Towards Just-in-Time Instruction Generation for Accelerated Sparse Matrix-Matrix Multiplication on GPUs

Seth David Kay
GraphLab
George Washington University
sdkay@gwu.edu

H. Howie Huang
GraphLab
George Washington University
howie@gwu.edu

***Abstract*—Sparse Matrix-Matrix Multiplication (SpMM) is a fundamental operation in neural networks and high performance graph algorithms. Most popular solutions to SpMM adhere to ahead-of-time (AOT) compilation, where the entire program is compiled before execution. With the introduction of GPU threading in SpMM there have been significant leaps in performance. However, AOT compilation for threaded SpMM has four major limitations: unnecessary memory access, greater branch overhead, unnecessary retained memory, and slow memory transfers between the host and the device. These limitations are due to the nature of AOT compilations where key information is unaccessible during the compilation time but later becomes available during the programs run-time. In this paper, we propose a SIMD GPU threaded just-in-time (JIT) compilation solution to SpMM, improving SpMM resource usage and run time on GPUs. The most significant advantage of our framework is that JIT compilation provides our solution with runtime information. This information is used to dynamically create precise thread structures based on input matrices. Operations are evenly distributed to each thread in their smallest denominations to achieve maximum efficiency and a balanced workload distribution. Runtime information is also used to manage a self-maintained memory pool which increases memory efficiency, bandwidth, and SpMM speed. Improved forms of memory allocation and storage are used, decreasing the amount of transfers needed between the device and host, as well as increasing GPU memory bandwidth and decreasing memory access times. We conduct a performance evaluation of our framework and compare it to a AOT baseline provided by NVIDIA, a threaded matrix multiplier using the CUDA framework, adjusted for SpMM using an array in compressed sparse row (CSR) format. Our JIT framework consistently delivers a significant performance enhancement over the native CUDA solution, achieving on average, a 1.3x improvement in execution time and a 3x improvement in GPU memory footprint.**

***Index Terms*—SpMM, Just-in-Time Compilation, GPU Threading, Performance Optimization**

I. Introduction

Over recent years sparse data has seen greater and greater prevalence. As a result, sparse computation - algorithms that leverage the understanding of operating with sparse data to minimize the number of fundamental operations performed - is receiving large amounts of attention, in hopes to improve preexisting algorithms using the knowledge that the operational data will be sparse [1]–[5], [7]. SpMM (Sparse Matrix-Matrix Multiplication), is no exception to this trend. SpMM is a form of matrix multiplication that is performed on one sparse matrix, A (a matrix in which most elements are zero) [1], and one dense matrix, X (a matrix in which most elements are non-zero) [1]. Each index j each row i in the sparse matrix A is multiplied by index j in the corresponding column i in the dense matrix, and products of the multiplications are then added to form one index in the resulting dense matrix Y, stored at Y[i][j]. SpMM has found extensive applications, being used in matrix factorization algorithms, graph clustering algorithms, and is a key component in Graph Neural Networks, being used in graph convolution [7]–[9].

SpMM frameworks popular today are limited by design challenges. These limitations include memory transfer between the host and the device, unnecessary branch overhead, limiting memory storage techniques, and unnecessary retained device memory. Our framework aims to solve these limitations. To do so we use the following approaches. First, we use just-in-time (JIT) compilation to gain access to valuable runtime information which allows us to perfectly balance thread structures and deallocate memory when needed, extending our prior work on multicores [11] to GPUs. Second, we employ CSR format for sparse matrices to improve memory and runtime efficiency. Third, we utilize a highly efficient self-maintained memory pool using improved forms of memory allocations. Using the framework described above, our SpMM solution - using both JIT compilation and GPU threading - addresses the limitations of popular SpMM solutions today. By addressing these limitations, our JIT threaded SpMM exhibits a 1.3x improvement in execution time and a 3x improvement in GPU memory efficiency, meaning problems 3x larger are able to be processed without exhausting GPU memory.

The rest of the paper is organized as follows. Section 2 provides a background on technologies used in our SpMM solution. Section 3 discusses current iterations of SpMM limitations and the opportunities provides by these limitations for our framework to improve on SpMM. In section 4, we present our framework's design and optimization techniques. Section 5 describes our findings and discusses their implications and potential origins. Finally, section 6 concludes the paper.

TABLE I
LIST OF NOTATIONS.

Symbol	Description
A	Sparse Matrix of Dimensions $m \times n$
X	Input Dense Matrix of Dimensions $n \times d$
Y	Output Dense Matrix of Dimensions $m \times d$
m	Number of Rows of Matrix A
n	Number of Columns of Matrix A, Number of Rows of Matrix X
d	Number of Columns of Matrix X
$M[i]$	The i-th Row of a Matrix M
$M[i][j]$	The Element at the i-th Row and j-th Column of a Matrix M
$n_{i,j}$	An Element n, which is at Row i and Column j

II. Background and Related Work

To increase the performance of SpMM operations and decrease the amount of memory used to store sparse matrices, many sparse matrix storage techniques have been designed. Matrices are commonly represented as two-dimensional arrays in software. Each element, $n_{i,j}$, is accessed using two indices - i for the row and j for the column. Rows are numbered from top to bottom, while columns are numbered from left to right. This means, for an $m \times n$ matrix, the memory required to store this two-dimensional array representation of a matrix scaled linearly with the product of its dimensions ($m \times n$), excluding the storage needed for the matrix's dimensions themselves. Table I, a table of frequently used notations throughout this paper, is provided for the convenience of the reader.

To improve memory usage, the Compressed sparse row (CSR) formatting of sparse matrices, also known as compressed row storage (CRS) or Yale format, and many formats like it, where created. CSR format is represented as follows (zero-basing is used). An $m \times n$ sparse matrix A is represented not by using one two-dimensional array but by using three one-dimensional arrays: a values array (V), an array to hold column indices (col_index), and an array to hold the number of non-zero elements above each row (row_index). CSR format compresses row indices, only storing nonzero values. The values array, V, holds all nonzero elements from the matrix, and is the same length as the number of nonzero elements. The column indices array, col_index, at it's i'th element contains the column location of the value at V's i'th element. col_index also has length equal to the number of nonzero elements in the sparse matrix. The row_index array has a length of m + 1, and stores at index i the total amount of non-zero elements that occur above row i. This results in the final element of row_index being equal to the total number of nonzero values in the sparse matrix A. Figure 1 presents an example of the process. CSR formatted sparse matrices drastically reduce the amount of memory needed to store the sparse matrix.

When performing SpMM with a sparse matrix A in CSR format and a dense matrix X, we can greatly reduce the amount of individual multiplications necessary during execution by doing SpMM using a CSR-formatted sparse array. The algorithm performs SpMM on a sparse matrix A in CSR format and a dense matrix X. First, the row_index is accessed to locate the initial position of row i. Segments of col_index and V are traversed to perform the necessary multiplications on all non-zero entries in the sparse row i. For each nonzero entry, the algorithm utilizes the column index from col_index to find the corresponding row in the dense matrix X. The value at X is then multiplied by the nonzero value at A, and the product is added to the row-column total, yielding the final value stored at $Y[A$ col$][X$ row$]$.

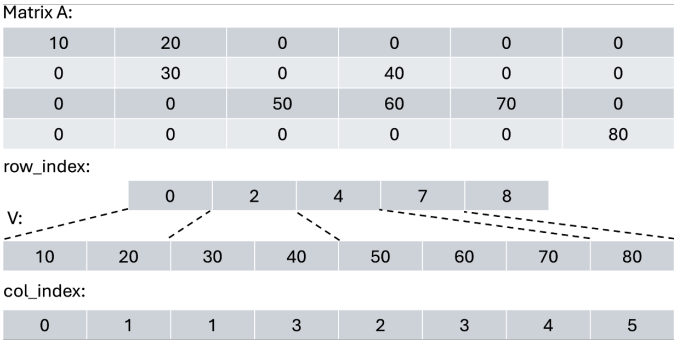

Fig. 1. Example of CSR format (bottom) and the matrix representation (top).

Multiplying a sparse matrix in CSR format by a dense matrix using GPU threading is shown in Algorithm 1. First, we get the position of the thread as a vector, where the first position is the position within the thread block and the second position is the position of the thread block in relation to the overarching thread grid. Then for each thread, the return value is initialized, and the thread iterates over the row_index array of the sparse matrix from the thread's position to the next thread's position ($row\_index[posx]$ to $row\_index[posx+1]$). Then the result of all nonzero multiplications in the thread's row are added to the result, which is then inserted into the

Algorithm 1 A threaded implementation of SpMM based on CSR

Require: Sparse matrix A of size $m \times n$, i.e., A.row_index, A.col_index, A.V, dense matrix X of size $n \times d$, and return matrix Y of size $m \times d$.
Ensure: Dense matrix $Y = AX$.
  $thread\_posx, thread\_posy = cuda.grid(2)$ ▷ Get thread positions using cuda framework
  if $thread\_posx < n$ **and** $thread\_posy < d$ **then** ▷ Make sure the thread is needed
    $ret = 0$
    for $idx = A$.row_ptr$[i]$ to A.row_ptr$[i+1]$ **do**
      $k = A$.col_indices$[idx]$
      $ret+ = A$.vals$[idx] \times X[k][thread_posy]$
    end for
    $Y[thread\_posx][thread\_posy] = ret$
  end if
  return Y

correct position in Y (the dense array that is a result of the SpMM). When doing this for every thread in a specialized thread and thread block format for the specific input arrays, every position in Y is filled in by one thread. Done simultaneously, the longest the algorithm can run for is the time taken for the largest amount of multiplications for a single row-column pair to be done, added, and inserted into Y.

III. Motivations and Opportunities

As SpMM has become more prevalent, ways to improve runtime have been proposed. A popular method is to massively thread the operation using GPUs [13]. To achieve this threading, we use NVIDIA's API framework, CUDA. NVIDIA provides an extensive API for threading using their GPUs, allowing for thread structures to be created, memory pool manipulation, and optimized data transferred between the host and the device. It also allows for the creation of unified and shared memory pools, increasing memory efficiency and transfer speeds. The use of GPUs allows for individual multiplication operations to be executed at once, allowing for a significantly faster SpMM.

Leveraging NVIDIA's CUDA framework for GPU threading can provide powerful efficient parallelization, and is utilized in our JIT threaded approach to SpMM. A core aspect of CUDA's framework is their extremely customizable thread hierarchy. This hierarchy consists of three denominations of threading: threads, blocks, and grids. Threads are used in SIMD architecture, each one performing one iteration of a given GPU kernel method. Programs typically base thread execution on the specific thread architecture used. Threads can be grouped into blocks, which subsequently can be grouped into grids. Precise thread architecture is decided at runtime, which allows for dynamic thread structure allocation that corresponds to a GPU kernel method's specific requirements. This allows for only threads that are necessary for computation to be deployed for execution, which decreases memory and GPU bandwidth consumption. Each thread is given its own registers and local memory, whereas thread blocks in CUDA can cooperate by sharing data through shared memory pools, as well as synchronize their execution. Grids are a collection of thread blocks that do not share memory but are useful in terms of simplifying the code needed by adding another layer of thread grouping often used to deploy individual threads to each index in three-dimensional arrays.

The CUDA framework also provides many ways to store memory, and understanding what memory type to use and where is pivotal for optimizing performance. There are three main types of memory CUDA offers. First, global memory, which is accessible by all threads in every thread block and grid. Global memory is not cached, leading to latency in reading and writing. Second, shared memory, which is accessible only by all threads in a specific thread block. Our solution typically uses shared memory, as it offers synchronization, automatic data locking, and quick access. Third, there is local memory, which is specific to each thread. Because it stored in the same way global memory is, it is prone to the same latency.

Our SpMM solution only utilizes shared memory when storing data on the GPU. This is because shared memory latency is roughly 100x lower than other memory formats able to be used for NVIDIA GPU computation, and when a bottleneck of the algorithm stems from reading and writing to arrays, it becomes very important to improve [16]. Shared memory also allows for data cashes in a custom memory pool, which our solution uses. Shared memory can also be read-only, further reducing the runtime of reading data from input arrays when performing SpMM. Using shared memory in our framework allows for both faster speeds at runtime and also increased memory bandwidth on the GPU.

Popular threaded and non-threaded solutions to SpMM adhere to AOT (ahead-of-time) compilation. This involves the entire program being compiled before execution. AOT compilation for SpMM has limitations in the form unnecessary memory access and greater branch overhead. To address these limitations, we propose the use of JIT (just-in-time) compilation for threaded SpMM. JIT compilation compiles GPU kernel functions into machine code at runtime, allowing runtime information to be available for optimizations. Using information known at the program's runtime, we are able to address each of these limitations in turn and greatly improve the speed of SpMM.

NVIDIA does not provide a JIT framework with the CUDA API, so to perform JIT compilation using NVIDIA GPUs, Numba, a Python JIT framework for generating machine code from Python code using the LLVM compiler infrastructure [14]. Numba provides decorators for Python functions to allow them to be compiled to machine code at runtime. Numba integrates with the CUDA framework almost seamlessly, which allows for the same benefits of the CUDA framework, such as pinned and shared memory pools, while also allowing for the access of runtime information.

Current solutions to SpMM have many limitations. Our solution is able to alleviate them by using the strategies and technologies described above. First, CSR format allows us to reduce the time taken for SpMM to run, as well as greatly reduce memory consumption and data transfer speeds. Second, by threading SpMM on the GPU we are able to improve SpMM speeds. Third, by using JIT compilation we gain access to important runtime information. Fourth, using this runtime information, our solution is able to precisely allocation thread structures perfectly suited for GPU execution. And fifth, the use of initially writing to pinned memory and by utilizing only shared memory on the GPU, we are able to decrease the speed of memory transfers as well as reduce memory usage on the GPU. Using the technologies described above, as well as highly specific techniques such as loop unrolling, memory coalescing, and minimized memory access latency, our proposed solution achieves less memory usage and faster runtime.

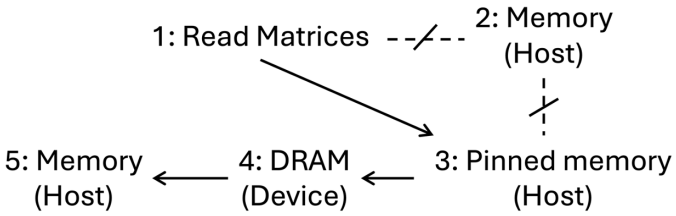

Fig. 2. How our solution uses pinned memory directly during host to device data transfers. Writing data directly into pinned memory when read circumvents the need to transfer a host array to pinned memory before copying memory the device.

Algorithm 2 An implementation of thread block allocations

Require: Dense matrix X size $n \times d$, and return $blocks\_per\_grid$, a description of thread and thread block count and shape.

Ensure: $blocks\_per\_grid$ = thread block allocations.

$threads\_per\_block = (16, 16)$

$blocks\_per\_grid\_x = math.ceil(n/threadsperblock[0])$

$blocks\_per\_grid\_y = math.ceil(d/threadsperblock[1])$

$blocks\_per\_grid = (blockspergrid\_x, blockspergrid\_y)$

return $blocks\_per\_grid$

IV. Framework

The implementation of the above frameworks and solutions culminates in our proposed solution. Here, we will discuss how each technology is used in our solution to improve on previous iterations of SpMM. The first technology to discuss is our threading model. Due to the nature of global and local memory and the latency they produce, our solution uses only shared memory structures. This meant that thread grids were impractical, and they were not necessary or useful for our solution from the beginning, so our solution implements threads only as part of thread blocks. This allows for easily shared memory that can be accessed quickly, and it allows for all of the threads to interact with one array simultaneously due to the automatic locking and synchronization capabilities of the CUDA shared array framework. This also allowed for simple calculations to determine how many threads were needed in each block, as well as how many blocks were needed in total, given the dimensions of the matrices we are multiplying. This allows for the perfect amount of threads and thread blocks to be allocated for our specific GPU method, see Algorithm 2.

The decision to use shared memory was a valuable one. While it did allow for threads to easily access data in the same array, it also improved data read, write, and even transfer speeds, because it is able to be cached. Shared memory has significant improvements on alternative types of GPU memory for SpMM implementations, as it is much faster at accessing memory, able to be allocated from the custom memory pool our solution employs, is able to be accessed safely by every thread in a grid at once, and can be made to function as a read-only array, even further reducing memory usage for input arrays into the SpMM.

Another less commonly used but important form of memory is pinned memory. As shown in Figure 2, host data allocations are pageable by default when normally allocating memory. Because of this, the GPU cannot access data directly, so the CUDA driver first must allocated a page-locked, or pinned, host array, which can then be copied to device memory. This causes slower data transfer speeds. To circumvent this bottleneck in data transfer between the host and the device, we utilize directly allocating data into pinned memory, allowing for the transfer to host data structures to improve greatly [15].

To further optimize performance and memory usage, our solution implements a self-regulated memory pool. Numba's memory pool increased calls to the CUDA API for memory retention while executing on the GPU, and made it difficult to individualize the storage of each array. By managing memory allocation and deallocation dynamically, memory is able to be freed when it has been used, reducing memory usage and improving runtime performance. This also allowed us to write data into pinned arrays when initially reading them, reducing the amount of data transfers done. See 2. This initial pinned memory is stored on the host. While this is necessary for initializing the SpMM, the CPU memory becomes a burden for CUDA's framework to upkeep while executing on the GPU. Having the ability to remove this pinned memory as soon as it is transferred to device memory improves the runtime and memory usage of our solution. A dynamic memory pool also allows for the allocation of read-only arrays for input arrays. This improves memory access speeds and increases memory bandwidth using caching mechanisms that are more effective for read-only data. Because our solution is able to dynamically allocate to a memory pool it controls, it allows for greater memory bandwidth and faster runtime speeds when executing on the GPU.

JIT compilation is also a critical part of our solution. JIT offers many benefits to SpMM, as it allows for increased performance overall for SpMM as well as flexibility during runtime. By compiling CUDA GPU kernel code at runtime, JIT compilation allows for optimizations suited for specific hardware, which Numba employs. This alone results in faster execution times for SpMM when compared to an AOT algorithm. JIT compilation also allows our solution to dynamically adapt to specific inputs, allowing for easily thread block allocations, which optimizes the algorithm for specific matrix structures during runtime. Because thread allocation, memory usage, and runtime is highly dependent on the nature of the data on GPU's, having access to and using runtime information through a JIT framework allows for more dynamic, fast, and memory efficient SpMM. Choosing to use a JIT compilation framework in our proposed solution has enhanced the efficiency and scalability the SpMM, as well as allowing us to understand and process unique data by dynamically allocating thread structures during runtime based on the input data.

The use of CSR format in our solution was important. When working with matrices of a known specific type, using storage and computational methods uniquely made for that matrix can

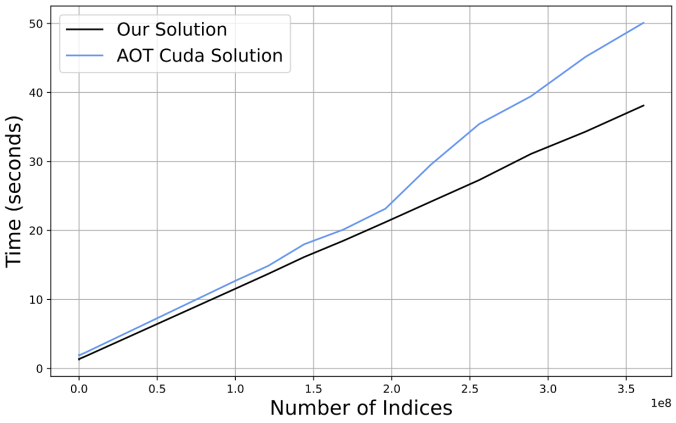

Fig. 3. Time vs. array total number of elements of the resulting array ($m \times d$) in 100 millions, comparing our solution and the AOT CUDA solution.

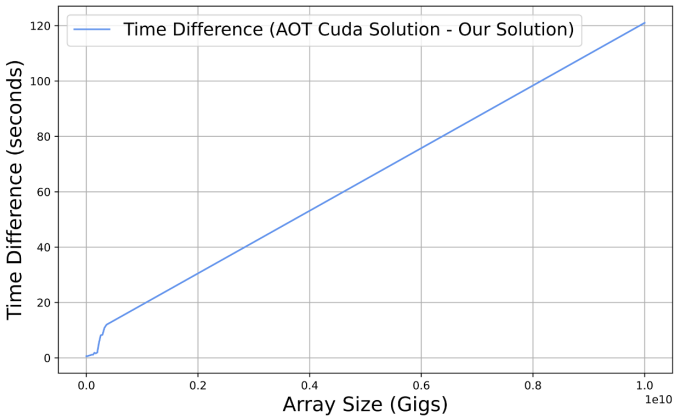

Fig. 4. Time difference (AOT Cuda Solution - Our Solution) vs. array storage size in gigabytes.

improve an algorithm significantly. CSR format for SpMM is no exception. Since we know an input matrix is sparse, and because the sparse matrix is stored in CSR format, accessing the arrays and doing the few multiplications necessary for one thread is highly efficient, which also leads to a drastically lower runtime than a basic SpMM would produce. Using CSR format not only allows us to rewrite our SpMM function to be much faster, it also drastically reduces the amount of memory used to store the sparse matrix. This in turn decreases the amount of time taken to transfer read the data as well as dramatically decreasing the transferring of data between the device and host.

To optimize our solution we employed smaller scale specific optimized techniques to the general computation. First, we use loop unrolling, as shown in Algorithm 1, to thread the SpMM and achieve a balanced thread workload. Second, we use memory coalescing to optimize memory access patterns and GPU bandwidth usage during GPU kernel execution. Third, we minimize memory copying latency by only returning the array resulting from the SpMM back to host memory and removing the input arrays from both device and host memory as soon as soon as they are no longer needed. These techniques maximize the utility of the frameworks that our solution uses, and they achieve significant performance improvements when scaling up the SpMM.

V. Evaluation

The hardware used to test our implementation of SpMM were as follows: GPU type - Tesla V100-SXM2, GPU count - 4, driver version - 515.65.01, cuda version - 11.7, Numba version - 0.59.1. All tests run were done utilizing these systems to their full capacities. When creating data sets, two two-dimensional arrays are created using a python script of type int32. The dense matrix is generated randomly using Python's random number generation framework, and the sparse matrix is generated similarly, the only difference being that is generated with a sparsity of 0.00001, or .001 percent. This sparsity of matrices is only used for the largest generated, when generating smaller matrices for testing a sparsity of 0.001, or 0.1 percent, is used. Numpy is then used to house the sparse matrix, so it can then be separated into its CSR components using the Scipy framework. The sparse and dense arrays are then written to files, the sparse array being written as three lines, one line for each of its components, V, col_index, and row_index. The dense array is stored with one line being one row in the matrix, and spaces separate the indices. There is also a file in which the resulting array dimensions are written into, to avoid unnecessary iterations over the input matrices to gather dimension data. When running each solution, the files are first read into pinned memory to gather the correct input data.

The AOT CUDA SpMM baseline algorithm run was created by NVIDIA and listed in the CUDA documentation. In order to identify the effect that JIT compilation was having on the algorithm, the AOT implementation was adjusted to fit SpMM best. First, memory is read from files holding the data directly into pinned memory, allowing faster allocation from host to device memory when calling the GPU kernel. Shared memory is used to improve latency in reading and writing arrays and simplify thread memory access. Threads structure allocations are the same as our solution utilizes, see Algorithm 2, meaning only thread blocks are used to organize threads. CSR format is used when processing and iterating over the sparse matrix, improving memory usage and memory transfer speeds. Our CSR specific algorithm, see Algorithm 1, is used when executing the SpMM in the GPU kernel to improve memory usage and runtime speeds. Only the resulting array, Y, is copied from device memory back to host memory. These changes ensure that the baseline algorithm is as fast as possible given its constraints.

Running on the above architecture, our solution to SpMM outperformed its AOT official CUDA counterpart in every tested evaluation. First, we testing the proposed solution against the AOT baseline for time taken to run. Here we see an average difference in runtime of 34 percent. In other words, our algorithm performed on average 1.3 times faster than our baseline. In figure 3, shown is the time taken for our solution

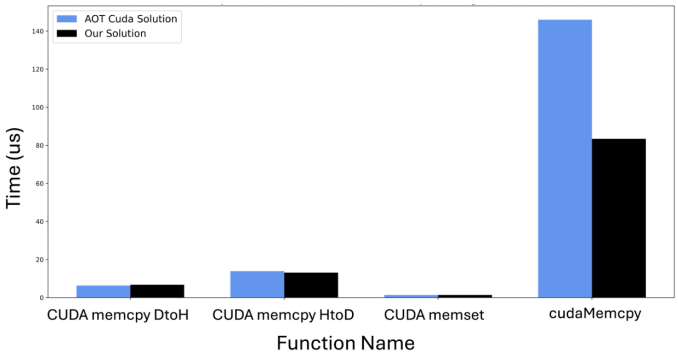

Fig. 5. API calls recorded on a small SpMM example for both our solution and the AOT CUDA baseline.

and the AOT CUDA baseline. This shows how our SpMM runs faster at every point than its counterpart.

Scalability is a critical factor for modern SpMM solutions to focus on. Shown in 4 is a larger graph showing much larger arrays than 3, up to the largest arrays the SpMM could run on with the GPU memory available, and depicts the difference in runtime between the two solutions. As is clearly shown, as the matrices get larger, our solution consistently performs better than the AOT CUDA baseline at an increasing rate. This graph further emphasizes the runtime improvements our SpMM solution offers, especially when scaling up matrix sizes.

After observing inconsistencies between the GPU memory usage of the two algorithms, we also ran GPU memory usage tests. We found that the percent difference in memory able to be stored on the GPU in our solution compared to the AOT CUDA baseline was, on average, a 203 percent difference. In other words, our algorithm was able to store 3 times as much memory on the GPU as its counterpart, allowing our solution to handle problems that are 3 times are larger than the baseline. This is due to the dynamic self-hosted memory pool our solution utilizes, which is able to deallocate memory using runtime information. It also allows for our program to circumvent unnecessary memory expenditure by allowing us to create read-only arrays on device memory, as well as remove any excess memory usage caused by CUDA API calls.

Finally, we ran performance evaluations on CUDA API calls for both of the algorithms using CUDA's nvprof [17]. Figure 5 shows comparisons of the largest api calls that both algorithms ran. The data was filtered to pertain just to memory allocation, retention, and copying. Data was collected by running randomly sized SpMM calculations and averaging the nvprof results. The graph shows that memory copying between the host and device takes close to the same amount of time between the algorithms, but the copying of memory from pinned memory to shared memory (cudaMemcpy) is improved in our solution.

VI. Conclusion

Sparse Matrix-Matrix Multiplication is an important algorithm and is growing more so rapidly. Increased attention and research into increasing memory usage run time is important. In this paper, we proposed a JIT compiled and GPU threaded SpMM algorithm that improved on the cutting-edge implementation of SpMM from NVIDIA. Our solution improved on time taken to run by 34 percent. It also improved memory consumption by 102 percent. This clearly shows that despite most popular SpMM algorithms adhere to ahead-of-time compilation, the information and control over thread allocation and memory given using the runtime information provided only by JIT compilation reduces the limitations of AOT compilation for SpMM. Our solution decreases memory consumption, memory access time, time taken to run, and memory copying between the host and the device.

For future work, we plan to explore a number of research directions, including studying data partitioning methods to scale to multiple GPUs [18], [19], leveraging tensor cores [20], as well as investigating I/O support for semi-external memory based SpMM algorithms [21], [22].

Acknowledgment

The authors thank the anonymous reviewers for their suggestions. This work was supported in part by National Science Foundation grant 2127207.

References

[1] D. Yan, T. Wu, Y. Liu and Y. Gao, "An efficient sparse-dense matrix multiplication on a multicore system," 2017 IEEE 17th International Conference on Communication Technology (ICCT), Chengdu, China, 2017, pp. 1880-1883, doi: 10.1109/ICCT.2017.8359956.

[2] A. Bik, P. Koanantakool, T. Shpeisman, N. Vasilache, B. Zheng, and F. Kjolstad, "Compiler Support for Sparse Tensor Computations in MLIR," ACM Trans. Archit. Code Optim., vol. 19, no. 4, pp. 1–25, Dec. 2022, doi: 10.1145/3544559.

[3] W. W. Sun, J. Lu, H. Liu, and G. Cheng, "Provable Sparse Tensor Decomposition," Journal of the Royal Statistical Society Series B: Statistical Methodology, vol. 79, no. 3, pp. 899–916, Jun. 2017, doi: 10.1111/rssb.12190.

[4] G. Blelloch, M. Heroux, and M. Zagha, "Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors," Oct. 1993.

[5] E. T. Phipps and T. G. Kolda, "Software for Sparse Tensor Decomposition on Emerging Computing Architectures," SIAM J. Sci. Comput., vol. 41, no. 3, pp. C269–C290, Jan. 2019, doi: 10.1137/18M1210691.

[6] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," SIAM J. Matrix Anal. Appl., vol. 13, no. 1, pp. 333–356, Jan. 1992, doi: 10.1137/0613024.

[7] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, in PPoPP '19. New York, NY, USA: Association for Computing Machinery, Feb. 2019, pp. 300–314. doi: 10.1145/3293883.3295712.

[8] G. Huang, G. Dai, Y. Wang, and H. Yang, "GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks," in SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Nov. 2020, pp. 1–12. doi: 10.1109/SC41405.2020.00076.

[9] Qiu, Shenghao, Liang You, and Zheng Wang. "Optimizing sparse matrix multiplications for graph neural networks." International Workshop on Languages and Compilers for Parallel Computing. Cham: Springer International Publishing, 2021.

[10] Saad, Youcef. SPARSKIT: A basic tool kit for sparse matrix computations. No. NAS 1.26: 185876. 1990.

[11] Q. Fu, T. B. Rolinger and H. H. Huang, "JITSPMM: Just-in-Time Instruction Generation for Accelerated Sparse Matrix-Matrix Multiplication," 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Edinburgh, United Kingdom, 2024, pp. 448-459, doi: 10.1109/CGO57630.2024.10444827.

[12] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, Calgary AB Canada: ACM, Aug. 2009, pp. 233–244. doi: 10.1145/1583991.1584053.

[13] A. Mehrabi, D. Lee, N. Chatterjee, D. J. Sorin, B. C. Lee, and M. O'Connor, "Learning Sparse Matrix Row Permutations for Efficient SpMM on GPU Architectures," in 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Mar. 2021, pp. 48–58. doi: 10.1109/ISPASS51385.2021.00016.

[14] "Overview — Numba documentation." Accessed: Jul. 09, 2024. [Online]. Available: https://numba.readthedocs.io/en/stable/user/overview.html

[15] "How to Optimize Data Transfers in CUDA C/C++," NVIDIA Technical Blog. Accessed: Jul. 12, 2024. [Online]. Available: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

[16] "Using Shared Memory in CUDA C/C++," NVIDIA Technical Blog. Accessed: Jul. 12, 2024. [Online]. Available: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[17] "NVIDIA Profiler." Accessed: Jul. 12, 2024. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/

[18] Y. Hu, P. Kumar, G. Swope and H. H. Huang, "TriX: Triangle counting at extreme scale," 2017 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 2017, pp. 1-7, doi: 10.1109/HPEC.2017.8091036.

[19] Jianhua Gao, Weixing Ji, and Yizhuo Wang. 2024. Optimization of Large-Scale Sparse Matrix-Vector Multiplication on Multi-GPU Systems. ACM Trans. Archit. Code Optim. Just Accepted (July 2024). https://doi.org/10.1145/3676847

[20] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), Vol. 3. Association for Computing Machinery, New York, NY, USA, 253–267. https://doi.org/10.1145/3620666.3651378

[21] Pradeep Kumar and H. Howie Huang. Falcon: Scaling IO Performance in multi-SSD Volumes. In Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference . USENIX Association, 41–53.

[22] D. Zheng, D. Mhembere, V. Lyzinski, J. T. Vogelstein, C. E. Priebe and R. Burns, "Semi-External Memory Sparse Matrix Multiplication for Billion-Node Graphs," in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 5, pp. 1470-1483, 1 May 2017, doi: 10.1109/TPDS.2016.2618791.