
Gallatin: A General-Purpose
GPU Memory Manager

Introduction

Dynamic memory management is crucial for modern data pipelines that use GPU.General purpose memory management implementation faces the problems of parallelism and weak memory coherence.The state of the art memory managers employ classic data structures for memory management failing to support modern applications.

Nowadays the use of GPUs is increased with uses in:

- Databases
- Data analytics
- Genomics
- Sparse linear algebra
- Graph analytics

The lack of dynamic memory management limits the performance of the applications as CUDA allocator faces great latency.Dynamic resizes are not supported and static allocations lead to over approximation of space in the limited space of GPUs.

The paper present Gallatin a memory manager that implements van Emde Boas data structure supporting allocations of any size,constant time insertion and deletion and succession for constant memory size.Therefore achieving greater performance than state of the art. By minimizing fragmentation and allowing large allocations.

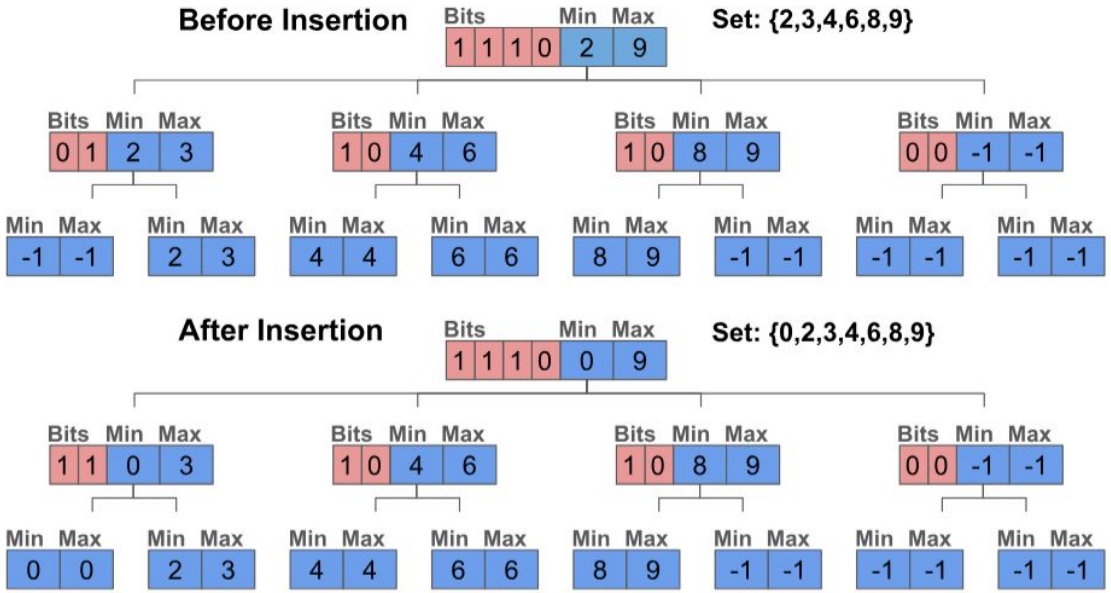
Van Emde Boas Tree

Van Emde Boas tree assumes a universe $U = 0,1,2,...,u-1$ storing a subset $S \subseteq U$,with $|U| = u$ and $|S| = n$,the following operations can be executed on the tree:

- insertion(x): Add $x \in U$ to S, i.e. $S \leftarrow S \cup x$.
- delete(x): Remove x from S, i.e. $S \leftarrow S - x$.
- query(x): Return whether $x \in S$.
- succ(x): Return the minimum $y \in S$, such that $y \geq x$.
- pred(x): Return the maximum $y \in S$, such that $y \leq x$.

For each node of the tree, we store the maximum and minimum values among its child nodes, as well as a bit array indicating the presence or absence of each child.

Insertion and deletion



Note that every node stores $\sqrt{U}$ items in the array.

The following modifications made on the vEB tree structure

- Each node has limited size in 64 bits
- The min and max are removed

  The above allow atomic operations and consistency therefore gaining concurrency.

Memory Partition

1. Segment
2. Block
3. Slice

From 1 to 3 the size and allocation time decreases but reusability is reduced.

Segments

1. Allocated with cuda malloc,segments take 16MB of space.
2. We store a vEB tree called segment tree storing information whether or not the segment is used by the system.
3. We use successor search in order to determine if all segments <k are allocated when the k-th element is needed to be allocated,therefore the fragmentation is reduced and the space is reserved for larger allocations.When the allocation requests size greater than the segment size then an allocation is taking place at of the continuous region using first fit strategy.

The algorithm

Algorithm 1 Allocate segment

```
procedure GetSegment(treeId)                                    ▷
Gather new segment(s) from segmentTree
    if treeId >= numBlockTrees then
        segment = segmentTree.claimMultiple(treeId - num-
BlockTrees)
        ▷ Allocate multiple contiguous segments from back of
the tree
        return segment
    else
        while true do
            segment = segmentTree.successor(0)
            if segment == -1 then                  ▷ All segments in use
                return false
            end if
            if !segmentTree.claimIndex(segment) then
                Continue ▷ Another thread claimed segment, retry
            end if
            memoryTable.initSegment(segment, treeId)
                                          ▷ Segment claimed, initialize
            blockTrees[treeId].insert(segment)
                                ▷ Broadcast availability to all threads
            return true
        end while
    end if
end procedure
```

Blocks

1. Blocks are smaller than segments with size ranging from 4 KB to 16 MB increasing in power of 2.
2. Blocks are stored in vEB trees one for each size of block.
3. When we need to format a segment to a block we remove it from segment tree and add it to the block tree.

The algorithm

Algorithm 2 Allocate block

```
procedure GetBlock(treeId)
    while true do
        segment = blockTrees[treeId].findSegment()
        if segment == -1 then
            if !getSegment(treeId) then
                return nullptr                    ▷ No allocations left
            end if
            continue
        end if
        blockId = memoryTable[segment].getBlock()
        if ldcv(memoryTable[segment].treeId) != treeId then
                                    ▷ Assert tree read was not stale
            memoryTable[segment].releaseBlock(blockId)
            continue
        end if
        myBlock = memoryTable.getBlock(blockId)
        myBlock.init(treeId)
        return myBlock
    end while
end procedure
```

Slices

1. Range size from 16 B to 4096 B
2. We reuse memory only in the block level in order to reduce the allocations to one atomic operation.
3. Slices are allocated from blocks were every block contains 4096 slices.
4. When the same size is requested we coalesce the allocations.

The algorithm

Algorithm 3 Slice allocation

```
procedure MallocSlice(size)
    treeId = log(size) - log(minSize)
    while true do
        wholeWarpTeam = cg::coalesced_threads()
        mallocTeam = cg::ballot(wholeWarpTeam, size)
        if thread == mallocTeam.leader() then
            myBlock = blockBuffer[treeId].getBlock()
            blockId = memoryTable.getBlockId(myBlock)
        end if
        blockId = mallocTeam.broadcast(leader, blockId)
        myBlock = mallocTeam.broadcast(leader, myBlock)
        allocCount = mallocTeam.exclusiveScan(1)
        myAllocation = myBlock.malloc(mallocTeam, allocCount)
        bool valid = (myAllocation < 4096)
        bool replace = mallocTeam.ballot(!valid)
        if replace and (thread == mallocTeam.leader()) then
            newBlock = getBlock(treeId)
      blockBuffer[treeId].replaceBlock(newBlock)
        end if
        if valid then
            return offset(blockId*4096 + myAllocation)
        end if
        continue                                  ▷ Alloc failed, loop
    end while
end procedure
```

Freeing memory

1. Identify the segment we want to free,by treating the pointer as offset.
2. After that we need to identify the size of free.

The algorithm

Algorithm 4 Free

```
procedure FREE(allocation)
    segment = (allocation − memoryStart) / segmentSize
    treeId = ldcv(&memoryTable.ids[segment])
    if treeId < numBlockTrees then
                ▷ return slice and block allocations to their block
        myBlock = memoryTable.locateBlock(segment, treeId,
allocation)
        if myBlock− >free(allocation == 4095 then
            memoryTable.free(myBlock)
        end if
    else
        ▷ Segment allocation, return allocation(s) to segment tree
        numSegments = treeId - numBlockTrees
        segmentTree.insert(segment, numSegments)
    end if
end procedure
```

Evaluation

Experiments set up

Hardware: NVIDIA A40 with 48GB of DRAM and 10,752 CUDA cores.

Benchmark modifications:Experiments in the original benchmark used 10000 to 100000 threads in the modification it increased to 1 million.Furthermore the allocators reset between runs.

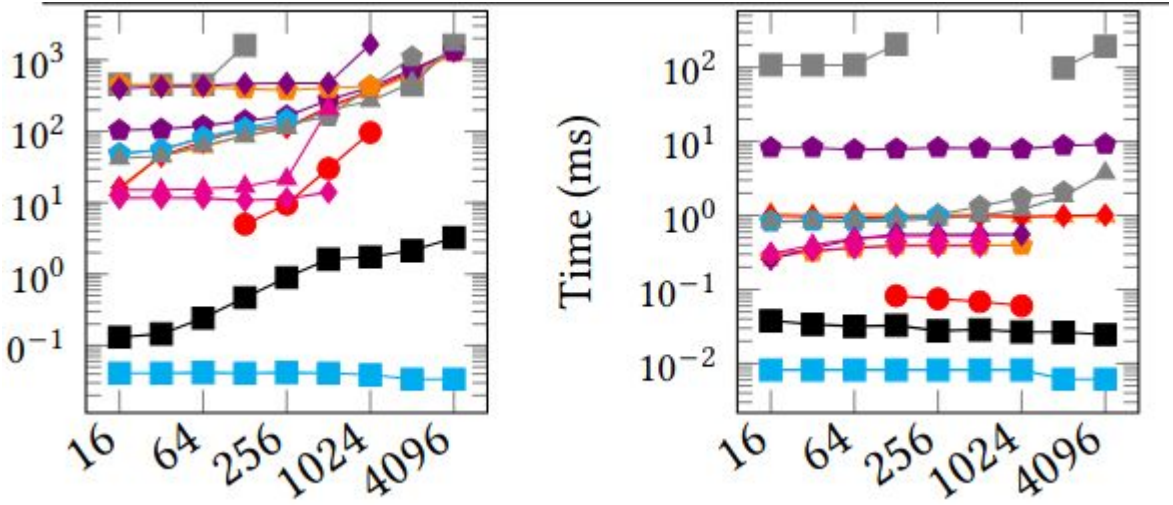
Only CUDA, Ouroboros, RegEff, ScatterAlloc, XMalloc are present in evaluation as they compile in the current cuda version,are publicly available and complete the tests in modified benchmark. RegEff-AW is only present to show the optimal performance as it makes all allocations in one atomic add and all free in one atomic no-op,but it does not manage the memory and therefore multiple addresses are given to multiple threads.

Allocator initialization overhead

All alocators initialize in 32 ms and average is 27ms the best allocator in initialization is ouroboros taking 12ms and galatin takes 31ms.

Single-sized test

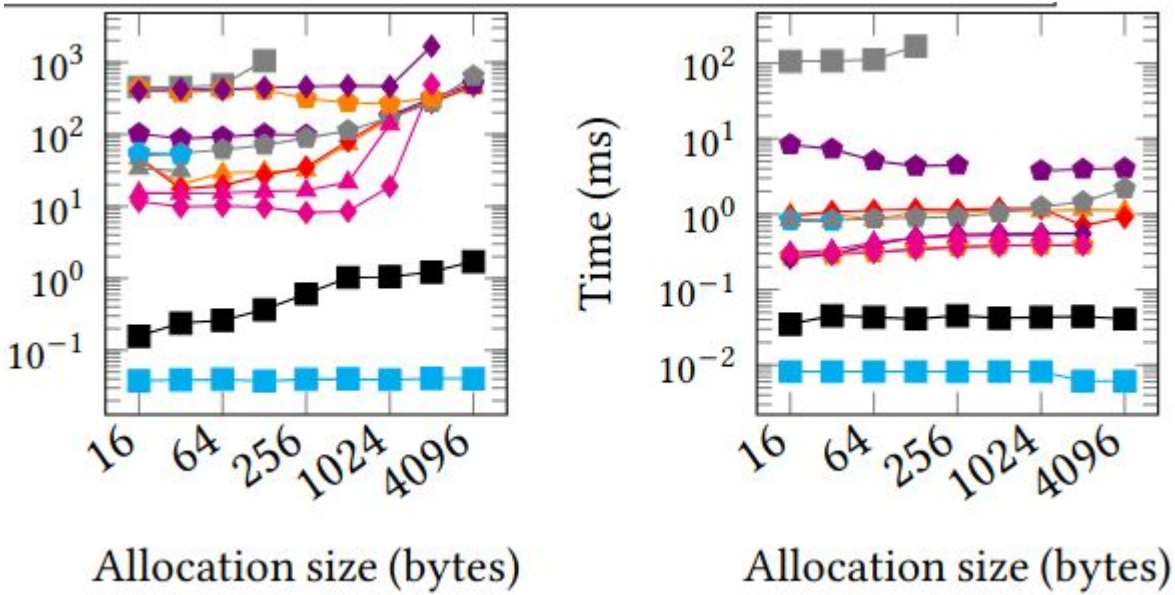
We perform allocations on 1M memory with single size ranging from 16 to 4096 bytes



Gallatin is the fastest allocator and deallocator
for single-sized allocations, and is between 8 − 374× faster for allocations and 2−39× faster for frees.

Mixed-size test

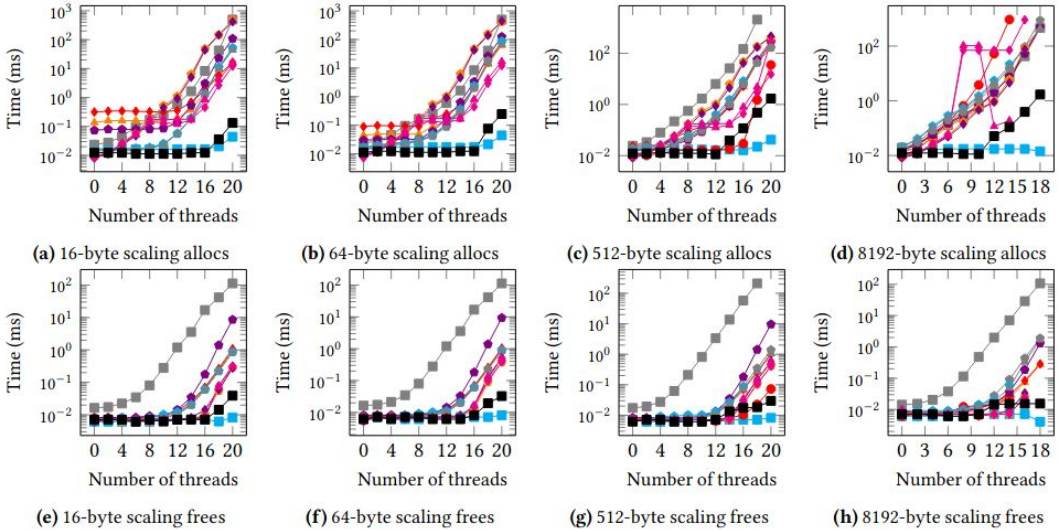
1 M of variant size allocations in ranges of 16 to 4096 bytes.



Gallatin is the fastest allocator for mixed allocations being 8-264× faster and the best deallocator bring 6−22× faster than the next best allocator.

Scaling test

Allocation size static and number of threads increase from 2^0 to 2^20.



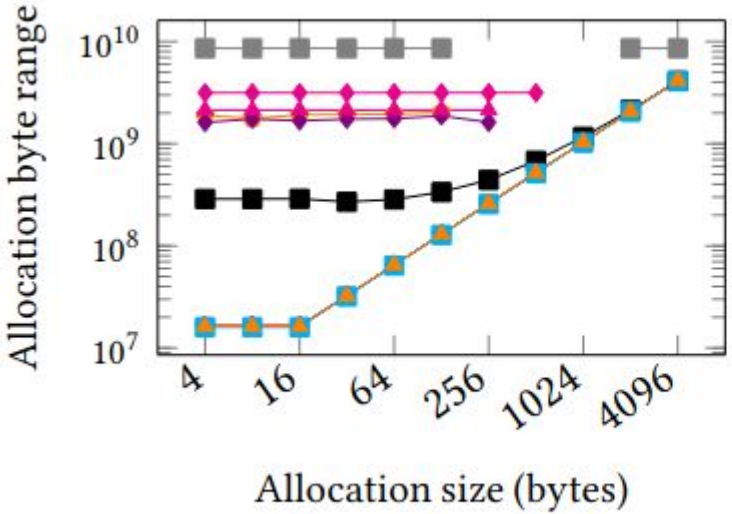
Gallatin has the lowest variance across all sizes for single and mixed-size allocations, with a variance that is 4−87× lower than the next best allocator. Gallatin is the best or close to the best for variance during frees for all experiments, with variance between .57 − 74×lower than the best allocator.

Experiments with warmed-up allocators

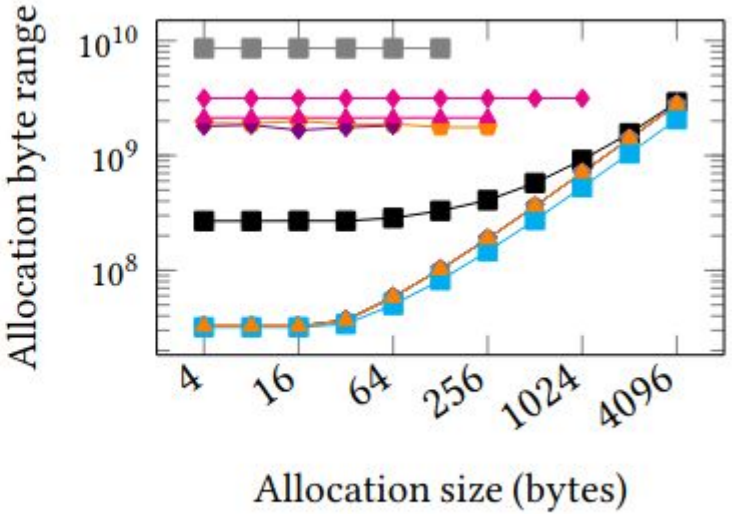
Most allocators, including Gallatin, show no difference in performance when running warmed-up,the only changes were in Gallatin were latency went from 2.13197 ms to 2.15344 ms for 2 KB allocations and Ouroboros-P where median performance went from 15.0069 ms to 0.224256 ms for 16-byte allocations.

Fragmentation tests

Fragmentation test measures the fragmentation of an allocator as the difference between the highest and lowest address given out when performing an allocation.

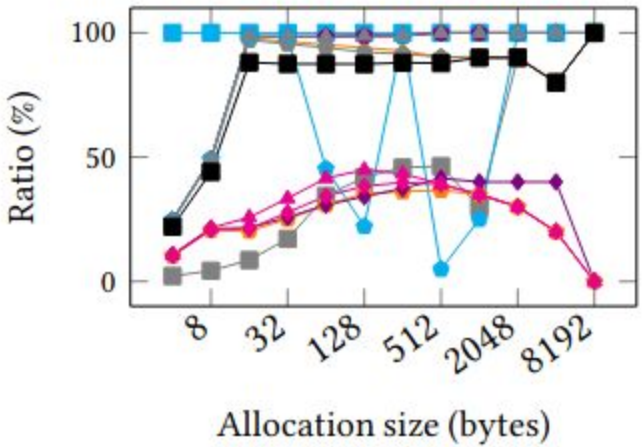


(a) Single-size fragmentation



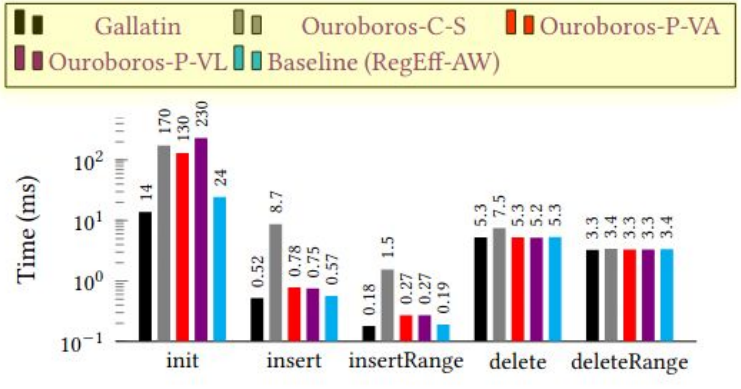
(b) Mixed-size fragmentation

Utilization test

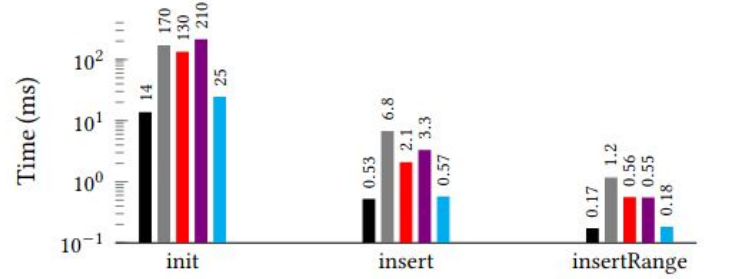


(c) Memory utilization

Graph tests



(a) Performance for insertions and deletions



(b) Performance for expansions only

Figure 7. Performance on the graph tests. The y-axis is the mean runtime over 100 iterations. Lower is better.

Conclusion and future work