Parallel Memory Defragmentation on a GPU

Ronald Veldema, Michael Philippsen
University of Erlangen-Nuremberg
Germany

Informatik 2 • Programmiersysteme • Martensstraße 3 • 91058 Erlangen

Motivation

- Application programmers want Java/C#/Python + Performance
  - ☐ Our idea: Java + OpenMP (JaMP)

```
//#omp parallel for
for (int i=0; i<N; i++)
    a[i]++;
```

Runs on GPU

  - ☐ Parallel loops automatically converted to Cuda/OpenCL code on the fly

Motivation

- Application programmers want Java/C#/Python + Performance
  - Our idea: Java + OpenMP (JaMP)

```
//#omp parallel for
for (int i=0; i<N; i++)
      a[i] = new Object();
```


  - Parallel loops automatically converted to Cuda/OpenCL code on the fly
    - Java/C# programmers rely on automatic memory
      - Even in parallel context / OpenMP
    - Now need a memory allocator+garbage collector on the GPU

Cuda++

- We need a simple prototype for experimentation
  - Going for Java/OpenMP is Future Work

Our Cuda++

```
//@GPU
class Test {
 void zoo(int[] a) {}

 kernel static void foo() {
  int id = kernel_id();
  Test t = new Test();
  if (t != null)  {
   int x = random();
   t.zoo(new int[x]);
}}

void main() {
  parallel_call[1024] Test.foo()
  garbage_collect();
}
```

Nvidia's CUDA

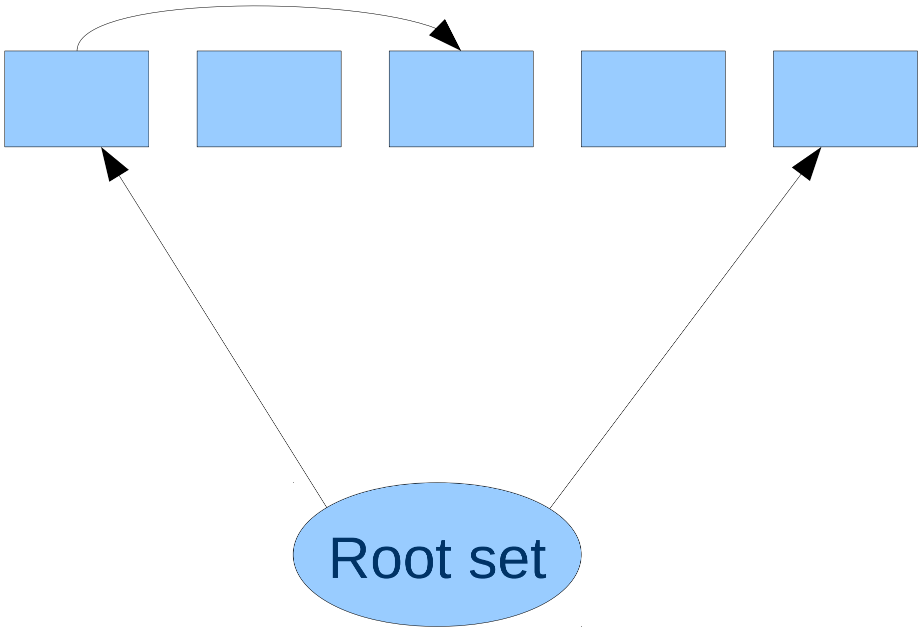
```
__global__ void Test__foo() {
  int id =  ( (blockIdx.x * blockDim.x) + threadIdx.x );
  Test *t = alloc_object(sizeof(Test));
  If (t != null) {
     int x = random();
    Test__zoo(t, alloc_array(sizeof(int),x));
  }
}

void main() {
    Test__foo<<<1024/64, 64>>> ();
    garbage_collect_and_defragment();
}
```

Motivation

- Garbage collection on a GPU
  - Needs to be highly parallel, high-throughput
  - Shouldn't copy memory much (bandwidth costs, potential allocation bottleneck to use a single to-space)
  - Mark-and-sweep collector
      - Mark all objects in parallel
      - Sweep all objects in parallel
    - See paper
      - “Iterative data-parallel mark&sweep on a GPU”
      - ISMM'11
  - Don't want to pay for defragmentation all the time
    - If a large array allocation on the GPU fails
      - Try GC
      - If that GC fails, then defragment
  - Don't want to defragment the whole heap, only enough to cont.

Motivation

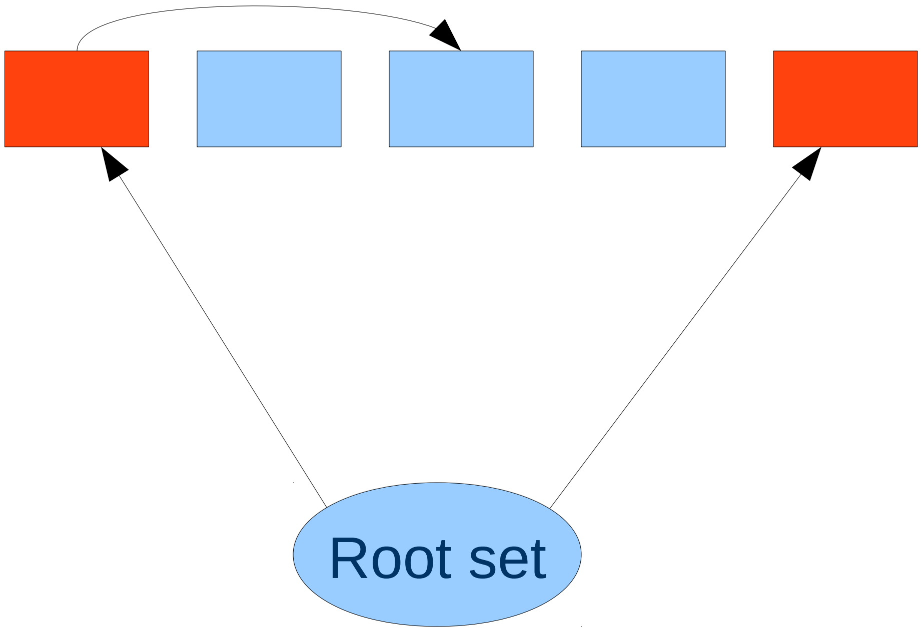
- Problem: mark&sweep does not guarantee a unfragmented heap:



Step #1: find root set (all global variables in parallel)

Motivation

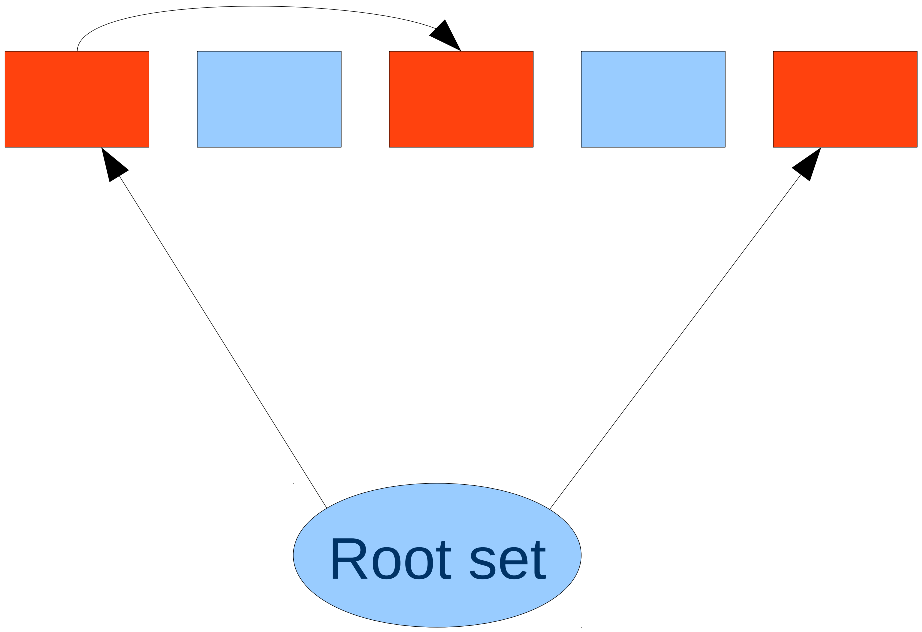
- Problem: mark&sweep does not guarantee a unfragmented heap:



Step #2: scan directly reachable objects (all in parallel)

Motivation

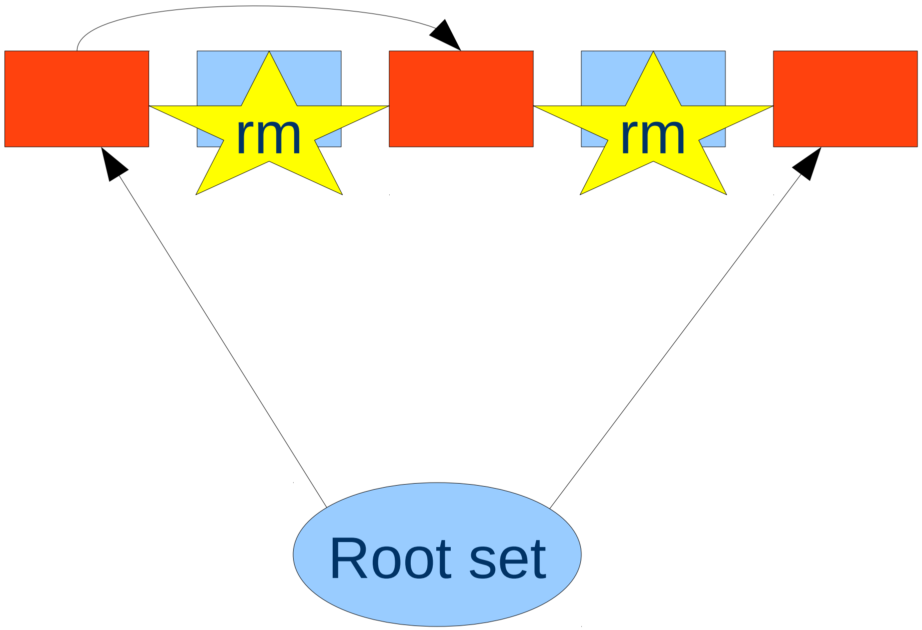
- Problem: mark&sweep does not guarantee a unfragmented heap:



Step #3: scan indirectly reachable objects (all in parallel)

Motivation

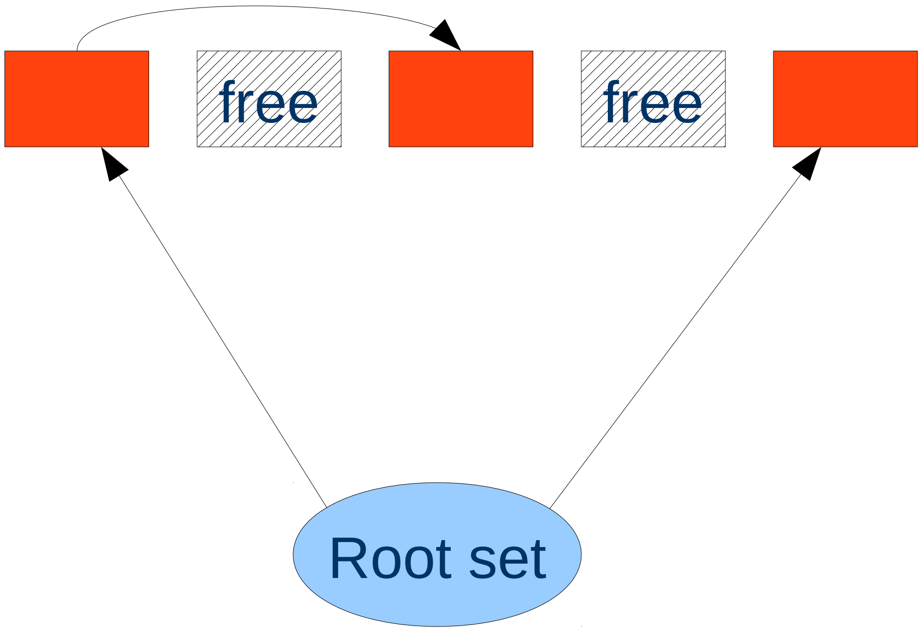
- Problem: mark&sweep does not guarantee a unfragmented heap:



Step #4: look at all objects in parallel, remove non-marked ones

Motivation

- Problem: mark&sweep does not guarantee a unfragmented heap:

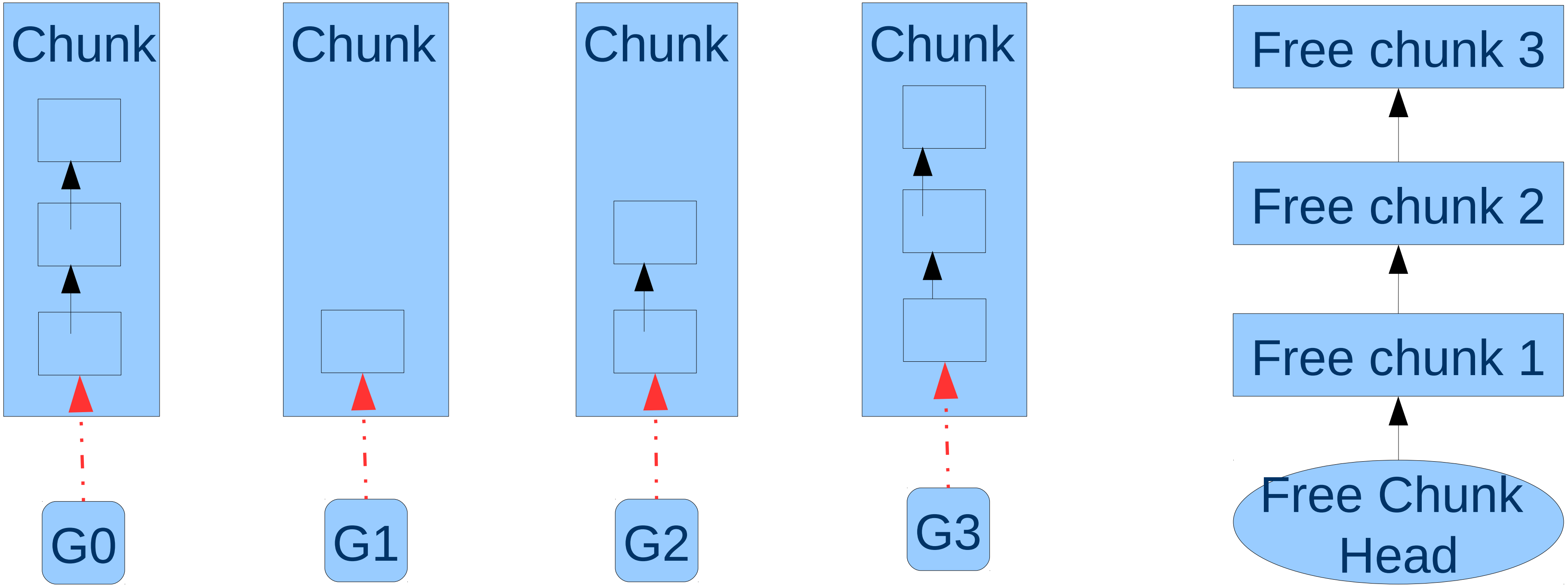


Cannot allocate:  because no long enough stretch exists

The Allocator

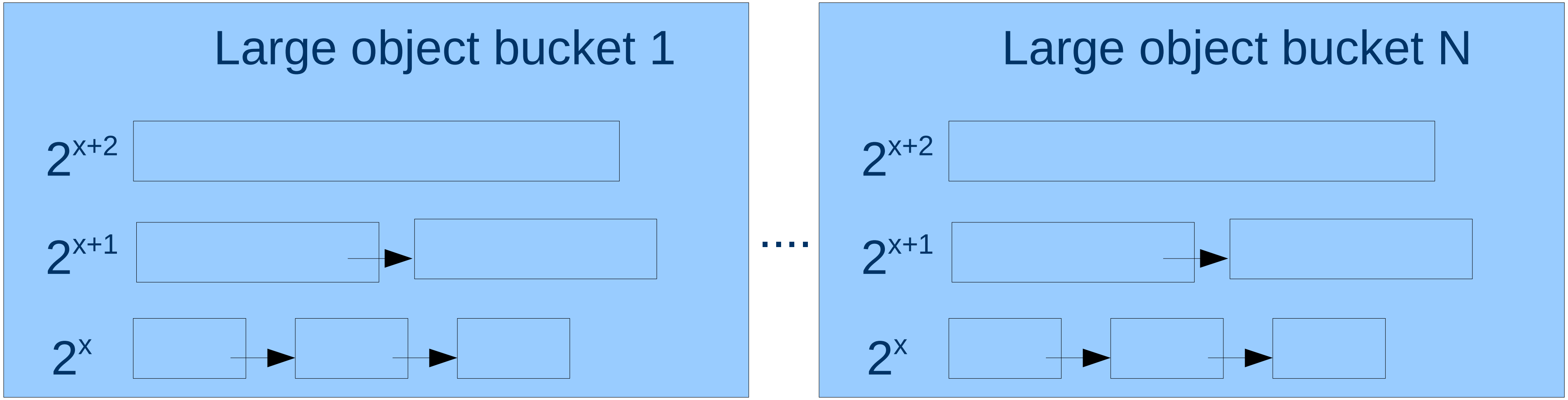
Defragmentation: allocator data structures

- Memory is partitioned into 64 K chunks
  - Each chunk has a free-list of small (64 byte) holes
  - Each GPU-thread has a reference to the chunk to allocate from
  - Multiple global lists of chunks with free-memory exist
    - When a GPU-thread exhausts its chunk, it can get a new chunk



Defragmentation: allocator data structures

- Buckets of power-of-2 size large memory holes
  - Single linked list with next pointer in each hole
  - Multiple large object buckets to reduce contention
- If list 2^x is empty, get memory from 2^{x+1}, etc.
  - Leftover memory is placed back into list 2^x and any small-object chunks
    - A cause for memory to become fragmented!



Defragmentation: allocator data structures

- The allocator/GC maintain a map with 1 byte per 64 bytes of memory
  - allocation_map[address/64] == 1, if address starts an allocated object
- Trick: we can now start a kernel per potentially allocated object:

```
kernel void foo() {
    void *address = heap + (kernel_id() * 64);
    If (allocation_map[address/64] == 1)
            parallel_work_allocated_object(address);
}


parallel_for i=0 to max-address / 64:
        foo();
```

The Defragmentator

Defragmentation

- Definitions:
  - Defragmentation = compaction + coalescing
  - Compaction = move objects together
  - Coalescing = combine free-memory holes
- Compaction+Coalescing both parallel
  - Need to select where to apply parallelism to avoid excessive synchronization
- “Got-ya” problems on a GPU
  - 'divergence': need a best-effort to make each GPU-thread do the same thing in lock step
  - Cannot use a spin-lock on a GPU because the hardware scheduler will always mark the spin-locker 'runnable'
    - Will then not schedule the 'winner' but the loser of the try-lock operation....

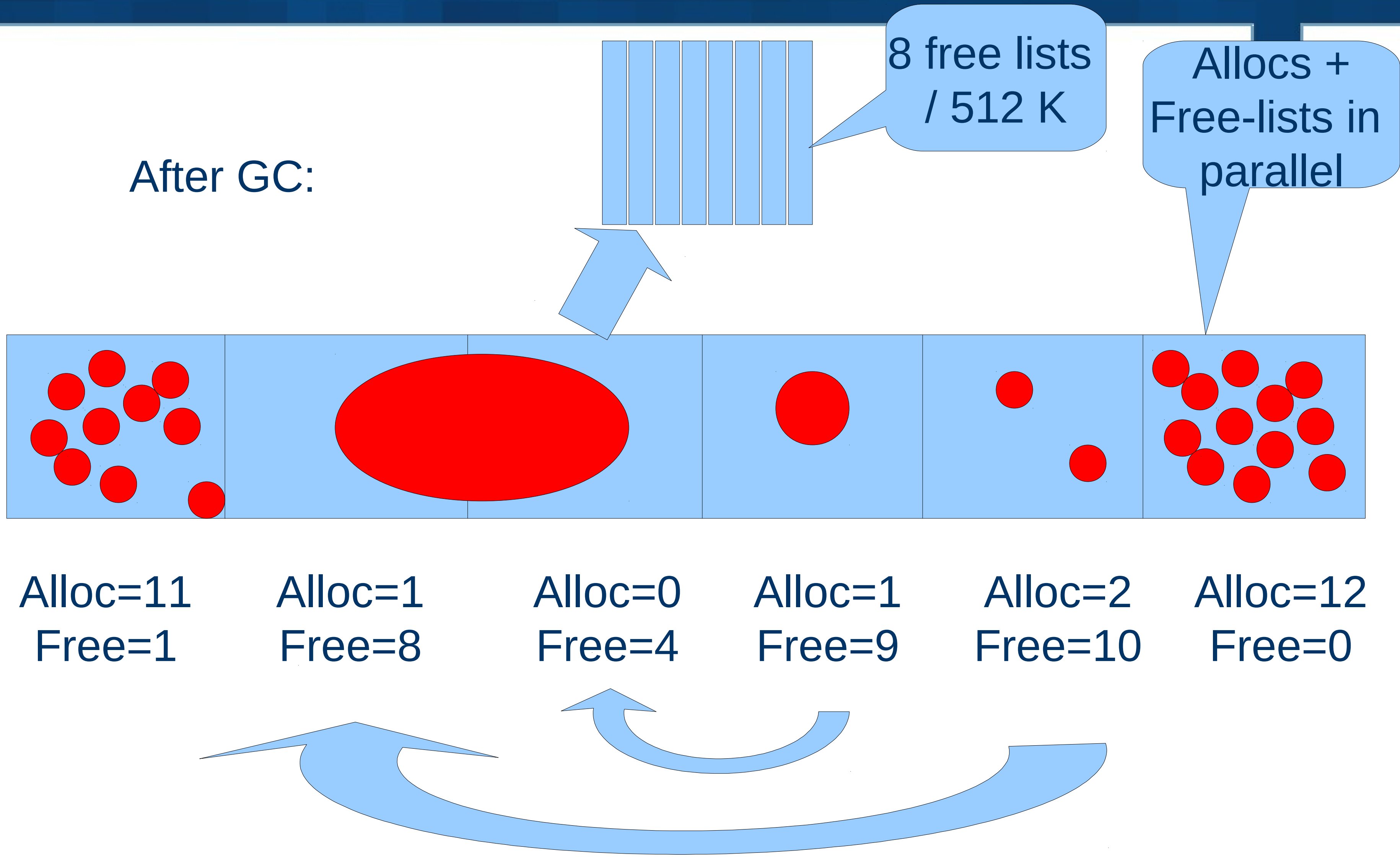
Defragmentation: Compaction

- 3 steps:
  - 1) Select regions of memory to defragment
    - defragmenting all of the memory would be expensive
    - Gathering information about each region of memory can be done in parallel
  - 2) Move objects inside regions
    - Each object can be moved in parallel
    - Need to allocate memory in the destination region
      - Potentially expensive: contention over the free-lists in destination regions
  - 3) Retarget pointers everywhere to moved objects
    - All pointers can be moved in parallel
    - Trick: we can exclude regions of memory we are sure contain no pointers to the source region(s)

Defragmentation: Compaction

- Selection of regions to defragment
  - Want to defragment 10% of the heap
- Partition the heap in 512 Kbyte compaction regions
  - Count the number of allocated objects in a region
    - A GPU-thread per potential object
  - Count the number of elements in the free-lists
    - A GPU thread per free-list
      - A small-free-chunk is maintained per 64 Kbyte
        - 512 Kbyte / 64 Kbyte = 8 GPU threads per region
- Analyze up-to 8 regions in parallel at a time, starting from the top of memory (source) and bottom (destination)
- If the region is non-empty AND is at-most 75% full
  - Add to the set of compaction-region set

Defragmentation: Compaction

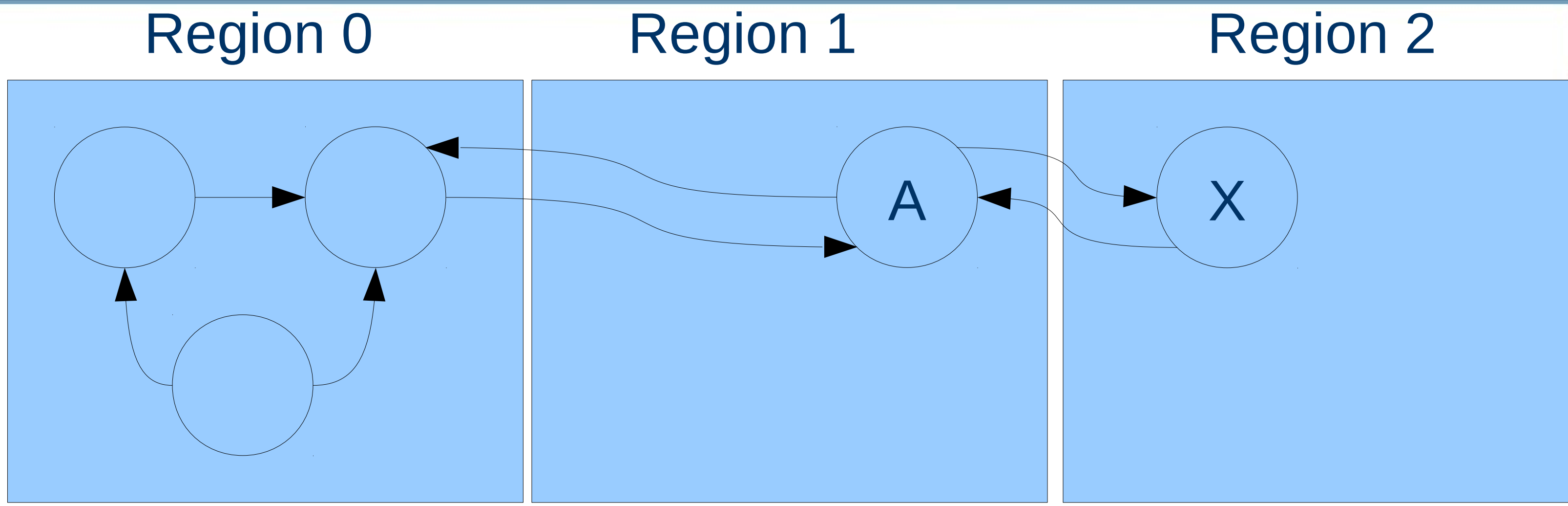


Defragmentation: Compaction: Pointer-rewriting

- The mark-and-sweep collector already traversed memory once
  - Extend it to keep track of which regions of memory contain pointers to which other region of memory
- Cheap:
  - Overhead only during the GC's mark phase

```
bool  have_pointers[NUM_REGIONS][NUM_REGIONS]

void mark_record_seen_pointer(void *object, void *field) {
      unsigned from = object / REGION_SIZE; // a shift...
      unsigned to   =  field / REGION_SIZE;
      have_pointers[from][to] = true;
}
```

Defragmentation: Compaction

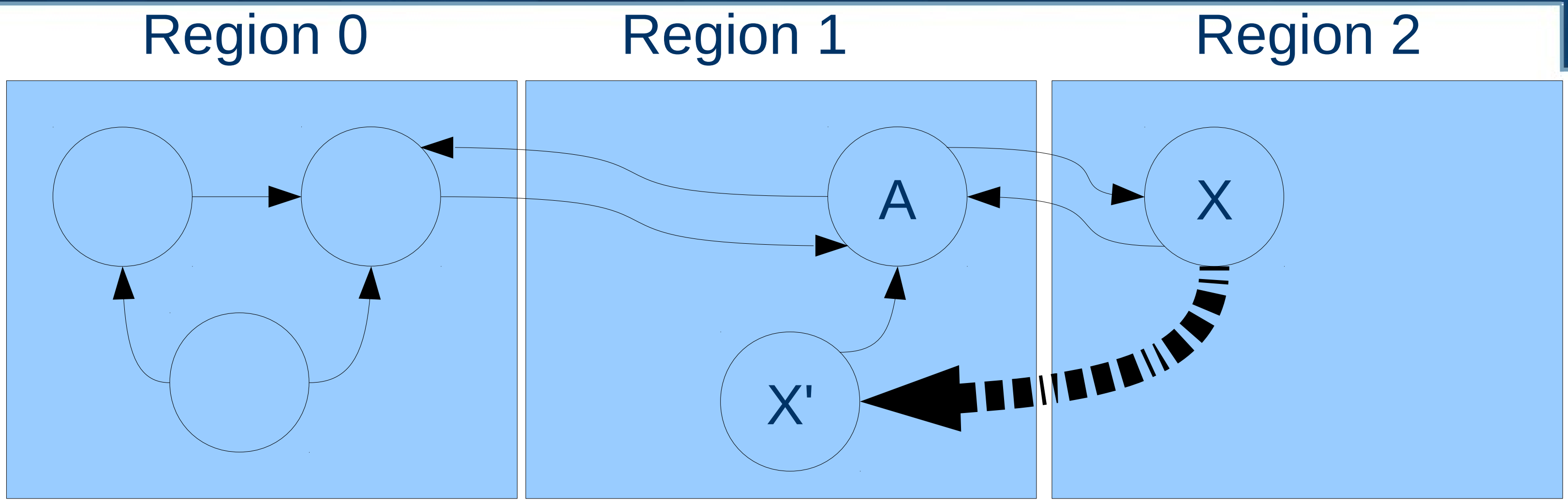


Region 0: contains pointers to { region 0, region 1}
Region 1: contains pointers to { region 0, region 2}
Region 2: contains pointers to { region 1}

Pointer-to Table:

1	1	0
1	0	1
0	1	0

Defragmentation: Compaction

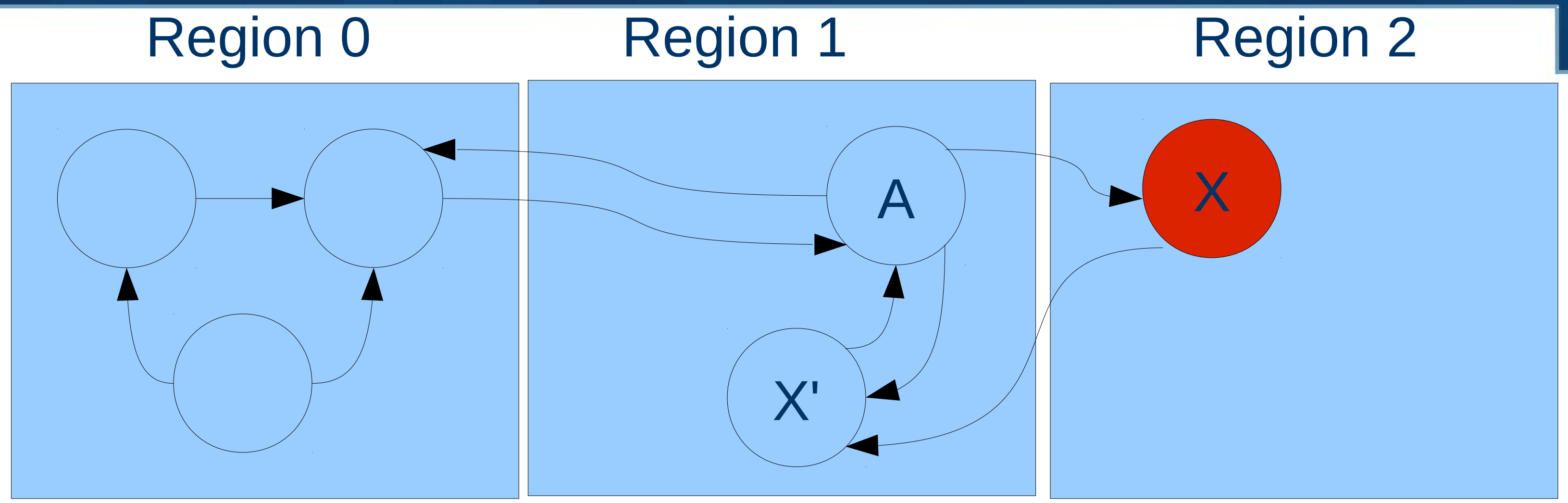


- Assume region 2 is compacted (source) to region 1 (dest)
  - X is moved to clone X'
  - Now need to retarget all pointers in regions that can point to region 2
    - According to table that is only region 1
  - Region 2 has no pointers to region 2, therefore object X' itself does not need to be rewritten

Pointer-to Table:

1	1	0
1	0	1
0	1	0

Defragmentation: Compaction



- region 2 is now empty -> added to free-chunk list(s)
  - In X a forwarding pointer is set to Y
  - X is freed

Pointer-to Table:

1	1	0
1	0	1
0	1	0

Defragmentation: compaction

- We use a gpu-thread per potential object for:
  - Processing all objects in a region to determine if a region is a compaction (src/dst) candidate
  - All objects in a region copied in parallel
  - All objects in a region rewritten in parallel
- Problems:
  - There are 8 small-object-free-chunks per compaction region
    - N kernels in a src-region fight for space from the 8 linked lists for the dst-region
  - Only compacting small (up to 64 byte) objects
    - Arbitrary decision, could also do larger (128 byte, 1024 byte?)

Defragmentation: Coalescing

- After compaction the free lists contain many free holes, need to combine them to larger holes
  - Holes inside the small-object-chunks's linked lists are unsorted, not necessarily adjacent, not all of the same size
    - Reduction/pointer doubling algorithms will not work to merge adjacent holes
- Performance
  - Could use atomic instructions to remove adjacent holes, BUT:
    - Performance will be bad + race conditions galore:
    - Therefore
      - Single individual lists are coalesced sequentially
      - Each list coalesced in parallel

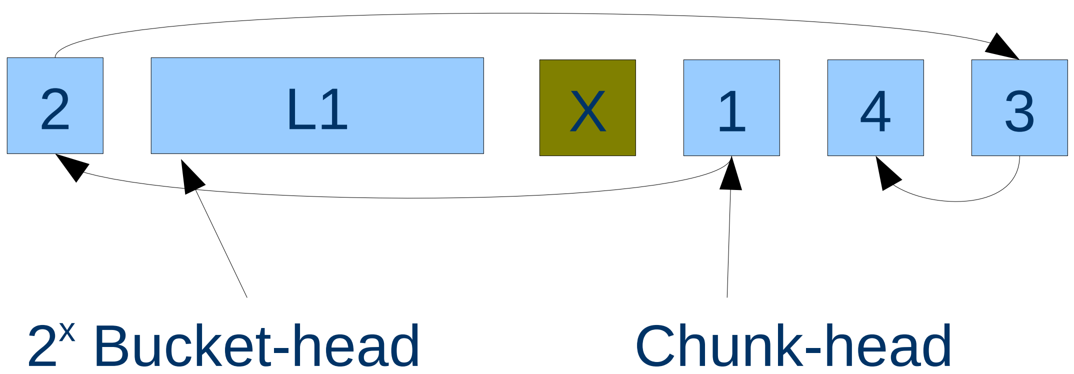
Defragmentation: Coalescing

- Example single list coalescing (many lists in parallel):
  - Step 1: (sequentially) mark which hole can be absorbed into a previous one
  - Step 2: (sequentially) put a pointer to all holes NOT absorbable into an array
  - Step 3: (parallel) coalesce all holes in the array
  - Step 4: (sequentially) rebuild the free-lists

Allocated =

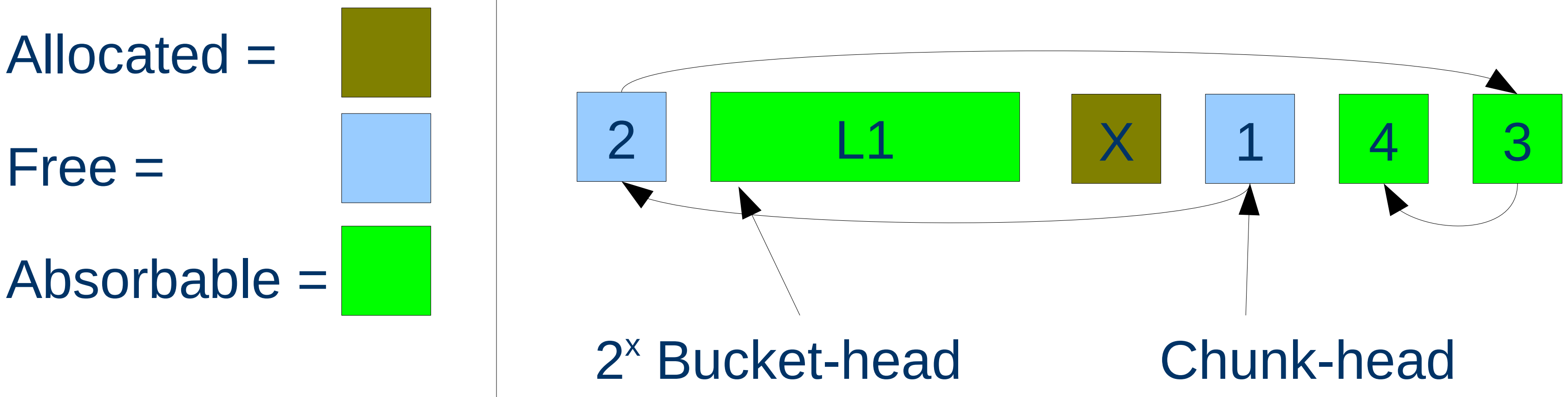
Free =

Absorbable =



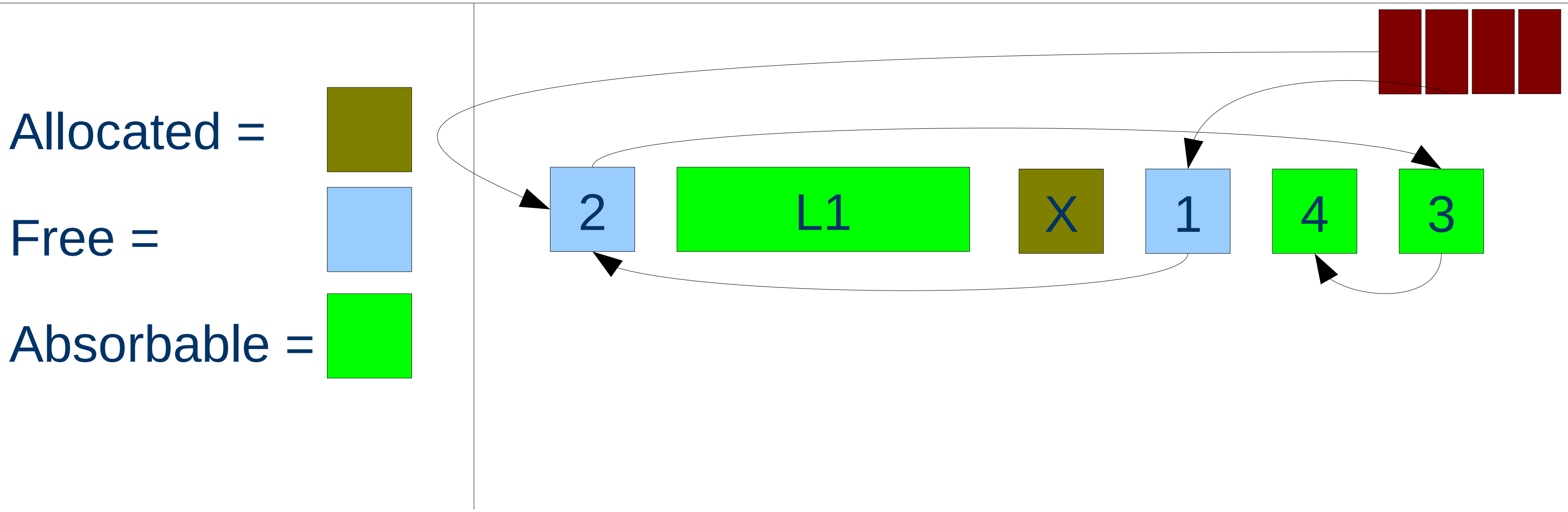
Defragmentation: Coalescing

- Example single list coalescing (many lists in parallel):
  - Step 1: (sequentially) mark which hole can be absorbed into a previous one
  - Step 2: (sequentially) put a pointer to all holes NOT absorbable into an array
  - Step 3: (parallel) coalesce all holes in the array
  - Step 4: (sequentially) rebuild the free-lists



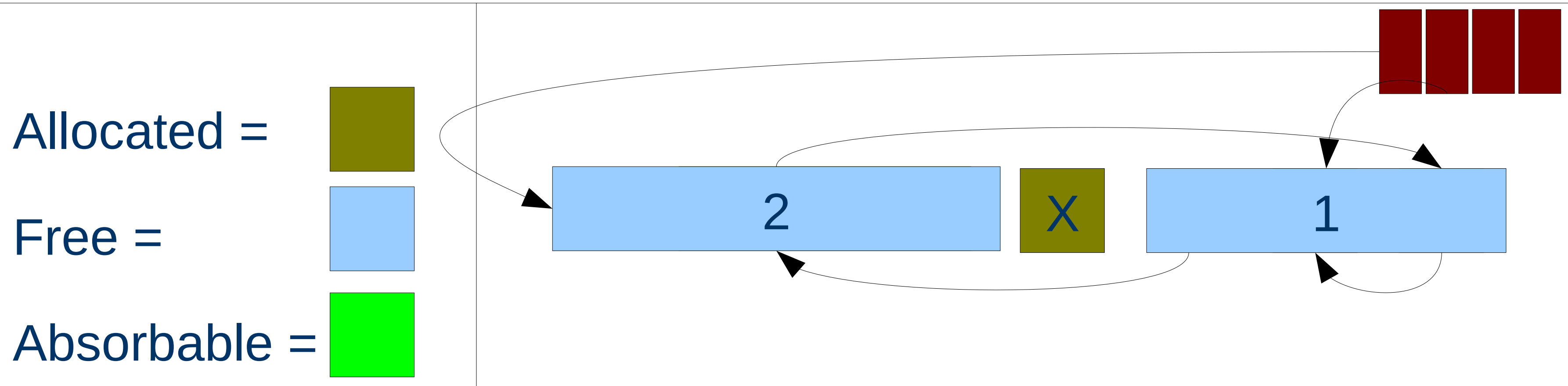
Defragmentation: Coalescing

- Example single list coalescing (many lists in parallel):
  - Step 1: (sequentially) mark which hole can be absorbed into a previous one
  - Step 2: (sequentially) put a pointer to all holes NOT absorbable into an array
  - Step 3: (parallel) coalesce all holes in the array
  - Step 4: (sequentially) rebuild the free-lists



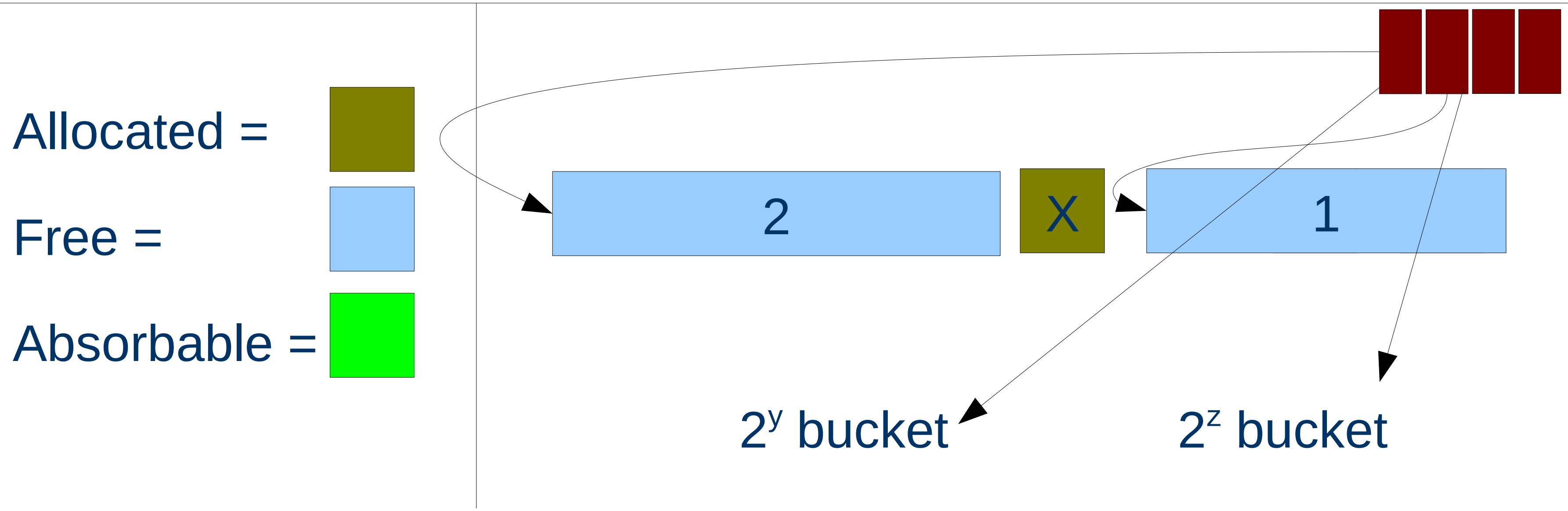
Defragmentation: Coalescing

- Example single list coalescing (many lists in parallel):
  - Step 1: (sequentially) mark which hole can be absorbed into a previous one
  - Step 2: (sequentially) put a pointer to all holes NOT absorbable into an array
  - Step 3: (parallel) coalesce all holes in the array
  - Step 4: (sequentially) rebuild the free-lists



Defragmentation: Coalescing

- Example single list coalescing (many lists in parallel):
  - Step 1: (sequentially) mark which hole can be absorbed into a previous one
  - Step 2: (sequentially) put a pointer to all holes NOT absorbable into an array
  - Step 3: (parallel) coalesce all holes in the array
  - Step 4: (sequentially) rebuild the free-lists



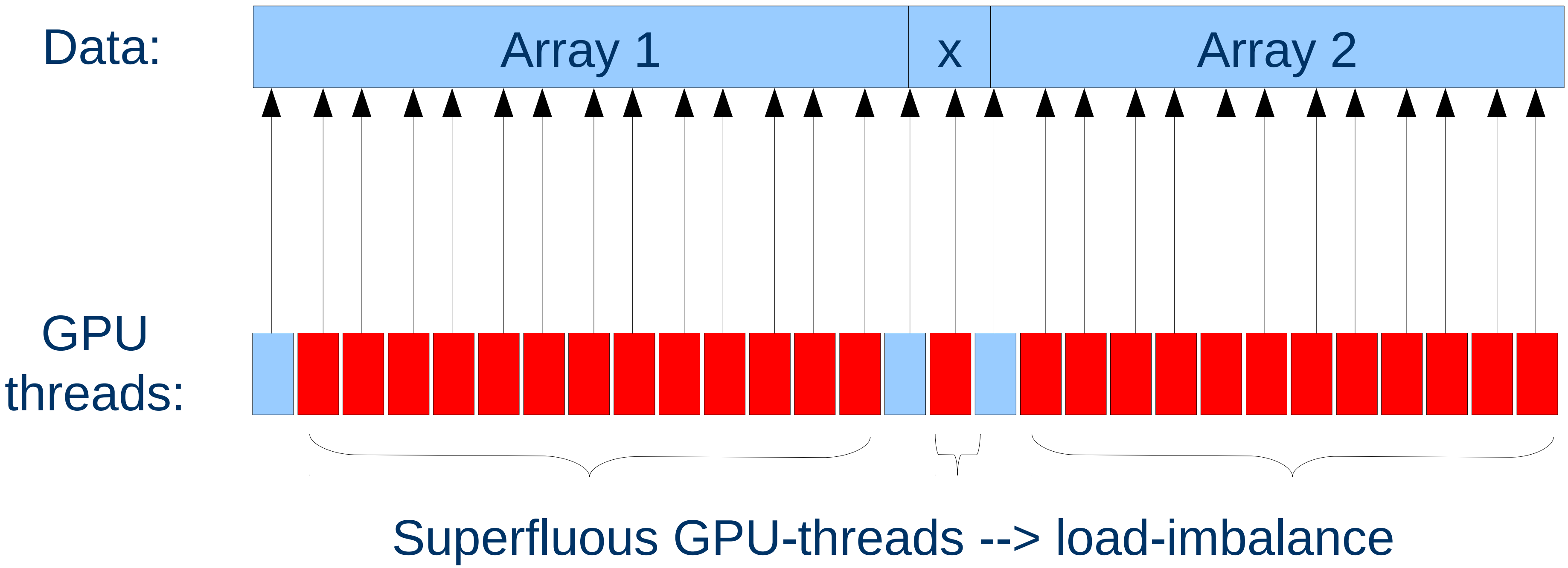
Defragmentation: fast-path

- The defragmentator at startup progressively searches for 512K regions of memory with few objects
  - If a region of memory is found with no allocated objects AND the 8 small-free-chunks it comprises are maximally full
    - Reset the associated small-free-chunks
    - Add the whole 512K region as a single large free memory block to a 2^{19} memory bucket list

- Saves 3 sequential traversals of a maximal length free-list

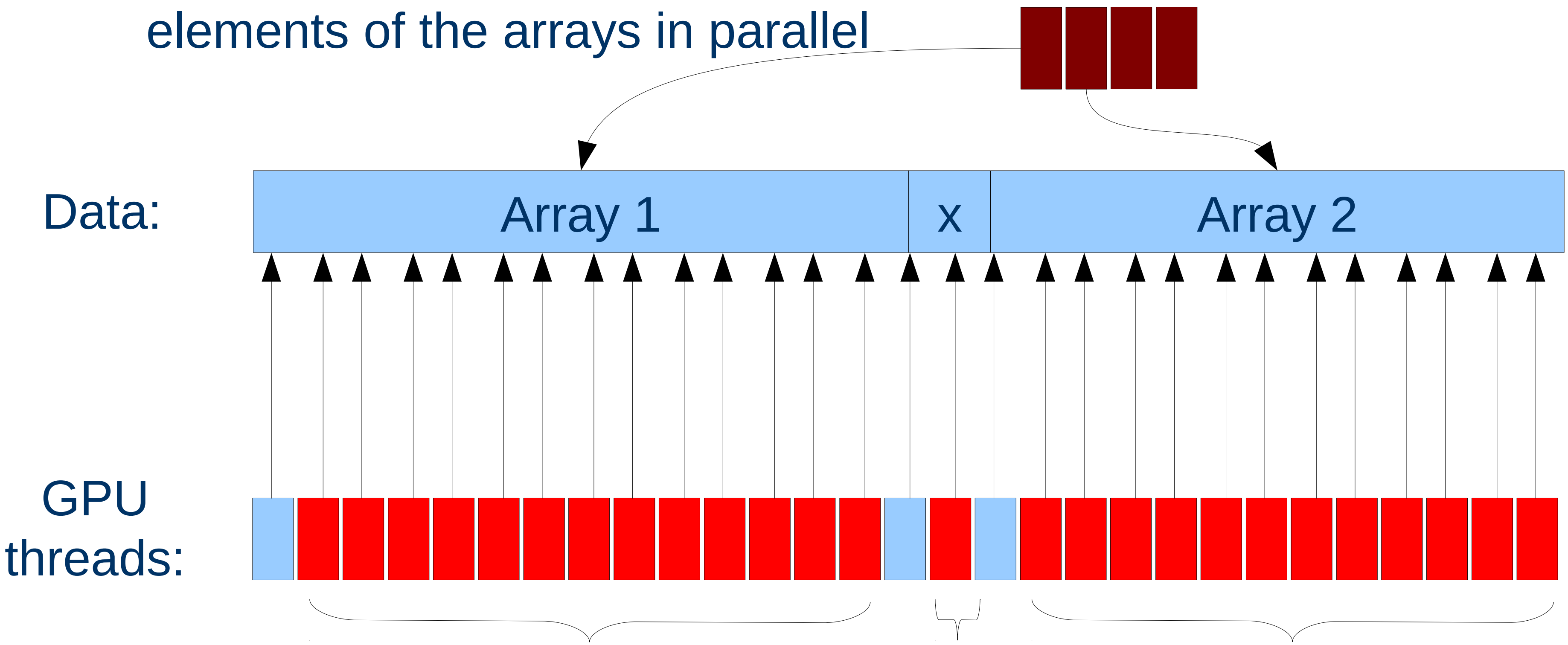
Defragmentation: array rewriting

- When rewriting references, we start a kernel per potential object:



Defragmentation: array rewriting

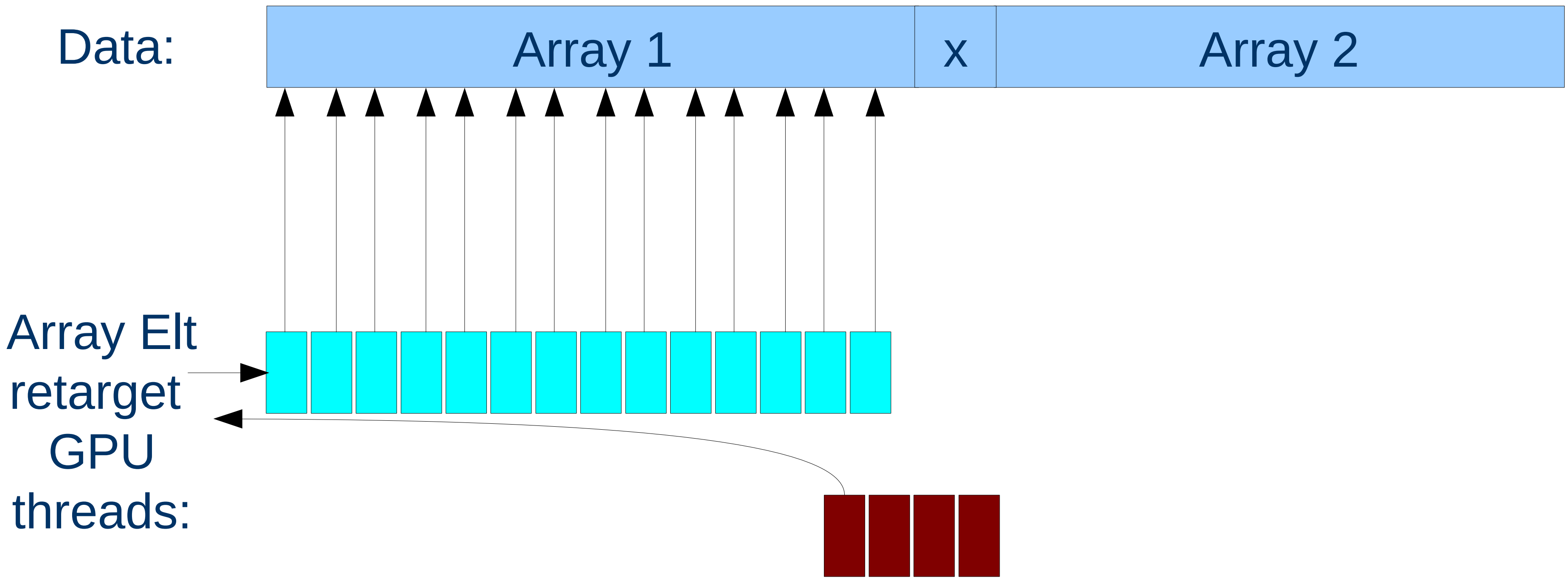
- When rewriting references, we start a kernel per potential object:
  - When detecting a large array: push pointer and later retarget all elements of the arrays in parallel



Still superfluous but take just as long as the start-array gpu-thread

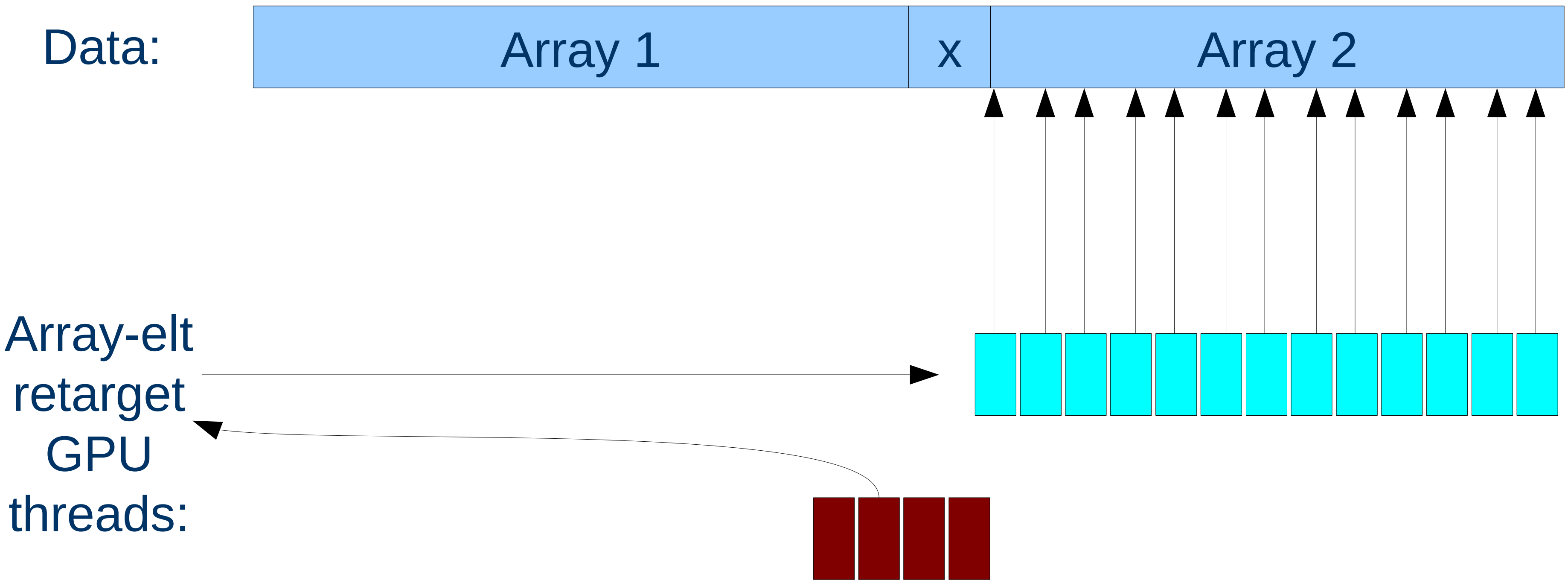
Defragmentation: array rewriting

- When rewriting references, we start a kernel per potential object:
  - When detecting a large array: push pointer and later retarget all elements of the arrays in parallel



Defragmentation: array rewriting

- When rewriting references, we start a kernel per potential object:
  - When detecting a large array: push pointer and later retarget all elements of the arrays in parallel

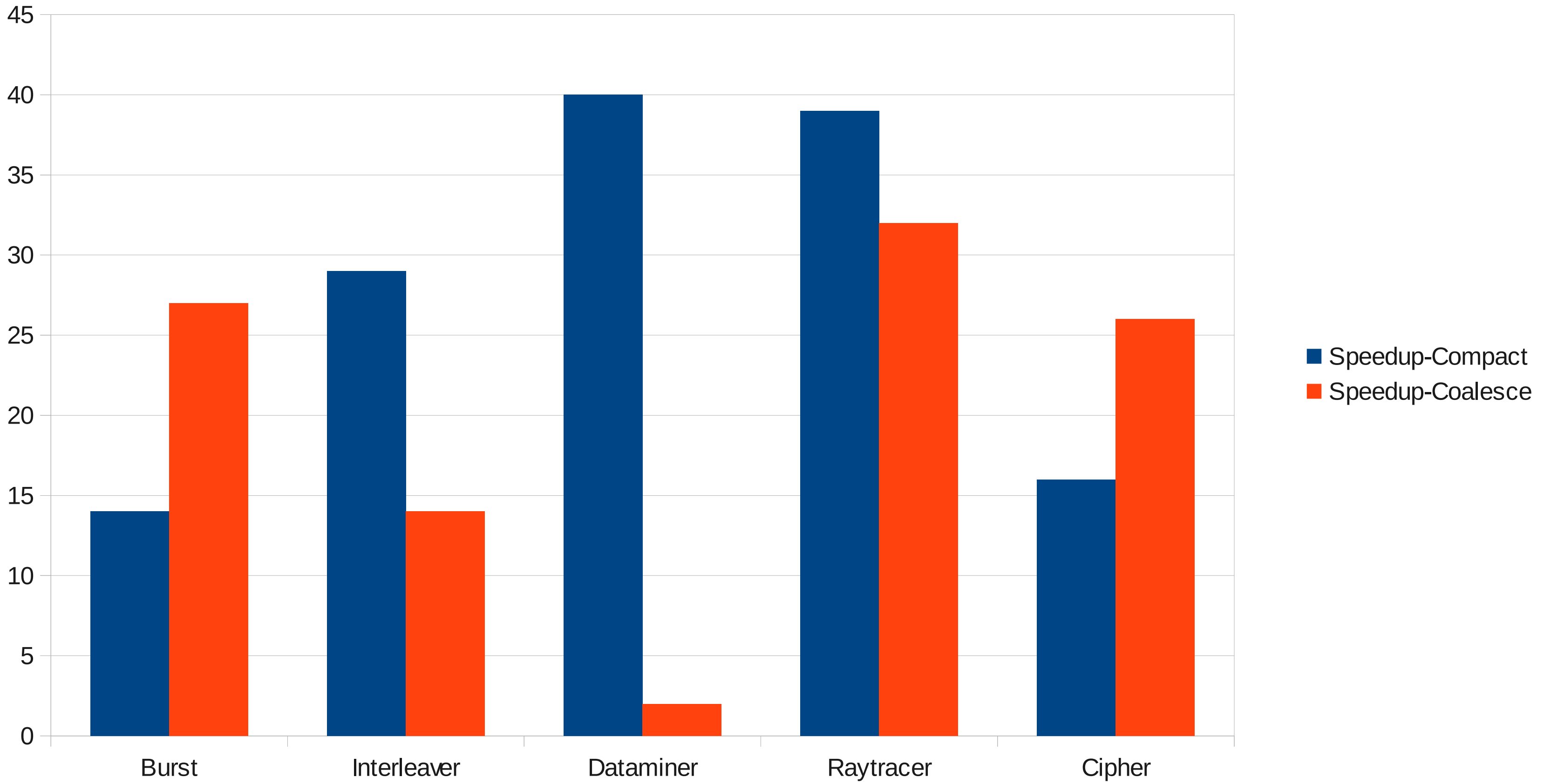


Measurements

- Cuda 4.0 with a GeForce 560 Ti GPU (384 CUDA cores)
  - Array optimization helps a little bit (+/- 10%)
    - Depends on if the large arrays are of int/float or of pointers
  - >90% of all regions fall into the compaction fast path
  - 2% - 4 % of the heap needs its references to be rewritten

Measurements

- Cuda 4.0 with a GeForce 560 Ti GPU (384 CUDA cores)
  - Speedup over using 1 GPU thread:



Measurements

- Compaction speedup is good when
  - A src-region is moderately full (lots of work to do in parallel)
- Coalescing speedup is good when
  - seeing patterns:
    - <allocated><free><free>, <allocated><free><free>, etc
    - Can do most work in a region in parallel
    - Happens when <allocated> is a smallish array
      - Compaction didn't work well
- Coalescing speedup is bad when
  - Few regions are selected for defragmentation
    - Few linked lists to be processed in parallel starving the GPU
  - A region is mostly empty and of different sizes
    - Long linked lists that are sequentially processed
    - Different sizes cause SPMD divergence

Conclusions

- Parallel defragmentation works OK on a GPU
  - If you avoid splitting large holes to smaller ones, much defragmentation can be avoided
    - Will cost you memory...
- Careful selection of what to do in parallel saves costs of atomic operations