FAST SDDMM ON GPUS

E. Schreiber, N. Roemer, L. Erlacher, M. Möller, D. Knecht

Department of Computer Science
Design of Parallell and High Performance Computing
ETH Zürich

ABSTRACT

In this paper we implement SDDMM on GPUs. We choose SDDMM as an algorithm to optimize as it is a primitive used in many machine learning applications. We first implement baselines and benchmark different optimization techniques to find the ones that lead to the biggest speedup. We then further optimize one implementation to benchmark against math library implementations. This paper was inspired by the paper "Sampled dense matrix multiplication for high-performance machine learning." [1] and we also compare our implementation to the one of that paper. Additionally, we develop density boundaries that guide the choice of how shared memory should be used.

1. INTRODUCTION

Motivation. The fundamental operations that machine learning applications repeatedly perform can typically be decomposed into linear algebra primitives. One such primitive is Sampled Dense-Dense Matrix Multiplication (SDDMM). Running machine learning applications on GPUs (or other accelerators such as TPUs) is suitable since they require many instances of the same operation. Due to the iterative nature of machine learning algorithms small performance gains in the implementation of the underlying linear algebra primitives can lead to a significant runtime reduction. While GPUs are suited for a wide variety of linear algebra operations, sparsity (which is frequent in machine learning applications) poses a challenge, the main issues being irregular access patterns to memory as well as an irregular distribution of work across the input data. In this paper, we explore optimization options for improving SDDMM on GPUs.

Related work. The SDDMM operation was first introduced in the context of big data and machine learning in the paper "Big data analytics with small footprint: squaring the cloud"[2] in 2013, as a custom kernel which the authors describe to be "non-standard matrix operations that provide significant acceleration for one or more learning algorithms". We used the paper "Sampled Dense Matrix Multiplication for High-Performance Machine Learning" [1] (which implements a variety of optimizations such as vectorization, tiling, and the use of warp-level primitives for reduction in order to optimize load-balancing on extremely sparse matrices) as a starting point for our exploration of SDDMM optimizations. The paper proposes two different algorithms: the first option ("SM-L2") loads one matrix into shared memory and tries to reuse the other matrix by keeping it in L2 cache. In the other variant ("SM-SM") both matrices are kept in shared memory. Based on the sparsity of the sampling matrix the appropriate option is chosen.

The authors thank Jelena Kovacevic. This paper is a modified version of the template she used in her class.

2. BACKGROUND

SDDMM. The input to the SDDMM operation are the following three matrices: two dense matrices $A \in \mathbb{R}^{M \times k}$ and $B \in \mathbb{R}^{K \times N}$ as well as a sparse matrix $S \in \mathbb{R}^{M \times N}$. The output is another sparse matrix $P \in \mathbb{R}^{M \times N}$. SDDMM first performs a matrix-matrix multiplication between A and B. Afterwards we sample from S by setting all elements $P_{ij} = 0$ where $S_{ij} = 0$. Since S is typically highly sparse the sampling step will set most of the intermediary results to zero. Therefore, it is inefficient to calculate the complete matrix-matrix product of A and B and a more efficient approach is to compute the inner product of a row in A and a column in B only if the corresponding entry in S is non-zero. Formally this becomes the following:

$$P_{ij} = A_{i,n} * B_{n,j} * \mathbb{1}_{S_{i,j} \neq 0}$$

3. INTRODUCTION OF THE IMPLEMENTATIONS

In this section we discuss the different SDDMM implementations that we have developed as part of this project. We will start by talking about different baselines. Afterwards, we introduce and discuss a selection of possible optimizations before introducing the implementation of the paper [1]. In the end we also propose density boundaries for how to optimally use shared memory.

One optimization is applied to all implementations. For memory alignment reasons B is saved as it's transpose in memory. For this paper we only consider the densities where loading both A and B into shared memory does not lead to improved performance. The bound for this will be defined in SM-L2. Our code can be found at [3].

Baselines. We have implemented four different baselines. The first baseline, **sampled_cuBLAS**, calls "cublasSgemm" from the cuBLAS library (see [4]) and subsequently samples the resulting dense matrix on the GPU. The intent behind this baseline is to show that it is indeed beneficial to develop an algorithm that takes density into account. The second baseline **cuSPARSE_baseline** invokes "cusparseSDDMM" from the cuSPARSE library (see [5]). This is our primary baseline as this function is used, among others, by PyTorch (see [6]). The third baseline **naive_coo** is used to develop and evaluate optimizations. It implements a naive version of SDDMM in which every thread calculates the vector-vector product between one row of A and one column of B for a single non-zero entry in the sampling matrix. The last baseline, **naive_csr**, is a bit more sophisticated as each block calculates all the inner products of one row of the sampling matrix and each result is aggregated by warp-level reductions. It spawns 2 blocks of 1024 threads per SM.

Improvements. The first thing to note is that the cuSPARSE_baseline implementation outperforms the sampled_cuBLAS version. The reason for this is that the latter computes unnecessary entries while the cuSPARSE implementation only computes the needed ones. It also holds, that the sparser a matrix is, the more SDDMM gets memory bound. It follows that memory management is the main area that our improvements are aimed at. The following improvements are also mentioned in [1]. The first optimization we test is loop unrolling with a factor of 4 (coo_loop_unrolled). This allows us to skip three quarters of control flow in the for-loops where the loop unrolling is applied. More importantly, this helps with memory alignment. Adding to this, the second optimization improves the memory loading scheme. CUDA offers vector loading operations that take fewer cycles than regular loading operations. For this the float array needs to be cast to a float2 or float4 array. In the the case of float4, 4 floats are loaded in two cycles. The downside of this optimization is that the memory needs to be 128 bit aligned. This can be achieved either by designing the algorithm to enforce correct alignment (for example by using padding) or by implementing control flow to deal with incorrect alignment. The third optimization is to use shared memory for one matrix. The choice which matrix to load into shared memory is dependent on the size of the number of rows of A and the number of columns of B as well as the algorithm used. Since in all of our implementations the matrix B is saved as its transpose, matrix A and matrix B can be switched to get the transposed result. Therefore choosing which matrix to load into shared memory is arbitrary if the transposition of the result is handled as post processing. The fourth optimization that is mentioned in the paper is to perform the SDDMM block-wise. This is strange. If the block size is chosen correctly not only data that has been loaded into shared memory can be reused, but also L2-cache can be used efficiently. The paper [1] focuses on this aspect. The paper [7] introduces the notion of reordering rows and columns to create dense and sparse regions. This can either be done for the whole matrix or in the case of this paper for each block. Creating dense regions allows the use of more efficient algorithms that are less memory bound. Additionally, this creates very sparse regions that lend themselves to specialized algorithms. The last improvement is to eliminate bad worst case run-times by giving small sets of computations to randomized threads or warps.

Our versions. We looked at both COO and CSR for possible storage formats for the sparse matrix and implemented both a naive_coo and naive_csr version. For our final versions we preferred CSR as it results in less memory loads. Additionally we realized that the value vector does not need to be copied to the GPU as the matrix S is only used for sampling and not for a Hadamard product. coo_vectorized and coo_loop_unrolled focus on improving the control flow and memory loading. Both use naive_coo as the basis and then add optimizations. **coo_loop_unrolled** unrolls the for loop that calculates the vector-vector product by a factor of 4. **coo_vectorized** does the same thing by casting matrix A and matrix B to float4 and then multiplying the float4 values. Between naive_csr and naive_coo, naive_csr is more performant than naive_coo most of the time. The reason for this is that it allows L2 reuse for matrix A and uses warp reductions. For small k naive_csr does not perform well. This is because all 1024 threads always work on one vector-vector product even if the vector only has length 50. In this case the other 974 threads remain idle throughout the computation which results in bad occupancy. To remedy this, **dynamic_warp_csr** calculates how many rows can be worked on at the same time to still get reuse of matrix A in L2 cache. We then calculate the number of warps that need to cooperate on a row to distribute the threads to get high occupancy.

SM-L2. In the paper [1] the authors introduce the notion of an "SM-L2" scheme in which one matrix is loaded to shared memory and the other matrix gets implicitly reused through L2 cache. We agree that for a specific density range this is the correct approach. However, for densities below

$$p \leq \frac{\alpha}{min(max(M,N), \frac{\text{shared memory size in bytes}}{4})}$$

this method in expectation does not have enough reuse to warrant the use of shared memory. Instead implementations like naive_csr, that do not use shared memory, perform better. We introduce α as the expected reuse of a row in shared memory. By our measurements the usage of shared memory is justified when $\alpha \geq 1.7$ on the used GPU. This minimal α however, is dependent on GPU characteristics like memory bandwidth and therefore kept general in this paper. For big matrices A and B the upper bound for which using shared memory of p is $p \approx 0.007\%$. As mentioned in the paper [1] for high enough densities having both matrices in shared memory performs best. For the V100 this is the case from 1.5%, the paper mentions 5% as they based it on measurements of a now old GPU. The general formula for this bound is given by

$$p \geq \frac{\alpha}{\lfloor \sqrt{\frac{\text{shared memory size in bytes}}{2*4}} \rfloor}$$

. We propose this bound as it is not dependent on a specific GPU but rather on the GPU architecture and therefore more relevant in the future. The paper then calculates the block size of the sampling matrix as

$$\sqrt{\frac{\text{L2 size}}{c * p}} * \frac{\text{shared memory size}}{\sqrt{\text{L2 size * c*p}}}$$

for p the density and c the cost of the representation of the sparse matrix (3 for COO). These are the correct bounds for optimal L2 reuse. The derivation of these bounds can be found in [1].

4. EXPERIMENTAL RESULTS

This section introduces the hardware as well as the parameters used to evaluate the implementations and discusses the performance differences between them.

Experimental setup. We evaluate the performance of all implementations on a Tesla V100 PCIE 32GB. The V100 consists of 80 streaming multiprocessors, has a L2 cache size of 6 MiB, and 96 KiB of shared memory per multiprocessor. We use GCC 10.2.0 and CUDA 12.1.1 for compilation. Our CUDA flags are –gencode arch= compute_70,code= sm_70, –use_fast_math, –Xptxas –O3, ––m64 and ––fmad=true . Our CXX flags are –std=c++17, –Ofast, – funroll –loop, – flto , – finline – functions , –march=native, –mtune=native and –fopenmp. For timing we used the "CUDATimer.cuh" timing function [8]. We ran a collection of different experiments where we kept $M = N$ while varying $M \in [100, 10^9]$, $k \in \{50, 100, 500, 1000\}$ and the density $p \in [0.0001\%, 1.5\%]$. Choosing $M = N$ gives us the worst case regarding shared memory usage as we can not take advantage of choosing which matrix to load to shared memory. The values of matrix A and matrix B are created uniformly at random and for the sampling matrix we use a mix of self generated random matrices, marked as gen in all plots, and matrices from matrix market (see [9]), marked with MM in all plots. We checked that the advertised density was accurate and corrected the density where needed. For a correctness check we used the sampled_cuBLAS and cuSPARSE_baseline implementations. We also adapted the code from the paper [1] to work with a more recent CUDA version and our pipeline but found that it occasionally yields incorrect results, prompting us to only use it as a baseline and not as ground truth. To check all our implementations we compare them to the naive_coo implementation. We do not include the runtime of transposing the matrices in the reported runtimes. The reason for this is that in a real world scenario matrix B often already is in transposed form to optimize other operations. We also do not include the time to copy the memory from the host to the device and back as for small matrices we can assume that they are already on the device. Specifically for the code of the paper [1] we are not timing preparation functions as they were not timed in their paper. They are quite extensive and would add a big runtime overhead in realworld scenarios, that our implementations avoid. Our plots show the median of the data along with a 98% confidence interval which we plotted using Seaborn's errorbar option that does not assume normality of the data. In the case of plots where the percentage of a baseline is shown we divide the measurements by the median of the baseline and multiply by 100 to get the percentage.

Results. We will first explore how the baselines perform in comparison to each other. Subsequently we will discuss the impact of the improvements as well as the performance of the implementation of the paper [1]. Lastly, we discuss the performance of some implementations using a roofline model.

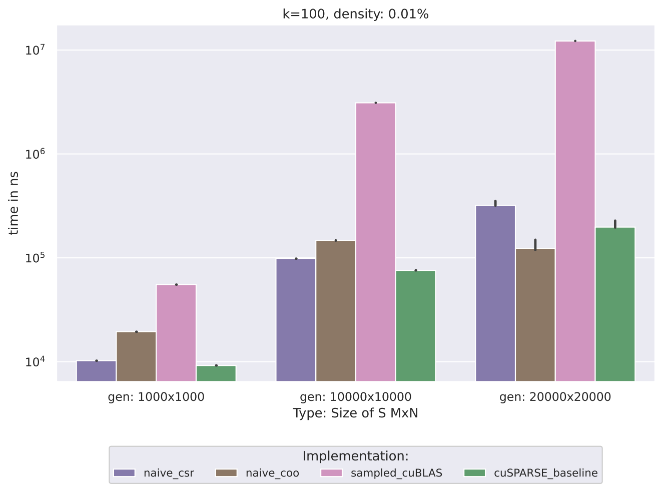

Fig. 1. Performance of our baseline models

Baselines. Figure 1 shows the runtimes in nanoseconds on a log-axis of the different baselines over different sizes of S for $k = 100$ and a density of 0.01%. The x-axis displays the size of S with each color representing a different baseline. We can see that sampled_cuBLAS is outperformed by all other baselines for all sizes. The reason for this is that it performs a dense-dense matrix product between matrix A and matrix B and samples the resulting dense matrix. With the largest density tested being 1%, this means that at least 99% of work is done in vain. All other implementations only calculate the necessary inner products. Especially the comparison with naive_coo shows the validity of developing specialized algorithms and matrix storage formats for sparse matrix computations as this very naive version outperforms the heavily optimized library call. The performance of naive_csr, as discussed in the previous chapter, is dependent on the value of k. The performance lead of cuSPARSE_baseline is significantly bigger for large matrices than for small matrices Which has two reasons. Firstly for very small matrices the number of non-zeros is too small to get enough benefits from a more optimized parallelism compared to the runtime. The second reason is that the benefits of using an optimized memory loading scheme only start to get obvious for larger data sizes.

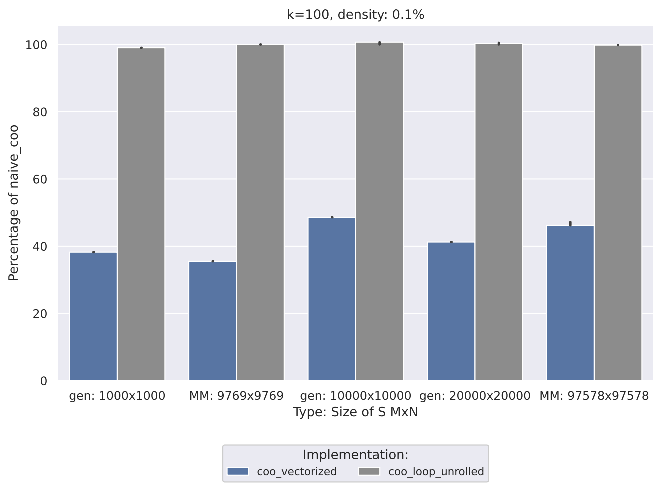

Fig. 2. Performance of loop unrolling and vectorization relative to the baseline naive_COO

Improvements. Figure 2 shows how loop-unrolling and vectorization affect performance in comparison to the naive_coo baseline. The colors correspond to the different implementations. On the x-axis we see the different sizes for matrix S and the y-axis displays the relative runtime of the implementations to naive_coo. We can see that using loop unrolling minimally affects performance in both directions. Our expectation from this optimization was a more significant boost in performance. We suspect that because naive_coo is already bottle-necked by the memory throughput, since all threads of the GPU want to access the memory at the same time, we can not take advantage of any improvements in the control flow. In comparison to this we see a significant increase in performance when using vectorization (float4). The speedup ranges from 90% to almost 170%.

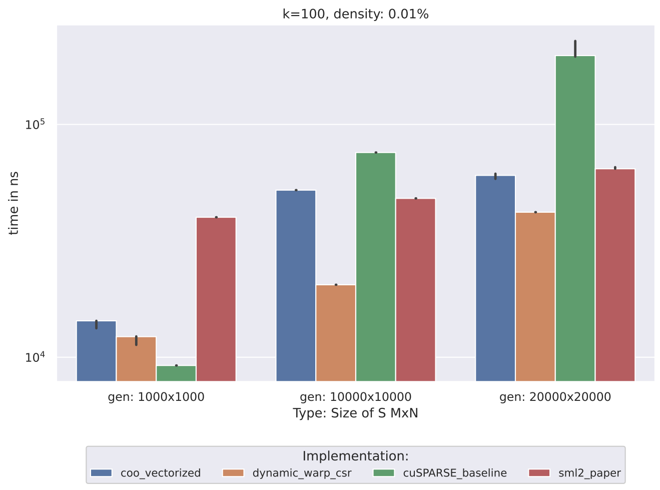

Fig. 3. Performance of our best implementations

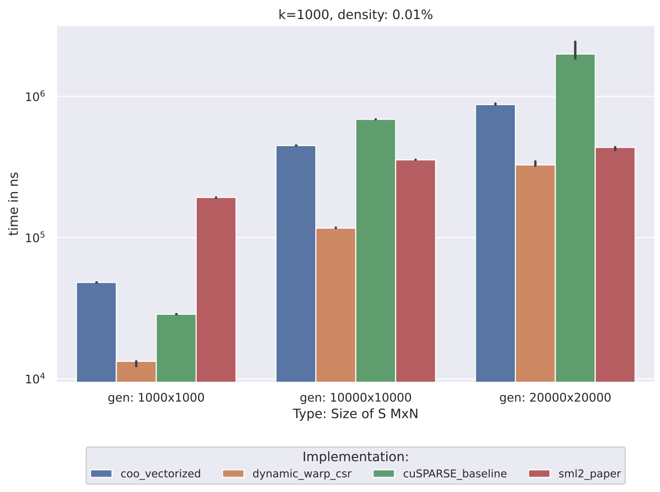

Fig. 4. Performance of our best implementations

Figures 3, 4 and 5 show the performance of coo_vectorized, dynamic_warp_csr, cuSPARSE_baseline and sml2_paper for different matrix sizes, different k and different densities. For small densities of 0.01% dynamic_warp_csr is almost always the best implementation, always beating sml2_paper. The cuSPARSE implementation is the fastest implementation for $k = 100$ and the smallest matrix. In general it performs best for small matrices. For a higher density of 1% sml2_paper is the best implementation for the larger matrices. Noteworthy is that its runtime stays almost constant for the larger matrices instead of the increase seen by all

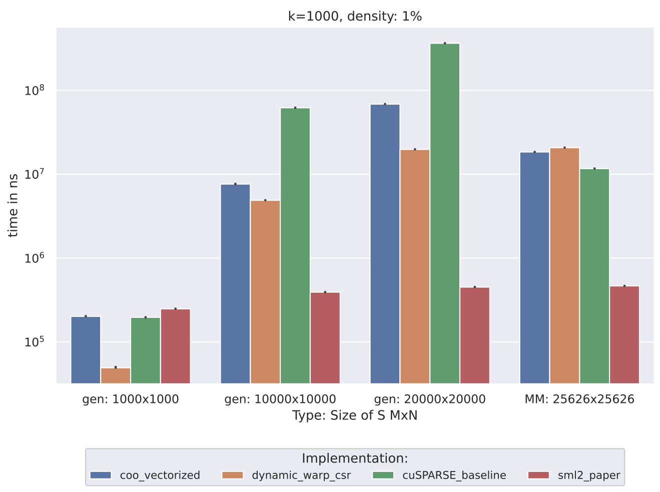

Fig. 5. Performance of our best implementations

other implementations. For the biggest matrix S of size $25626x25626$ coo_vectorized and cuSPARSE_baseline see an improvement of their runtime. The reason for this is that this is not a generated matrix and hence the non zeroes of this matrix are not equally distributed. This allows coo_vectorized a better L2 cache reuse. The reason for the improvement of cuSPARSE_baseline is either that because of the size a different algorithm is used or that it uses optimizations for sparse and dense regions.

In Figure 6 we show a best implementation plot for fixed $k = 500$. It shows for each combination of size of S and density p which implementation is the most performant. We also included the theoretical boundary to the left of which using shared memory for one matrix does not boost performance. In theory sml2_paper should be the best implementation for every point right of this line. In reality the kernel of sml2_paper is not optimized enough to compete with our dynamic_warp_csr implementation. For very small sizes naive_csr outperforms dynamic_warp_csr as it has a slightly smaller overhead.

Roofline.

In order to obtain an objective measure of the proximity of our implementations to the GPU's theoretical maximum performance we use a roofline plot. The plot is based on two papers "Instruction roofline: An insightful visual performance model for gpus"[10] from 2019 and "An instruction roofline model for gpus"[11] from 2022. Our plotting code is based on their Matlab implementations that can be found here [12]. We obtained the relevant GPU measurements of our implementations via $\text{nvprof} \text{ --analysis-metrics --cpu-thread-tracing on --print-api-trace --metrics all --profile-child-processes --system-profiling on --trace api,gpu --track-memory-allocations on --events all --csv --export-profile}$.

In this measurement, the kernel of coo_vectorized took

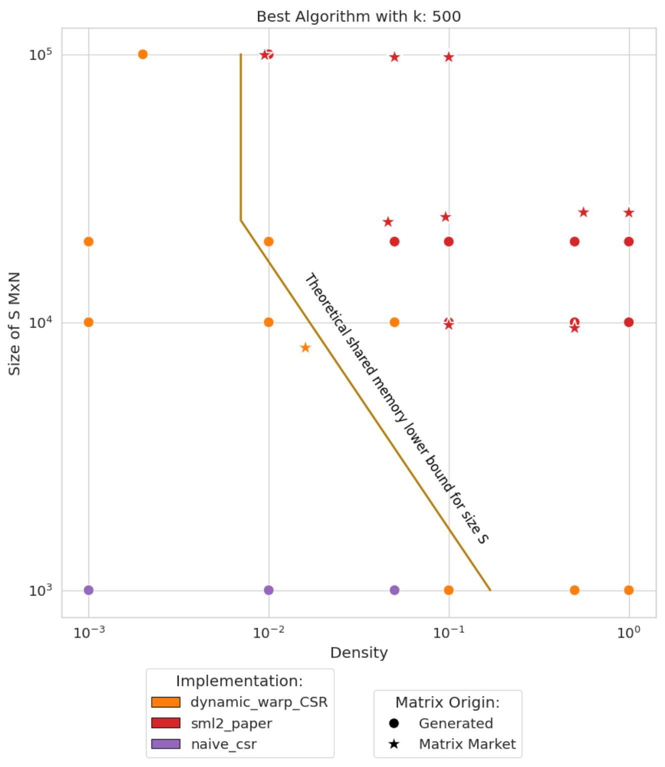

Fig. 6. Best implementation given k = 500

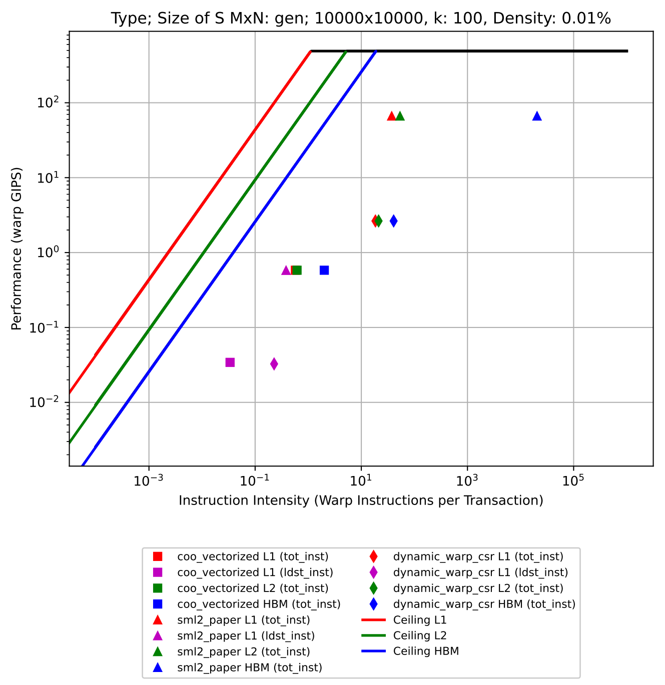

Fig. 7. Roofline plot for coo_vectorized, sml2_paper and dynamic_warp_csr

48 microseconds, of sml2_paper took 37 microseconds and the kernel of dynamic_warp_csr took 13 microseconds. Nonetheless we can see that sml2_paper uses the most instructions per second which should be the impact of using shared memory. Additionally the bandwidth to main memory is not used well. We can also see that dynamic_warp_csr uses more operations per second than coo_vectorized. This is because its memory access pattern is better. The roofline plot seems to indicate that SM_L2_paper is the most performant implementation. However, seeing that it is slower than the other implementations, we can argue that they issue unnecessary instructions, thereby distorting the roofline plot.

5. CONCLUSIONS

In this paper we have shown that while using the cuSPARSE SDDMM implementation leads to decent results, for almost all sizes other implementations outperform it. We have also shown that SDDMM is memory bound and the biggest room for improvement is optimizing memory accesses on the device. One of the key optimizations for the memory accesses on the device is the use of vectorization with the use of float4. We strongly recommend the use of float4 over float wherever possible for vector-vector multiplications. Additionally we introduced density boundaries which determine when the shared memory is best used.

Next Steps. We propose to clean up and optimize the sml2_paper implementation to reach the theoretical bounds of the shared memory use. The next step is to design an implementation that interleaves and parallelizes memory movement between the host and the device a blueprint can be found here [1]. In this implementation the row and column reorder optimization can be implemented to create dense and sparse regions. Additionally which rows and columns a thread works on should be randomized to guarantee the absence of worst case inputs. At the end this implementation can then be optimized for multi-node and multi-GPU systems.

[1]SDDMMlib/src/SDDMM/optimized_sml2_blueprint.txt

6. REFERENCES

[1] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan, "Sampled dense matrix multiplication for high-performance machine learning," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, 2018, pp. 32–41.

[2] John Canny and Huasha Zhao, "Big data analytics with small footprint: Squaring the cloud," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, 2013, KDD '13, p. 95–103, Association for Computing Machinery.

[3] E. Schreiber, N. Roemer, L. Erlacher, M. Möller, D. Knecht, "Dphpc23-smokyrhino," online: https://github.com/ericschreiber/DPHPC23-SmokyRhino, 2023.

[4] "cublas library," online: https://docs.nvidia.com/cuda/cublas/index.html?highlight=cublasSgemmcublas-t-gemm, 2023.

[5] "cusparse library," online: https://docs.nvidia.com/cuda/cusparse/generic-api/generic-api-functions.html?highlight=cusparseSDDMMcusparsesddmm, 2023.

[6] "Pytorch documentation," online: https://github.com/pytorch/pytorch/blob/172dd13ecff965a549bf4f2a58f5b2900a00497a/aten/src/ATen/native/sparse/cuda/SparseBlasImpl.cppL1501, 2023.

[7] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2019, PPoPP '19, p. 300–314, Association for Computing Machinery.

[8] Fabian Wermelinger, "Cudatimer.cuh," online: https://gitlab.ethz.ch/hpcse-public-repos/hpcse-ii-spring-2023-lecture/-/blob/main/homeworks/hw04/skeleton_code/include/CUDATimer.cuh, 05 2021.

[9] "Suitesparse matrix collection," online: https://sparse.tamu.edu, 2023.

[10] Nan Ding and Samuel Williams, "An instruction roofline model for gpus," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 7–18.

[11] Nan Ding, Muaaz Awan, and Samuel Williams, "Instruction roofline: An insightful visual performance model for gpus," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, pp. e6591, 2022.

[12] Nan Ding, "Instruction roofline model profiling and plotting scripts," online: https://github.com/nanding0701/Instruction_roofline_scripts, 2023.