東京工業大学
Tokyo Institute of Technology

Massively Parallel GPU Memory Compaction

Matthias Springer, Hidehiko Masuhara
Tokyo Institute of Technology

ISMM 2019

Introduction / Motivation

- *Goal:* Make GPU programming easier to use.
- *Focus:* Object-oriented programming on GPUs/CUDA.
  - Many OOP applications in high-performance computing.
  - DynaSOAr [1]: Dynamic memory allocator for GPUs.
  - **CompactGpu:** Memory defragmentation for GPUs, to make allocations more space/runtime efficient.

[1] M. Springer, H. Masuhara. **DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access.** ECOOP 2019.

Outline

1. Background: GPU Architecture

2. Memory Defragmentation: Concept and Main Ideas

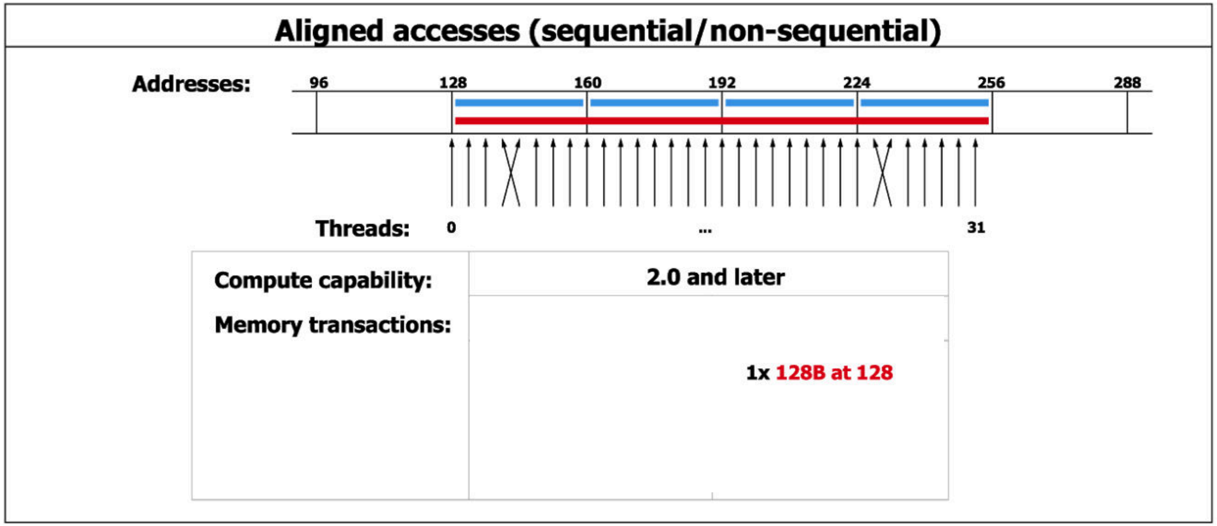
3. Defragmentation: Step by Step

4. Benchmarks

5. Conclusion

Background: GPU Architecture

Memory Coalescing

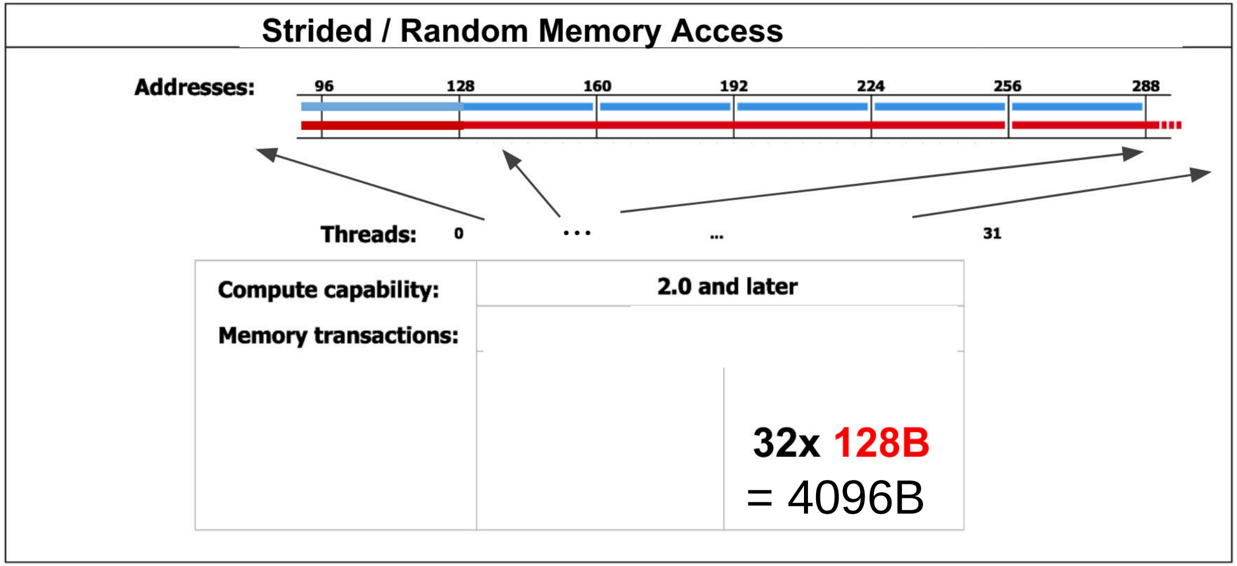


Source: CUDA C Programming Guide

If the threads of a physical core access memory within the same **aligned 128-byte window** (L1/L2 cache line), the those accesses are **combined into 1 memory transaction** by the memory controller.

Because the hardware really operates on **128-byte vector registers**.

Worst Case: No Memory Coalescing



Threads of a physical core (*warp*) access memory of totally different L1/L2 cache lines.

Before attempting any other optimization, try to improve memory coalescing!

Why GPU Memory Defragmentation?

東京工業大学
Tokyo Institute of Technology

- *Space Efficiency:* Reduce overall memory consumption.
  - Avoid premature out-of-memory errors.
- *Runtime Efficiency:* Vectorized access is more efficient.
  - **Accessing compact data requires fewer vector transactions (→more memory coalescing)** than accessing fragmented data.

Memory Defragmentation: Concept and Main Ideas

Dynamic Memory Allocation on GPUs

東京工業大学
Tokyo Institute of Technology

- Until recently, not supported well and not widely utilized yet
- Existing dynamic GPU memory allocators
  - CUDA allocators (`new`/`delete`): Extremely slow and unoptimized
  - Halloc [1], ScatterAlloc/mallocMC [2]: Very fast (de)allocation time
  - DynaSOAr [3]: Fast (de)allocation time, efficient access of allocations
- Memory allocation characteristics on GPUs
  - Massive number of concurrent (de)allocations
  - **Most allocations are small and have the same size** (due to mostly regular control flow)

Allows us the implement memory defrag. **more efficiently than on other platforms**.

[1] A. V. Adinetz and 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.
[3] M. Springer, H. Masuhara. **DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access.** ECOOP 2019.

Overview

- *CompactGpu:* A memory defragmentation system for the DynaSOAr memory allocator.
  - *Basic Idea:* Defragmentation by block merging.
  - *Optimization:* Fast pointer rewriting based on bitmaps.
  - Main CompactGpu techniques could be implemented in other allocators.

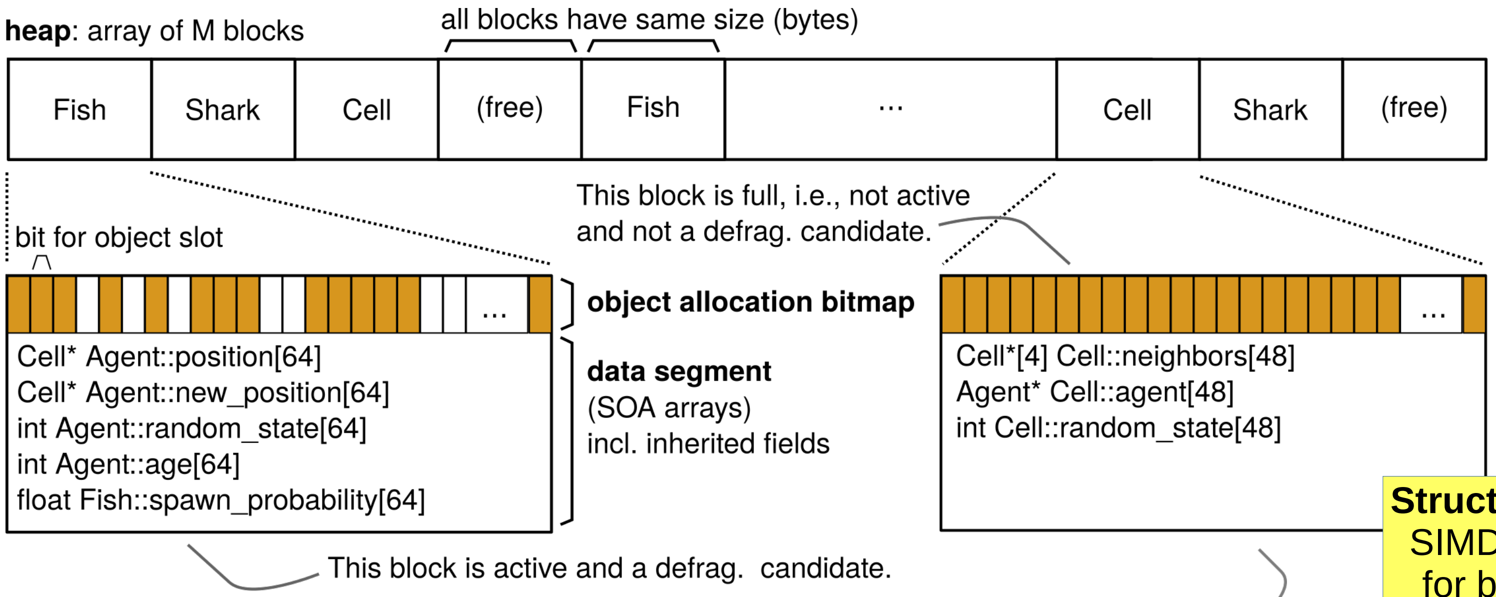
Main Design Choices and Requirements

東京工業大学
Tokyo Institute of Technology

- **In-place** defragmentation: To save space...
  - Defrag. by **block merging**: Combine blocks that are partly full.
- **Fully parallel** implementation
  - CompactGpu is a set of CUDA kernels.
- **Stop-the-world** approach: Run defragmentation when no other GPU code is running.
- **Manual**: Programmers initiate defragmentation manually or use a heuristic (e.g., defrag. after a large number of deallocations).

Overview: DynaSOAr Mem. Allocator [1]

東京工業大学
Tokyo Institute of Technology



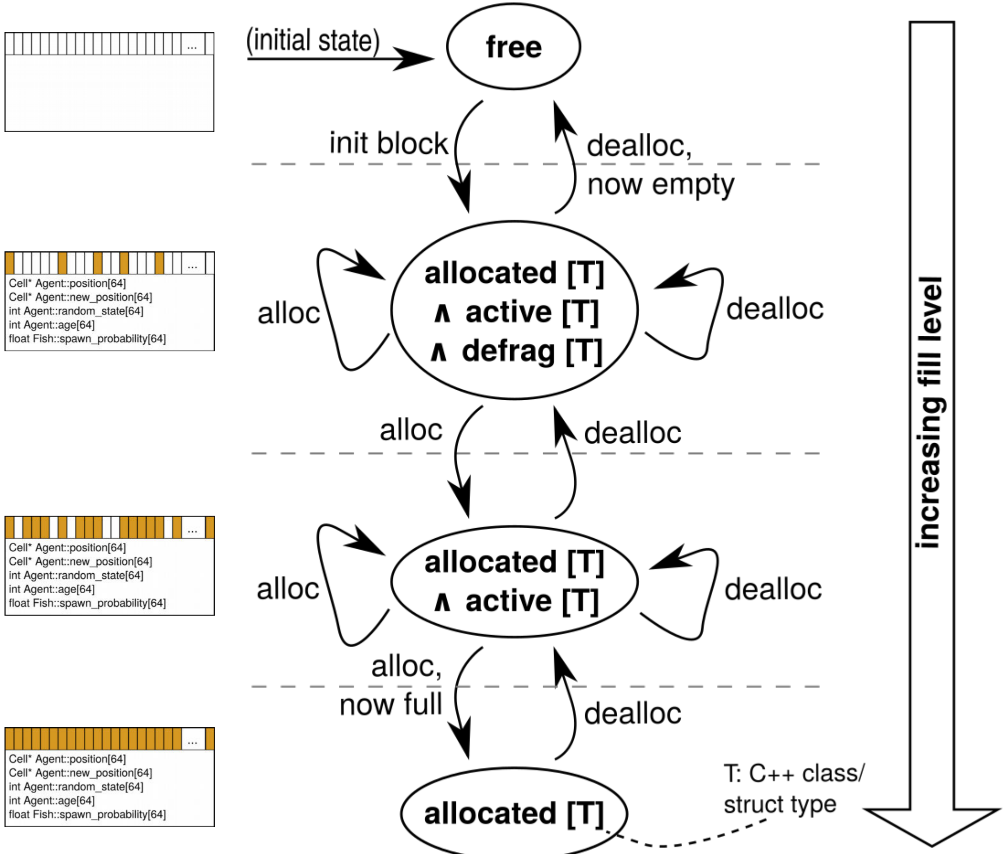
Running example:
Fish-and-Sharks simulation

Structure of Arrays (SOA):
SIMD/GPU **best practice**
for better vector access/
memory coalescing.

- Always allocate in active (non-full) blocks.
- Objects of same type stored in blocks in SOA data layout.

[1] M. Springer, H. Masuhara. **DynaSOAr: A Parallel Memory Allocator for Object-oriented Programming on GPUs with Efficient Memory Access.** ECOOP 2019.

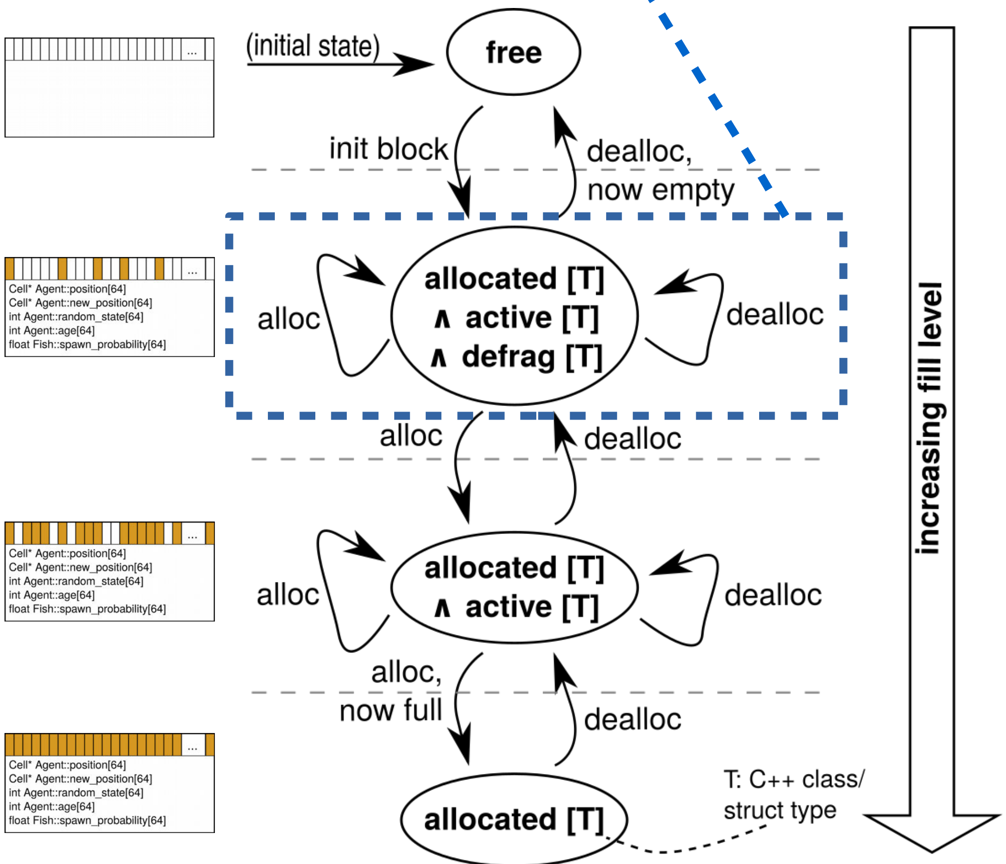
Block States



- **free:** Block is empty
- **allocated [T]:** Block contains at least 1 object of type T.
- **active [T]:** Block is allocated [T] and has at least 1 free slot.
- **defrag [T]:** Block is active [T] and is a *defragmentation candidate* (block with low fill level).

new with CompactGpu

Block States

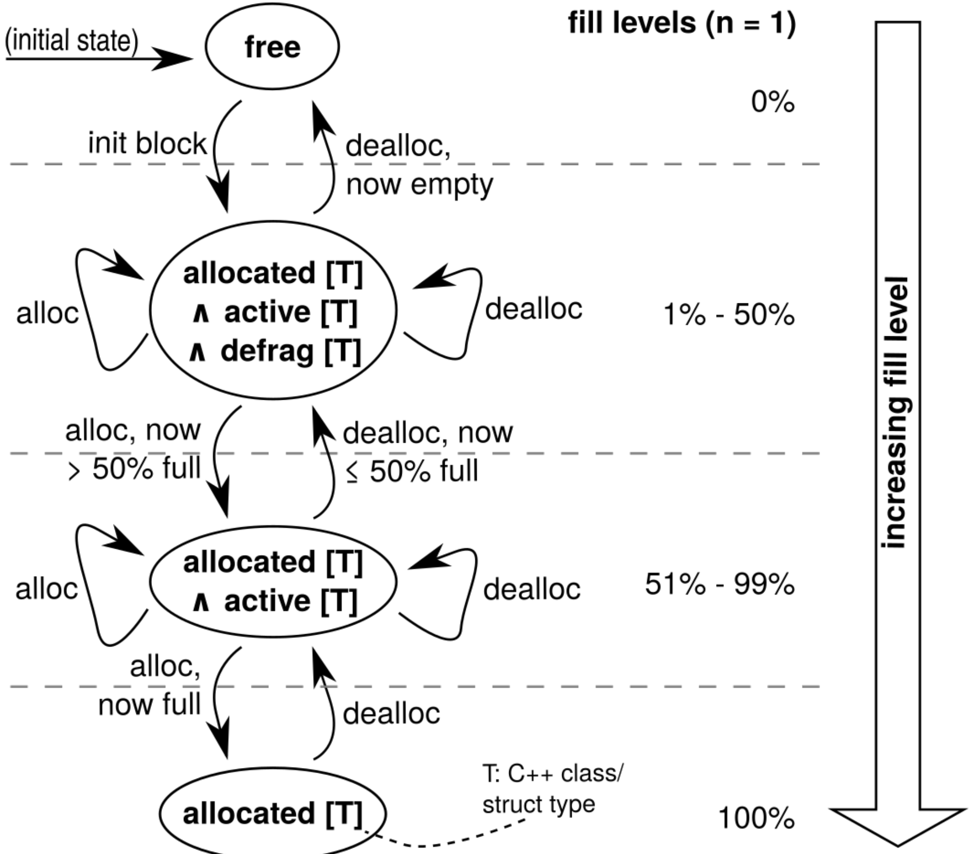


- **free:** Block is empty
- **allocated [T]:** Block contains at least 1 object of type T.
- **active [T]:** Block is allocated [T] and has at least 1 free slot.
- **defrag [T]:** Block is active [T] and is a *defragmentation candidate* (block with low fill level).

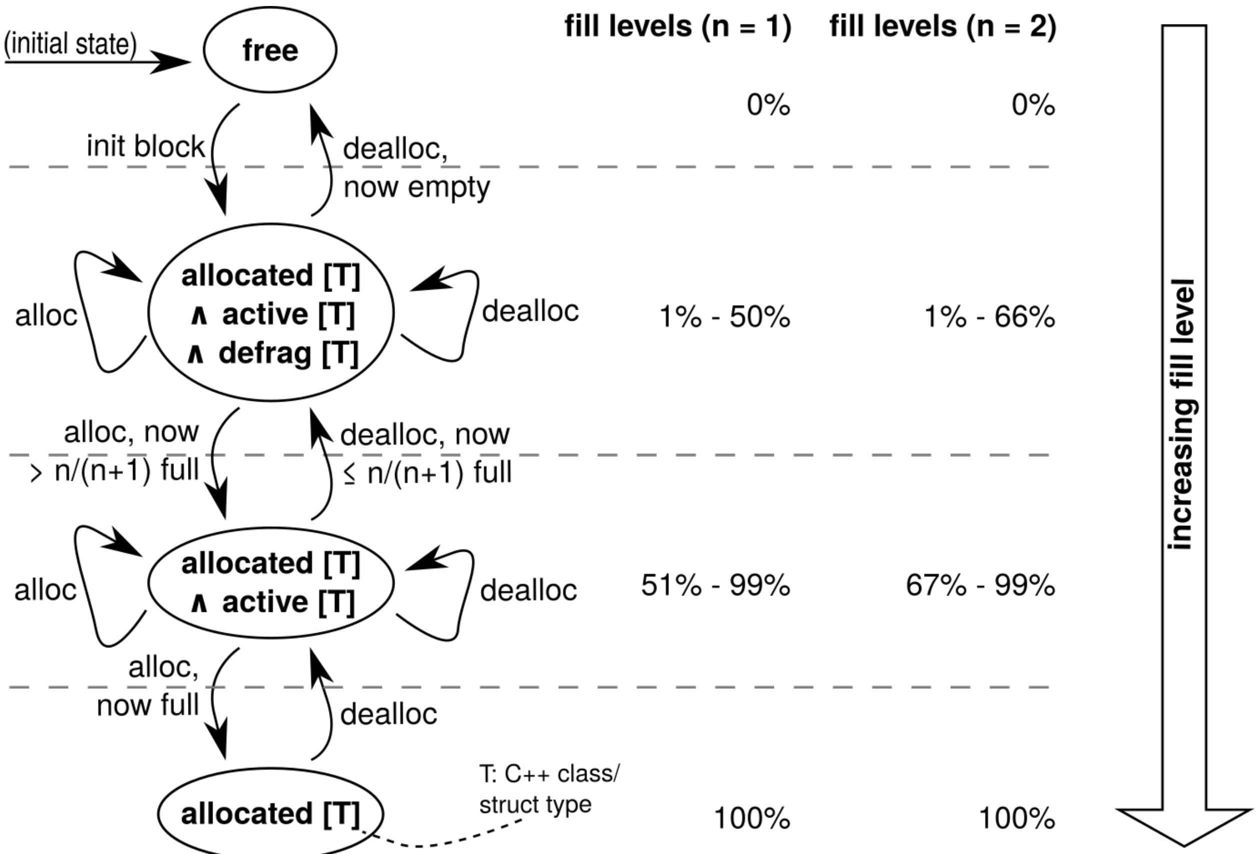
Defragmentation Factor

- *n* is the problem-specific **defragmentation factor** that must be chosen at compile time.
  - **Consider only blocks of fill level ≤ n/(n+1)** for defragmentation (*defrag. candidates*).
  - Move objects **from 1 source block into *n* target blocks**.
  - One defragmentation pass eliminates 1/(n+1) of all defragmentation candidates. Run **multiple passes** to eliminate all candidates.
  - Example: n = 1: Merge 2 blocks of fill level ≤ 50%.
  - Example: n = 2: Merge 3 blocks of fill level ≤ 66.6%.
  - In each case, the **source block is eliminated** by defragmentation.
- Higher *n* → More defragmentation
- Lower *n* → Less defragmentation, but faster (less work)

Block States

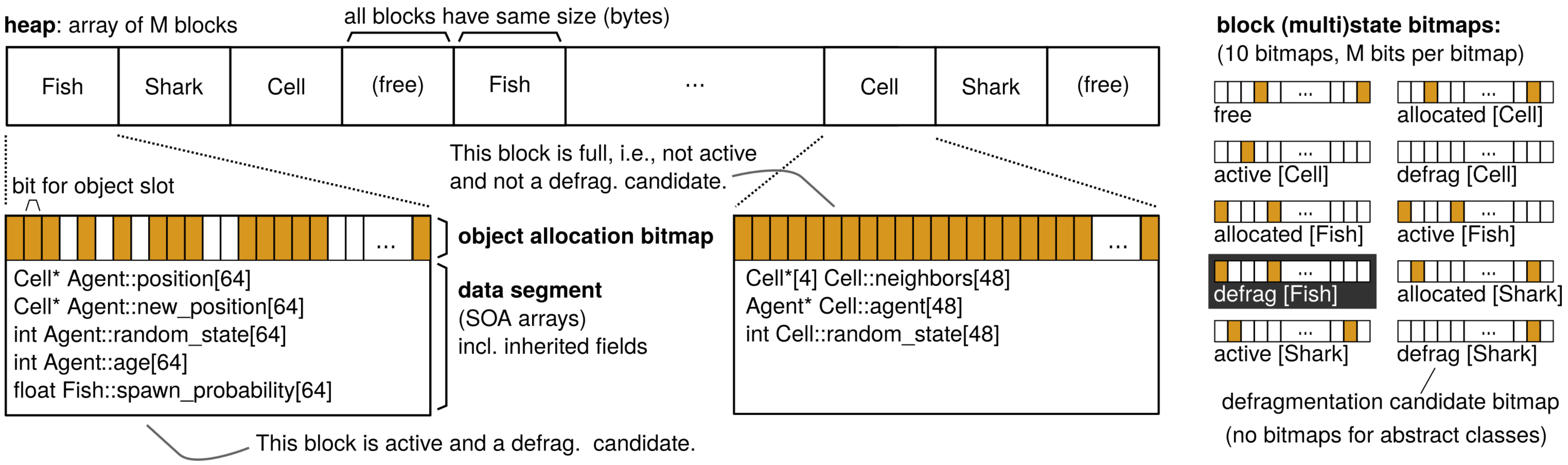


Block States



東京工業大学
Tokyo Institute of Technology

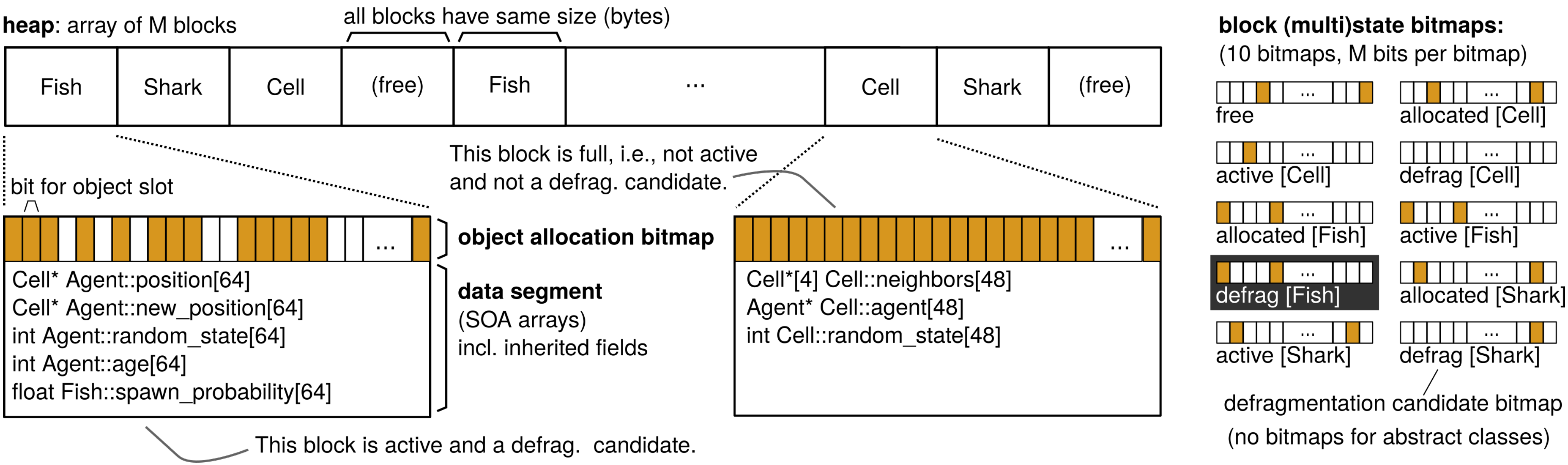
Block State Bitmaps



- DynaSOAr/CompactGpu indexes states in **block state bitmaps**.
- Newly introduced with CompactGpu: **defrag[T]**

Definition of Fragmentation

東京工業大学
Tokyo Institute of Technology

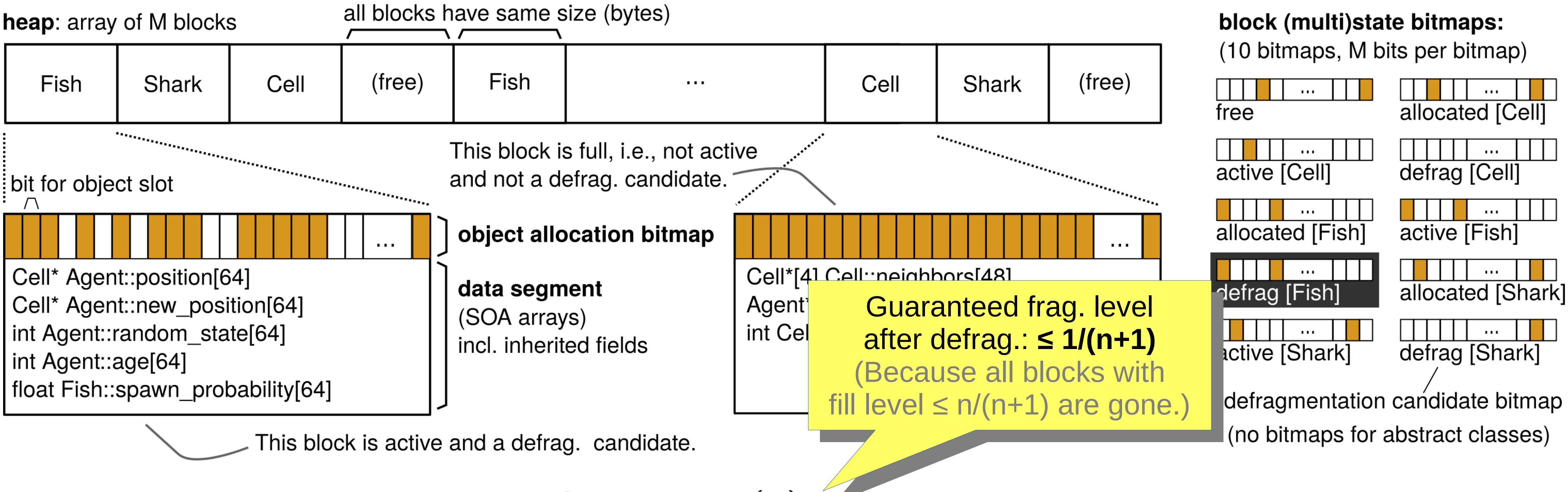


$$F = \frac{1}{\#\text{Blocks}} \sum_{b \in Blocks} \frac{\#\text{free slots}(b)}{\#\text{slots}(b)}$$

(considering only *allocated[?]* blocks)

Definition of Fragmentation

東京工業大学
Tokyo Institute of Technology

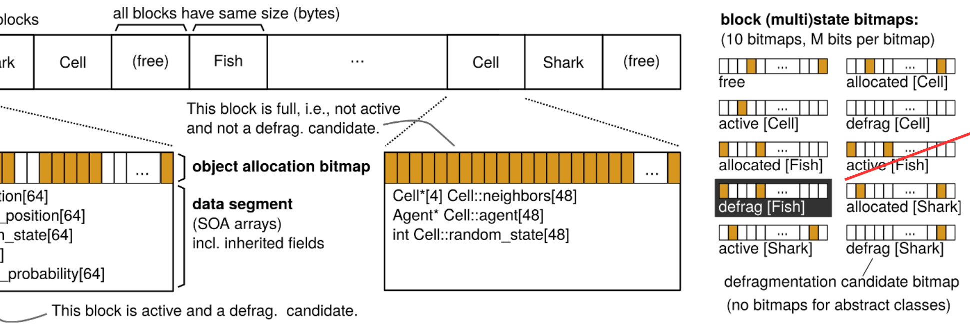


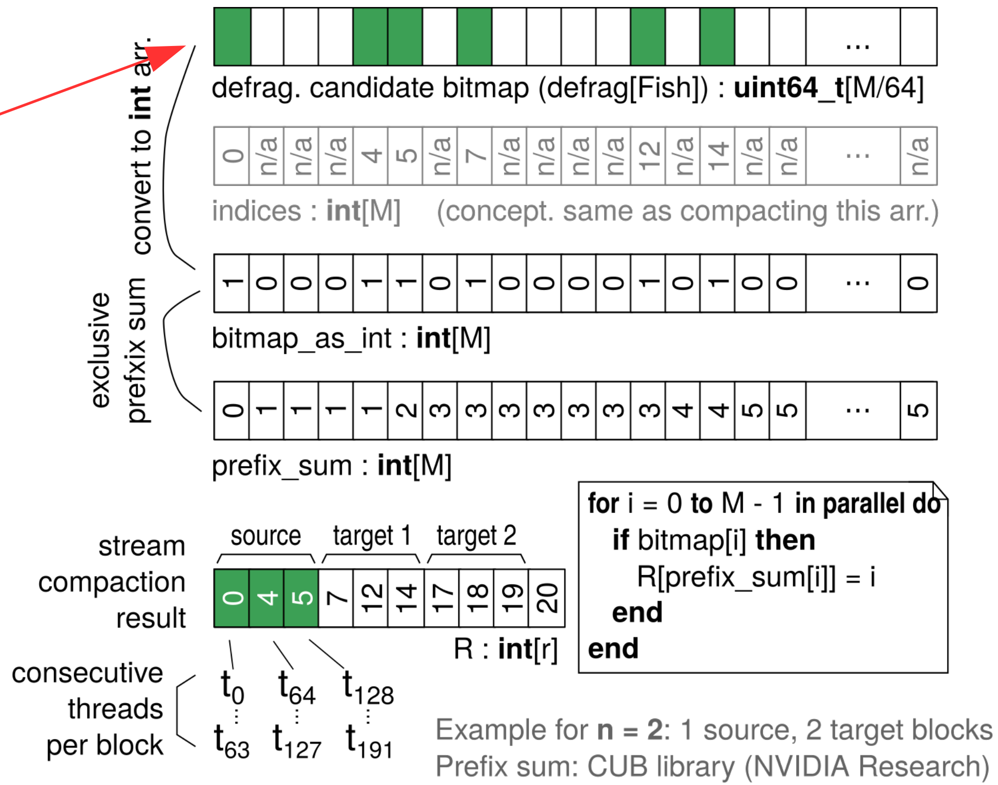
$$F = \frac{1}{\#\mathrm{Blocks}} \sum_{b \in Blocks} \frac{\#\mathrm{free\ slots}(b)}{\#\mathrm{slots}(b)}$$

(considering only *allocated[?]* blocks)

Defragmentation: Step by Step

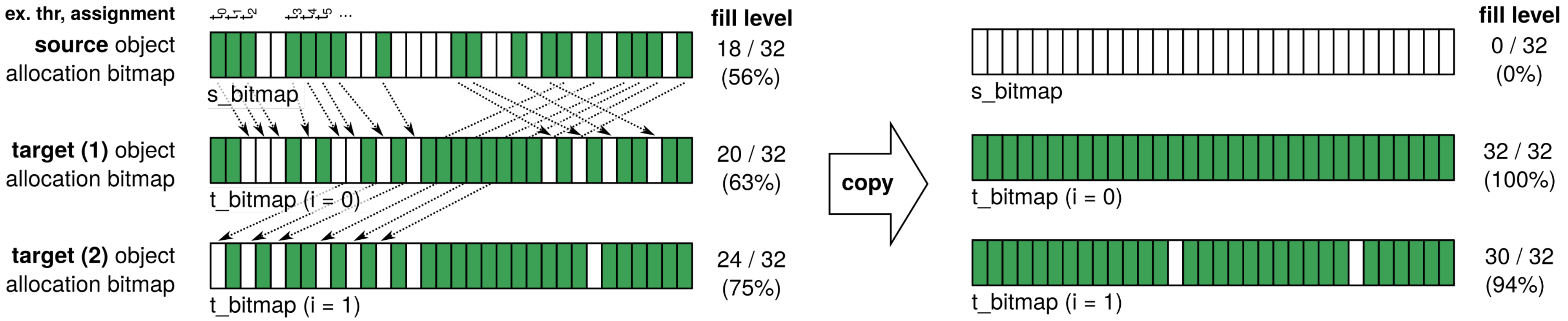
Choose Source/Target Blocks





- Compact defrag[T] bitmap. (exclusive prefix sum)
- Choose n target blocks for each source blocks.

Defragmentation by Block Merging

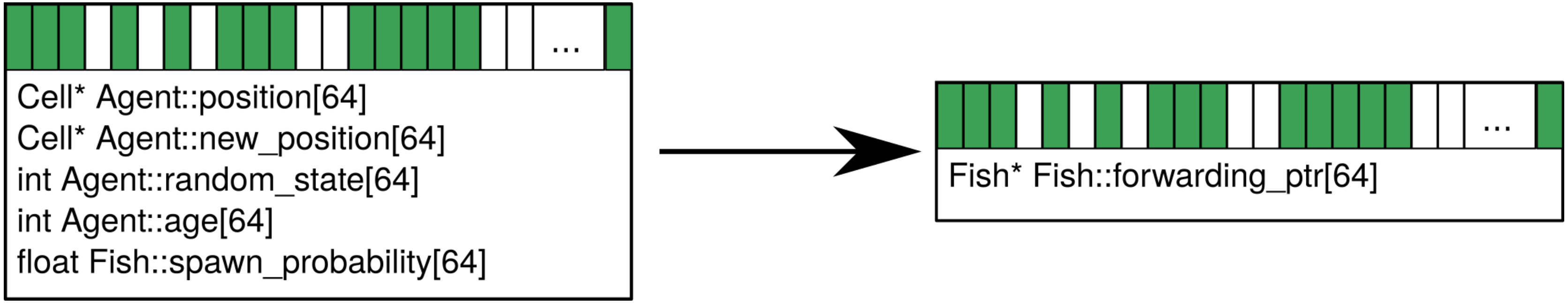


- Copy objects from a source block to *n* target blocks (in parallel).
- Source block is empty (new state: **free**), reducing fragmentation.
- **In-place** defragmentation mechanism.

Rewriting Pointers to Old Locations

東京工業大学
Tokyo Institute of Technology

- Store forwarding pointers in source blocks.



- *Afterwards:* Scan heap and find pointers to relocated objects. Rewrite those pointers.

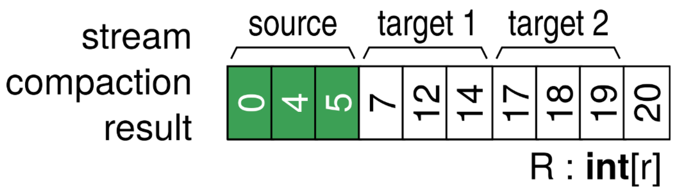
Rewriting Pointers to Old Locations

東京工業大学
Tokyo Institute of Technology

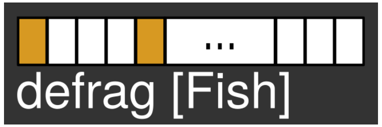
- Scan heap and look for anything that looks like a pointer.
- Rewrite if **bid < R[r/n]** and block is a defrag. candidate.

```
for all Fish*& ptr in parallel do
    s_bid = extract_block_id(ptr)
    if s_bid < R[r/n] && defrag[Fish][s_bid] then
        s_oid = extract_object_id(ptr)
        ptr = heap[s_bid].forwarding_ptr[s_oid]
    end
end
```

Condition 1: bid < 7 (i.e., source range)



Condition 2: defrag[Fish][bid] (i.e., defrag. cand.)



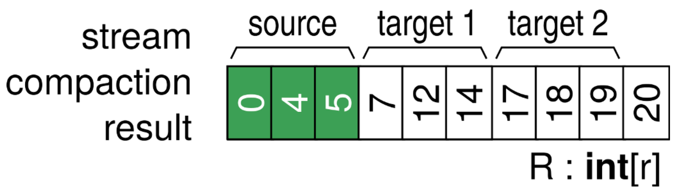
Rewriting Pointers to Old Locations

- Scan heap and look for anything that looks like a pointer.
- Rewrite if **bid < R[r/n]** and block is a defrag. candidate.

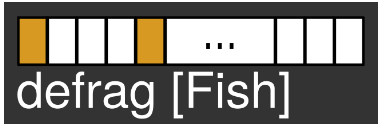
```
for all Fish*& ptr in parallel do
    s_bid = extract_block_id(ptr)
    if s_bid < R[r/n] && defrag[Fish][s_bid] then
        s_oid = extract_object_id(ptr)
        ptr = heap[s_bid].forwarding_ptr[s_oid]
    end
end
```

- Defrag bitmap largely cached.
- **2 mem. reads + 1 write** if pointer rewritten
- **1 mem. read** otherwise

Condition 1: bid < 7 (i.e., source range)

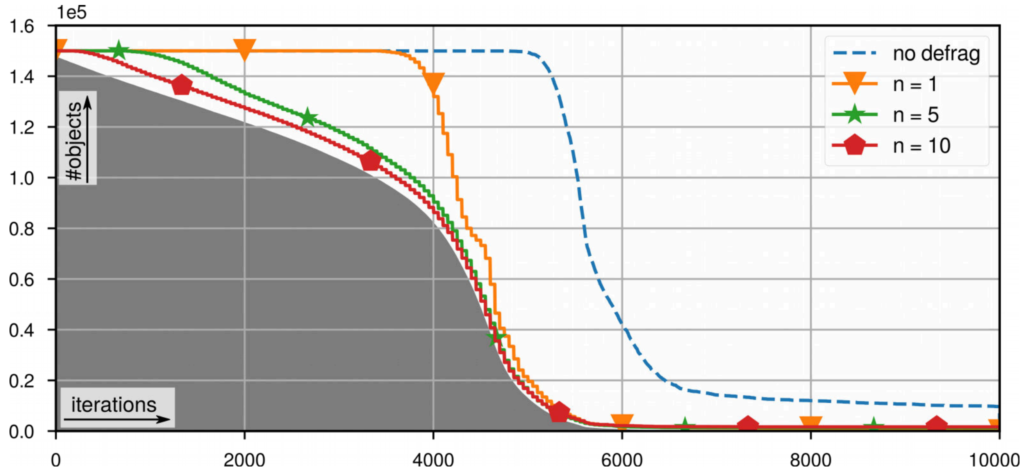


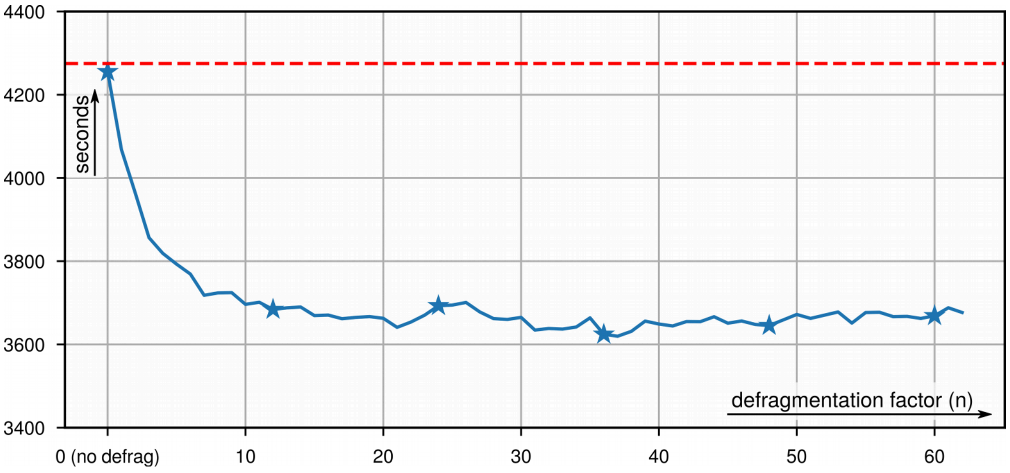
Condition 2: defrag[Fish][bid] (i.e., defrag. cand.)



Benchmarks

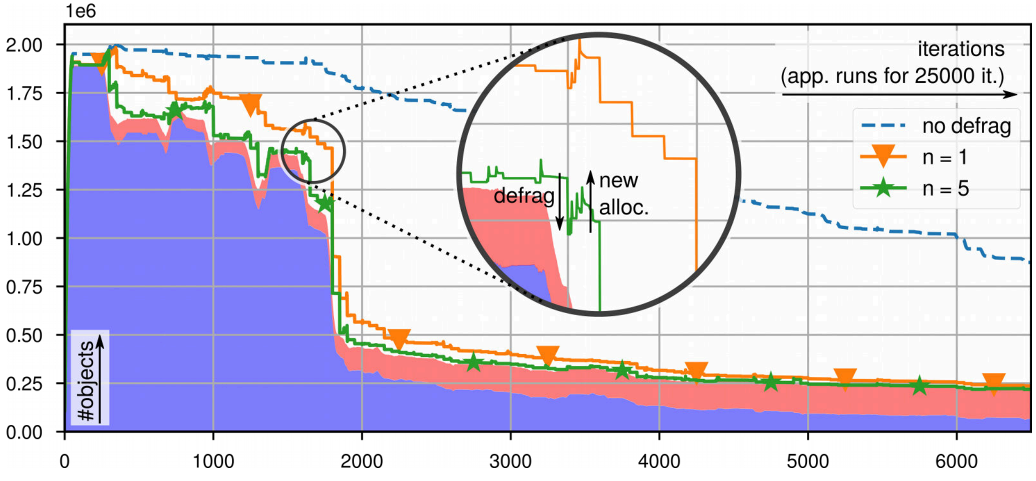
Benchmark: N-Body with Collisions

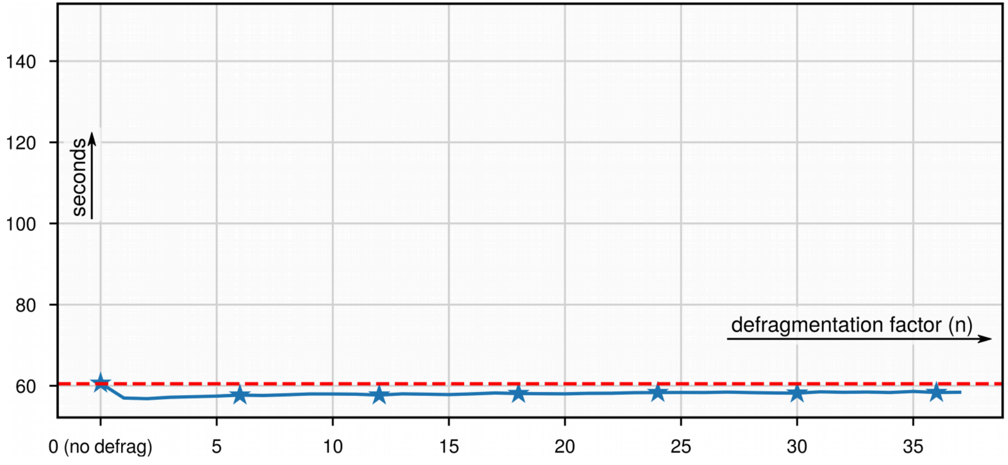




- Memory consumption drops faster.
- Performance improvement: 12%

Benchmark: Generational Cellular Automaton





- Memory consumption drops faster.
  - *Too much* defragmentation leads to **overcompaction**.
- Performance improvement: 6%

東京工業大学
Tokyo Institute of Technology

Conclusion

Conclusion

東京工業大学
Tokyo Institute of Technology

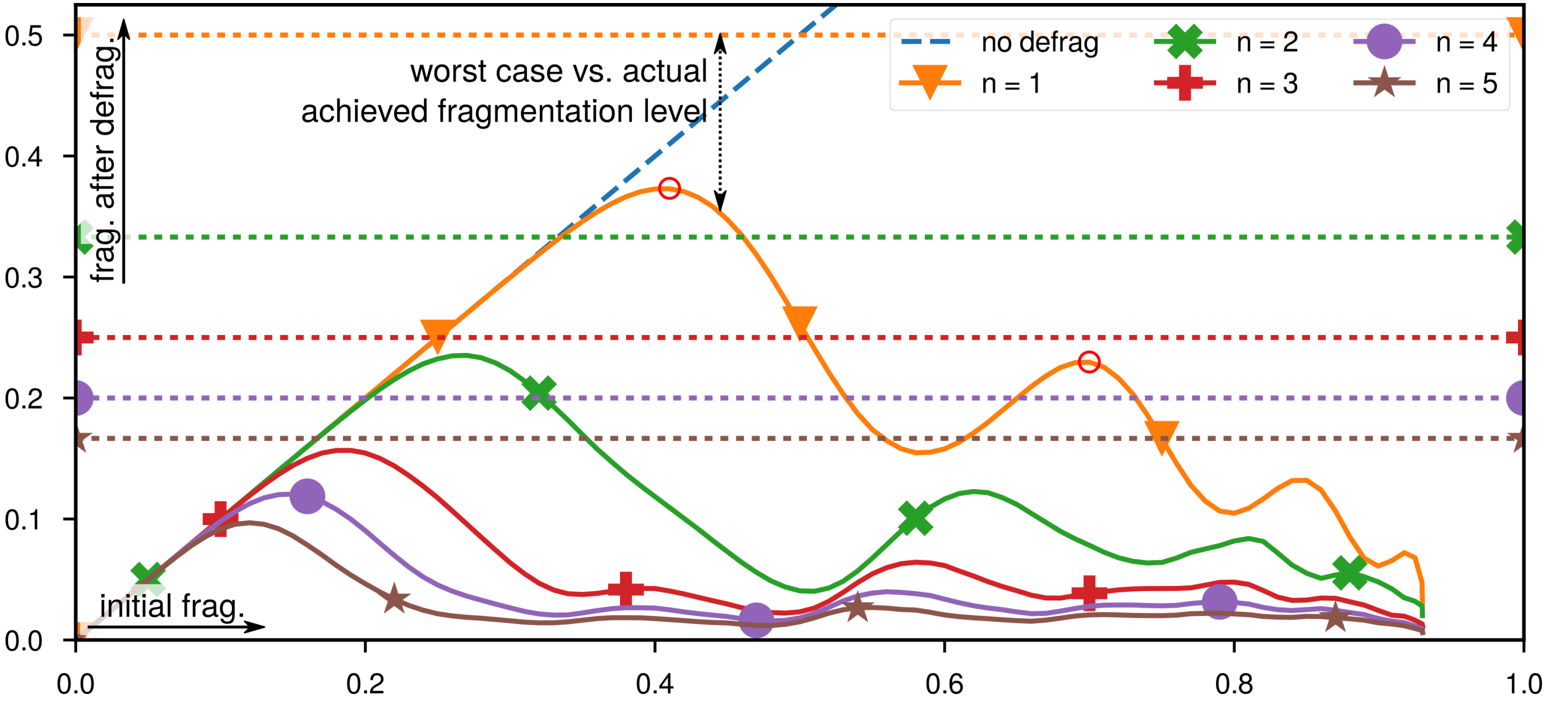
- Efficient memory defragmentation is **feasible on GPUs**.
- Besides saving memory, defragmentation makes usage of allocated memory more efficient (**better mem. coalescing**).
- GPU memory allocation patterns allow us to implement defragmentation efficiently.
- Certain CPU technqiues (e.g., recomputing forwarding pointers on the fly [1]) do not pay off on GPUs.

[1] D. Abuaiadh, Y. Ossia, E. Petrank, U. Silbershtein. An Efficient Parallel Heap Compaction Algorithm. OOPSLA 2004

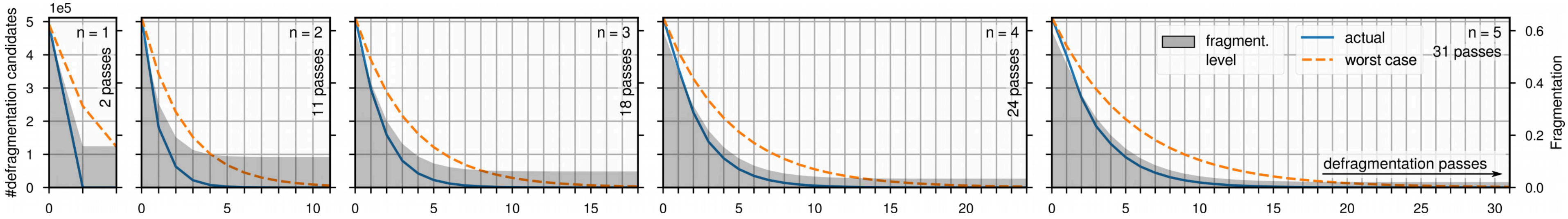
Appendix: Microbenchmarks

Achieved Fragmentation Level

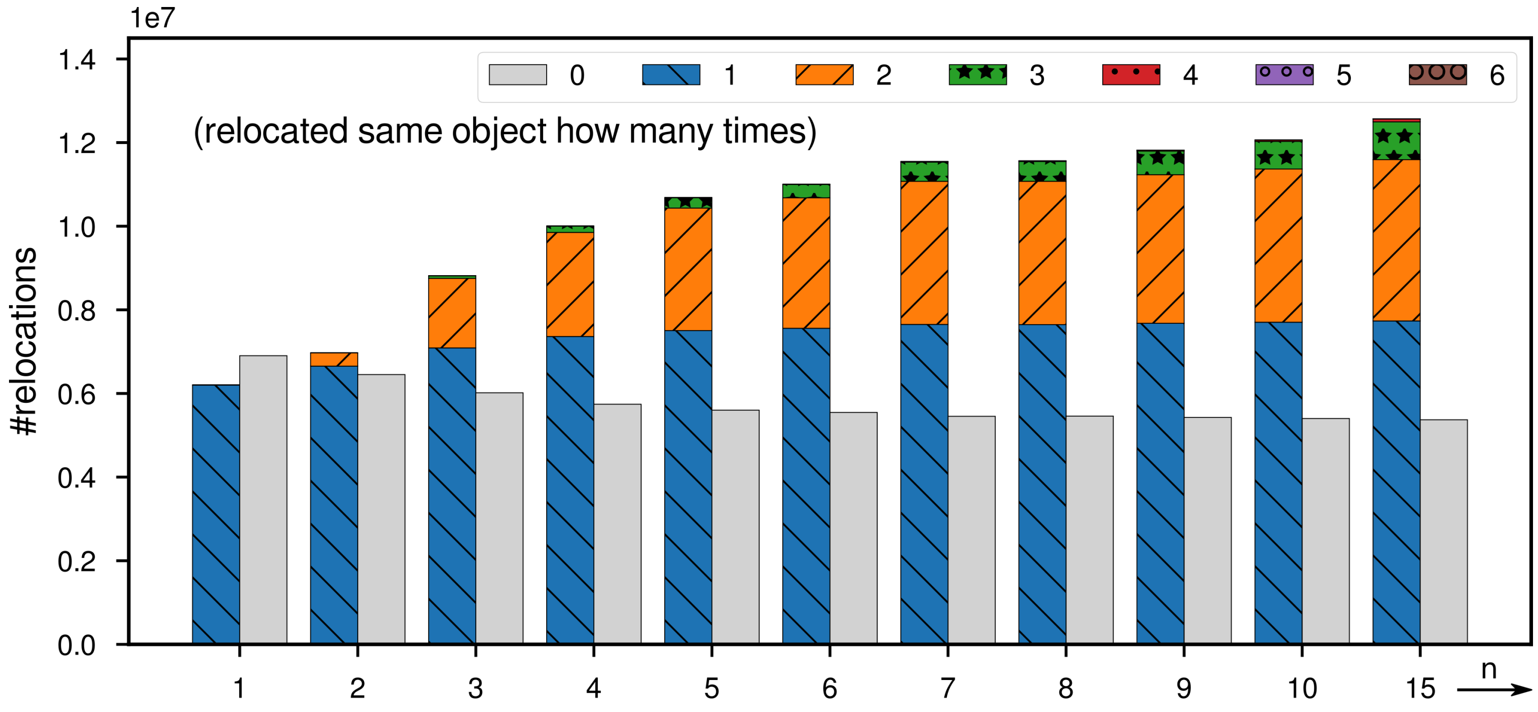
東京工業大学
Tokyo Institute of Technology



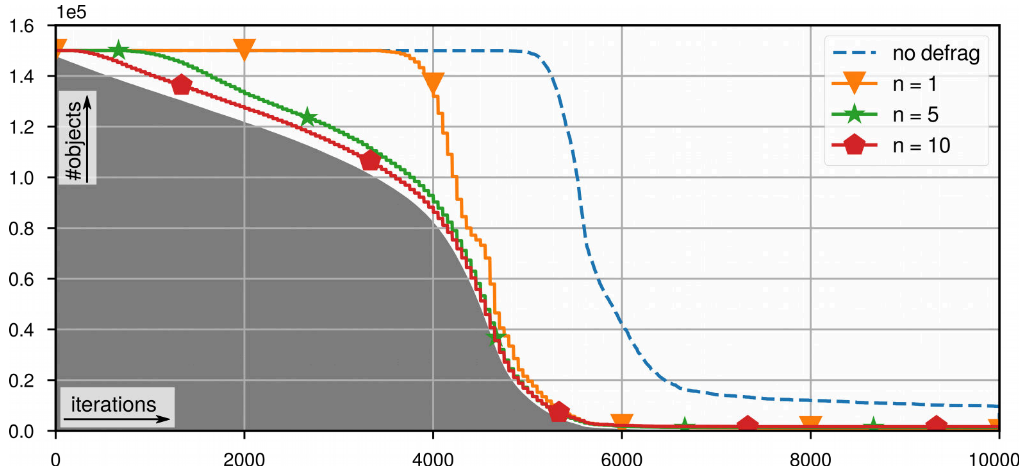
Number of Defragmentation Passes

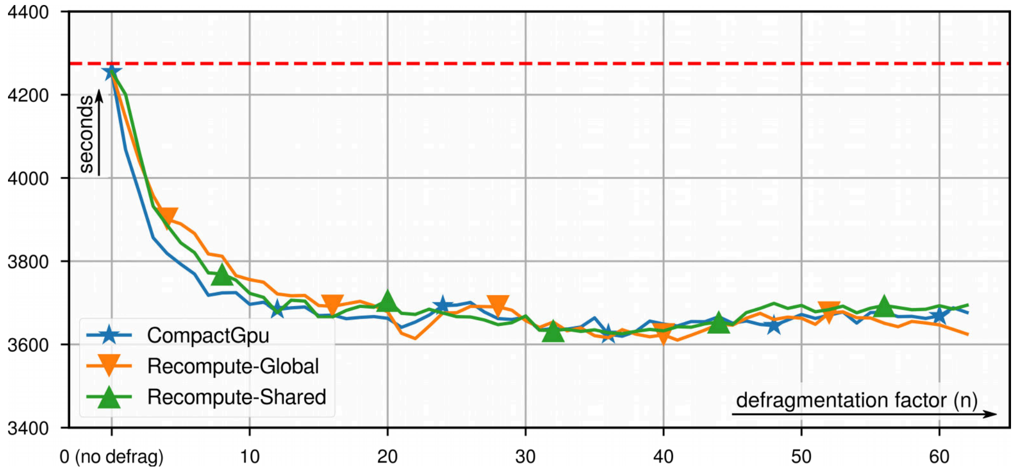


Number of Object Copies



Benchmark: N-Body with Collisions

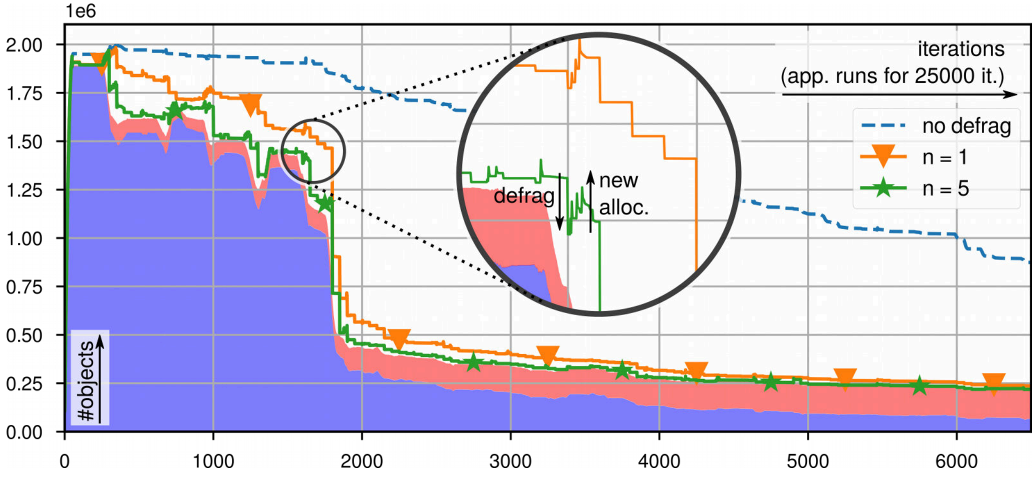


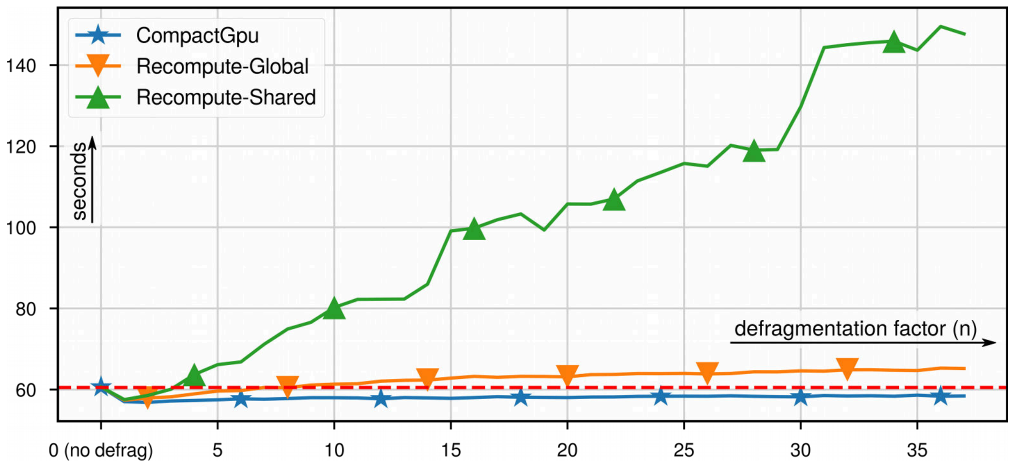


- Memory consumption drops faster.
- Performance improvement: 12%

Benchmark: Generational Cellular Automaton

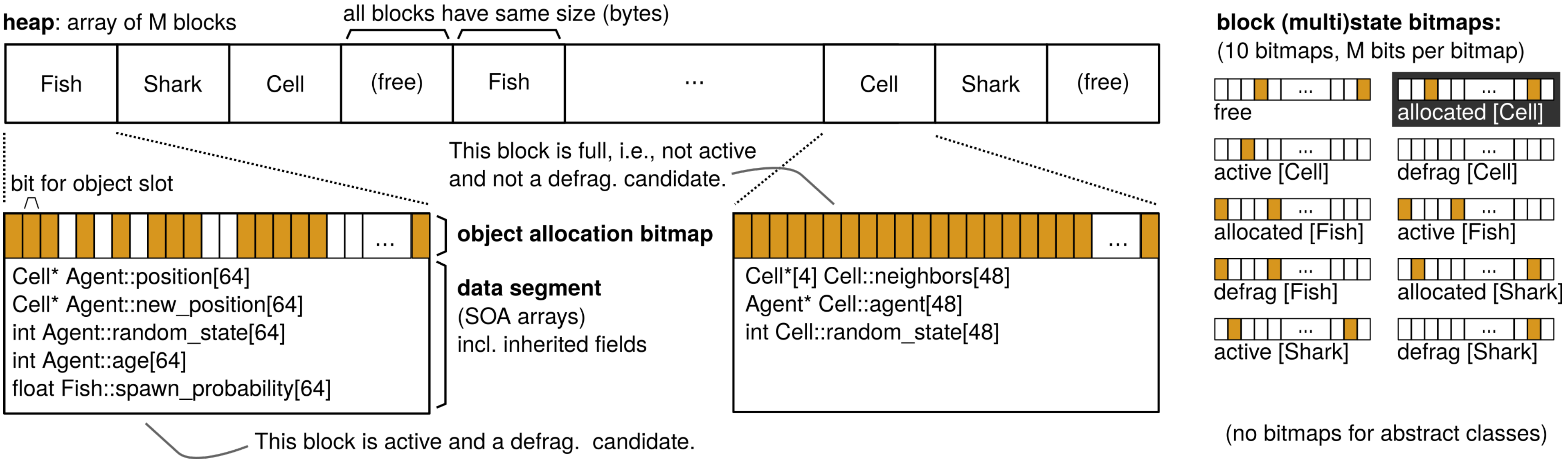
東京工業大学
Tokyo Institute of Technology





- Memory consumption drops faster.
  - *Too much* defragmentation leads to **overcompaction**.
- Performance improvement: 6%

Reducing Heap Scan Area



- Allocator has detailed information about the **structure of allocations**.
- Only `Cell` has a pointer to `Agent`. Only look into *allocated[Cell]* blocks.

Background: GPU Architecture

- 20 symmetric multiprocessors (SMs)
- 128 CUDA cores per SM
- *Total:* 20*128 = 2560 CUDA cores
- *But in reality:* 20*4 physical cores, each operating on **128-byte vector registers**

CUDA gives programmers the **illusion** of having 2560 cores.

Memory controller accesses memory in 128-byte blocks

Source: NVIDIA GeForce GTX 1080 Whitepaper