Dynamically Determined Output Space For SpGEMM Using GPU Virtual Memory Management

By

MARJERIE SURESH
THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Dr. John Owens, Chair

Dr. Jason Lowe-Power

Dr. Houman Homayoun

Committee in Charge

2023

To my parents, Suresh and Snophia, and my grandparents.

Contents

LIST OF FIGURES

List of Tables

ABSTRACT

Dynamically Determined Output Space For SpGEMM Using GPU Virtual Memory Management

Sparse general matrix-matrix multiplication (SpGEMM) is a costly yet fundamental operation used across numerous scientific computing domains, such as machine learning, graph algorithms, and computational fluid dynamics. As GPUs are increasingly being used to accelerate scientific computing applications, optimizing SpGEMM for GPUs is a critical research area. One of the challenges in implementing SpGEMM on GPUs is determining the size of the output matrix, as the sparsity of the input matrices is not known beforehand. Precisely determining the size of the output matrix can add to the latency of the SpGEMM kernel, especially for large graphs.

This work presents an algorithm that dynamically expands the size of the output matrix as the SpGEMM kernel proceeds with the computation. This technique can improve memory utilization efficiency by avoiding unnecessary memory allocation at a modest performance cost. However, implementing a dynamic data structure like a vector to store the output matrix using the default `cudaMalloc`-based allocator is not memory efficient. This is because it has to allocate more memory than necessary when growing the vector, and there is latency associated with the `cudaMemcpy()` calls to copy the data from the old allocation to the new allocation. Hence, this work also presents a custom virtual memory-based allocator. Virtual memory-based allocations are better for growing allocations because new allocations can be created and mapped to a contiguous virtual address range with ease.

Our experiments show that the dynamic SpGEMM algorithm proposed in the work gives 70–99% memory efficiency with a modest performance cost when compared to the SpGEMM algorithm that uses the upper bound method for estimation. The proposed algorithm also consistently better performance for a modest loss in memory efficiency when compared to the SpGEMM algorithm that uses the precise estimation method. We also demonstrate that our custom virtual memory-based allocator can improve the memory efficiency and overhead latency of dynamic data structures on GPUs.

ACKNOWLEDGMENTS

I am deeply grateful to my advisor, Professor John D. Owens, for their guidance, support, and encouragement throughout this project. His expertise and insights were invaluable to me, and I am truly grateful for his mentorship.

I would also like to thank our postdoc Serban D. Porumbescu and the members of my research group for their help and feedback. I am grateful for their willingness to share their knowledge and expertise, and for making my time in the lab so enjoyable.

I would like to extend my gratitude to Professor Jason Lowe-Power and Professor Houman Homayoun for their valuable feedback and for dedicating their time to be a part of my thesis committee.

I would like to extend my sincerest thanks to Stephen Jones (NVIDIA) who was always there to answer all our questions and for taking the time to give intriguing talks during group meetings and course lectures.

I am also grateful to my friends for their love and support. They have been there for me through thick and thin, listening to me rant and rave about graduate school, and always offering words of encouragement. I am so grateful for their friendship.

Finally, I would like to thank my parents, Suresh and Snophia, and grandparents, Rajamoney, Selvabai, Sither, and Florence, for everything they have done for me. They have supported me in every way possible, and I am so grateful for their love and sacrifice. I am truly blessed to have them in my life.

This thesis would not have been possible without the support of all of these amazing people. I am truly grateful for each and every one of them.

Chapter 1

Introduction

In this thesis, we propose an algorithm that can dynamically determine the output space for Sparse General Matrix-Matrix Multiplication (SpGEMM). To aid with efficient growing vector allocation, we also provide a custom virtual memory-based allocator, which is implemented alongside the library Thrust. The algorithm proposed, when used with virtual memory-based vector allocation, can provide better memory efficiency than SpGEMM algorithm using the upper bound estimation method, and faster runtimes than the SpGEMM algorithm using precise estimation method to determine the size of the output matrix.

1.1 Overview

Sparse General Matrix-Matrix Multiplication (SpGEMM) is an expensive, fundamental primitive that is used in numerous scientific computing domains like machine learning, graph algorithms, etc. Hence, any optimization to the implementation positively impacts a wide range of applications. With the increasing use of Graphics Processing Units (GPUs) for accelerating scientific computing applications, optimizing for GPU specific scenario is an interesting research topic. One main issue with using GPUs for such big data applications is the low device memory. So, the algorithms have to be memory efficient.

The general implementation of SpGEMM has a challenging task of estimating the number of non-zeros in the output matrix which determines the size of the output matrix. This is because the sparsity of the input graphs are not known prior to computation. Precisely estimating the size by iterating through the matrices twice can add to the latency and this only increases with larger graphs. The methods that estimate an upper bound of the size leads to unnecessary overhead of the available memory resource. The overestimation factor for this method is $\sim 1.5 - 20$ with worse estimation for sparser matrices. Libraries like cuSPARSE and Intel Math Kernel precisely allocate the output space by iterating twice while the library CUSP uses the upper bound method to estimate the number of non-zeros in the output matrix. These approximate methods that overestimate the size leads to wastage of precious memory resources which can make or break the application in low memory cases and the ones that precisely calculate the output space adds to latency by iterating through the matrices twice. Hence, an option is to dynamically grow the output size of the data structure as the SpGEMM kernel progresses with the computation and requires more memory.

1.2 The Problem

An ideal algorithm for SpGEMM will have low cost, iterate through the input matrices only once, and minimizes over-allocation of memory resources. An algorithm that dynamically determines the output space is ideal as it will eliminate the need to iterate through the matrices twice and help with using the available memory resources wisely.

Implementing a dynamic data structure like vector to store the output matrix using the default `cudaMalloc`-based allocator is inefficient. `cudaMalloc()` has to allocate more memory than necessary, because, to grow, it has to have enough space for the old allocation and the new allocation. Additionally, there is also latency associated with the `cudaMemcpy()` calls to copy the data from the old allocation to the new allocation during resizing the vector.

Memory efficiency for dynamic data structures can be achieved using virtual memory-based allocators as they provide better control of memory. Virtual memory-based allocations are better for growing allocations due to the ease with which new allocations can be created and mapped to a contiguous virtual address range. Hence, using this custom virtual memory-based data structure for dynamically determining the output space of an SpGEMM implementation is an interesting solution for low memory situations.

1.3 The Approach

To be able to manage virtual memory allocations, we require a system that can be called when needed. We implement this system as a memory allocator which, at its core, does allocations and de-allocations based on the principles of virtual memory. This custom virtual memory-based memory allocator can then be used to allocate and grow the memory as required on the fly for the SpGEMM algorithm.

The Thrust library is the most commonly used standard template library for dynamic data structures like vectors in GPUs. An abstraction for virtual memory management that makes sense for Thrust is a custom memory allocator. This custom memory allocator can be specified when creating a vector so that the underlying memory allocations are based on virtual memory management techniques. Since CUDA 10.2, low-level driver APIs are provided for GPU virtual memory management. Using these APIs, we implement the custom memory allocator which, when called along with the `thrust::device_vector`, provides us a custom virtual memory-based vector. We can use this vector to dynamically grow the output matrix space for the SpGEMM algorithm.

1.4 Contributions

We implemented the following two main contributions in this work:

1. A custom allocator implemented as an abstraction for virtual memory-based vector that can be used alongside the Thrust library. This hides the complexities associated with the memory allocation and de-allocation from the programmer and improves productivity.

2. An SpGEMM algorithm that uses the custom virtual memory-based vector to dynamically grow the output space on the fly as needed. This algorithm avoids the need to iterate through the input matrices twice to estimate the size of the output and provides memory efficiency of 70–99%. This helps relieve the stress on the limited device memory resource available on the GPU.

1.5 Benefits

With this thesis, we show that using the custom virtual memory-based allocator can yield the following benefits.

1. **Memory efficiency:** When growing, it does not have to create a new larger allocation along with the old allocation.

2. **Performance:** It eliminates the latency caused by the `cudaMemcpy()` calls when the contents of the older allocation have to be copied to the new allocation. In addition, the wasteful use of bandwidth for the copy is also eliminated.

3. **Simplicity:** It makes the implementation of algorithms like resize, append, etc. for vector manipulation simpler, as the complexities associated with creating and copying between new and old allocation are not present.

4. **Ease of use:** With the custom allocator, it is easy to add a virtual memory-based system to any application. Only the definition of the vector has to be changed to specify the use of the custom virtual memory-based allocator. The functions to manipulate the vector works the same as the default vector.

Using the benefits shown by the custom virtual memory-based vector, we can implement an efficient dynamically determined output space for SpGEMM which can eliminate the need for estimation methods.

1.6 Related Work

SpGEMM is used in a wide variety of domains, including engineering, chemistry, physics, machine learning, and others. As the data size for these problems increases, there is a growing need for parallel implementations of SpGEMM.

Some popular parallel implementations of SpGEMM on the GPU include the NVIDIA CUDA Sparse library (cuSPARSE) [8], CUSP [3] and Intel Math Kernel Library (MKL) [7]. The libraries cuSPARSE and MKL precisely allocate the required memory for the resultant matrix by iterating through the input matrices. CUSP employs the upper bound method to estimate the size of the output matrix. J. J. Elliott and C. M. Siefert proposed the Low Thread-count Gustavson algorithm [5] designed for CPU using OpenMP, which uses dynamic memory allocation. However, current implementations on the GPU do not use dynamic memory allocation to store the resultant matrix.

This work proposes a method to dynamically determine and allocate space for SpGEMM and the memory allocations are managed using a custom virtual memory-based allocator to overcome the disadvantages of using a traditional memory allocator.

1.7 Outline of the Thesis

In Chapter 2, we will cover the background information about the libraries, techniques and algorithms used throughout the thesis work. Chapter 3 describes the implementation of the custom virtual memory-based allocator and its application in the dynamically determined output space for SpGEMM. In Chapter 4, we talk about the methodology used to test the system developed. In Chapter 5, we discuss the results obtained from the experiments, and conclude the thesis in Chapter 6. We conclude with remarks on the work done and also talk about possible future work that can be done to improve on the current implementation.

Chapter 2

Background

In this chapter, we introduce the foundational material for the work. We start with discussing SpGEMM and the different challenges with implementing a parallel SpGEMM for GPUs in Section 2.1. Section 2.2 talks about the general design for a parallel SpGEMM algorithm. We follow it with sections on why different components in the algorithm are inadequate. Finally, Section 2.5 discusses the basics on virtual memory and we end with Section 2.6 briefing about the APIs provided with CUDA to implement GPU virtual memory management.

2.1 Sparse General Matrix-Matrix Multiplication

Sparse General Matrix-Matrix Multiplication (SpGEMM) is an elementary matrix multiplication operation that is specific to sparse input matrices. SpGEMM is a building block that is widely used in complex machine learning and data analysis algorithms. The two sparse input matrices are represented using formats like Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC).

2.1.1 Challenges with Parallel SpGEMM Implementation

There are three main challenges that arise with the parallel implementation of SpGEMM [6].

- The first challenge is the estimation of size of the output matrix due to the unknown sparsity of the input matrices. This estimation is crucial as it dominates the memory resource overhead with larger matrices.

- The second challenge is load balancing of the multiplication among the threads of the GPU. A proper balancing of the work is important for efficient use of the computational resources available in the GPU.

- The third challenge is accumulating the result properly in the output space. Due to the unknown sparsity, it is a difficult task to identify which element of the output space the computational unit should accumulate the result on.

This work addresses the first challenge with an algorithm that can dynamically determine the output space.

2.2 Parallel Implementation of SpGEMM

There are many parallel SpGEMM implementations available. Some of the popular implementations are in GPU libraries like NVIDIA CUDA Sparse Library (cuSPARSE), CUSP, etc. For our baseline implementation, we use the parallel SpGEMM provided with Gunrock, an open-source graph-processing library designed for the GPU using CUDA.

The general parallel implementation of SpGEMM as provided by Gunrock [9, 11] is shown in Listing 2.1. The first input matrix is represented using the Compressed Sparse Row (CSR) format, while the second input matrix is represented using the Compressed Sparse Column (CSC) format. This is because to compute an element in the output matrix, we have to traverse along the row of the first matrix and the column of the second matrix. This can be done optimally when they are in CSR and CSC formats respectively.

```
 auto naive_spgemm = [=] __host__ __device__(vertex_t const& row)
-> bool {
   // Get the number of non-zeros in row of sparse-matrix A.
   auto a_offset = A.get_starting_edge(row);
   auto a_nnz = A.get_number_of_neighbors(row);
   auto c_offset = thread::load(&row_off[row]);
   auto n = 0;
   bool increment = false;

   // Iterate over the columns of B.
   for (edge_t b_col = 0; b_col < B.get_number_of_vertices(); ++
b_col) {
     // Get the number of non-zeros in column of sparse-matrix B.
     auto b_offset = B.get_starting_edge(b_col);
```

```
      auto b_nnz = B.get_number_of_neighbors(b_col);
      auto b_nz_idx = b_offset;
      auto a_nz_idx = a_offset;

      // For the row in A, multiple with corresponding element in B
.
      while ((a_nz_idx < (a_offset + a_nnz)) &&
             (b_nz_idx < (b_offset + b_nnz))) {
        auto a_col = A.get_destination_vertex(a_nz_idx);
        auto b_row = B.get_source_vertex(b_nz_idx);

        //  Multiply if the column of A equals row of B.
        if (a_col == b_row) {
          auto a_nz = A.get_edge_weight(a_nz_idx);
          auto b_nz = B.get_edge_weight(b_nz_idx);

          // Calculate  C's nonzero index.
          std::size_t c_nz_idx = c_offset + n;
          assert(c_nz_idx < estimated_nzs);

          // Assign column index.
          thread::store(&col_ind[c_nz_idx], b_col);

          // Accumulate the nonzero value.
          nz_vals[c_nz_idx] += a_nz * b_nz;

          a_nz_idx++;
          b_nz_idx++;

          increment = true;
        } else if (a_col < b_row)
          a_nz_idx++;
        else
          b_nz_idx++;
      }
      // A non zero element was stored in C, so we increment n.
      if (increment) {
        n++;
        increment = false;
      }
    }
    return false;
  };
```

Listing 2.1: Parallel Implementation of SpGEMM

Each row of the output matrix is computed by a thread in parallel on the GPU. So, each thread iterates through a row of the first input matrix and computes a dot product with the corresponding column of the second input matrix. This is repeated until the row has computed a dot product with every column in the second matrix. This simplifies load balancing of the algorithm and the accumulation of the result in the output space allocated.

As mentioned earlier, the output space of SpGEMM computation is difficult to estimate due to the unknown sparseness of the input matrices. The precise method to estimate the size of the output matrix is done by iterating through the input matrices once before computation. The implementations of SpGEMM in cuSPARSE, Intel Math Kernel Library (MKL), etc. employ this method.

Another common method to estimate the size is the upper bound method as shown in Listing 2.2. This method estimates the upper bound of the number of non-zeros in each row of the output matrix. CUSP [3], a C++ template library for sparse linear algebra and graph computations on CUDA, applies the upper bound method to obtain the size of the result matrix.

In the upper bound method, for each non-zero in the row of the first matrix, the number of non-zeros present in the corresponding row of the second matrix is computed and added to the estimate. This is implemented in parallel for the GPU with each thread working on a non-zero value from the first matrix. We need to use an atomic add operation as the threads working on non zeros from the same row can compute an addition to the same vector location storing the estimate for the respective row. We can observe from the tests performed on different datasets as shown in Section 5.2 that it overestimates the size of the result matrix by a factor of 7.018. This was obtained by taking the geometric mean of the overestimation factor observed across the different datasets. The amount of overestimation varies depending on the sparsity of the input matrices.

```
   auto upperbound_nonzeros = [=] __host__ __device__
       (vertex_t const& m,   // ... source (row index)
       vertex_t const& k,   // neighbor (column index)
       edge_t const& nz_idx,   // edge (row -> column)
       weight_t const& nz      // weight (nonzero).
       ) -> bool {
     // Compute number of nonzeros of the sparse-matrix C for each
  row.
     math::atomic::add(&(estimated_nz_ptr[m]),
                         B_csr.get_number_of_neighbors(k));
     return false;
   };
```

Listing 2.2: Estimation of the upperbound of non-zeros per row of the output matrix

The output sparse matrix is stored using the CSR format. So, the row offsets are calculated using the estimation we obtain from Listing 2.2. This is used in the SpGEMM kernel to aid with the accumulation of the intermediate results after computation. The non-zero values and the column indices vectors are also allocated based on the size we obtain using the estimation method.

Looking at the algorithm implementation we discussed, there is a need for memory allocations on the GPU for keeping track of different components of the result matrix. The memory allocations for data structures like vectors to be used in the algorithms on the GPU are generally done using the Thrust library. We will discuss what Thrust has to offer in the following sections.

2.3 Thrust

The C++ Standard Template Library (STL) is implemented for the GPU using CUDA for parallel algorithms in the Thrust Library [1]. It is a header-only library, which means that in order to be used, it does not need to be compiled or installed. It has a high-level interface that improves the developer productivity by making it easier to write parallel code for GPUs. The library is scalable and can be used to develop and accelerate high-performance applications with CUDA alongside C++. Due to its ease of use, it hastens the time taken to prototype and develop parallel algorithms.

A vast range of parallel algorithms like sort, scan, reduce, etc. are provided with Thrust. It also provides vector containers for ease of manipulation of data, both on the CPU and GPU memory.

2.3.1 Vectors

There are two vector containers in the Thrust Library, `thrust::host_vector` and `thrust::device_vector`. `thrust::host_vector` is used to store data in the host or CPU memory, while the `thrust::device_vector` is used to store data in the device or GPU memory. They are implemented to resemble the C++ STL's `std::vector` container. Hence, they also have features like insert, erase, resize, etc., and can be expanded and contracted dynamically. One important feature that vector provides is the ability to specify memory allocators. Listing 2.3 shows the example usage of the vector containers `thrust::host_vector` and `thrust::device_vector` of the Thrust library. As mentioned earlier, since Thrust is a header-only library we include `thrust/device_vector.h` and `thrust/host_vector.h` to use the respective vectors.

```
// include for Thrust host and device vector
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>

size_t vector_size = 10;
// creates a device vector of given size and fills with value
thrust::device_vector<int> vector_d(vector_size, 15);
// reserves capacity of given size
vector_d.reserve(2 * vector_size);
// inserts the given number of elements at position and fills
with value
vector_d.insert(vector_d.begin(), vector_size, 0);

// creates a host vector
thrust::host_vector<int> vector_h;
// push back the given value
vector_h.push_back(10);
// resize to the given size
vector_h.resize(2 * vector_size);
// assign values using the operator []
vector_h[1] = 1;
vector_h[2] = 2;
vector_h[3] = 3;
```

Listing 2.3: Example usage of `host_vector` and `device_vector`

The allocation of memory that happens behind the scenes in the vector is managed using memory allocators. Let us look into memory allocators and the different ones provided by Thrust.

2.4 Memory Allocators

The memory allocator is a fundamental part of a library that is used to allocate and de-allocate memory for data structures or containers, such as vector, list, set, map, etc. They keep track of which memory is allocated and by whom, and free memory that is no longer necessary. They are used when dynamic memory allocation is required. This helps to hide the complexity of the memory management model from the programmers.

There are many types of memory allocators available, each with its own advantages and disadvantages. There is a default allocator for general-purpose applications, but there are other specific memory allocators available as well. There is also the option for the programmer to implement custom memory allocators. The choice of memory allocator can have a significant impact on the performance of a program. Some memory allocators are designed for speed and not necessarily for memory efficiency. So, one must be careful when choosing an appropriate memory allocator based on the specific needs of the application in consideration.

Thrust provides three different allocators based on different memory resources. They are the default allocator, universal allocator, and pinned allocator. We will go into detail about each of the allocators in the following subsections.

2.4.1 Default Allocator

The default allocator is the simplest allocator and is suitable for most applications. It uses the system's default memory allocation routines. It uses the functions, `cudaMalloc()` and `cudaFree()`, to allocate and de-allocate memory on the device. This is the default memory resource for the system and it wraps the allocation with the `cuda::pointer` type.

The default allocator is a good choice for most applications. It is simple to use and provides good performance. However, for applications that requires frequent efficient transfer of data between host and device, the universal or pinned allocators may be a better choice.

2.4.2 Universal Allocator

The universal allocator is more sophisticated than the default allocator and can be used to allocate memory from different sources, including the host memory and the device memory. This uses the universal memory resource which can be accessed both from the host and the device. The function `cudaMallocManaged()` is used for allocation and `cudaFree()` is used for de-allocation, similar to the default memory allocator. It wraps the allocation using `cuda::universal_pointer`.

2.4.3 Pinned Allocator

The pinned allocator is designed for applications that require a staging area between the host and the device for exchange of data. The memory allocated by this allocator is page-locked to the host. This ensures that the device can always read this memory with higher bandwidth when compared to other types of allocations. To allocate and de-allocate the host pinned memory resource, it uses the functions, `cudaMallocHost()` and the `cudaFreeHost()`. Similar to the universal allocator, the pinned allocator also wraps its allocation using the `cuda::`

`universal_pointer` type.

2.4.4 Disadvantages of Traditional Allocators

The memory allocators such as default allocator, universal allocator, and pinned allocator use traditional allocators for their memory allocations.

As mentioned earlier, vector containers with memory allocators are used to implement dynamic data structures. When expanding a vector, if contiguous virtual memory address space is not available immediately after the current allocation, then the vector has to take the long route. It has to create a new allocation with the larger size and then copy the contents of the vector from the older allocation using `cudaMemcpy()`. Once the copy is complete, the older allocation is freed. This leads to two issues:

1. **The need for extra space when resizing.** The old allocation is kept to hold the data in the vector, even though it is no longer being used. This leads to inefficient use of the GPU memory.

2. **The extra time spent to copy the data from the old allocation to the new allocation.** `cudaMemcpy()` calls are not cheap, and the time taken to copy increases with the increase in size of the vector.

The custom virtual memory-based allocation proposed in the thesis overcomes both of the issues mentioned. We will look into implementing it for GPUs in the next sections.

2.5 Virtual Memory

Virtual memory is a memory management technique that provides the illusion of a large, contiguous storage to the programmer. The operating system, using a combination of hardware and software resources, maps fragmented chunks of physical memory with physical addresses to a contiguous virtual memory address space.

When the allocation has to be expanded, the operating system will allocate the required physical memory and map it to a virtual memory address range of the appropriate size. The physical memory can be fragmented but will appear contiguous in the virtual address space.

It is especially useful for applications that need to allocate large amounts of memory, such as scientific simulations and graph algorithms.

Using a custom virtual memory management as the underlying virtual allocation system for Thrust vectors can be advantageous and help overcome the inadequacies shown by the standard memory allocators. It allows the vector to expand by reserving larger virtual address ranges. When the vector is expanded, the new allocation is simply mapped to the old allocation. This avoids the need to copy the data and also avoids the need to keep the old allocation.

So, it is clear that the custom virtual memory-based allocation is more efficient than traditional allocation. Now, let us look at how to create such allocations on the GPU.

2.6 GPU Virtual Memory Management APIs

Custom virtual memory-based allocation can be done on the GPU using four low-level driver functions [10], which were introduced since CUDA 10.2, as shown in Figure 2.1.

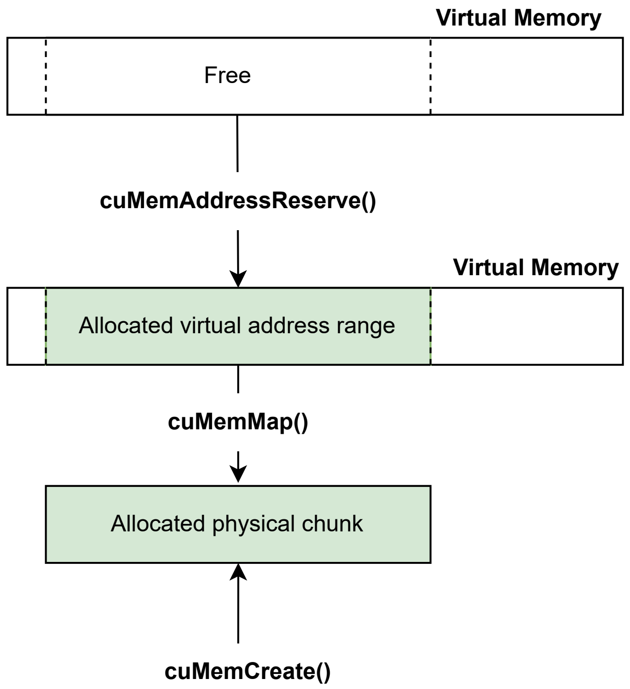


Figure 2.1: GPU Virtual Memory Management APIs.

`cuMemAddressReserve()` reserves a contiguous virtual memory address range as required. `cuMemCreate()` is used to allocate physical memory of the required size. It returns a physical memory handle which is mapped to the reserved virtual address space using `cuMemMap()`. The device can access the allocation after `cuMemSetAccess()` is used to set the appropriate memory access rights.

To de-allocate, we are provided with a similar set of CUDA driver APIs. `cuMemUnmap()` unmaps the physical memory backing the allocated virtual address range. `cuMemRelease()` releases the physical memory handle that was created, and `cuMemAddressFree()` frees the reserved virtual address range.

We can implement a custom virtual memory-based memory allocator using the above presented virtual memory management APIs, which is a part of the work done for the thesis.

Chapter 3

System Design

This chapter presents a high-level system design of the two contributions of this work: the custom virtual memory-based vector and an algorithm to dynamically determine the output space for SpGEMM.

3.1 Custom Virtual Memory-Based Vector

Dynamic data structures like vectors that use traditional memory allocation methods are inadequate as we have mentioned earlier. So, our goal is to implement a custom virtual memory-based vector to be able to efficiently allocate more memory and grow. The three different components of this system are as follows.

1. **A custom virtual memory-based memory allocator:** The vector, at its core, must have memory allocations based on the concept of virtual memory.

2. **Functionalities of the vector:** The different methods in the vector container must be updated to take advantage of the benefits of using the custom virtual memory-based allocations.

3. **Interfacing with the vector:** Using the vector must be simple and intuitive. To achieve this, we integrate the system with a commonly used C++ standard template library for GPUs called Thrust.

3.1.1 Custom Virtual Memory-Based Memory Allocator

A standard memory allocator manages the allocation, de-allocation and keeps track of the state of the allocation. The basic idea of allocating more memory in this case is to be able to create new physical memory and map it to a virtual address range that is of the appropriate size. There are two basic cases that must be covered by the allocator.

1. **Initial allocation:** This is when the allocation is created for the first time or when it is resized from zero to a non-zero size.
2. **Growing allocation:** This is when the allocation grows from a non-zero size to a larger size.

As shown in Figure 3.1, the process for allocating memory for the first case, if the size requested is less than the current capacity of the allocation is as follows. Since this is based on virtual memory, we have a few steps to follow to complete the allocation.

1. **Virtual address range reservation:** A contiguous address range in the virtual memory is reserved for the requested size. This is the address space that the user will interact with.
2. **Create physical memory allocation:** A physical memory region of the requested size has to be created. Physical memory allocations can be fragmented, and the user does not need to be aware of this.
3. **Mapping the physical memory allocation to the virtual address range:** The physical memory allocated in step 2 is mapped to the contiguous virtual address range reserved in step 1. This creates the illusion of a contiguous address space for the user to work with. Figure 3.2 shows an example of this mapping.

Once the allocation is complete, we update the pointer to the start of the memory and the size of the allocation. These values denote the state of the allocation.

In the second case, there are two possible paths that the procedure can take, as shown in Figure 3.3. The ideal case is that there is enough space at the end of the current virtual address reservation to accommodate the growing allocation, which is shown by the path on the left.

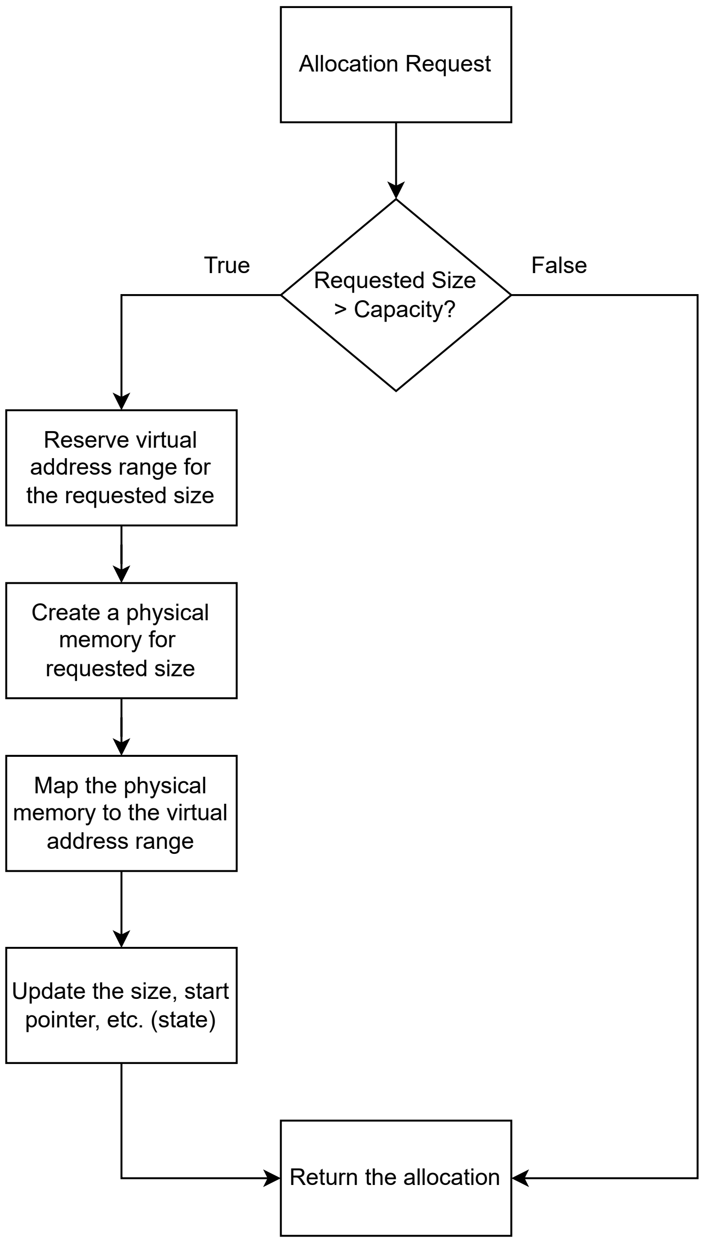

Allocation Request
Requested Size > Capacity?
True
False
Reserve virtual address range for the requested size
Create a physical memory for requested size
Map the physical memory to the virtual address range
Update the size, start pointer, etc. (state)
Return the allocation

Figure 3.1: Basic allocation

If this is the case, we can reserve the required space at the end of the current allocation by requesting it. The larger virtual address reservation is returned, and it is then mapped to the physical memory that was created for the appropriate size. This is shown in Figure 3.4.

If there is not enough space at the end of the current virtual address reservation, as shown by Figure 3.5, we have to take the longer route depicted by the path on the right of the flowchart in Figure 3.3. First, we have to unmap all of the current mappings. Then, we have to release

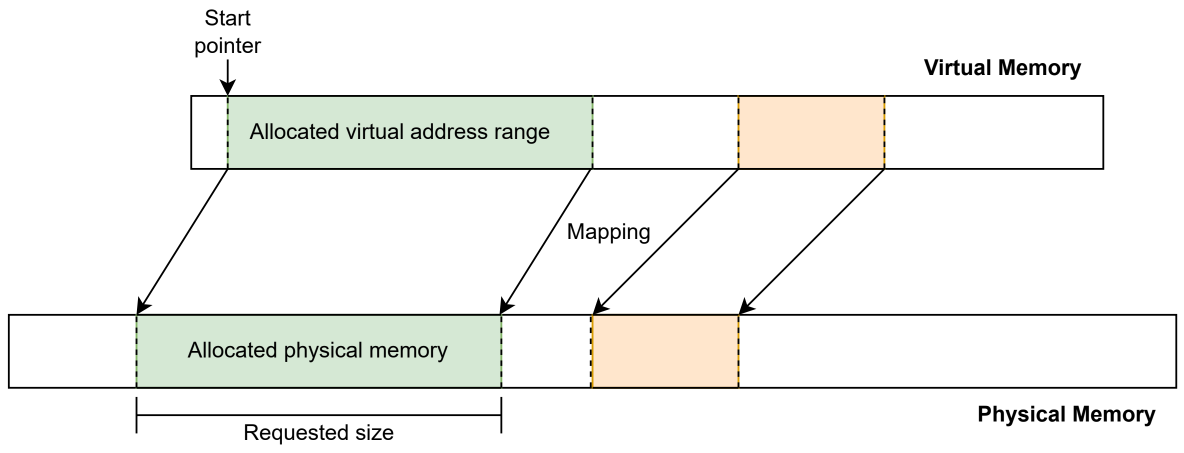


Figure 3.2: Memory mapping for a basic allocation

the existing virtual address range so that it can be used by another process. Next, we restart the allocation by requesting a new virtual address range of the larger size, as shown by Figure 3.6. This time, we should be able to get a contiguous virtual address range if there is enough memory available in the device. One thing to note here is that the start address is different for this allocation. Therefore, we need to update the start pointer of the allocation along with the size. Finally, we remap the physical allocations to the new virtual address range and return the new, larger virtual address range.

As seen in this subsection, to implement a custom memory allocator based on virtual memory, we need fine-grained control of the memory. The GPU virtual memory management APIs provided by CUDA, described in Section 2.6, helps us achieve this.

3.1.2 Adding to Thrust

As mentioned earlier in Section 2.3, the library Thrust is commonly used for dynamic data structures on the GPU Therefore, adding this functionality to Thrust can be beneficial to many programmers who are interested in using the system. To integrate the allocator to Thrust, we implement a class called `thrust::mr::virtual_memory_resource_allocator` which includes all the methods to provide virtual memory-based allocations. Appendix A.1 gives a detailed look into how this was implemented in Thrust.

For ease of use, we added the custom virtual memory-based allocator to the namespace `thrust`. So, it can now be called as `thrust::virtual_allocator<T>`. This ensures that

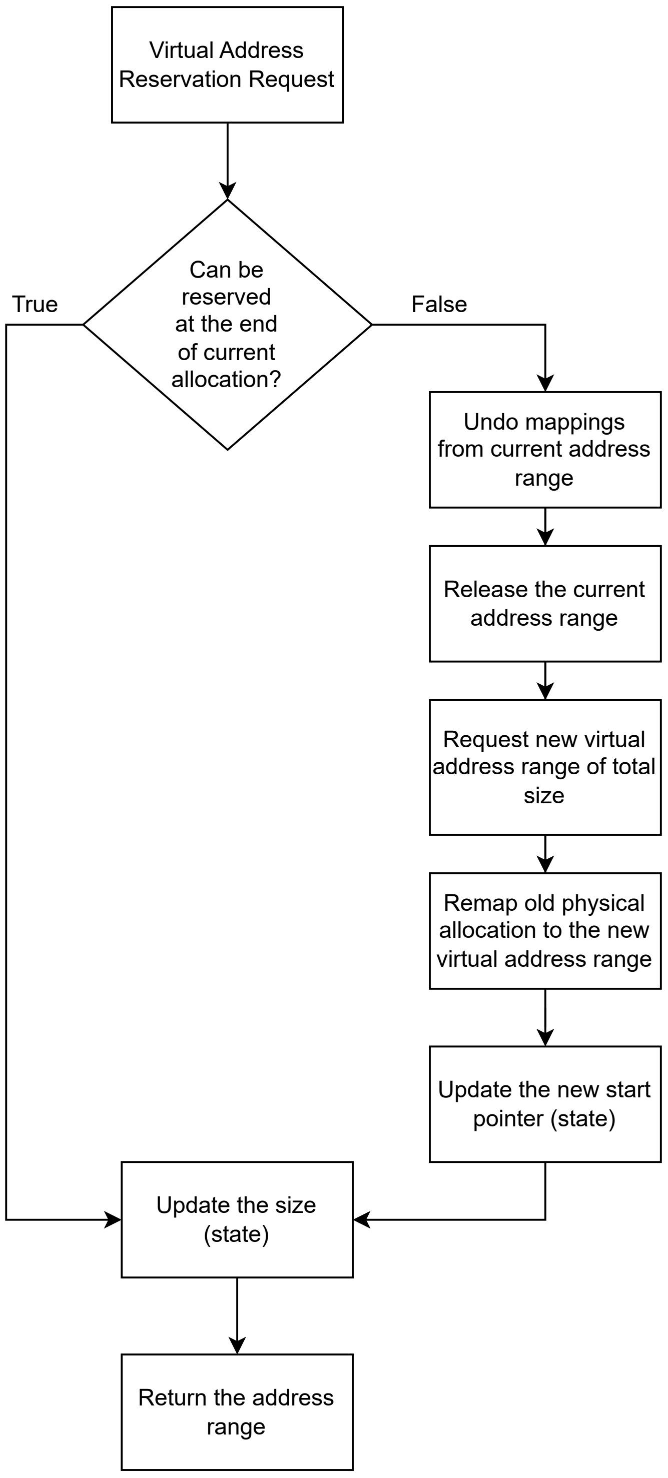

Virtual Address Reservation Request
Can be reserved at the end of current allocation?
True
False
Undo mappings from current address range
Release the current address range
Request new virtual address range of total size
Remap old physical allocation to the new virtual address range
Update the new start pointer (state)
Update the size (state)
Return the address range

Figure 3.3: Virtual address reservation for a growing allocation

the user does not have to interact with the underlying memory management details mentioned in the previous section. This allocator uses the CUDA device memory, which is denoted

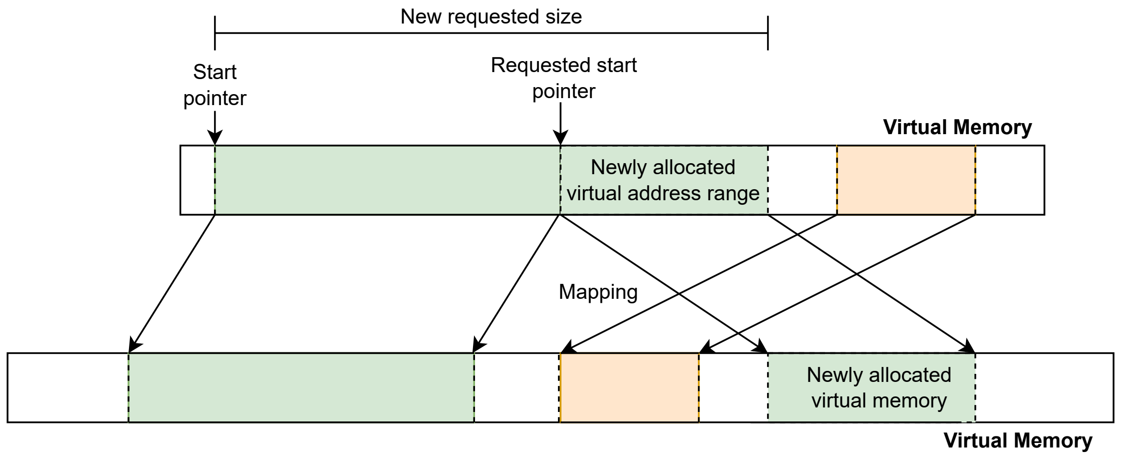


Figure 3.4: New virtual address reservation at the end of the current reservation

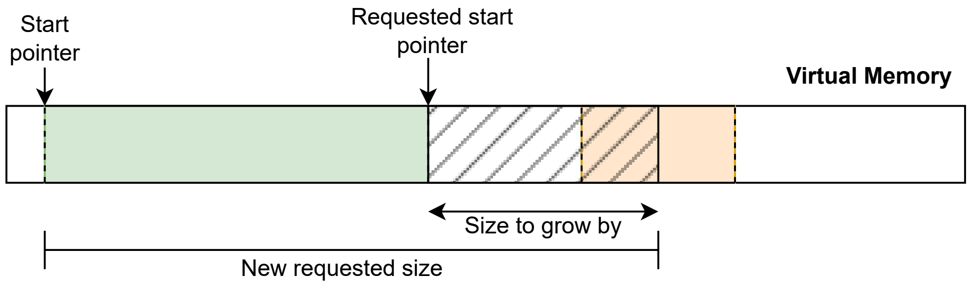


Figure 3.5: Failed virtual address reservation at the requested start address

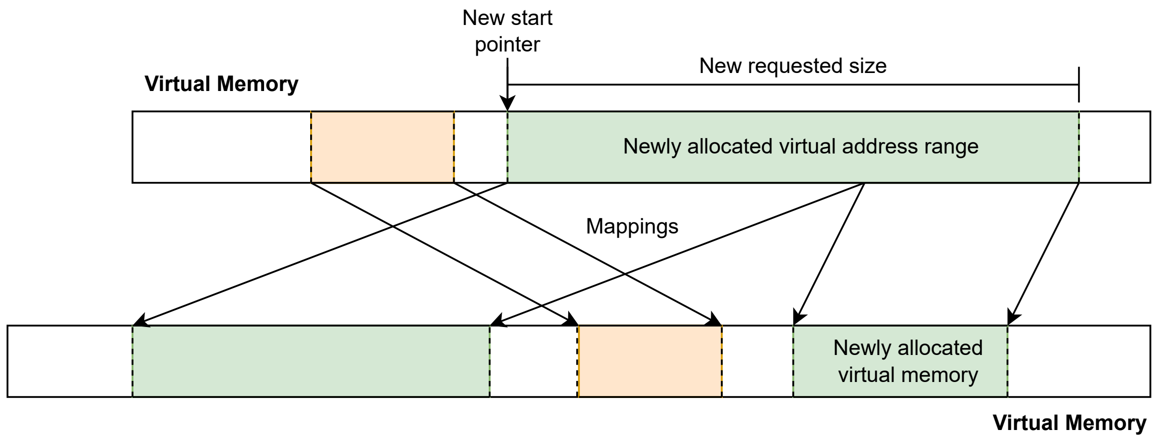


Figure 3.6: New virtual address reservation for the requested size

with `thrust::system::cuda::memory_resource`. The definition of the custom allocator, `virtual_allocator`, is shown in Listing 3.1.

```
namespace thrust
{
    template<typename T>
    using virtual_allocator = thrust::mr::
virtual_memory_resource_allocator<
    T, thrust::system::cuda::memory_resource
    >;
}
```

Listing 3.1: The definition of `virtual_allocator` for the `thrust` namespace

3.1.3 Functionalities of the Vector

Now that our custom allocator, `virtual_allocator`, is added to Thrust, we need to look into using it for the vector container. The vector container has functions to create and manipulate the vector, such as copy, reserve, resize, insert, fill, erase, etc. The methods used to manipulate the vector are written for a `cudaMalloc`-based allocation, which is the default. Therefore, some changes have to be made to such methods in order to observe the benefits of using the custom virtual memory-based allocator.

The functions like reserve, resize, append, insert, push_back, etc. in the vector container lead to growing allocations in the device memory. First, let us look at how growing allocations are handled in the default functions given by Thrust. Algorithm 1 describes the procedure to reserve more memory for the default vector. The following steps are followed:

1. A new storage with the new capacity requested is constructed.
2. The contents of the current storage are copied to the new storage.
3. The destructor is called on the current storage.
4. The new storage is swapped with the current storage.

This is done because the underlying memory allocation has to be contiguous for the default vector. With virtual memory-based allocations, this is not a necessity, as we can reserve a contiguous virtual address range and map the old allocations to it. Algorithm 2 shows how the reserve is implemented for a custom virtual memory-based vector. It can be observed that the difference between the current capacity and new capacity is calculated and this difference is

Algorithm 1 The function `reserve()` in the default vector

```
1: Input: $n$, new size to be reserved
2: Output: $storage$, an allocation of size $n$
3: if $n > capacity()$ then
4:     $new\_capacity \leftarrow n$
5:     construct $new\_storage(new\_capacity)$
6:     copy $storage$ to $new\_storage$
7:     $storage.destroy()$
8:     $storage \leftarrow new\_storage$
9: end if
```

allocated. At its core, as we saw in the previous section, a new physical memory is allocated for the difference, and a contiguous virtual address range is reserved and they are mapped.

Algorithm 2 The function `reserve()` in the custom virtual memory-based vector

```
1: Input: $n$, new size to be reserved
2: Output: $storage$, an allocation of size $n$
3: if $n > capacity()$ then
4:     $new\_capacity \leftarrow n$
5:     $size \leftarrow new\_capacity - capacity()$
6:     $storage.allocate(size)$
7: end if
```

Similar changes are made to functions like insert, append, etc. as well. A detailed look at these functions are available in Appendix A.2.

3.1.4 Interfacing with the Vector

Thrust provides the functionality to mention memory allocators along with the vector container when creating them. The Listing 3.2 shows how the vector container can be used along with the custom virtual memory-based allocator, `virtual_allocator`. The only difference with the usage of a custom allocator is when declaring the vector. We need to mention the required allocator along with the data type for the vector. The use of the vector is the same as using with the default allocator as shown in Section 2.3.1. So, it is simple to add the custom virtual memory-based system to any application on the GPU that depends on Thrust vectors.

```
    size_t vector_size = 10;
    // creates a vector using the virtual_allocator of given size and
     fills with value
```

```
 thrust::device_vector<int,thrust::virtual_allocator<int>>
vector_d(vector_size, 15);
 // reserves capacity of given size
 vector_d.reserve(2 * vector_size);
 // inserts the given number of elements at position and fills
with value
 vector_d.insert(vector_d.begin(), vector_size, 0);
```

Listing 3.2: Example usage of `device_vector` along with `virtual_allocator`

3.2 Dynamic Output Space Allocation for SpGEMM

The main idea behind the algorithm is to add more space to store the results as needed by the kernel as it progresses with the computation. The Algorithm 3 shows the implementation of the dynamic output space allocation for SpGEMM.

Algorithm 3 Dynamic Output Space Allocation for SpGEMM

```
 1: Input: A, input graph
 2: Input: num_of_vertices, number of vertices in the input graph
 3: Input: size_per_row, size to be allocated per row for a kernel iteration
 4: Input: active_row, vector to store if a row has completed computation
 5: Output: nz_vals, vector to store the non-zero values of the output matrix
 6: Output: col_ind, vector to store the column indices of the output matrix
 7: Output: row_off, vector to store the row offsets of the output matrix
 8: num_active_row ← num_of_vertices
 9: repeat
10:     size ← size_per_row x num_active_row
11:     nz_vals.resize(size)
12:     col_ind.resize(size)
13:     nz_vals_ptr ← nz_vals.data().get()
14:     col_ind_ptr ← col_ind.data().get()
15:     thrust::exclusive_scan(row_off.begin(), row_off.end(), size_per_row)
16:     spgemm_kernel(A, active_row, nz_vals_ptr, col_ind_ptr, row_off_ptr)
17:     num_active_row ← thrust::count(active_row.begin(), active_row.end(), 1)
18: until num_active_row == 0
```

To start, a given size, $size\_per\_row$, in terms of the number of elements in the output vector is allocated for each row of the output matrix and the kernel is launched. The threads perform the computation, filling the allocated space for each row, and exit when the space is exhausted. The threads also mark if the computation for the given row is completed or not using a vector, $active\_row$. After the first kernel call, the host computes the number of rows that have not completed computation due to insufficient memory and allocates more space for them. The kernel is then launched again. This is done in a loop until all the rows in the resultant matrix have been computed.

The output is stored using the Compressed Sparse Row (CSR) format. So, we require three vectors to store the non-zeros, the column indices and the row offsets, nz_vals, col_ind and *row_off*, of the output sparse matrix. We have to resize both the nz_vals and col_ind vectors before every kernel iteration. After resizing, we need to make sure to update the pointer to the vector as it may change, as we have seen from Section 3.1. To calculate the *row_off* vector of the output sparse matrix, we use the exclusive scan functionality provided with the library Thrust.

Chapter 4

Experimental Setup

This chapter provides a brief description about the experimental setup, the datasets used to evaluate the performance of the implementation, and the parameters used to compare the different algorithms.

4.1 Experiment

We compared the performance of the dynamic SpGEMM algorithm described in Section 3.2 against SpGEMM that uses the two different estimation methods, namely, the upper bound estimation method and the precise estimation method. The performance metrics we focus on are the end-to-end time elapsed for the algorithms and the memory efficiency of the estimation techniques.

For the dynamic SpGEMM algorithm, we test eight different sizes with which the output space grows for each kernel iteration. These sizes are denoted by $size\_per\_row$ in Section 3.2. The sizes used in the tests are 8, 16, 32, 48, 64, 96, 128 and 256 elements in the output vector for each row. We also compared the dynamic SpGEMM algorithm which uses the default non-virtual memory-based vector against one using the custom virtual memory-based vector. This is done to observe if the custom virtual memory-based vector provides any benefit with the growing allocations.

The tests were done on the NVIDIA GV100 GPU with multiple square matrices, which are introduced in Section 4.2. A test entails multiplying the square matrices with itself. For each test, 50 runs of the algorithm were performed. The first 25 runs are excluded to ensure a stable environment with respect to the cache, power, and temperature of the device. The last 25 runs were averaged to obtain the runtime. This was done because we observed that the variations in the runtimes reduced as the number of runs increased. The runtimes are measured using CUDA Events. The following chapter will tabulate and discuss the results in detail.

4.2 Datasets

SpGEMM is used to solve a variety of problems in areas such as chemistry, physics, engineering, machine learning and healthcare. To test the implemented algorithm, we chose datasets from these different domains. We chose graphs that are uniformly degree distributed because the SpGEMM implementation is not load balanced. This is sufficient to showcase the technique this work proposes, as we only focus on the memory allocation for the output matrix and not on the actual computation of the SpGEMM. The extension for this work using a load balanced SpGEMM is discussed in Chapter 6. The datasets were obtained from the publicly available SuiteSparse Matrix Collection [4], which hosts sparse matrices from different domains for the development and testing of sparse algorithms. The description and the size of the datasets used are given in Table 4.1.

Table 4.1: The description of the datasets

Dataset	Number of rows	Number of columns	Number of non-zeros	Description
atmosmodl	1,489,752	1,489,752	10,319,760	Atmospheric modeling
pwtk	217,918	217,918	11,524,432	Structural engineering
af_shell10	1,508,065	1,508,065	52,259,885	Sheet metal forming problem
cage14	1,505,785	1,505,785	27,130,349	DNA electrophoresis model
nv2	1,453,908	1,453,908	37,475,646	Semiconductor device simulation
Hook_1498	1,498,023	1,498,023	59,374,451	3D mechanical problem
BenElechi1	245,874	245,874	13,150,496	2D/3D problem
atmosmodm	1,489,752	1,489,752	10,319,760	Atmospheric modeling
Geo_1438	1,437,960	1,437,960	60,236,322	geomechanical problem

af_3_k101	503,625	503,625	17,550,675	Sheet metal forming problem
Hardesty1	938,905	938,905	12,143,314	Surface fitting problem
Serena	1,391,349	1,391,349	64,131,971	Structural problem
boneS10	914,898	914,898	40,878,708	Model reduction problem
Emilia_923	923,136	923,136	40,373,538	geomechanical problem
dgreen	1,200,611	1,200,611	26,606,169	Semiconductor device simulation
ss	1,652,680	1,652,680	34,753,577	Semiconductor process simulation
Fault_639	638,802	638,802	27,245,944	Structural problem
Transport	1,602,111	1,602,111	23,487,281	Structural problem
nlpkkt80	1,062,400	1,062,400	28,192,672	Optimization problem
cage15	5,154,859	5,154,859	99,199,551	DNA electrophoresis model

Chapter 5

Results

This chapter contains the results obtained by conducting the experimental setup described in Chapter 4. Section 5.1 shows the results obtained and draws a comparison between the different algorithms. It also provides insights into why and where using the dynamic SpGEMM algorithm and the custom virtual memory-based vector implemented by this work can be useful. Section 5.2 discusses interesting questions regarding the dynamic SpGEMM algorithm and Section 5.3 concludes the chapter with the limitations in the current implementation of the algorithm.

5.1 Performance of the SpGEMM Algorithms

In general, we observe from the results obtained that the dynamic SpGEMM algorithm provides memory efficiency of 70–99%, which is consistently better when compared to the SpGEMM algorithm that uses the upper bound method for estimation, with a modest performance cost. For 17 out of the 20 datasets, it also provides solutions with memory efficiencies between 70 and 99% for faster runtimes tested when compared to the SpGEMM algorithm that uses the precise method for estimation. Although we observed runtime being larger by $\sim$8% for the 3 datasets, there were other solutions on the graph that provided decent memory efficiency for faster runtimes.

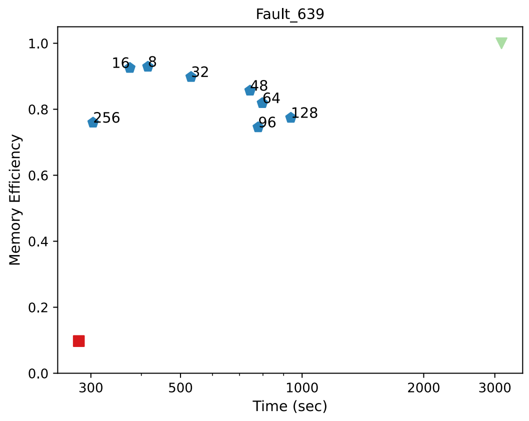
We constructed graphs for each of the datasets used for testing, with the time in seconds on the x-axis and the memory efficiency of the algorithms on the y-axis. The memory efficiency is computed by dividing the size of the ideal vector, which is the total number of non-zeros in the output matrix, by the estimated vector size for the different algorithms under test. In the graph, the optimal solution in terms of memory efficiency will be on the topmost part of the graph, and the optimal solution for faster runtimes will be on the leftmost part of the graph. A point in the top leftmost part of the graph would be an optimal solution for a good tradeoff between the two parameters.

Figure 5.1 shows the graphs for the 20 different datasets tabulated in Section 4.2. In the graphs, the red square represents the SpGEMM algorithm that uses the upper bound method for estimation, the green triangle represents the SpGEMM algorithm that uses the precise estimation method, and the blue pentagons are the dynamic SpGEMM algorithm implemented in this work for different sizes, as mentioned in Chapter 4.

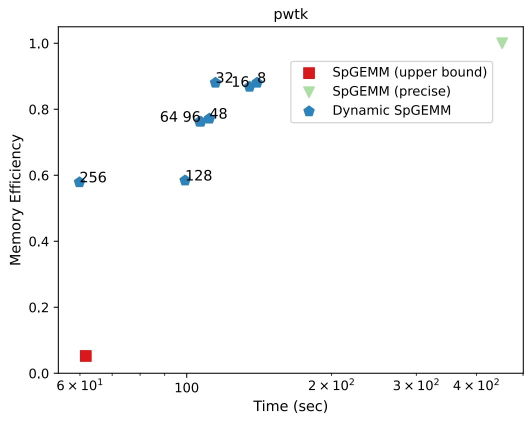
The smaller datasets like pwtk (5.1b), af_shell10 (5.1c), af_3_k101 (5.1g), atmosmodm (5.1h), Transport (5.1i), atmosmodl (5.1n), Hardesty1 (5.1p), nlpkkt80 (5.1q), ss (5.1s), and BenElechi1 (5.1t) have similar graphs. The dynamic SpGEMM algorithm performs best in terms of memory efficiency with the smallest *size_per_row* of 8. It also achieves higher memory efficiency than the SpGEMM that uses the upper bound estimation method with the same runtimes for optimal solutions for *size_per_row* values varying between 32 and 256 for the different datasets. For the datasets, atmosmodm and atmosmodl, the efficiency gets worse for larger values of *size_per_row* because of their high sparsity. Most of the rows in the resultant matrix has less than 32 elements leading to overallocation for *size_per_row* values greater than 32.

For the larger datasets like Fault_639 (5.1a), cage14 (5.1d), nv2 (5.1e), Hook_1498 (5.1f), boneS10 (5.1j), cage15 (5.1k), dgreen (5.1l), Emilia_923 (5.1m), Geo_1438 (5.1o), and Serena (5.1r), the smaller *size_per_row* values perform better than the larger *size_per_row* values in terms of memory efficiency and runtimes.

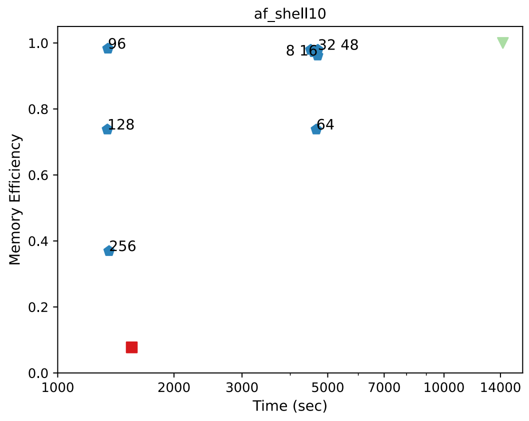
We can observe from the graphs that the SpGEMM algorithm that uses the precise method to estimate the size of the output space has the ideal memory efficiency, but is slower compared to the others due to the need to iterate through the input matrices twice. Table 5.1 shows the time taken for estimation and the total runtime for the SpGEMM algorithm with the precise estimation method. It can be observed that the estimation takes more than 50% of the total runtime with a geometric mean of 61.46% for the different datasets leading to longer runtimes.



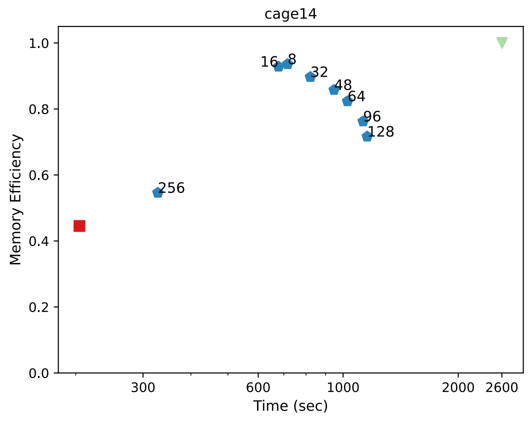
(a) Fault_639



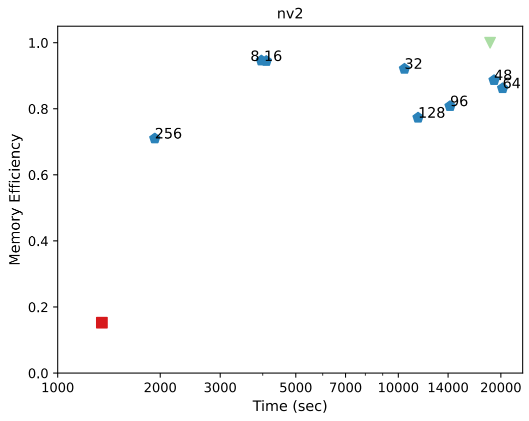
(b) pwtk



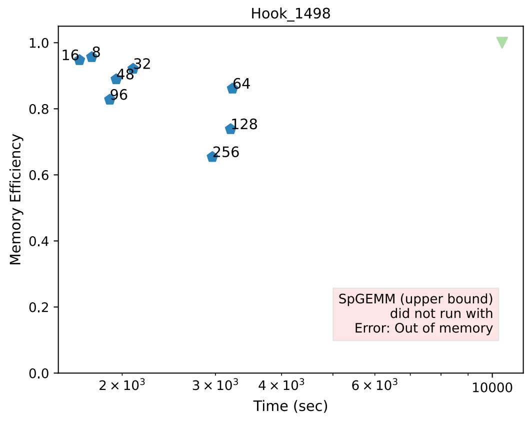
(c) af_shell10



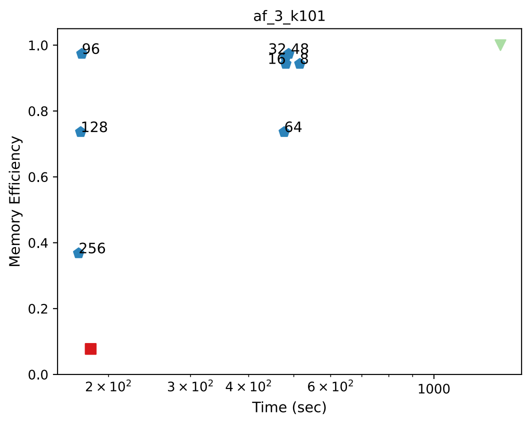
(d) cage14



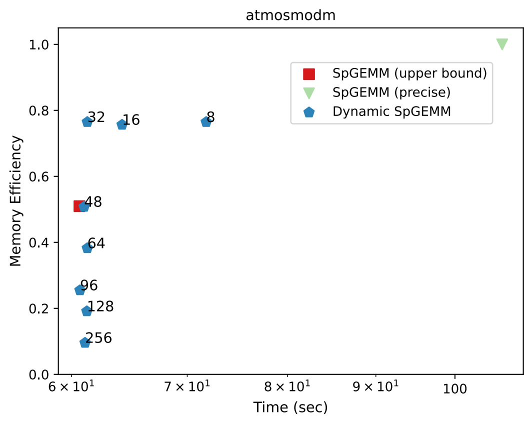
(e) nv2



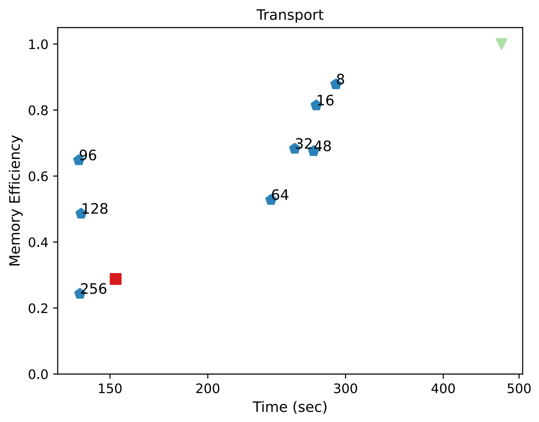
(f) Hook_1498



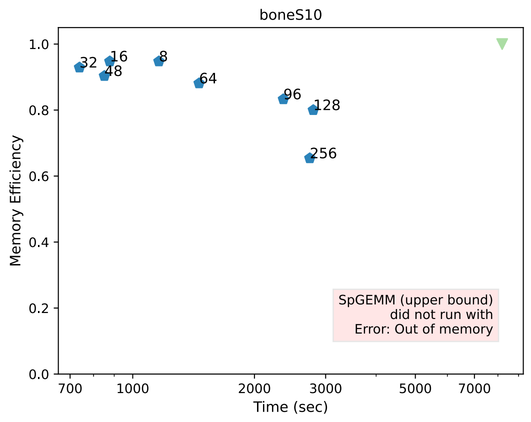
(g) af_3_k101



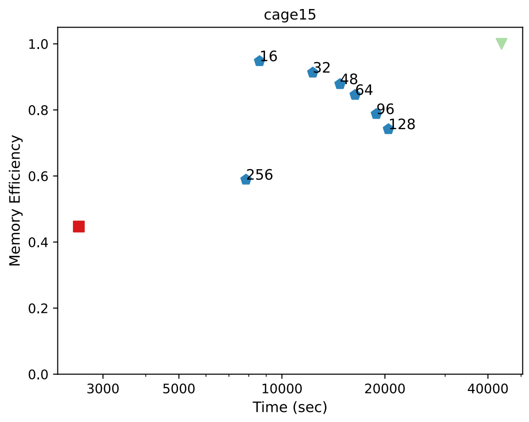
(h) atmosmodm



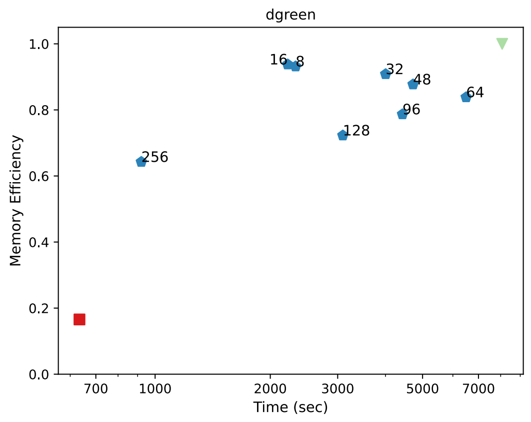
(i) Transport



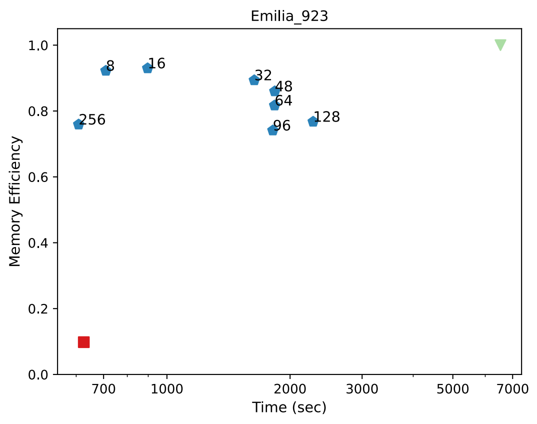
(j) boneS10



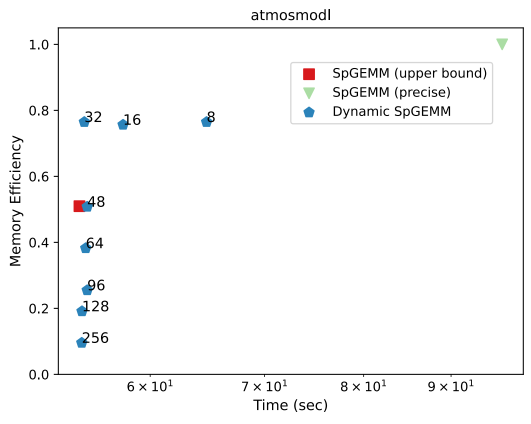
(k) cage15



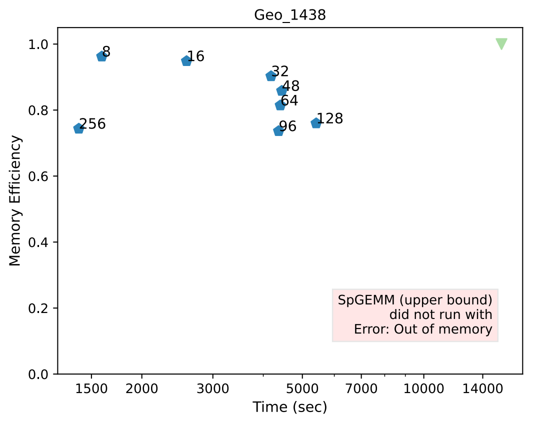
(l) dgreen



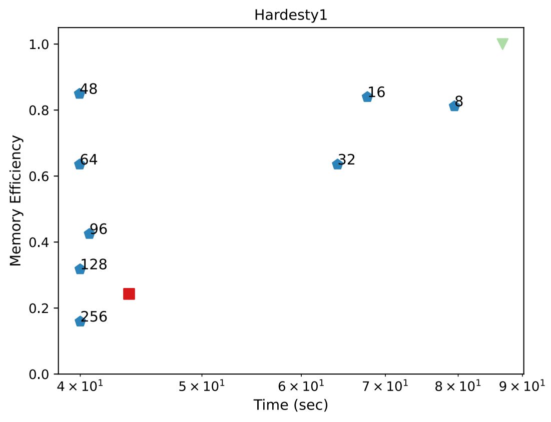
(m) Emilia_923



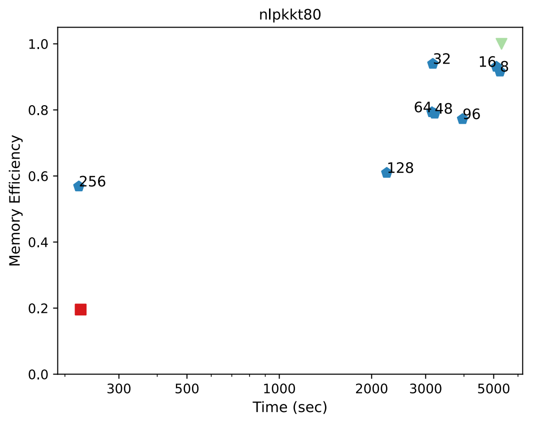
(n) atmosmodl



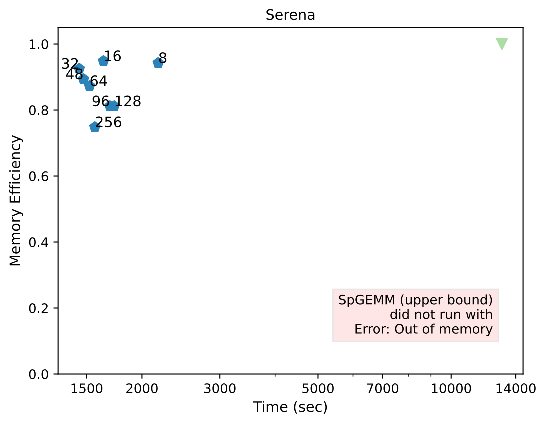
(o) Geo_1438



(p) Hardesty1



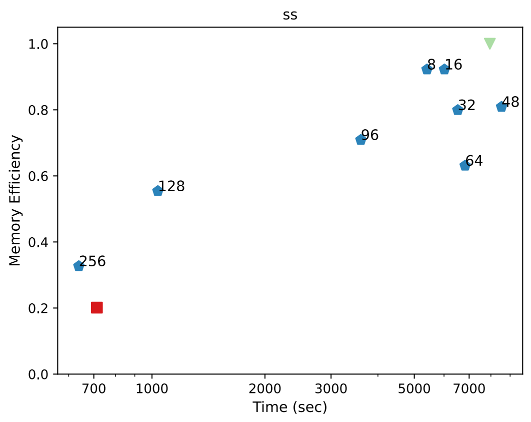
(q) nlpkkt80



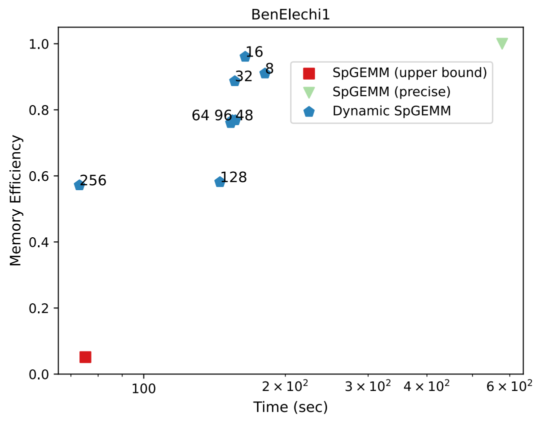
(r) Serena

Table 5.1: The estimation time compared to the total runtime for the SpGEMM algorithm using the precise estimation method for the datasets

Dataset	Estimation Time (ms)	Total Runtime (ms)	Percentage of Total Runtime
pwtk	225728	449094	50.26
atmosmodl	45445.2	96971.5	46.86
cage14	2413180	2600780	92.79
af_shell10	7333460	14171800	51.75
nv2	12387200	18745500	66.08
Hook_1498	8452150	10437300	80.98
af_3_k101	697222	1378340	50.58
atmosmodm	46424.2	97763.6	47.49
BenElechi1	306939	575953	53.29
boneS10	5131020	8181740	62.71
dgreen	6525400	8042480	81.14
Emilia_923	3658290	6549040	55.86
Fault_639	1830820	3130430	58.48
Geo_1438	8646480	15593000	55.45
Hardesty1	38607.7	76762	50.30
Serena	9043490	12956000	69.80
Transport	314604	451029	69.75
ss	5983500	7867570	76.05
nlpkkt80	3865260	5333560	72.47
Geometric Mean (Percentage of Total Runtime)			61.46



(s) ss



(t) BenElechi1

Figure 5.1: Memory efficiency and the runtime plotted for the algorithms for different datasets

The SpGEMM algorithm that uses the upper bound method for estimation is mostly the fastest, but has worse memory efficiency. Table 5.2 shows the overestimation factor for the upper bound method for the datasets. The upper bound method overestimates by an average factor of 7.018, which we determined by taking the geometric mean for the different datasets. This overestimation of the resultant matrix size leads to low memory efficiency and can be a significant problem in low memory scenarios. For example, for the datasets Hook_1498, boneS10, Geo_1438, and Serena, shown in Figures 5.1f, 5.1j, 5.1o, and 5.1r, it does not complete the run due to an out-of-memory error. From the Table 5.2, we can observe that the upper bound method overestimated the output matrix by a factor of 8.863, 16.741, 10.207, and 9.854, for the datasets Hook_1498, boneS10, Geo_1438, and Serena respectively, which led to the GPU running out of memory.

The dynamic SpGEMM takes the middle ground by providing solutions that have between 70 and 99% memory efficiency, with runtimes closer to the SpGEMM algorithm with the upper bound estimation. This shows that the dynamic SpGEMM algorithm is a good candidate for low memory cases.

5.2 Discussion

In this section, we discuss the answers to some interesting questions regarding the dynamic SpGEMM algorithm.

Table 5.2: Overestimation for the datasets for the upper bound estimation method.

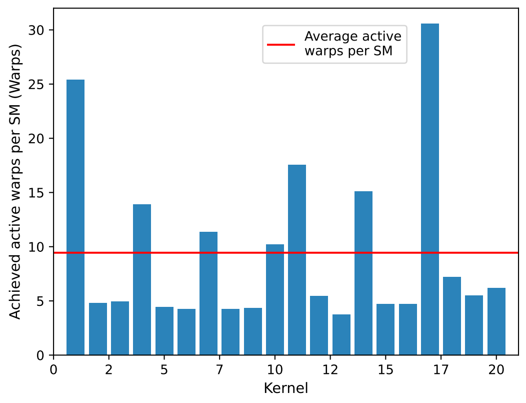
Dataset	Actual	Estimation	Overestimation Factor
pwtk	32,772,236	626,054,402	19.103
atmosmodl	36,486,008	71,590,768	1.962
cage14	236,999,813	532,205,737	2.246
af_shell10	142,742,975	1,840,916,875	12.897
nv2	310,543,360	2,034,624,478	6.552
Hook_1498	312,415,749	2,768,898,411	8.863
af_3_k101	47,472,025	612,413,375	12.901
atmosmodm	36,486,008	71,590,768	1.962
BenElechi1	36,261,344	705,555,870	19.458
boneS10	223,561,566	3,742,712,082	16.741
dgreen	228,101,325	1,376,588,233	6.035
Emilia_923	179,867,448	1,834,209,036	10.198
Fault_639	126,633,024	1,298,780,298	10.256
Geo_1438	274,478,976	2,801,728,386	10.207
Hardesty1	38,292,287	157,652,676	4.117
Serena	315,805,689	3,111,966,351	9.854
Transport	99,905,089	346,715,709	3.470
ss	138,289,810	687,203,895	4.969
nlpkkt80	154,663,144	790,384,704	5.110
Geometric Mean (Overestimation Factor)			7.018

Why do changes in *size_per_row* values cause performance changes? Is there a trend with larger/smaller *size_per_row*? A fixed number of elements, *size_per_row*, is added to the output row for every iteration of the kernel. This means that there is wasted space if the row does not completely utilize the space allocated. This is also why there is higher memory efficiency when the fixed size allocated for each kernel iteration is smaller, as this provides finer-grained control over memory usage. Therefore, we see that the run with a larger size allocated per kernel iteration leads to lower memory efficiency.

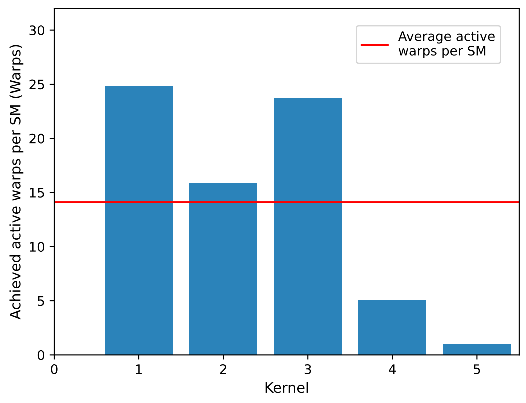
For runtime, there is no straightforward trend that we can observe. For smaller datasets, the larger values of *size_per_row* lead to faster runtimes, while smaller values of *size_per_row* are slower in comparison. From profiler analysis for the kernels, we understand that the kernels are latency bound from the following insight from NVIDIA Nsight Compute. "This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues." This can also be seen in Figures 5.2a and 5.2b, which show the achieved active warps per SM for the kernel calls with the dataset pwtk. When *size_per_row* is 64, the average active warps per SM is more than when *size_per_row* is 8. More active warps per SM allows for better latency hiding, leading to lower total runtime.

However, for larger datasets, the reverse can be observed, with smaller *size_per_row* values providing better runtimes compared to the larger *size_per_row* values. This could be due to better latency hiding by the threads in the larger datasets. The threads that have finished computation do not have to wait as long for other threads that are still active, because only smaller amounts of computation are done per kernel launch. These kernels are also observed to be latency bound from the profiler as seen earlier. Figures 5.2c and 5.2d showcase this behavior for the dataset Hook_1498. For the value of 16 for *size_per_row*, the achieved active warps per SM is consistently large for most of the kernel calls, leading to a smaller total runtime due to good latency hiding. For the latter case, there is low achieved active warps per SM for the kernel calls, therefore it has a worse runtime.

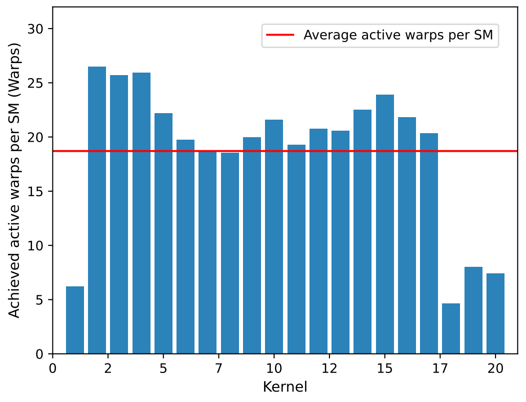
If you had no other information at all, and didn't get to exhaustively search, how can you pick the right *size_per_row* value for the matrix? This is an interesting research problem,



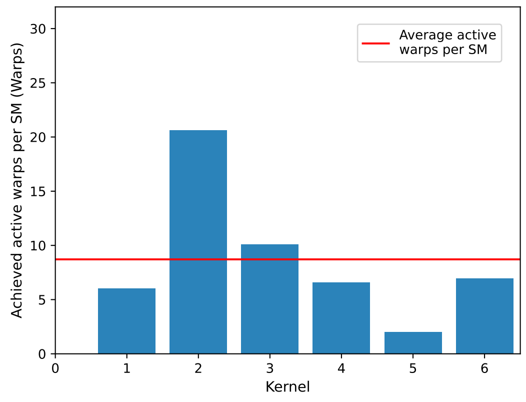
(a) pwtk with *size_per_row* = 8



(b) pwtk with *size_per_row* = 64



(c) Hook_1498 with *size_per_row* = 16



(d) Hook_1498 with *size_per_row* = 64

Figure 5.2: Achieved active warps per SM observed for dynamic SpGEMM kernels for a large dataset, Hook_1498, and a small dataset, pwtk, for different *size_per_row* values

and some ideas to tackle it are provided in Section 6.2. Based on the observations we have made through the results of this work, the following suggestions can be made:

- If memory efficiency is what you are optimizing for, then you should be choosing smaller *size_per_row* values. The smaller *size_per_row* value provides finer-grained control over the memory allocation.

- On the other hand, if you are optimizing for runtime, then depending on the size of the dataset, you have to choose either the smaller or the larger *size_per_row* value. For larger datasets, we observed the smaller *size_per_row* values to be faster, and for small datasets, the larger *size_per_row* values are faster.

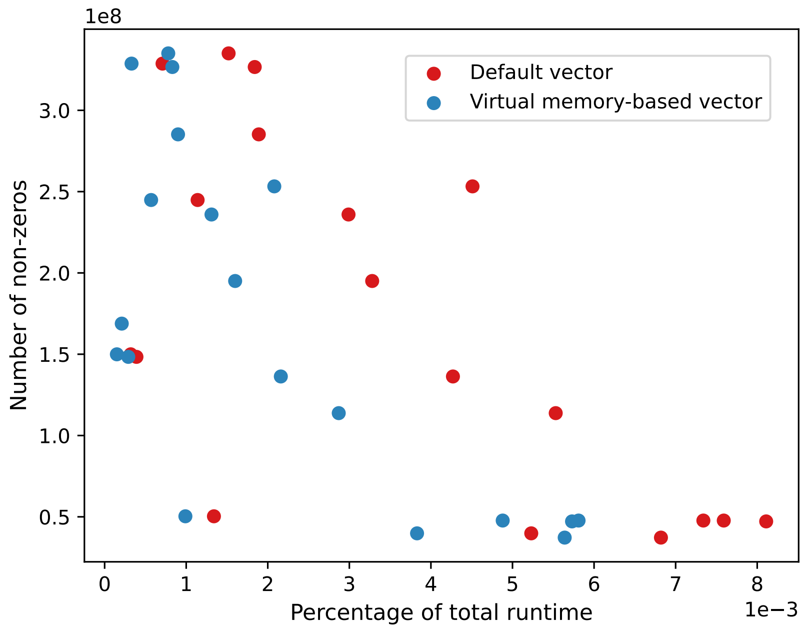


Figure 5.3: Resize time for default vector and virtual memory-based vector with dynamic SpGEMM as a function of the number of non-zeros in the output matrix

- If you are looking for a trade-off between the two parameters, then a *size_per_row* value is hard to predict. In general, for larger datasets, a smaller *size_per_row* value would be a good candidate, but for smaller datasets, it is difficult to conclude on a *size_per_row* value.

Why is it advantageous to use the custom virtual memory-based vector to store the resultant vector in the dynamic SpGEMM algorithm? The dynamic SpGEMM algorithm needs to resize the output vector between the kernel calls due to the need for more memory. We learned earlier the disadvantages of using the default non-virtual memory-based vector for growing allocations. To reiterate, it requires extra space to resize and adds to latency from the `cudaMemcpy()` calls to copy the contents of the vector from the old allocation to the new allocation. Therefore, we use the custom virtual memory-based vector for the dynamic SpGEMM algorithm. Figure 5.3 shows the percentage of time taken to resize as a function of the number of non-zeros in the output matrix. Here, resize time includes the time taken for memory management like allocation, copying memory from the old allocation to the new one and deallocation. We can observe that the custom virtual memory-based vector takes smaller fraction of the total runtime than the default vector.

5.3 Limitations

The current implementation of the dynamic SpGEMM algorithm does not work well for skewed matrices because it depends on an SpGEMM algorithm that is not load balanced. The memory efficiency for this case is good, but it takes longer time than the SpGEMM algorithms that use the precise and upper bound methods for estimation. Some engineering can be done to make this work with a load balanced SpGEMM implementation, which will be discussed in the future work section in Chapter 6.

Another issue with the current implementation is the use of kernel launch as a way to synchronize threads across the device. When we explored persistent kernels to achieve this, we ran into issues with running the dynamic SpGEMM kernel and the memory allocation at the same time. This is because we used the Thrust library for memory allocation, which requires cub kernels to allocate and initialize the memory. These kernels were blocked when enough resources were not available, due to them being used by the dynamic SpGEMM kernel, which are waiting for the memory allocation to complete before continuing its computation. Secondly, CUDA provides cooperative groups, which can be used to synchronize across a grid using grid groups. We did not explore this option.

The third issue, which is an interesting research challenge, is identifying the right value for *size_per_row* based on the needs of the user. For larger matrices that take longer times for SpGEMM computation, it is not ideal to iterate over different values of *size_per_row* to identify the optimal solution. Some ideas to tackle this are mentioned in the future work found in Chapter 6.

Chapter 6

Conclusion and Future Work

This chapter will provide concluding remarks in Section 6.1 for the custom virtual memory-based allocator and the dynamic output space allocation algorithm for SpGEMM. We will also include potential improvements and interesting ideas to continue on the work done in this thesis in the Section 6.2.

6.1 Conclusion

In conclusion, we have presented an algorithm to dynamically determine the output space for SpGEMM and a custom virtual memory-based vector that can used alongside applications that use Thrust vectors. The dynamic SpGEMM algorithm was evaluated by performing experiments on a collection of sparse matrices from the SuiteSparse Matrix Collection.

From the experiments, we can conclude that the dynamic SpGEMM algorithm can be used in low memory applications with runtimes that are close to the SpGEMM algorithm using the upper bound estimation method. Even though the SpGEMM algorithm with precise estimation method provides us with the ideal memory efficiency, its runtimes are slower than the dynamic SpGEMM.

Based on the graphs and the insights provided, the user can choose the optimal $size\_per\_row$ value as per their requirements to provide the desired tradeoff between runtime and memory efficiency. We can conclude that the smallest $size\_per\_row$ value provides us with the solution that is optimal in terms of memory efficiency. For smaller datasets, a larger $size\_per\_row$ value provides faster runtimes, but for larger datasets, a smaller $size\_per\_row$ value provides the faster runtimes.

The custom virtual memory-based vector provides a memory efficient and faster way to resize growing vector allocations. Due to its integration to a header-only library like Thrust, it makes it very simple to port this system to any application. Therefore, this custom virtual memory-based system can be used in any algorithm that requires growing allocations.

6.2 Future Work

The integration of the custom virtual memory-based system with the library Thrust must be done efficiently. The current implementation replicates classes to do partial template specialization due to complexities associated with using templates. This was done to prototype and speed up the development of the system for testing. An efficient way to redefine only a few selected functions must be determined and implemented.

The custom virtual memory-based vector implemented in this work can be used for other graph algorithms that use growing allocations as well. Some examples of such algorithms include depth-first search, breadth-first search, and single-source shortest path. The impact of using the custom virtual memory-based allocation on the runtimes and memory utilization efficiency can be studied.

The dynamic output space allocation algorithm for SpGEMM currently uses a fixed value for $size\_per\_row$ because optimizing for different parameters leads to different values for $size\_per\_row$. An algorithm or formula to identify an ideal $size\_per\_row$ value for the requirements of the user would be a useful addition to the system. Cohen [2] proposes a probabilistic method to predict the size of the resultant matrix using the Monte Carlo algorithm for sparse graph problems, which can be an interesting way to approach the problem.

The dynamic output space allocation algorithm as mentioned earlier does not work with a load balanced SpGEMM implementation. To be able to work with a load balanced implementation, there needs to be some facility for the threads that work on the same row of the output matrix to communicate and store the results efficiently. This is an interesting but necessary research that needs to done to further improve the algorithm.

The current implementation of the dynamic SpGEMM algorithm uses kernel launch for synchronization across the device. However, CUDA provides grid groups, which can be used to synchronize across the grid. This functionality can be used to avoid the multiple kernel launches, and the performance of the algorithm needs to be observed.

REFERENCES

[1] Nathan Bell and Jared Hoberock. Chapter 26 - Thrust: A productivity-oriented library for CUDA. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 359–371. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-385963-1. doi: 10.1016/B978-0-12-385963-1.00026-5.

[2] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997. doi: 10.1006/jcss.1997.1534.

[3] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. URL `http://cusplibrary.github.io/`. Version 0.5.0.

[4] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1), December 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663.

[5] James J. Elliott and Christopher M. Siefert. Low thread-count Gustavson: A multithreaded algorithm for sparse matrix-matrix multiplication using perfect hashing. In *2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA)*, pages 57–64, 2018. doi: 10.1109/ScalA.2018.00011.

[6] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. A systematic survey of general sparse matrix-matrix multiplication. *ACM Computing Surveys*, 55(12), 2023. doi: 10.1145/3571157.

[7] Intel. Intel Math Kernel Library. URL `https://software.intel.com/en-us/mkl`. Accessed: 05.15.2023.

[8] NVIDIA. NVIDIA CUDA Sparse library. URL `https://developer.nvidia.com/cusparse`. Accessed: 05.15.2023.

[9] Muhammad Osama, Serban D. Porumbescu, and John D. Owens. Essentials of parallel graph analytics. In *Proceedings of the Workshop on Graphs, Architectures, Programming, and Learning*, GrAPL 2022, pages 314–317, May 2022. doi: 10.1109/IPDPSW55747.2022.00061.

[10] Cory Perry and Nikolay Sakharnykh. Introducing low-level GPU virtual memory management. Technical report, NVIDIA, 2020. URL `https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/`.

[11] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing*, 4(1):3:1–3:49, August 2017. doi: 10.1145/3108140.

Appendix A

Implementation

A.1 Custom Virtual Memory-Based Memory Allocator

We implemented a custom memory allocator based on GPU virtual memory management called `thrust::mr::virtual_memory_resource_allocator` on the Thrust library shown in Listing A.1. It is a class publicly derived from the class `thrust::mr::allocator`. The class, `thrust::mr::allocator`, fulfills the C++ requirements for memory allocators.

This class has a constructor, copy constructor, and member functions for reserving, growing and de-allocating memory. It also holds state of the memory allocator, such as the vector of virtual address ranges, allocation handles and handle sizes, pointer and size of allocation, properties of the mapping, etc. as shown on lines 8-22 in Listing A.1.

```
1     class virtual_memory_resource_allocator : public thrust::mr::
     allocator
2     {
3         // properties of the allocation
4         CUmemAllocationProp prop;
5         CUmemAccessDesc accessDesc;
6
7         // holds the state of the allocation
8         struct Range {
9             CUdeviceptr start;
10            size_t sz;
11        };
12        // vector of virtual address ranges
13        std::vector<Range> va_ranges;
14        // vectors of CUDA memory handles and its sizes
15        std::vector<CUmemGenericAllocationHandle> handles;
16        std::vector<size_t> handle_sizes;
```

```
        // CUDA device pointer to the start of the allocation
        CUdeviceptr d_p;
        // sizes of minimum granularity, allocation and total
    reservation
        size_t chunk_sz;
        size_t alloc_sz;
        size_t reserve_sz;

        public:
            // constructor
            virtual_memory_resource_allocator();
            // allocates memory of size n
            pointer allocate(size_t n);
            // reserves virtual address range of size new_size
            CUresult reserve(size_t new_size);
            // de-allocates memory of size n at address pointed by p
            void deallocate(pointer p, size_t n);
            // destructor
            ~virtual_memory_resource_allocator();
    }
```

Listing A.1: Overview of the class `virtual_memory_resource_allocator`

We look into the different parts of the class `virtual_memory_resource_allocator` in the following subsections.

A.1.1 Construction

The setup required for using the virtual memory management driver APIs is performed in the constructor call.

Firstly, the CUDA driver APIs have to be initialized for every process created by calling `cuInit()`. Each physical memory chunk allocation has properties and a handle associated with it, which is stored in the structs `CUmemAllocationProp` and `CUmemGenericAllocationHandle` respectively.

The struct `CUmemAllocationProp` holds fields for the type and location of the memory allocation. We use pinned memory in the device, so `CU_MEM_ALLOCATION_TYPE_PINNED` is given as input for the allocation type. Here, pinned memory means that the allocation cannot migrate from the current location while the application is actively using it. The struct `CUmemLocation` is used to hold the location and the device ID of the allocation. `CU_MEM_LOCATION_TYPE_DEVICE` denotes that the location of the memory allocation is a device and the ID is its ordinal.

To set access rights to the memory allocation, we use the struct `CUmemAccessDesc`. The first field describes the location of the memory allocation, which is the same as the location field in `CUmemAllocationProp`. The second field is used to specify the memory protection flags for the allocation. We assign `CU_MEM_ACCESS_FLAGS_PROT_READWRITE` for the flag as we would like to be able to read from and write to the memory allocated. `CU_MEM_ACCESS_FLAGS_PROT_READ` should be used if the address range only has to be read accessible.

Finally, we need to obtain the minimum granularity with which we can allocate memory. The function `cuMemGetAllocationGranularity()` returns the minimal granularity when provided with the properties of the allocation for which we need the granularity. The multiples of the granularity returned can be used for alignment and size of the memory allocation. Putting it all together, the Listing A.2 shows the constructor implemented.

```
 virtual_memory_resource_allocator
     ::virtual_memory_resource_allocator()
 {
     // to initialize the CUDA driver APIs
     cuInit(0);
     // assign properties of the allocation
     prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
     prop.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
     int device_id;
     cudaGetDevice(&device_id);
     prop.location.id = device_id;
     // assign access rights for the allocation
     accessDesc.location = prop.location;
     accessDesc.flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
     // returns minimum granularity for the given allocation
properties
     cuMemGetAllocationGranularity(&chunk_sz, &prop,
CU_MEM_ALLOC_GRANULARITY_MINIMUM);
 }
```

Listing A.2: Constructor for the class `virtual_memory_resource_allocator`

A.1.2 Allocation

As mentioned earlier, allocation happens in multiple steps involving multiple driver APIs. We implemented an `allocate()` function depicted by Listing A.3 which receives the number of elements, n, for allocation. Then, the size of memory required is calculated. To request a virtual address range, we need to first pad the size to align with the minimum granularity, $chunk\_sz$ which is done in line 6. A function, `reserve()`, is used to reserve the required virtual address range.

```
pointer virtual_memory_resource_allocator
    ::allocate(size_t n)
{
    CUmemGenericAllocationHandle handle;
    // pad size to multiple of minimum granularity
    size_t sz = ((n * sizeof(T) + chunk_sz - 1) / chunk_sz) * chunk_sz;
    // reserves virtual address range for given size
    CUresult status = reserve(alloc_sz + sz);

    if (status == CUDA_SUCCESS) {
        // creates physical memory handle
        if ((status = cuMemCreate(&handle, sz, &prop_new, 0ULL)) == CUDA_SUCCESS) {
            // maps physical handle to virtual address range
            if ((status = cuMemMap(d_p + alloc_sz, sz, 0ULL, handle, 0ULL)) == CUDA_SUCCESS) {
                if ((status = cuMemSetAccess(d_p + alloc_sz, sz, &accessDesc_new, 1ULL)) == CUDA_SUCCESS) {
                    // store state
                    handles.push_back(handle);
                    handle_sizes.push_back(sz);
                    // extend the size
                    alloc_sz += sz;
                }
                // if unsuccessful, then un-map
                if (status != CUDA_SUCCESS) {
                    (void)cuMemUnmap(d_p + alloc_sz, sz);
                }
                // release the physical handles
                if (status != CUDA_SUCCESS) {
                    (void)cuMemRelease(handle);
                }
            }
        }
    }
    return (pointer((void *)d_p));
}
```

Listing A.3: The function `allocate()` for the class `virtual_memory_resource_allocator`

Assuming the virtual address range reservation was a success, we then allocate the physical memory using `cuMemCreate()`. It creates and returns a CUDA memory handle denoted by `CUmemGenericAllocationHandle`, which represents a physical memory allocation with the requested properties and size. Then, the physical memory allocation has to be mapped to the virtual address range, which is taken care of by `cuMemMap()`. This driver function maps the CUDA allocation handle to the reserved virtual address range, $d\_p + alloc\_sz$. d_p is the pointer to the start of the address range and $alloc\_sz$ is the total size of the allocation. Finally, `cuMemSetAccess()` sets access flags for each location specified by the struct `CUmemAccessDesc` passed to it. If the allocation is successful, the state of the allocation is updated. $handles$ and $handle\_sizes$ are vectors that house the memory handles and sizes of the physical memory allocations. This is used to remap allocations in the case of when virtual address range reservation changes.

Now, let us look closer at what happens during virtual address reservation. The `reserve()` function we implemented is passed the total size, new_sz, of the virtual address range required, as observed in Listing A.4. This is padded for alignment with the minimum granularity to get the aligned size, $aligned\_sz$. `cuMemAddressReserve()` is the driver API used to make the address range reservation.

```
CUresult virtual_memory_resource_allocator
    ::reserve(size_t new_sz)
{
    CUresult status = CUDA_SUCCESS;
    // pad size to multiple of minimum granularity
    size_t aligned_sz = ((new_sz + chunk_sz - 1) / chunk_sz) *
chunk_sz;
    // try to reserve new size at the end of current virtual
address range
    status = cuMemAddressReserve(&new_ptr, (aligned_sz -
reserve_sz), 0ULL, d_p + reserve_sz, 0ULL);
    // if unsuccessful, we take the longer route
    if (status != CUDA_SUCCESS || (new_ptr != d_p + reserve_sz))
{
        // free if virtual address range was allocated at random
start address
        if (new_ptr != 0ULL)
            status = cuMemAddressFree(new_ptr, (aligned_sz -
reserve_sz));
        // try to reserve for total allocation size
        status = cuMemAddressReserve(&new_ptr, aligned_sz, 0ULL,
0U, 0);

        if (status == CUDA_SUCCESS && d_p != 0ULL) {
            CUdeviceptr ptr = new_ptr;
            // un-map old allocation
            status = cuMemUnmap(d_p, alloc_sz);
            // map the new virtual address range to old physical
handles and set access rights
            for (size_t i = 0ULL; i < handles.size(); i++) {
                    const size_t hdl_sz = handle_sizes[i];
                    if ((status = cuMemMap(ptr, hdl_sz, 0ULL, handles[i], 0ULL)) != CUDA_SUCCESS)
                        break;
                    if ((status = cuMemSetAccess(ptr, hdl_sz, &accessDesc_new, 1ULL)) != CUDA_SUCCESS)
                        break;
                    ptr += hdl_sz;
                }
                // un-map and throw error if unsuccessful
                if (status != CUDA_SUCCESS) {
                    status = cuMemUnmap(new_ptr, aligned_sz);
                    assert(status == CUDA_SUCCESS);
                    status = cuMemAddressFree(new_ptr, aligned_sz);
                    assert(status == CUDA_SUCCESS);
                }
                // if successful, we free the old virtual address range reservations
                else {
                    for (size_t i = 0ULL; i < va_ranges.size(); i++) {
                        status = cuMemAddressFree(va_ranges[i].start, va_ranges[i].sz);
                    }
                    va_ranges.clear();
                }
            }
            // store state of the allocation - update pointer, size, etc.
            if (status == CUDA_SUCCESS) {
                Range r;
                d_p = new_ptr;
                reserve_sz = aligned_sz;
                r.start = new_ptr;
                r.sz = aligned_sz;
                va_ranges.push_back(r);
            }
        }
        // store state of the allocation - update pointer, size, etc.
        else {
            Range r;
            r.start = new_ptr;
            r.sz = aligned_sz - reserve_sz;
            va_ranges.push_back(r);
            if (d_p == 0ULL) {
                d_p = new_ptr;
            }
            reserve_sz = aligned_sz;
        }
        return status;
    }
```

Listing A.4: The function `reserve()` for the class `virtual_memory_resource_allocator`

It takes the size of reserved virtual address range requested and a fixed starting address for the requested reservation, and returns a pointer to the start of the virtual address range allocated if successful. The easiest way to increase the size of the address range is to add the required size range to the end of the current address. So, an attempt to reserve at starting address $d\_p + reserve\_sz$ of size $aligned\_sz - reserve\_sz$ is made as shown by the line 8. Here, $reserve\_sz$ is the total size of the current address range. If the reservation was a success, then we can store the state of the allocation, which is the pointer to the start of the address range and the size of the handles. This is done in the code by the vector va_ranges in the lines 57-64.

If reserving at the end of the current virtual address range is a failure, then we have to free any reservations done and start over. Now, a virtual address range for the total aligned size, $aligned\_sz$, is requested. If this is a success, then we have to un-map the previous mappings to the old virtual address range and map the physical memory handles to the new virtual address range. This is implemented in the lines 10-29. The access rights are also set for the new address range. Finally, the old virtual address reservation is freed and the current state of the allocation is updated to the vector va_ranges. We will go into more detail of the driver functions used to un-map and free reservations in the next subsection.

A.1.3 De-allocation

We implemented a `deallocate()` function, given in Listing A.5, that receives the pointer and the size of memory to be de-allocated. Similar to the procedure for allocation, we need to execute multiple steps to un-map and free the physical and virtual address reservation.

Firstly, we un-map the backing memory of the given virtual address range using the driver function `cuMemUnmap()`. Then, we free the virtual address range reservations stored by the state variable va_ranges using `cuMemAddressFree()`. Lastly, we release the CUDA physical memory handles kept in track with the state variable $handles$. The state variables are also cleared and reset for future new allocations.

```
void virtual_memory_resource_allocator
    ::deallocate(pointer p, size_t n)
{
    CUresult status = CUDA_SUCCESS;
    if (d_p != 0ULL) {
        // un-map the allocations
```

```
            status = cuMemUnmap((CUdeviceptr) p.get(), n * sizeof(T))
    ;
            assert(status == CUDA_SUCCESS);
            // if successful, free the virtual address range
    reservations
            for (size_t i = 0ULL; i < va_ranges.size(); i++) {
                status = cuMemAddressFree(va_ranges[i].start,
    va_ranges[i].sz);
                assert(status == CUDA_SUCCESS);
            }
            // release the CUDA physical handles
            for (size_t i = 0ULL; i < handles.size(); i++) {
                status = cuMemRelease(handles[i]);
                assert(status == CUDA_SUCCESS);
                }
            // reset state
            va_ranges.clear();
            handles.clear();
            handle_sizes.clear();
        }
    }
```
Listing A.5: The function `deallocate()` for the class `virtual_memory_resource_allocator`

A.2 Using the Custom Virtual Memory-based Allocator with Vectors

The functionalities of `std::vector` are implemented for the GPU on the Thrust library in the class `thrust::detail::vector_base`. This class contains functions to create and manipulate the vector, like copy, reserve, resize, insert, fill, erase, etc. It also has the pointer, reference, iterators and properties of the vector. The methods used to manipulate the vector are written for a `cudaMalloc`-based allocation, which is the default. Hence, some changes have to be made to such methods in order to observe the benefits of using the custom virtual memory-based allocator. Since some of the methods have to be changed, we specialized the class `thrust::detail::vector_base` for `thrust::virtual_allocator<T>` to redefine the functions according to our needs. Let us look at some of the functions that had to be changed in the following subsections.

A.2.1 Reserve

Reserve is one of the most important functions in the vector container. It is used to reserve memory for a given number of elements. The Listing A.6 showcases the default implementation of this function. Let us look at a gist of what happens when reserve is called:

If the new size is greater than the current capacity of the vector, then a new storage with the new size is created. Then, the contents of the vector is copied from the old storage to the newly created storage. Then, the old storage is destroyed and the new storage is added to the state of the vector.

```
template<typename T, typename Alloc>
void vector_base<T,Alloc>
    ::reserve(size_type n)
{
    if(n > capacity()) {
        // compute the new capacity after the allocation
        size_type new_capacity = n;

        // create new storage
        storage_type new_storage(copy_allocator_t(), m_storage,
new_capacity);
        iterator new_end = new_storage.begin();

        // construct copy all elements into the newly allocated
storage
        new_end = m_storage.uninitialized_copy(begin(), end(),
new_storage.begin());

        // call destructors on the elements in the old storage
        m_storage.destroy(begin(), end());

        // record the vector's new state
        m_storage.swap(new_storage);
    }
}
```

Listing A.6: The function `reserve()` for default allocators

There are a few inefficiencies with the way `reserve()` is done in the default implementation in Listing A.6. To grow the allocation, both the old and new storage must be in memory. This significantly reduces by how much we can reserve more memory. For example, if the GPU has 4 GB of memory and you need to grow a 2 GB vector to 3 GB, it is not possible. This is because there is no space to allocate a new larger allocation of size 3 GB along with old 2 GB allocation. Therefore, in low memory scenarios, this implementation can fail to run the application. Secondly, there is a copy called to transfer the data of the vector from the old vector to the new storage. This adds to the latency of the application. It also occupies the bandwidth when it replicates data, which is very wasteful for high performance applications.

Using `virtual_allocator` resolves the inefficiencies mentioned above. Listing A.7 shows the implementation for the function `reserve()` when specialized for `virtual_allocator`. From the listing, we can observe that the creation of a new larger storage to grow is eliminated. Instead, we allocate the difference between the new required capacity and the current capacity. In the backend, the `virtual_allocator` allocates the required physical memory and virtual address range as discussed in the previous section. Then, it maps the physical memory handle to the virtual address range and updates the pointer to the allocation in case of a change.

```
template<typename T>
void vector_base<T,thrust::virtual_allocator<T>>
    ::reserve(size_type n)
{
    if(n > capacity()) {
    // compute the new capacity after the allocation
    size_type new_capacity = n;

    // find size difference to allocate
    const std::size_t sz = new_capacity - capacity();

    // allocate takes the number of elements to allocate as input
    m_storage.allocate(sz);
    }
}
```

Listing A.7: The function `reserve()` specialized for `virtual_allocator`

A.2.2 Append

Similar to the function `reserve()` mentioned in the previous subsection, `append()` is another important and frequently used function in the vector container. Listing A.8 shows the implementation for `reseve()` for default allocators, and the Listing A.9 shows the definition of the same specialized for the `virtual_allocator` to yield the benefits of using virtual memory.

```
template<typename T, typename Alloc>
void vector_base<T,Alloc>
    ::append(size_type n)
{
    if(n != 0) {
        if(capacity() - size() >= n) {
            // we've got room for all of them
            // add to end and increase size
            // ...
        }
11         else {
12             const size_type old_size = size();
13 
14             // compute the new capacity after the allocation
15             size_type new_capacity = old_size + thrust::max(
   old_size, n);
16             new_capacity = thrust::max(new_capacity, 2 * capacity
   ());
17 
18             // create new storage
19             storage_type new_storage(copy_allocator_t(),
   m_storage, new_capacity);
20             iterator new_end = new_storage.begin();
21 
22             // construct copy all elements into the newly
   allocated storage
23             new_end = m_storage.uninitialized_copy(begin(), end()
   , new_storage.begin());
24 
25             // construct new elements to insert
26             new_storage.default_construct_n(new_end, n);
27             new_end += n;
28 
29             // call destructors on the elements in the old
   storage
30             m_storage.destroy(begin(), end());
31 
32             // record the vector's new state
33             m_storage.swap(new_storage);
34             m_size = old_size + n;
35         }
36     }
37 }
```

Listing A.8: The function `append()` for default allocators

In general, when `append()` is called, if the current capacity cannot hold the new elements to append, then it creates a new large storage. This is followed by a copy of the elements from the old storage to the new, and adding the elements to append to the end of the new storage. Finally, the old storage is destroyed and the new state of the vector is recorded. This is shown in lines 19-34 in the Listing A.8.

Similar to Listing A.7, when specializing for `virtual_allocator`, the function `append()` only allocates the necessary storage and appends as normal. In addition to the performance benefits of using virtual memory, we can also observe that this simplifies the algorithm for vector manipulation as seen in Listing A.9 when compared to the default implementation in

Listing A.8.

```
template<typename T>
void vector_base<T,thrust::virtual_allocator<T>>
    ::append(size_type n)
{
    if(n != 0) {
        if(capacity() - size() >= n) {
            // we've got room for all of them
            // add to end and increase size
            // ...
        }
        else {
            const size_type old_size = size();

            // compute the new capacity after the allocation
            size_type new_capacity = old_size + thrust::max(
old_size, n);
            new_capacity = thrust::max(new_capacity, 2 * capacity
());

            // find size difference to allocate
            const std::size_t sz = new_capacity - capacity();

            // allocate takes the number of elements to allocate
as input
            m_storage.allocate(sz);

            append(n);
        }
    }
}
```

Listing A.9: The function `append()` specialized for `virtual_allocator`

A.2.3 Insert

The `insert()` is another useful functionality provided by vectors. It can insert elements at a given position in the vector. There are a couple of different helper functions to support the insert operation. One of them is `fill_insert()`. We will take a closer look at it in this subsection.

`fill_insert()` inserts n number of elements to the index given by *position* and fills it with the value x. If the capacity of the vector is sufficient for the insertion of the n elements, then the functionality is the same for both the default and the custom virtual memory-based allocator.

When there is a need to increase the capacity of the vector, the default implementation as mentioned previously creates a new large storage and copies the elements from the old storage to the new one in addition to inserting the elements. This is depicted in the Listing A.10.

```
template<typename T, typename Alloc>
void vector_base<T,Alloc>
    ::fill_insert(iterator position, size_type n, const T &x)
{
    if(n != 0) {
        if(capacity() - size() >= n) {
            // we've got room for all of them
            // insert the elements
            // ...
        }
        else {
            const size_type old_size = size();

            // compute the new capacity after the allocation
            size_type new_capacity = old_size + thrust::max(old_size, n);
            new_capacity = thrust::max(new_capacity, 2 * capacity());

            storage_type new_storage(copy_allocator_t(), m_storage, new_capacity);
            iterator new_end = new_storage.begin();

            // construct copy elements before the insertion to the beginning of the newly allocated storage
            new_end = m_storage.uninitialized_copy(begin(), position, new_storage.begin());

            // construct new elements to insert
            m_storage.uninitialized_fill_n(new_end, n, x);
            new_end += n;

            // construct copy displaced elements from the old storage to the new storage
            new_end = m_storage.uninitialized_copy(position, end(), new_end);

            // call destructors on the elements in the old storage
            m_storage.destroy(begin(), end());

            // record the vector's new state
            m_storage.swap(new_storage);
            m_size = old_size + n;
        }
    }
}
```

Listing A.10: The function `fill_insert()` for default allocators

When the same function is specialized for the `virtual_allocator`, we need to be cautious of the change in the start pointer that could happen if a contiguous virtual address range is not available at the end of the current reservation. Hence, we make sure to obtain the index of the insertion before `allocate()` is called as shown in line 22. We can compute the new position, $position\_new$, using the index and the pointer pointing to the start of the vector as implemented in line 29. This $position\_new$ is used to call `fill_insert()` on the vector now that there is enough capacity required for insertion of the elements. This can be seen in the Listing A.11.

```
1     template<typename T>
2     void vector_base<T,thrust::virtual_allocator<T>>
3         ::fill_insert(iterator position, size_type n, const T &x)
4     {
5         if(n != 0) {
6             if(capacity() - size() >= n) {
7                 // we've got room for all of them
8                 // insert the elements
9                 // ...
10            }
11            else {
12                const size_type old_size = size();
13
14                // compute the new capacity after the allocation
15                size_type new_capacity = old_size + thrust::max(
      old_size, n);
16                new_capacity = thrust::max(new_capacity, 2 * capacity
      ());
17
18                // find size difference to allocate
19                const std::size_t sz = new_capacity - capacity();
20
21                // get index of the insertion
22                size_type index = thrust::distance(begin(), position)
      ;
23
24                // allocate takes the number of elements to allocate
      as input
25                m_storage.allocate(sz);
26
27                // update position in case the begin pointer changes
      during allocation
28                iterator position_new = begin();
29                thrust::advance(position_new, index);
30
31                fill_insert(position_new, n, x);
32            }
33        }
34    }
```

Listing A.11: The function `fill_insert()` specialized for `virtual_allocator`