bits

8 **return** success;

Figure 5. Example: Hier. bitmap of size $N = 32$ with container size 4 (instead of 64). This example illustrates how (1) a *clear*(18) operation triggers (2) a *clear*(4) operation in the nested bitmap, which triggers (3) a *clear*(1) operation in the next nested bitmap.

3 Hierarchical Bitmaps

DynaSOAr uses hierarchical bitmaps to find find free and active blocks with try_find_set() in Alg. 1. This is a key optimization because block bitmaps can reach multiple megabytes in size and checking every bit is too slow.

Data Structure   A hierarchical bitmap of size N bits consists of two parts: an array of size $\lceil N/64 \rceil$ of 64-bit *containers* (uint64_t), and a *nested bitmap* of size $\lceil N/64 \rceil$ if $N > 64$. A container C_i^l consists of bits $b_{64 \cdot i}^l, ..., b_{64 \cdot i+63}^l$ and is represented by one bit b_i^{l+1} in the nested (higher-level) bitmap (Fig. 5). That bit is set if at least one bit is set in the container.

$$b_i^{l+1} = \bigvee_{k=0}^{63} b_{64 \cdot i+k}^l \qquad (container\ consistency)$$

Bits in are changed with atomic operations. Higher-level bits (and thus bitmaps) are *eventually consistent*[7] with their containers. Keeping both consistent all the time is difficult without locking, because two different memory locations cannot be changed together atomically. However, due to the design of the bitmap operations, the bitmap is guaranteed to be in a consistent state when all bitmap operations (of all threads) are completed. Bitmap operations retry or give up (*FAIL*) to handle temporary inconsistencies.

clear(pos) with Atomics   Alg. 2 illustrates how to clear a specific bit. We clear the bit with an atomic operation (Line 4). If this operation actually changed the bit (success) and cleared the last bit of the container, it is this thread's *responsibility* to clear the respective bit in the nested bitmap. Since the bitmap may be in an inconsistent state, we have to retry until the bit was cleared, thus clear(pos) instead of try_clear(pos) in Line 7.

---

[7]This is a key difference from other lock-free hier. data structures such as SNZI [7], which have stronger runtime consistency guarantees and require complex algorithms.

4 Evaluation

We evaluated the running time (Fig. 7) and memory usage (Fig. 8) of 8 SMMO applications (Fig. 6) with different allocators on an NVIDIA Titan Xp GPU (12 GB device memory). We compared DynaSOAr with two state-of-the-art GPU allocators (*mallocMC* [22], *halloc* [1]) and with another baseline allocator BitmapAlloc that we developed. BitmapAlloc allocates objects in a large array and uses a bitmap to find empty slots. If possible, we also implemented variants without any dynamic allocation (*Baseline AOS/SOA*).

Running Time   We break down application running times by the amount of time spent on parallel enumeration (i.e., Fig. 4) and the remaining running time. Other allocators do not support parallel do-all out of the box, so we had to reimplement it for the benchmarks. With higher engineering efforts we could likely build a more efficient implementation. For a fair comparison of allocators, we should not take parallel enumeration time into account.

All applications except for sugarscape benefit from SOA (compare baselines AOS/SOA). DynaSOAr always exhibits better performance than the other allocators. The CUDA profiler indicated that this is due to better memory coalescing (SOA) in most cases. nbody and collision use little memory, so these applications benefit also from better L1/L2 cache utilization due to DynaSOAr's compact allocation policy.

Space Efficiency   To measure how efficient the allocators manage heap memory, we gave every allocator the same amount of memory and experimentally determined the maximum problem size before the application crashes with an out-of-memory error.

MallocMC and Halloc use a hashing approach to find empty memory locations during allocations. This requires little thread synchronization and usually results in fast allocations. However, the performance of every hashing approach decreases with the number of collisions. We believe this is why MallocMC and Halloc can utilize only a much smaller fraction of the heap compared to DynaSOAr and BitmapAlloc.

Many applications (e.g., traffic) cannot be implemented space-efficiently without dynamic allocation because the number of run-time objects of a type cannot be predicted accurately or changes over time.

5 Conclusion

DynaSOAr is a CUDA framework for SMMO applications on GPUs. The main insight of our work is that optimizing only for fast (de)allocations is not enough. Optimizing the access of allocated memory can result in much higher speedups, because memory access is the biggest bottleneck of many GPU applications. DynaSOAr achieves this by controlling both data layout (SOA) and data access patterns (parallel do-all), combined with a dense allocation policy that is optimized with hierarchical bitmaps.

Acknowledgments

References

[1] A. Adinetz and D. Pleiter. 2014. Halloc: A High-Throughput Dynamic Memory Allocator for GPGPU Architectures. https://github.com/canonizer/halloc. In *GPU Technology Conference 2014*.

(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
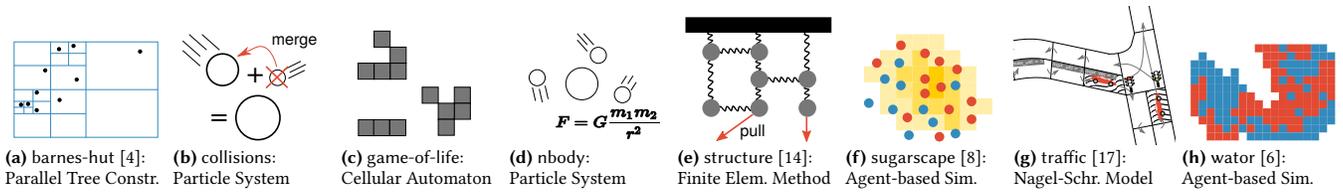**(h)** wator [6]: Agent-based Sim.

Figure 6. Visualization and classification of SMMO benchmark applications. The SMMO structure of each app. is explained in detail in the GitHub wiki page[1].
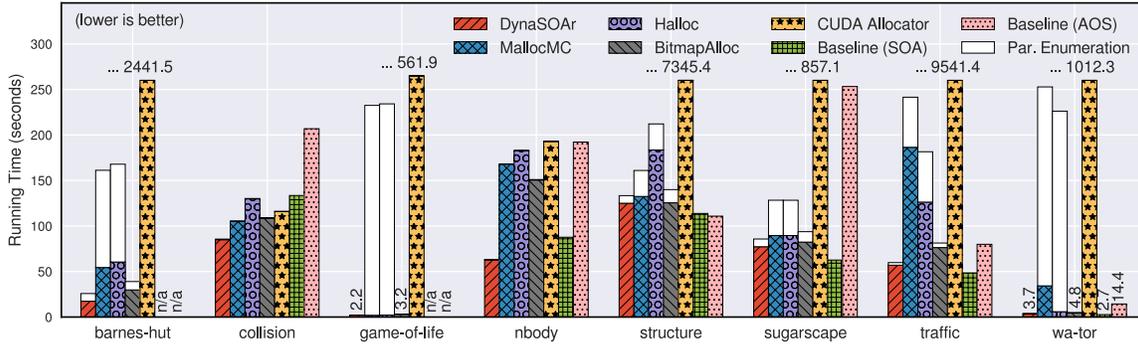

Figure 7. Running time of SMMO applications with different allocators.
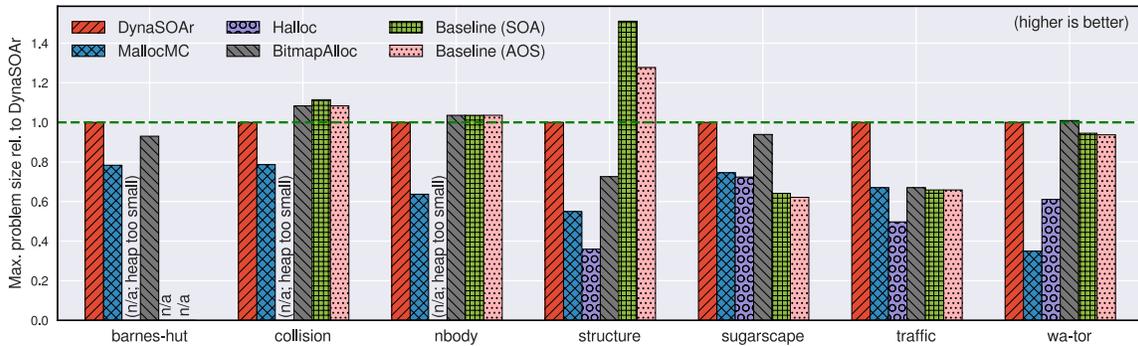

Figure 8. Space efficiency of SMMO applications with different allocators.

[2] D. Bakunas-Milanowski, V. Rego, J. Sang, and C. Yu. 2017. Efficient Algorithms for Stream Compaction on GPUs. *International Journal of Networking and Computing* 7, 2 (2017), 208–226.

[3] J. Bonwick. 1994. The Slab Allocator: An Object-caching Kernel Memory Allocator *(USTC '94)*. USENIX Association, 1–12.

[4] M. Burtscher and K. Pingali. 2011. Chapter 6 – An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm. In *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 75 – 92.

[5] S. G. De Gonzalo, S. Huang, J. Gómez-Luna, S. Hammond, O. Mutlu, and W. Hwu. 2019. Automatic Generation of Warp-level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs *(CGO 2019)*. IEEE Press, 73–84.

[6] A. K. Dewdney. 1984. Computer Creations: Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. *Scientific American* 251, 6 (12 1984), 14–26.

[7] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. 2007. SNZI: Scalable NonZero Indicators *(PODC '07)*. ACM, 13–22.

[8] J. M. Epstein and R. Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up* (1 ed.). Vol. 1. The MIT Press.

[9] I. Gelado and M. Garland. 2019. Throughput-oriented GPU Memory Allocation *(PPoPP '19)*. ACM, 27–37.

[10] M. P. Herlihy and J. M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.

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

[12] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. Hwu. 2010. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines *(CIT '10)*. IEEE Computer Society, 1134–1139.

[13] X. Li, W. Cai, and S. J. Turner. 2016. Supporting efficient execution of continuous space agent-based simulation on GPU. *Concurr. Comp.-Pract. E.* 28, 12 (2016), 3313–3332.

[14] X. Lu, B.Y. Chen, V.B.C. Tan, and T.E. Tay. 2018. Adaptive floating node method for modelling cohesive fracture of composite materials. *Eng. Fract. Mech.* 194 (2018), 240–261.

[15] T. Mattis, J. Henning, P. Rein, R. Hirschfeld, and M. Appeltauer. 2015. Columnar Objects: Improving the Performance of Analytical Applications *(Onward! 2015)*. ACM, 197–210.

[16] M. M. Michael. 2004. Scalable Lock-free Dynamic Memory Allocation *(PLDI '04)*. ACM, 35–46.

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

[18] P. Patel. 2006. *Object Oriented Programming for Scientific Computing*. Master's thesis. The University of Edinburgh.

[19] H. Song and S. Lee. 2017. Effects of wind and tree density on forest fire patterns in a mixed-tree species forest. *Forest Science and Technology* 13, 1 (2017), 9–16.

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

[21] M. Springer and H. Masuhara. 2019. DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access *(ECOOP '19)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (to appear).

[22] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg. 2012. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU *(InPar '12)*. IEEE Computer Society, 1–10.