Gallatin: A General-Purpose GPU Memory Manager

Hunter McCoy
University of Utah
USA
hunter@cs.utah.edu

Prashant Pandey
University of Utah
USA
pandey@cs.utah.edu

Abstract

Dynamic memory management is critical for efficiently porting modern data processing pipelines to GPUs. However, building a general-purpose dynamic memory manager on GPUs is challenging due to the massive parallelism and weak memory coherence. Existing state-of-the-art GPU memory managers, Ouroboros and Reg-Eff, employ traditional data structures such as arrays and linked lists to manage memory objects. They build specialized pipelines to achieve performance for a fixed set of allocation sizes and fall back to the CUDA allocator for allocating large sizes. In the process, they lose general-purpose usability and fail to support critical applications such as streaming graph processing.

In this paper, we introduce Gallatin, a general-purpose and high-performance GPU memory manager. Gallatin uses the van Emde Boas (vEB) tree data structure to manage memory objects efficiently and supports allocations of any size. Furthermore, we develop a highly-concurrent GPU implementation of the vEB tree which can be broadly used in other GPU applications. It supports constant time insertions, deletions, and successor operations for a given memory size.

In our evaluation, we compare Gallatin with state-of-the-art specialized allocator variants. Gallatin is up to 374× faster on single-sized allocations and up to 264× faster on mixed-size allocations than the next-best allocator. In scalability benchmarks, Gallatin is up to 254× times faster than the next-best allocator as the number of threads increases. For the graph benchmarks, Gallatin is 1.5× faster than the state-of-the-art for bulk insertions, slightly faster for bulk deletions, and is 3× faster than the next-best allocator for all graph expansion tests.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; • **Software and its engineering → Parallel programming languages**; **Concurrent programming structures**.

Keywords: GPU; Memory allocation; Concurrent data structures; High performance computing

PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0435-2/24/03.
https://doi.org/10.1145/3627535.3638499

1 Introduction

GPUs are increasingly used in large-scale data processing applications because they offer a substantial jump in terms of low-cost parallelism as long as data structures and algorithms can be designed and implemented to match the memory and parallelism requirements of GPUs. For example, GPUs are already used in domains such as databases [1, 12, 14], data analytics [3], genomics [7, 9, 13, 17], sparse linear algebra [25], and graph analytics [8, 10, 24, 27].

However, the penetration of GPUs into these applications is limited due to the lack of a generic, dynamic memory manager in GPUs. In most GPU applications, memory is managed by the host CPU, meaning threads in a kernel do not allocate or free memory during execution. Dynamic memory management is provided in CUDA in the form of the CUDA heap allocator, but the high latency of this allocator makes it challenging to build performant data structures. These limitations affect the capabilities of GPU data structures: most existing GPU data structures do not support dynamic resizes and are statically allocated; pointer-based data structures, such as trees and tries, do not achieve high performance on GPUs [28]; data structures on hard-to-align data, such as strings or vectors, or which have variable lengths, such as graphs, are not available.

Overcoming these limitations is critical for building scalable data processing applications. For example, k-mer (length-k strings) analysis is at the core of raw data processing [13, 17] in genomics. In k-mer analysis, the size of the k-mer multiset is not known in advance. Therefore, applications initialize the data structures (such as filters and hash tables) to a default size and then dynamically resize (mostly expand) based on the actual number of k-mers. The ability to resize is critical to perform k-mer analysis in a space-efficient manner. Static data structures such as hash tables and filters [17] currently available on GPUs are sized using an over-approximation of the number of k-mers, which leads to space inefficiency. Over-provisioning of memory makes things worse in GPUs due to the limited GPU memory.

Existing GPU allocators face a trade-off between ***performance*** and ***general usability*** that stems from their choice of data structures. These allocators can broadly be classified into two types depending on the data structures used for managing objects: *array-based* and *list-based*. Array-based solutions such as Ouroboros [23] and Halloc [4] partition GPU memory into an array of statically sized regions. These allow for fast and parallel allocations but limit the maximum allocable size, as all allocations must be smaller than a region. Linked-list

allocators such as RegEff [22] and ScatterAlloc [20] maintain a linked-list heap over all memory. This allows for more varied allocations but can limit performance, as linked lists tend to serialize and perform scattered memory accesses. To rectify this, these systems pre-split the heap, forming a binary heap in the case of RegEff and splitting memory into *super blocks* in the case of ScatterAlloc. These splits increase throughput but again limit the maximum possible allocation size.

Furthermore, state-of-the-art allocators such as Ouroboros [23] and RegEff [22] provide multiple allocator variants (6 and 5, respectively) specialized for different goals. This means that applications must make hard decisions about which variant to use, as the variants can run slowly in the wrong context. In addition to being specialized, these variants are not robust to changing workload profiles at runtime.

The only functioning GPU allocator capable of supporting allocations of any size is the CUDA allocator [2], which is often several orders of magnitude slower than the current state-of-the-art. To handle large allocations, other allocators often fall back to the CUDA allocator [2]. This lowers overall throughput and incurs hidden costs, as the CUDA heap must also be initialized and takes up extra space unrelated to the main allocator.

A high-performance and general-purpose allocator is required to serve modern data processing applications. For example, consider the problem of operations on a graph modeling a social network such as Twitter [6, 19]. This kind of graph grows over time and exhibits heavy skew, as a small percentage of the user base will have many social connections. In the Twitter graph, the average user vertex has less than 35 edges, while the most connected user has over 2.9 million edges. To handle such large-scale and skewed graphs, the allocators must efficiently support both small and large allocations and frees. Existing allocators typically reserve 500 MB in the CUDA heap for large allocations. They fail to process the graph if the total size of these skewed nodes exceeds 500 MB, even if the overall graph could fit in GPU DRAM.

Our contribution. In this paper, we propose a van Emde Boas (vEB) tree-based [21] allocation scheme to bridge the gap between fast, specialized allocators and the slower, general-purpose allocators. Using an ensemble of vEB trees as the underlying data structure design allows for large allocations to be quickly acquired and allows for the use of a fast successor search when looking for new allocations.

The vEB tree is a highly concurrent data structure that allows for faster insertions and deletions than a concurrent linked list. Successor search provides quick access to the first available region of the vEB tree. This enables it to maintain an ordering over the memory regions while being as fast as an array-based data structure. This ordering minimizes fragmentation and allows for large allocations to be performed.

In addition, we propose the use of a tiered, block-based allocation scheme for handling smaller allocations. Using a buffer that maps physical memory to blocks and opportunistic thread coalescing from the CUDA cooperative groups library [18], we can perform many allocations simultaneously and minimize memory pressure on the system.

The combination of these schemes is a high-performance and general-purpose GPU allocator that we call Gallatin. Gallatin can perform allocations of any size and efficiently utilize the entirety of GPU DRAM. Furthermore, unlike existing state-of-the-art allocators, Gallatin provides a single general-purpose variant that is suitable for all workflows and can adapt smoothly to changing workloads.

Our results. We employ the suite of benchmarks from the Winter et al. GPU memory manager survey paper [26] to evaluate Gallatin against existing GPU memory managers. We further update and extend the tests in the benchmark to better depict the scale and complexity of current real-world systems and applications.

We find that Gallatin is up to 374× faster than the next-best allocator on single-sized allocations and up to 264× faster than the next-best allocator on mixed-size allocations. Gallatin scales well as the number of threads increases and in scaling benchmarks is up to 254× faster for single-sized allocations as the number of threads increases.

In addition, Gallatin is competitive with more specialized allocator designs on real-world benchmarks. For the graph benchmark, Gallatin is 1.5× faster than the Ouroboros-P [23] series allocators for bulk insertions, slightly faster for bulk deletions, and is the fastest allocator for all graph expansion tests. These results show that a vEB tree-based allocator can bridge the gap between performance and general usability on GPUs, providing fast allocations of any size.

2 Previous work

In this section, we provide an overview of the existing GPU allocators. All allocators offer a standard malloc/free interface.

XMalloc. In XMalloc [11], allocations occurring in the same warp are combined by being packed into one large allocation with padding and an internal counter. One thread from the group is then sent to perform the allocation for all threads. Two tiers of buffers are used to reduce latency for allocations, and a doubly-linked list backs the allocator. The use of a linked list provides a point of serialization that slows down the allocator.

ScatterAlloc. ScatterAlloc [20] accelerates allocations by "scattering" them inside of memory via the use of a hash function. ScatterAlloc splits the memory pool into a set of fixed-size blocks called *super blocks*, which are chained together into a singly-linked list. Each super block is further subdivided into *pages*. Pages are fixed chunks of memory that can be specialized to different allocation sizes. In this scheme, allocations larger than a super block are not possible.

RegEff. RegEff [22] or *Register Efficient* allocator was introduced after ScatterAlloc with the goal of providing similar performance with less register pressure. The RegEff allocators

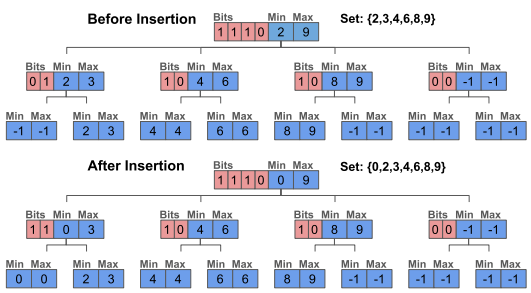


Figure 1. Insertion operation in a vEB tree of universe size 16. Each node has a summary structure (shown in red), min/max (shown in blue), and pointers to the next level. The size of the summary structure and the number of pointers is same. The leaf nodes have summary structure of size 2 and only contain min and max. While inserting 0 in the tree, it replaces the minimum value of nodes until it reaches the leaf. As the leaf is empty, the minimum and maximum are both set to 0.

are built using a lock-free singly-linked list. The list is fragmented into a binary heap to reduce thread contention when allocating. This improves the throughput of the linked list but limits the maximum allocation size possible. The faster variants trade fragmentation for throughput by introducing an array of memory offsets used for indexing into the linked list.

Ouroboros. The Ouroboros [23] allocators are based on a queue structure for allocations, with a semaphore being used to control operations on the queue. GPU memory is split into sections called pages, which are then grouped into larger contiguous sections called chunks of size 8192 bytes. All Ouroboros allocators have a set of queues, one for each power-of-two size supported. When an allocation is requested, the request is forwarded to the smallest queue that can support it. The allocators are split based on whether or not the queue holds chunks (C) or pages (P). The C series has full reuse, while in the P series freed allocations can only be reused for the same allocation size as the original. Each type of queue comes with a variant that is statically sized (S), a virtualized array (VA), or a virtualized list (VL). None of the variants natively support any allocation size larger than the chunk size (8192 bytes).

3 Preliminary

In this section, we will first give an overview of the van Emde Boas (vEB) tree and explain how to perform operations. We then describe how we adapt the traditional recursive design to develop a GPU implementation of the van Emde Boas tree.

The core data structure used in Gallatin is the van Emde Boas (vEB) tree. Originally described by van Emde Boas [21] for use in priority queues, the vEB tree operates on a *universe* $U = 0,1,2,\ldots,u-1$, maintaining a subset $S \subseteq U$ of elements from U, where $|U| = u$ and $|S| = n$. In this universe, the vEB tree performs the following operations in $O(\log \log u)$ time:

- **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 $y \leq x$.

Note that $x \in U$ but x need not be in S.

3.1 vEB tree design

A vEB tree is a recursive tree structure, with each node having three components: a min value, a max value, and a bit array (summary structure). This bit array has one bit for each of the node's children. The bit for each child is set to zero if the child (and corresponding subtree) is empty. Otherwise, the bit is set to one. For a universe U, the root node has $\sqrt{U}$ bits in the bit array, with each subtree below the root responsible for $\sqrt{U}$ items. A node's minimum and maximum values record the minimum and maximum stored in the entire subtree.

Insertions and deletions are done similarly to other tree data structures by recursively stepping into smaller subtrees until the item is found or the current subtree is empty. Insertion and deletion must update the minimum and maximum of each subtree traversed.

Successor search is also performed recursively, using the minimum and maximum values to determine how to recurse. Starting at the root, the maximum value is checked first. If $x > max$, then a larger item does not exist. Next, the minimum value is checked. If $x \leq min$, then the minimum is the successor. Otherwise, traverse the bit array until you find a set bit and recurse to the subtree the bit represents. As each subtree is sized to be the square root of the size of the previous tree, this gives a recurrence relation of $T(U) = T(\sqrt{U}) + O(1)$, which translates to an $O(\text{loglog}U)$ runtime for successor search.

3.2 Implementing vEB trees on GPU

The standard vEB tree design does not directly port efficiently to the GPU due to the large node sizes. As all data must be loaded atomically, having a total size larger than 64 bits means that nodes cannot be loaded/stored atomically. This can allow nodes to be viewed in an inconsistent state and requires a more complicated control flow to guarantee consistency.

To prevent inconsistent states, we fix the size of each node to 64 bits, and we remove the min and max values. Capping the number of bits in the bitarray (summary structure) to 64 bits enables atomic operations on the vEB tree nodes. However, modifying the vEB tree design this way results in losing the $O(\log \log U)$ performance bound on operations but allows for every vEB node operation to be performed in one atomic operation and guarantees that nodes cannot be viewed in an inconsistent state. Removing the min and max means that operations can no longer be shortcut, though for small trees, this is made up for by the reduced number of operations per

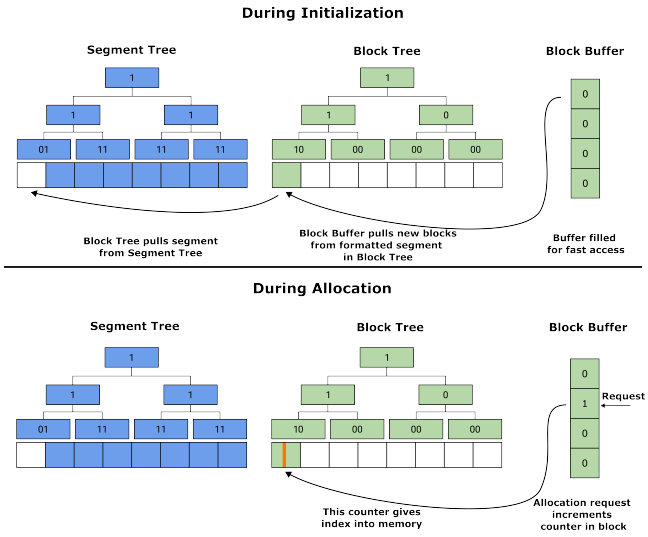


Figure 2. Data structures used in Gallatin during an allocation request. Memory is split into large regions called *segments*. Segments are managed by the segment tree. *Blocks* are segments formatted to a given size. Each block contains 4096 *slice* allocations of a set size. Blocks are managed by block trees. There are multiple block trees one for each pre-configured size. Pre-allocated block Buffers provide fast access to slices. The block buffer is loaded with blocks from segment 0 formatted to the given slice size. An allocation request comes to the block buffer, which sends the request to block 1 in the segment. An `atomicAdd` on the malloc counter of the block marks that this thread reserves the first slice in Block 1, and the address of the slice (labeled in orange) is handed to the requesting thread as an allocation.

node. In practice, losing the $O(\text{loglog}U)$ bound is not an issue because, for most practical memory managers, the size of the universe is known and small, so we can bound the height of the tree to a small constant.

Using atomic operations over the vEB nodes makes the tree highly-concurrent and makes the vEB tree the right candidate to exploit massive parallelism available in GPUs.

4 Design of Gallatin

In this section, we describe the design of Gallatin. We start with the memory partitioning scheme of Gallatin, followed by a discussion of the major data structures used and an overview of the allocation pipelines. Memory in Gallatin is partitioned into three types of allocations known as ***segment***, ***block***, and ***slice*** allocations. Moving from segments to slices, the allocations become smaller, less reusable, and significantly faster to allocate. This section starts with a discussion of GPU memory as a whole and covers the allocation sizes moving from largest to smallest.

Algorithm 1 Allocate segment

```
procedure GetSegment(treeId)                                   ▷
Gather new segment(s) from segmentTree
    if treeId >= numBlockTrees then
        segment = segmentTree.claimMultiple(treeId - numBlockTrees)
        ▷ 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
```

4.1 Segments

GPU memory in Gallatin is partitioned into large, contiguous 16MB regions called ***segments***. All memory segments are allocated using cudaMalloc [2] and form one contiguous array in the GPU memory.

To mark a segment as being free or in use, we use a van Emde Boas (vEB) tree called the ***segment tree***. This tree has one bit per segment. If the segment is free, the bit is set to 1. If any part of the segment is in use anywhere in the system, this bit is set to 0. Since each segment is composed of 16MB of memory, every leaf node in the vEB tree covers 64 segments or 1GB of GPU DRAM. In this setting, A 3-level vEB tree can cover up to 4 terabytes of memory in this design.

Using successor search, segments are allocated from the front of the segment tree in order to minimize the external fragmentation of memory. The use of successor search when finding new segments allows us to maintain the property that *the k^{th} segment can only be allocated if all segments $< k$ are allocated as well.* Minimizing the fragmentation from smaller allocations allows us to reserve space for larger allocations. When an allocation requires more than 16 MB or one segment of memory, it is pulled from the end of this contiguous region in a first-fit strategy using the predecessor search.

The use of a fixed-size van Emde Boas tree provides both an embarrassingly parallel method for finding free segments and maintains a total memory ordering. Previous allocators, such as Ouroboros [23], use a queue to provide obstruction-free

segment reclamation but did not provide an ordering. This ordering is crucial to provide large allocations and makes Gallatin the first GPU allocator to provide both high-throughput and arbitrary-sized allocations.

See Algorithm 1 for more information on segment allocations. Segment allocation either attaches a segment to a block tree or returns a set of contiguous segments as one large allocation. This is controlled by the input parameter treeId: If `treeId < numBlockTrees`, where `numBlockTree` is the number of block trees as described in Section 4.2, a single segment is allocated and attached to block tree with ID `treeId`. Otherwise, `treeID - numBlockTrees` segments are allocated from the end of the segment tree and returned as one allocation.

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
```

4.2 Blocks

Fulfilling smaller allocations using segments directly from the segment tree will result in high internal fragmentation. To avoid that, we format segments into smaller regions known as ***blocks*** to efficiently handle smaller allocation requests.

Blocks range in size from 4 KB to 16 MB, stepping in powers of two, meaning that a segment can be split into 1 to 256 blocks (as segments are 16 MB) depending on the block size needed. To format a segment into blocks, we give control of the block to another vEB tree known as a ***block tree***. There is one block tree for each block size. To mark a segment as being formatted into blocks, we remove it from the segment tree and insert it into the appropriate block tree.

Whenever a block is requested, a search is performed on the block tree to locate a formatted segment to hand out the block. If no segments are available, a new segment is taken from the segment tree, formatted into the appropriate block size, and handed to the block tree.

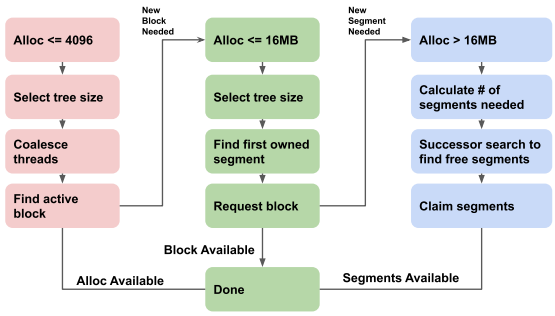


Figure 3. Pipelines in order from smallest to largest. Failure in a pipeline forwards a request to the next pipeline.

Blocks are allocated and returned to the segment using a constant-size per-segment ring queue. When all blocks are allocated from the queue, the segment is removed from the block tree to prevent over-subscription on the segment. The segment is re-added to the block tree when enough blocks are returned. When all blocks are returned, the segment is freed and re-added to the segment tree.

Please refer to algorithm 2 for pseudocode for block allocations. The `treeId` argument will always be strictly smaller than `numBlockTrees`, as blocks must come from a block tree. `ldcv` is the CUDA intrinsic for a global data load. This is used to fetch the `treeId` of the segment from the memory table.

4.3 Slice

The smallest allocations given by Gallatin are ***slices***. These small allocations are crucial to many dynamic data structures such as linked lists, skip lists, queues, trees, and hash tables [5] and range in size from 16 B–4096 B.

In a weak memory system, the minimum amount of work required to perform an allocation is one atomic instruction. To achieve this minimum atomic bound, memory reuse is only allowed at the block level. This makes the control of slice allocations relatively lightweight: one atomic operation on a 32-bit machine word is employed for malloc and free to control the slice allocations.

To minimize internal fragmentation, slices are allocated from blocks. When formatted, each block contains 4096 slices equally sized to be 1/4096-th of the block size. Slices are handed out in order, and reuse of slices is only allowed once all slices have been returned to the block.

Coalescing allocations. Using counters over a more sophisticated approach such as bit array allows Gallatin to serve multiple requests using the same atomic instruction. When multiple threads in the same warp request an allocation of the same size, we can group these requests and satisfy them simultaneously using a single atomic instruction. This grouping occurs opportunistically via the `cooperative_groups :: coalesced_threads` function, which returns a handle to all

threads performing the same request. The leader for these threads performs the allocation and distributes the allocations to the other threads in the team. If the block is exhausted, the leader is in charge of fetching a new block.

See Algorithm 3 for the pseudocode of slice allocation. In the code, `size` is the number of bytes requested for the allocation, and `offset` is a function that converts integers to pointers to live memory.

Faster access to blocks. To reduce the latency for the slice allocation operation, we store formatted blocks in a buffer. This buffer, called the ***block buffer***, maps streaming multiprocessors (the hardware in a GPU that executes thread blocks) to live block allocations. For the smallest slice size, this buffer has one block pointer for every streaming multiprocessor, with each larger allocation size having half as many slots as the next. In an A40 GPU, for example, the block buffer for 16-byte allocations holds pointers to 128 blocks (one per streaming multiprocessor). The block buffer for 32-byte allocations holds pointers to 64 blocks, the 64-byte block buffer holds pointers to 32 blocks, and so on. The minimum number of blocks held at a level is capped at four to reduce contention on large block sizes. When a block is exhausted, the thread that took the last allocation replaces the block in the buffer. Using block buffers improves the performance of slice allocations but slightly increases the memory overhead of the allocator, as the block buffer is always populated. Please refer to Figure 2 for more details on using the block buffer.

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
```

An overview of the allocation pipeline across the three memory constructs, segments, blocks, and slices, in Gallatin is described in Figure 3.

5 Freeing memory

We now describe how we perform the free operation in Gallatin. To free an allocation, we first identify the segment the allocation came from and the block size of the segment. The segment can be found by treating the pointer as an offset into the memory allocated by Gallatin. Dividing this offset by the segment size returns the `segmentId` for the allocation.

Once the `segmentId` has been acquired, a global read is used to identify the block size of the segment. For segments that have been partitioned into blocks, this value is in the range `(0, numBlockTrees)` and corresponds to the block tree the segment belongs to. For allocations greater than or equal to one segment in size, the value is equal to numBlockTrees + numSegments.

For smaller allocations, knowing the allocation offset and block size is sufficient to locate the block from which the allocation came. Once the block has been identified, the free counter is incremented to signal that the allocation has been returned. If all slices in the block are free, the slice is re-added to the segment's queue. If this was the last block in use in the segment, the entire segment is returned to the segment tree.

Larger allocations are returned to the system by reinserting them into the segment tree. See algorithm 4 for pseudocode for freeing allocations. `locateBlock` finds a block in constant time using the segment and `treeId` of the allocation by casting the allocation to a relative offset into the segment. Dividing this value by 4096 returns the relative `blockId` in the segment.

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
```

5.1 Memory table

Since the maximum possible number of blocks in a segment is known during construction, the metadata for each segment

is initialized during construction of Gallatin, with the block metadata being enabled or disabled depending on the current size of the segment. For example, in the default configuration of Gallatin, there is a maximum of 256 blocks per segment, so every segment comes with 256 block counters. When a segment is initialized to a larger slice size, such as 32 bytes, only 128 block counters are needed. In this example, only the first 128 block counters will be initialized, while the latter 128 block counters in the metadata will be unused. The queues used for performing malloc and free of blocks can also be initialized beforehand, as can the counters used inside blocks for malloc and free of slices. These items are grouped together in a table for efficient memory access and are known as the *memory table*.

6 Evaluation

In this section, we evaluate the performance of Gallatin, our GPU memory manager. For this evaluation, we use the benchmark from the Winter et al. [26] survey paper. This benchmark evaluates the following performance characteristics:

- Performance when allocating/freeing a single size. This performs all allocations of the same size, returns all allocated memory, and checks the correctness of allocated memory.
- Performance when allocating/freeing different sizes. This performs allocations in a range of sizes, returns all allocated memory, and checks the correctness of allocated memory.
- Performance scaling when increasing the number of active threads. Every iteration doubles the number of threads, and all allocations performed are the same size.
- Fragmentation as a measure of the gap between the highest and lowest addresses when performing a series of allocations and frees.
- Utilization of the allocator as a measure of the fraction of memory given to the allocator that can be used for allocations. Performs allocations in batches of 100,000 until the allocator returns *nullptr* or encounters an error. This benchmark was referred to as the "Out of Memory" benchmark in Winter et al. [26]. We have renamed it here for clarity.
- Performance of the allocator when running a dynamic graph simulation. This tests the performance when building and modifying edge-list graphs.

System specification. All experiments were run on an NVIDIA A40 with 48GB of DRAM and 10,752 CUDA cores. The code for Gallatin [15] and the benchmark [16] are publicly available.

6.1 Benchmark modifications

The benchmark from Winter et al. [26] covers all the major characteristics an allocator can exhibit. However, it does not fully depict the scale and complexity of major real-world scenarios. In practice, most GPUs are used for compute-intensive workflows where millions of threads can take advantage of the massive parallelism available on these devices. The original benchmark uses only a small number of threads, with most tests running with 10K threads and the largest running with 100K threads. To more accurately model the use cases of these allocators, we increased the number of threads used in the benchmark to one million.

Additionally, we modified the benchmark to reset allocators between runs for the allocation and scaling allocation benchmarks. The performance in these benchmarks is an aggregate of 50 different runs. The allocator is initialized once in the original benchmark, and all 50 tests are run back-to-back. This gives some allocators an unfair advantage as they cannot fully reset between runs, meaning that the memory is already reserved at the start of the next iteration. This results in a high variance in the overall performance, as the first run is significantly slower than subsequent iterations. Re-initializing the allocators before every iteration makes the performance representative of the performance when the allocators are run for the first time.

6.2 Allocators involved in the benchmark

The following is the list of allocators included and excluded in our benchmarks compared to the evaluation Winter et al. [26].

- **Allocators included**: CUDA, Ouroboros, RegEff, ScatterAlloc, XMalloc.
- **Allocators excluded**: FDGMalloc, Halloc, DynaSOAR, BulkAllocator. These allocators do not compile for the current version of CUDA, are not publicly available, or do not complete any tests in the extended benchmark.

RegEff-AW is included in the figures only as a baseline to show optimal allocator performance. This allocator performs all allocations in one atomicAdd and all frees in one no-op. However, this allocator does not manage memory and can give out the same address to multiple threads, making it unsuitable for real applications. The performance of RegEff-AW is not considered for comparison in any of the benchmarks.

6.3 Results summary

On the scaled tests, Gallatin is the fastest for single and mixed allocations and frees of any size, being up to 374× faster than the next best allocator for single-sized allocations, up to 39× faster for frees, up to 264× faster for mixed-size allocations, and up to 22× faster for mixed-size frees. In the scaling benchmark, Gallatin is the fastest allocator for nearly all allocation sizes. Gallatin is up to 254× faster than the next best allocator and is at worst 75% slower than ScatterAlloc when running with 65,336 threads and 512-byte allocations, and is only slower than RegEff-CFM when the number of threads is between 16,384 and 65,336.

Gallatin has the third-best fragmentation performance, only being beaten by the Ouroboros series allocators. Gallatin also has the third-best memory utilization and has the highest memory utilization when accounting for the extra space reserved in the CUDA heap by Ouroboros.

For the graph experiments, Gallatin is the fastest allocator for initialization, insertion, and bulk operations, and is

competitive with the Ouroboros-P allocators for deletion. In addition, Gallatin is 3x faster than the next-best allocator for all operations in the expansion graph tests.

Ouroboros and RegEff have multiple variants that specialize for either memory overhead or utilization. This leads to a variant of these allocators being performant in every benchmark, though the variant that performs well changes between benchmarks. In contrast, Gallatin has one generic version that is competitive for all benchmarks.

6.4 Allocator initialization overhead

All allocators initialize within 32 ms, with most allocators taking ~ 27 ms and Gallatin taking 31 ms. Ouroborous-C-S is fastest for initialization, taking only ~ 12 ms. We have excluded the cost of one-time initialization in the other benchmarks.

6.5 Single-sized test

The single-size tests measure performance on 1M allocations/frees of a single size, stepping in powers of two from 16–4096 bytes. All tests are run 50 times, and the results reported are the median time.

Single-sized allocation test. Figure 4a shows the results of the single-size allocation tests. Gallatin is the fastest allocator for single-sized allocations, and is between $8-374\times$ faster than the next-best allocator for these tests, depending on the size of the allocations. The high throughput of Gallatin on allocations depends on several factors. Firstly, most allocations occur in one atomic operation, as up to 4096 atomicAdds can occur before a new block must be loaded. In addition, threads are coalesced before allocation, meaning that if multiple threads in a warp request an allocation of the same size, the requests will be grouped, amortizing the cost of the atomic operation.

Single-sized free test. The results for this benchmark can be seen in Figure 4b. Gallatin is the fastest allocator for single-sized frees, being between $2-39\times$ faster than the next best allocator. Like allocations, the ability to coalesce multiple frees allows Gallatin to reduce the contention on blocks and achieve high throughput.

6.6 Mixed-size test

The mixed-size tests measure performance when all threads allocate and free from a range of possible allocation sizes, between 16-4096 bytes. The number presented is the median latency over all allocations. The x-axis is the upper end of the range, and the lower end is always 16 bytes, with all allocation sizes being a power of two. All tests perform 1M allocations and frees per round, resetting the allocators between each round.

Mixed-size allocation test. The results of the mixed-size allocation tests can be seen in Figure 4c. Gallatin is the fastest allocator for mixed allocations of all ranges and is 8-264× faster than the next best allocator, depending on the allocation range.

Mixed-size free test. The results of the mixed-size free tests can be seen in Figure 4d. Gallatin is the fastest allocator for mixed frees and is $6-22\times$ faster than the next best allocator, depending on the free range.

6.7 Scaling test

The scaling allocation tests hold the allocation size static at 16 bytes and vary the number of threads from 2^0 to 2^{20} to measure the effect of increased parallelism on the allocators.

Scaling allocations. For the 16 and 64-byte scaling tests, Gallatin is the best among all allocators when allocating more than four items. Against the non atomic_wrap (AW) allocators, Gallatin scales in performance from $1.17\times$ to over $91\times$ faster than the next best allocator, as the number of simultaneous allocations increases. In addition, Gallatin is faster than the AW allocators when running with less than 131,072 threads, likely due to the coalescing of atomic operations in Gallatin. For the 512-byte scaling test, Gallatin is the fastest allocator until 16,384 threads, at which point it is 64% slower than ScatterAlloc. ScatterAlloc is the fastest allocator until 65,336 threads, at which point Gallatin is the fastest again and is between $2-8\times$ faster than the next best allocator. For the 8192-byte scaling test, Gallatin is the fastest allocator for all allocation sizes, being between 1.25 and $254\times$ faster than the next best allocator.

Scaling frees. For scaling frees, Gallatin is the fastest allocator or competitive with the best. In the worst case, Gallatin is $1.9\times$ slower than the state of the art. For 16 and 64-byte scaling frees, the performance of Gallatin is between $.83-11\times$ faster than the best allocator. For the 512 and 8192-byte frees, Gallatin is slower than ScatterAlloc and regEff-CFM between 2^{12} and 2^{15} threads for 512-byte frees and is at worst 133% slower than the best allocator for 8192-byte frees when the number of allocs is between 2^{12} to 2^{14}. For all other free sizes, Gallatin is the fastest or competitive at $.86-17\times$ faster than the next-best allocator.

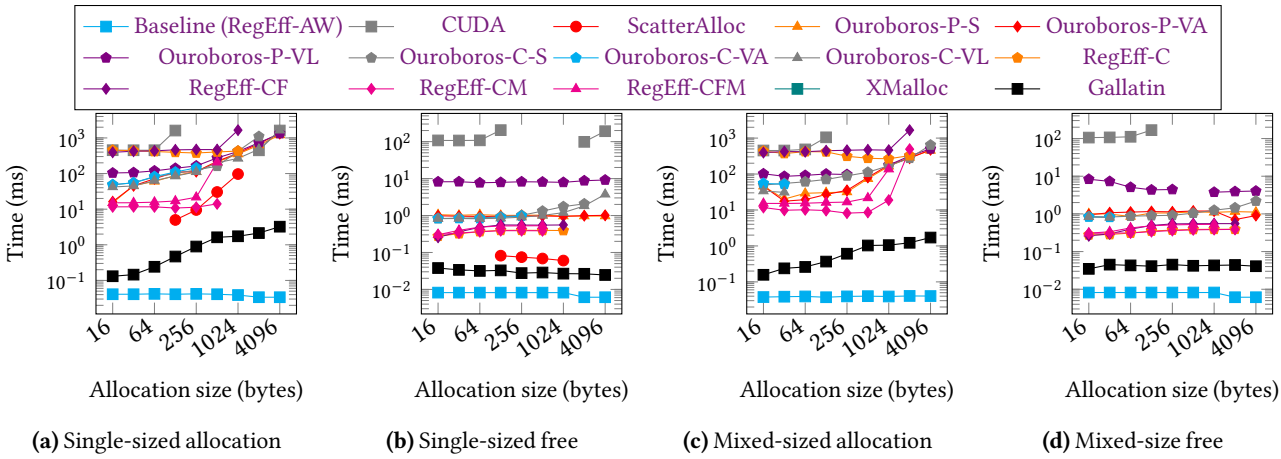
6.8 Variance in allocation and free latency

Gallatin has the lowest variance across all sizes for single and mixed-size allocations, with a variance that is $4-87\times$ lower than the next best allocator. When allocating more than 16 allocations for 16, 64, and 8192-byte scaling allocation tests, Gallatin has a variance between $.87-74\times$ lower than the best allocator. For 512-byte allocations, Gallatin loses to ScatterAlloc between $2^{13}-2^{16}$, although the difference is negligible: at 2^{16}, Gallatin has a variance of 0.0072 while ScatterAlloc has a variance of 0.0025.

Gallatin is the best or close to the best for variance during frees for all experiments, with variance between $.57-74\times$ lower than the best allocator.

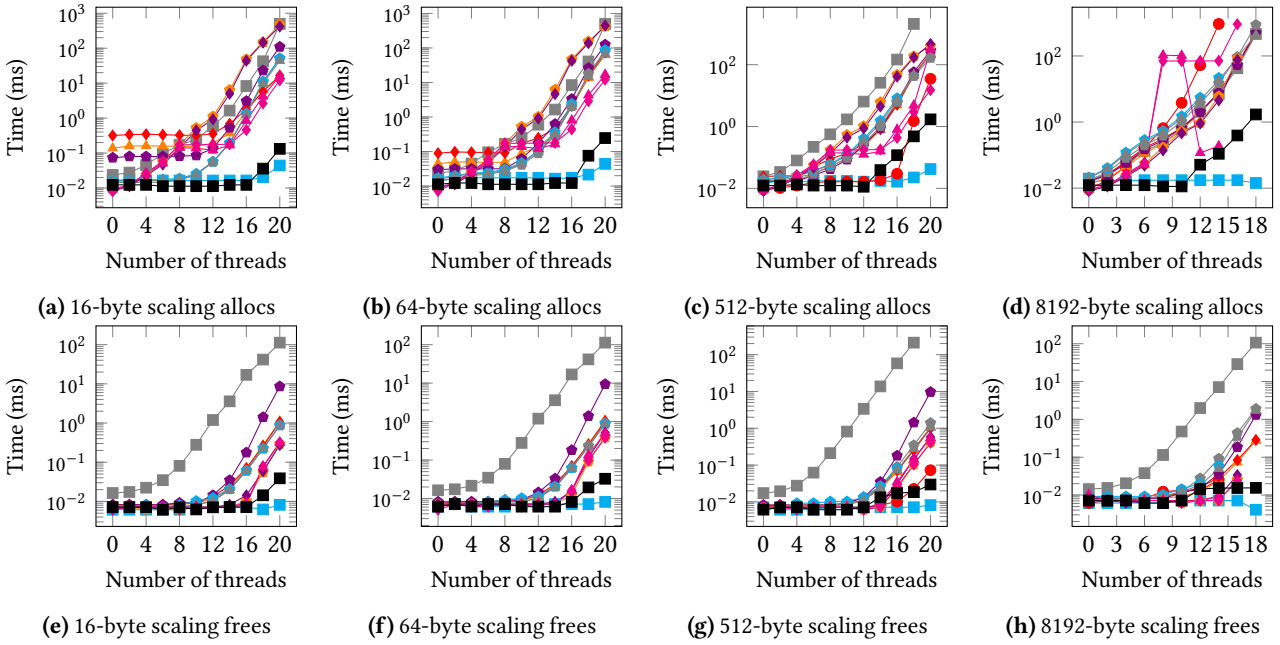
6.9 Experiments with warmed-up allocators

In addition to the main benchmark, we evaluated all allocators in a warmed-up state by running the first round of the



(a) Single-sized allocation **(b)** Single-sized free **(c)** Mixed-sized allocation **(d)** Mixed-size free

Figure 4. Allocation and free performance for single and mixed-size tests. Each allocation/free is handled by a unique thread. The x-axis is the allocation size for the single-size tests and the upper range for the mixed-size tests, with the lower range fixed at 16 bytes. The y-axis is median latency per thread across 50 runs. Lower is better.



(a) 16-byte scaling allocs **(b)** 64-byte scaling allocs **(c)** 512-byte scaling allocs **(d)** 8192-byte scaling allocs

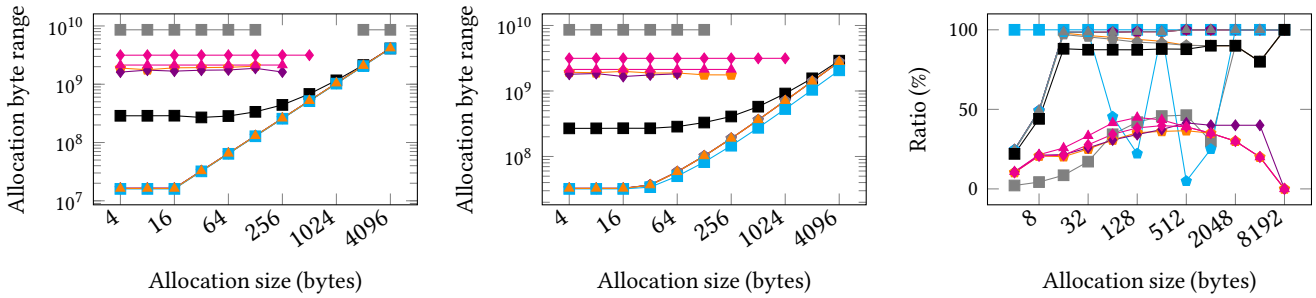
(e) 16-byte scaling frees **(f)** 64-byte scaling frees **(g)** 512-byte scaling frees **(h)** 8192-byte scaling frees

Figure 5. Scaling allocation tests and free tests for various allocation sizes. The x-axis is the $\log_2$ of the number of threads with the allocation size held constant. The y-axis is the median latency per thread over 50 runs. Lower is better.

benchmark before recording results and not resetting the allocator state, similar to how results are generated in Winter et al. [26]. Most allocators, including Gallatin, show no difference in performance when running warmed-up: on the single-sized allocation test, the largest change recorded for Gallatin was a change in latency from 2.13197 ms to 2.15344 ms for 2 KB allocations. The only allocators that do change performance are the Ouroboros-P series allocators, whose median performance lowers from 15.0069 ms to 0.224256 ms for 16-byte allocations. However, this is because these allocators cannot release memory once it has been acquired, meaning that subsequent runs start with the queue full of items. This inability to release allocations back to the system makes them unusable in many real applications.

6.10 Fragmentation tests

The fragmentation test measures the fragmentation of an allocator as the difference between the highest and lowest address given out when performing an allocation. All allocations occur as a static set of 1M allocations. The single-sized



(a) Single-size fragmentation **(b)** Mixed-size fragmentation **(c)** Memory utilization

Figure 6. Fragmentation performance for both single-sized and mixed-size allocations, along with the memory utilization performance. (a,b) For the fragmentation tests, the x-axis is the allocation size for the single-sized test and the upper bound of the range for the mixed-size test, with the lower bound being fixed at 16 bytes. The y-axis is the difference between the highest and lowest address given for an allocation. Lower is better. (c) For the utilization tests, the y-axis is a ratio compared to the maximum possible number of allocations that could be given. Higher is better.

fragmentation test uses a single set size, whereas the mixed fragmentation test allocates from a range of sizes using the generator from Section 6.6.

Single-size fragmentation. The results for the single-size fragmentation experiments can be seen in Figure 6a. Gallatin is the second best class of allocator in terms of fragmentation, following the Ouroboros series allocators. This is because allocations are always given from the first segment available. As the size of the allocations increases, Gallatin approaches the fragmentation of the best allocator, Ouroboros-C-VL, due to the wavefront used in Gallatin. During initialization, the first group of segments is pre-allocated to fill the block buffers of every allocation size. That means once the first new block is requested, it will come from a later memory segment. For example, the tree for 16-bit allocations will be initialized with segments 1 and 2, with segments 3–20 going to the other trees. Once a new block is required, the system will pull from the 21st memory segment instead of the 3rd, leading to a measurable gap in the fragmentation. As the allocation size increases, the segment used for the memory size approaches the end of the pre-allocated segment, meaning that segments are more adjacent to the new segments being used, bringing the fragmentation closer to optimal.

Mixed fragmentation. The results for the mixed fragmentation experiment can be seen in Figure 6b. The results are similar to those seen in Figure 6a, with the Ouroboros series allocators having the lowest fragmentation, followed by Gallatin, followed by the other allocators.

6.11 Utilization tests

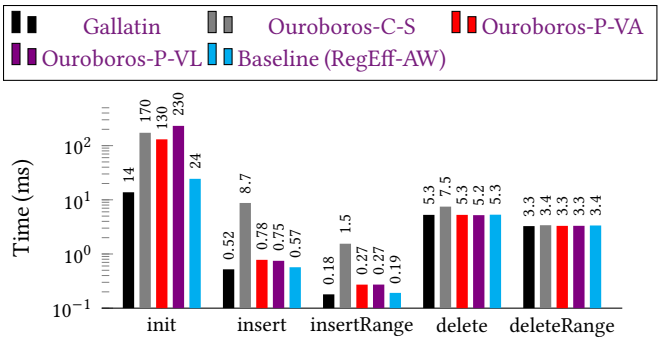
The utilization tests perform allocations until failure or timeout and report how often an allocation with 100K threads was possible as a ratio compared to the theoretical maximum number of allocations possible.

Figure 6c shows the results of the utilization test when run on allocation sizes between 4–8192 bytes with the allocators initialized with 2GB of memory. RegEff-AW has the highest utilization, as being a wrapper for an atomic operation it can hit and exceed 100% utilization by double allocating the same region of memory. After RegEff-AW, the Ouroboros C-VL, P-VL, and P-VA have the highest utilization, being able to use 98.8% of the memory when performing 256-byte allocations. Ouroboros P-S and C-S have the next highest utilization at 92.8% and 91.6%, respectively, followed by Gallatin at 89.1%. Gallatin has a lower utilization than the Ouroboros allocators due to the wavefront of blocks used in Gallatin. As all allocation sizes start with some blocks, live, allocating from only one size will leave the initialized blocks from other sizes untouched, which contributes to Gallatin's lower overall utilization. However, when calculating their overhead, the Ouroboros allocators do not consider the additional 500 MB of space used for the CUDA heap. This results in their total memory use being roughly 20% larger than reported. Including this space usage brings the space efficiency of all Ouroboros allocators below 89%, giving Gallatin the highest memory utilization.

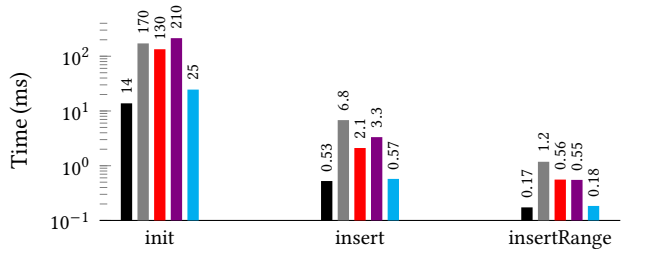
6.12 Graph tests

The graph tests measure the performance of the allocators when integrated into a graph workflow. Graph operations were measured on five operations: graph initialization, edge updates, bulk edge updates, edge deletes, and bulk edge deletes. The graphs are stored in memory as edge lists, with an edge list for a given vertex malloc the next largest power of two when it becomes too full.

The original experiments [26] were performed on the email dataset, a graph of 5,451 emails exchanged between 1133 members of the Univeristy Rovira i Virgili (Tarragona), and the 1138 bus dataset, a graph of 1138 nodes and 2596 edges of a bus power system. However, these tests do not fully stress the system as at most 2,600 threads are live per iteration. To more fully test the allocators, we use the Orkut dataset [19], a set of



(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.

3,072,626 vertices and 234,370,166 edges representing social connections on the website Orkut.

We test the Orkut dataset on two graph experiments, a scaled version of the graph test from Winter et al. [26], and an expansion test to measure performance when the graph grows over time. The scaled test initializes the graph and performs 100 iterations of adding and removing 1M edges from the graph. In the expanding graph test, the graph is initialized with a full set of 234,370,166 edges. Then over 100 rounds, 100M additional edges are added in batches of 1M. This mimics the growth of real-world datasets, where graphs often expand over time [19].

The results of these experiments are shown in Figure 7. For the original benchmark, Gallatin is the fastest for initialization, outperforming the non-AW allocators by an order of magnitude. Gallatin is the fastest allocator for insertion and is 1.46× faster than the next best allocator. For deletions, Gallatin is 1.1% slower than the best allocator. Gallatin is the fastest for the bulk operations, being 53% faster than the next best allocator for range insertion and 1.1% faster for range deletion. For the expansion tests, Gallatin is the fastest allocator of the fully functional allocators for all three operations, outperforming the next-best allocator by 12.24× for initialization, 3.94× for insertion, and 3.11× for bulk insertions.

The only RegEff variant that could complete the graph benchmarks was RegEff-AW. Other RegEff variants did not finish the benchmark.

6.13 Benchmark discussion.

Gallatin has the best performance when allocating from scratch and for most operations in the real-world graph experiments. For reuse, a more specialized allocator like Ouroboros-P-VA is the ideal choice if the size of the allocations remains consistent, as it can reuse allocations 56% faster than Gallatin. For the allocators that support full reuse, Gallatin is best in class. The ability to perform large allocations does affect the utilization and fragmentation of Gallatin. While this strategy does lower the overall utilization compared to the state-of-the-art, the ability to perform both small and large allocations from the same memory closes the utilization gap and allows for allocations that span the entirety of GPU memory.

7 Conclusion

We present Gallatin, a general-purpose and high-performance GPU memory allocator based on the van Emde Boas (vEB) tree. vEB trees offer fast, highly-concurrent operations in bounded time (loglog$u$). vEB trees are well suited for efficiently managing memory objects on massively parallel GPUs. Gallatin can exceed (or match in some cases) the performance of the state-of-the-art while also allowing for much larger allocations than were previously possible. Gallatin also provides full re-use of memory, which is not feasible for Ouroboros-P-VA and P-S, the fastest competing allocators.

Gallatin offers faster or competitive performance across various benchmarks when compared to more specialized variants designed for specific scenarios. Gallatin is a true general-purpose allocator. Gallatin is the ideal choice for tasks with allocation-heavy workflows, tasks with an unknown allocation pattern, and tasks requiring large allocations. It is positioned well as a general-purpose allocator, significantly improving over the default CUDA allocator in every context. In addition, the configurable nature of Gallatin allows it to be easily specialized for specific tasks. These results show that a tree-based allocation system is competitive with the existing linked list and queue-based solutions while providing extra features.

Acknowledgments

This research is funded in part by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the Department of Enery (DOE) under contract number DE-AC02-05CH11231, the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DEAC02-05CH11231.

References

[1] 2023. Kinectica. https://www.kinetica.com/

[2] 2023. NVIDIA Documentation Hub - NVIDIA Docs. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

[3] 2023. RAPIDS. https://rapids.ai/

[4] Andrew V. Adinetz and Dirk Pleiter. 2014. Halloc: a High-Throughput Dynamic Memory Allocator for GPGPU Architectures. https://github.com/canonizer/halloc https://github.com/canonizer/halloc.

[5] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. 2018. A Dynamic Hash Table for the GPU. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*. 419–429. https://doi.org/10.1109/IPDPS.2018.00052

[6] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[7] Matthias Becker, Umesh Worlikar, Shobhit Agrawal, Hartmut Schultze, Thomas Ulas, Sharad Singhal, and Joachim L. Schultze. 2020. Scaling Genomics Data Processing with Memory-Driven Computing to Accelerate Computational Biology. In *High Performance Computing*, Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief (Eds.). Springer International Publishing, Cham, 328–344.

[8] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2018.8547541

[9] Marius Erbert, Steffen Rechner, and Matthias Müller-Hannemann. 2017. Gerbil: a fast and memory-efficient k-mer counter with GPU-support. *Algorithms Mol. Biol.* 12, 1 (2017), 9:1–9:12. https://doi.org/10.1186/s13015-017-0097-9

[10] Oded Green and David A. Bader. 2016. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2016.7761622

[11] Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen-mei Hwu. 2010. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *2010 10th IEEE International Conference on Computer and Information Technology*. 1134–1139. https://doi.org/10.1109/CIT.2010.206

[12] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The Art of Balance: A RateupDB Experience of Building a CPU/GPU Hybrid Database Product. *Proc. VLDB Endow.* 14, 12 (2021), 2999–3013. https://doi.org/10.14778/3476311.3476378

[13] Hunter McCoy, Steven Hofmey, Katherine Yelick, and Prashant Pandey. 2023. Singleton Sieving: Overcoming the Memory/Speed Trade-Off in Exascale K-mer Analysis. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*. 213–224. https://doi.org/10.1137/1.9781611977714.19

[14] Hunter McCoy, Steven Hofmeyr, Katherine Yelick, and Prashant Pandey. 2023. High-performance filters for gpus. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 160–173.

[15] Hunter Mccoy and Prashant Pandey. 2024. Gallatin Source Code. https://github.com/saltsystemslab/gallatin

[16] Hunter Mccoy and Prashant Pandey. 2024. Modified Memory Manager Survey Source Code. https://github.com/saltsystemslab/memmansurvey

[17] Israt Nisa, Prashant Pandey, Marquita Ellis, Leonid Oliker, Aydın Buluç, and Katherine Yelick. 2021. Distributed-Memory k-mer Counting on GPUs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 527–536. https://doi.org/10.1109/IPDPS49936.2021.00061

[18] NVIDIA. 2023. CUDA Cooperative Groups. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cooperative-groups

[19] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluç. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1372–1385. https://doi.org/10.1145/3448016.3457313

[20] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. 2012. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *2012 Innovative Parallel Computing (InPar)*. 1–10. https://doi.org/10.1109/InPar.2012.6339604

[21] P. van Emde Boas. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.* 6, 3 (1977), 80–82. https://doi.org/10.1016/0020-0190(77)90031-X

[22] M. Vinkler and V. Havran. 2015. Register Efficient Dynamic Memory Allocator for GPUs. *Computer Graphics Forum* 34, 8 (2015), 143–154. https://doi.org/10.1111/cgf.12666 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12666

[23] Martin Winter, Daniel Mlakar, Mathias Parger, and Markus Steinberger. 2020. Ouroboros: Virtualized Queues for Dynamic Memory Management on GPUs. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 38, 12 pages. https://doi.org/10.1145/3392717.3392742

[24] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. faimGraph: High Performance Management of Fully-Dynamic Graphs Under Tight Memory Constraints on the GPU. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 754–766. https://doi.org/10.1109/SC.2018.00063

[25] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2019. Adaptive Sparse Matrix-Matrix Multiplication on the GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 68–81. https://doi.org/10.1145/3293883.3295701

[26] Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. 2021. Are Dynamic Memory Managers on GPUs Slow? A Survey and Benchmarks. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 219–233. https://doi.org/10.1145/3437801.3441612

[27] Martin Winter, Rhaleb Zayer, and Markus Steinberger. 2017. Autonomous, independent management of dynamic graphs on GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091058

[28] Jianting Zhang and Le Gruenwald. 2019. Efficient Quadtree Construction for Indexing Large-Scale Point Data on GPUs: Bottom-Up vs. Top-Down. In *ADMS@VLDB*. https://api.semanticscholar.org/CorpusID:198162358

A Artifact Appendix

Gallatin is entirely open-source, and the most up-to-date version of the source code is available at https://github.com/saltsystemslab/gallatin.git.

The testbed for reproducing the results in this paper is available at https://github.com/saltsystemslab/memmansurvey. A static version of these repositories is available at https://zenodo.org/records/10475796.

A.1 Reproducing results

The results of this paper can be reproduced in their entirety using the test suite in the `memmansurvey` benchmark. This can be pulled from the github URL above or from the static zenodo link. Once a copy of the repository is downloaded, run the following steps to build and execute the experiments.

1. **Determine architecture:** Determine your compute architecture from https://developer.nvidia.com/cuda-gpus.
2. **Clone repository:** Pull a fresh clone with submodules via `–recurse-submodules` or initialize the submodules with `git submodule init && git submodule update`.
3. **Curl graphs:** The graphs used in the graph benchmark are stored in a persistent zenodo link. Running `graph_curl.sh` will download these graphs and place them in `graph_tests/data`.
4. **Initialize tests:** Initialize the repository with `python init.py`. This verifies that boost has been installed.
5. **Build experiments:** Run `python setupAll.py –cc YOUR_ARCHITECTURE` to compile all tests. Each test suite has its own CMake file that can be built independently using the `setup.py` file located in the test directory.
6. **Run experiments:** To run a representative testsuite, simply call `python testAll.py -mem_size 8 -device 0 -runtest -genres`. The memory size is in Gigabytes, and the device ID of the device used for testing has to match the compute version passed in the build stage.
7. **Compile results:** Run `python process_results.py` to produce a pdf `artifact.pdf` with the results.

These instructions are packaged into an execuatable `download_and_run.sh` which will build and execute all tests.

In total, the tests take roughly 8 hours to execute on an A40. Executing `python3 testGallatin.py -mem_size 8 -device 0 -runtest -genres` will generate results only for Gallatin, and should finish in roughly half an hour.

To clean or reset the build folders, call `python cleanAll.py`. For more information on building and running the experiments, see the readME in the `memmansurvey` repository.

A.2 Using Gallatin

Gallatin is a header-only library, and once included can be used in a project by including `gallatin/allocators/gallatin.cuh`. Gallatin can be included in a project by including the repository as a git submodule or via the use of the CMake Package Manager (CPM). Once the repository has been linked it can be included in the project by linking it to a library or target executable via `target_link_libraries( EXE_NAME PRIVATE gallatin)`. More information on including and using Gallatin is available in the README of the Gallatin repository.

For convenience, Gallatin has a global variant which can be called with static device pointers. To use the global variant, include `global_allocator.cuh`. Once this has been inlcuded, the allocator can be initialized with `init_global_allocator(num_bytes)`. Once initialized, `void * global_malloc(num_bytes)` and `void global_free(void * alloc)` can be used in any device function to allocate and free memory. For more information see the readME in the `gallatin` repository.