EFFICIENT SDDMM ALGORITHMS ON GPU, A DYNAMIC APPROACH

François Costa, Lucas Bürgi, Marc Dufay, Marius Debussche, Paul Elvinger

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

The Sampled Dense-Dense Matrix Multiplication (SDDMM) is a fundamental operation proven to be instrumental in the redesign of various machine learning factor analysis algorithms, including Alternating Least Squares (ALS), Latent Dirichlet Allocation (LDA), Sparse Factor Analysis (SFA) and Gamma Poisson (GaP). This operation involves computing the product of two dense input matrices but exclusively at locations corresponding to non-zero entries in a sparse third input matrix.

This paper delves into the development of *GPU-Dynamic*, an efficient GPU implementation of SDDMM. *GPU-Dynamic* dynamically selects different algorithms at runtime depending on the density of the sparse matrix. With careful implementations for each of these algorithms, our implementation achieves competitive results with state-of-the-art libraries like DGL and outperforms Torch with speedups of at least 2x but up to 100x in many cases, without requiring any pre-processing of the input.

1. INTRODUCTION

Matrix multiplication is a fundamental operation in many mathematical and computational tasks. It is a critical component for various algorithms in Machine Learning (ML) such as Alternating least squares [10] and Sparse Factor Analysis [3], for Graph Analytics [16], and for scientific computing [11]. It has therefore unsurprisingly become a highly optimized operation and subject to countless previous works in the past decades. This mainly manifests in the specification and development of different levels of the BLAS library [1]. While BLAS1 incorporates dot product operations, BLAS2 operations revolve around dense matrix-vector products and BLAS3 around dense matrix-matrix products. None of these directly address their respective operations in the purely sparse setting however. That said, many formulations of current ML algorithms rely on *sparse* BLAS2 (matrix-vector product) operations and recent work [5] has proposed to reformulate ML algorithms using *sparse* matrix-matrix product primitives.

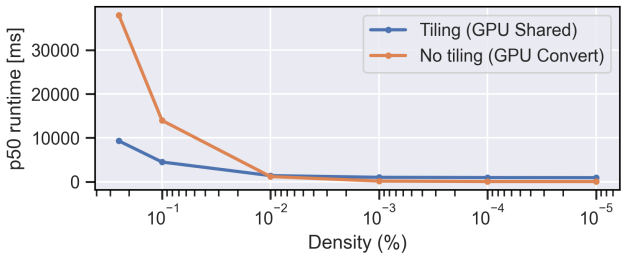

Fig. 1. The effect of tiling shown on two of our implementations. The plot shows the 50th percentile computation runtime over R = 20 runs on a A100 GPU for a randomly generated 20k $\times$ 20k sparse matrix S with different densities and $K = 32$

In this context, the SDDMM algorithm performs such a sparse matrix-matrix multiplication, involving sparse and dense matrices. It specifically targets the element wise multiplication of a sparse matrix S with two dense matrices A and B. To note here is that SDDMM can be computed using the GEMM operation exposed in BLAS3 between A and B followed by extraction of the sampled elements from S [12]. With growing input problems however, this becomes unpractical due to the excessive amounts of unnecessary computations arising from the zero elements in S. Therefore an efficient SDDMM implementation must take advantage of the sparsity pattern in S to reduce the number of operations of the multiplication compared to performing a dense matrix multiplication. This can be crucial for large-scale applications. While the use of the SDDMM kernel gains popularity for reformulating machine learning algorithms, there are still open opportunities for making it efficient.

State-of-the art implementations [12, 9, 6] of SDDMM kernels fail to provide both efficiency and zero pre-processing overhead on large matrices with only a few non-zero entries. These implementations internally use a technique referred to as *tiling*. The idea behind it is to break the matrix multiplication down into smaller chunks that can then be processed independently with the goal of exploiting data lo-

cality and optimizing memory access patterns. As Figure 1 highlights, the benefits of *tiling* however vanish as the density of the sparse matrix decreases. While this behaviour can be counter-acted with pre-processing the sparse input matrix [12], this effort requires the sparsity pattern to be known upfront and comes with an expensive overhead making it only attractive for repeated SDDMM computations that reuse the sparse matrix multiple times.

We were able to measure a number of inefficiencies in the SDDMM routines exposed by the C++ Torch API when input matrices grow in size and the density of the sparse matrix decreases. Despite the fact that the source code of the C++ Torch API is openly accessible [15], it is difficult to understand the exact implementation details of its GPU SDDMM kernel as the underlying cuSPARSE library [6] is closed-source.

In this paper, we make the following contributions. We start off by analyzing the effect of tiling techniques in the sparse setting, by evaluating different custom implementations. Given the newly gained insights, we propose to combine different implementations into what we call *GPU-Dynamic*, which aims at being both efficient and having zero pre-processing overhead on all kinds of sparse matrices. The remainder of this paper is structured as follows, section 2 provides the necessary background and formally defines the SDDMM problem. Sections 3, 4, 5 and 6 introduce some of our most promising implementations that build up to our final implementation *GPU-Dynamic* presented in section 7. Section 8 presents our results and we conclude the paper by reflecting on related and future work in sections 9, 10 and 11.

2. BACKGROUND

The SDDMM (Sampled Dense-Dense Matrix Multiplication) algorithm 1 multiplies a sparse matrix $S \in \mathbb{R}^{m \times n}$ element wise with the matrix product of two dense matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$ where m and n are typically very large while k is typically much smaller with $k \in [16, 512]$. For the remainder of this work we show results where $k \in \{32, 64, 128\}$ for consistency but note that our results are not limited to that. For each non-zero element s_{ij} of S, the operation performs a dot product between A_i and B_j^T which is scaled by s_{ij}, where A_i is the i-th row of A and B_j^T is the j-th column of B. The result is stored in a matrix $P \in \mathbb{R}^{m \times n}$.

Furthermore, it stands as a crucial matrix-matrix product operation with significant implications for the enhancement of diverse machine learning factor analysis algorithms. Notably, it plays a pivotal role in the optimization of algorithms such as Alternating Least Squares (ALS) [10], Latent Dirichlet Allocation (LDA) [20, 2], Sparse Factor Analysis (SFA) [3], and Gamma Poisson (GaP) [19, 4].

Algorithm 1 Naive SDDMM kernel (COO)

Input: CSR $S[M][N]$, float $A[M][K]$, float $B[N][K]$
Output: CSR $P[M][N]$

```
countPerThread = S.nnz/threadCount
startIdx = countPerThread * globalThreadId
endIdx = countPerThread * (globalThreadId + 1)
for i = startIdx to endIdx do
    row = S.row[i]
    col = S.col[i]
    result = 0
    for j = 0 to K do
        result+ = A[row * K + j] * B[col * K + j]
    end for
    P.val[i] = result * S.val[i]
end for
```

3. GPU-SHARED

This section introduces our GPU-Shared implementation which achieves a great speedup for sparse matrices with a relatively high density. As we will see in greater detail in section 6, when the matrix is not too sparse, tiling is a great way to reduce data movement and thus improve performance. The *shared* keyword comes from the fact that this implementation leverages shared memory on the GPU to decrease the bandwidth used. Because shared memory is quite limited, we must reduce the global SDDMM operation to multiple sub tasks which can each fit in shared memory and be parallelized. First, the $M \times K$ matrix A and $K \times N$ matrix B are divided into K/T_k smaller $M \times T_k$ and $T_k \times N$ matrices where T_k is the tile size along the K dimension. By linearity the desired result is the sum of the results for each of these sub-tasks. The $M \times N$ sparse matrix S is divided into smaller $T_i \times T_j$ tiles such that we are now left with some fixed-size matrices which can be loaded in the appropriate caches. The pseudo-code for the CUDA kernel can be seen in algorithm 2. Each thread has to process the row of S at my_row for a single tile of K.

Israt et al.[12] load the $T_i \times T_k$ sub-matrix A_s into GPU shared memory while the $T_k \times T_j$ sub matrix B_s was kept in L_2 cache. For each non-zero entry, its value is computed by exactly n_c threads. By a clever re-use of the coefficients of A_s for each thread, we were able to keep the content of A_s inside GPU registers. Meanwhile we loaded the content of B_s in shared memory. We note that the fact that we are dealing with the CSR representation of the sparse matrix, and not COO like [12] adds a few additional constraints.

In the original paper, the authors were doing a ingenious warp reduction before storing the result. We observed using benchmarks that performing an atomic operation instead yields a 35% throughput improvement. This is caused by the fact that NVIDIA GPUs have an instruction for atomically storing while reducing with the add operation. Moreover, this allows for better parallelization over the K/T_k

sub-tasks.

Algorithm 2 GPU-Shared Kernel

Input: CSR $S[M][N]$, float $A[M][T_k]$, float $B[N][T_k]$, my_row
Output: CSR $P[M][N]$ the result of the SDDMM operation

▷ Load the coefficients of A in GPU registers
coeffs $\leftarrow A[\text{my\_row}][0..\text{T}_\text{k}]$
for $t_j = 0$ **to** N **by** T_j **do**
  ▷ Load the coefficients of B in shared memory
  Thread barrier
  shared_B $\leftarrow B[t_j..t_j + T_j][0..T_k]$
  Thread barrier
  ▷ Compute the result for each entry in the tile
  for Each non-zero entry $(\text{my\_row}, col)$ at index idx for $col \in [t_j, T_j + t_j]$ **do**
    $v \leftarrow S.values[idx] * \text{dot}(\text{coeffs}, \text{shared\_B}[col - t_j][0..\text{T}_\text{k}])$
    Atomic store v at $P.values[idx]$
  end for
end for

4. GPU-PREPROCESSING

The *GPU-Preprocessing* is an improvement over *GPU-Shared* which achieves better results compared to the previous implementation for really sparse matrices, at the cost of a preprocessing step first. The preprocessing step only requires the location of the non-zero entries (and not their values).

We want to highlight that in the *GPU-Shared* implementation if a tile has an empty column, we still load all the coefficients from B. In particular if the matrix only has coefficients around the diagonal, we end up with most of the coefficients being loaded while they are not used. To solve this issue, the pre-processing step re-indexes the column entries such that each column from B loaded has at least one non-zero entry. If the matrix is diagonal, with the previous approach each thread block has to process N/T_j tiles while with the new approach only one tile has to be processed. However, it is important to note that the pre-processing steps are a significant overhead that is only worth paying if the SDDMM operation is repeated many times as we describe in more detail in the section 8.

Note that in algorithm 3, the result of the pre-processing I has size $M * N/T_i$ to simplify the pseudo code. In our implementation, its size is linear in the number of non-zero elements $nnz(S)$ of the matrix S.

5. GPU-CONVERT

GPU-Convert handles the computation on sparse matrices. This approach assumes that capturing memory reuse is either too expensive or too challenging. It aims to optimize towards perfectly balanced computation (meaning each thread has exactly the same amount of work, in our case amount of non-zero entries to compute) by first converting the CSR format into COO, as CSR may lead to an unbalanced workload. The conversion from CSR to COO can be parallelized, making it efficient. After the conversion, *GPU-Convert* is a straightforward COO SDDMM as shown in Algorithm 4 but additionally enhanced with loop unrolling and vectorization.

Algorithm 3 GPU Preprocessing - CPU Part

Input: CSR $S[M][N]$, S values are not required
Output: $I[M/T_i][N]$ B's column indices which have non-zero entries for each $T_i \times N$ block

for $t_i = 0$ **to** M/T_i **do**
  $idx \leftarrow 0$
  for $col = 0$ **to** N **do**
    if $S[t_i * T_i..(t_i + 1) * T_i][col]$ has a non-zero entry **then**
      $I[t_i][idx] \leftarrow col$
      $idx \leftarrow idx + 1$
    end if
  end for
end for

After computing the output, there is no need for conversion back into CSR, as both formats share the same structure for storing values in a flat 1D array and since positions of the sparse values do not change from the sparse input to the sparse output matrix.

Algorithm 4 GPU-Convert Kernel

Input: CSR $S1[M][N]$
Output: COO $S2[M][N]$

$S2.col = S1.col$
$S2.val = S1.val$
$i = globalThreadId$
for $j = S1.row\_start[i]$ **to** $S1.row\_start[i + 1]$ **do**
  $S2.row[j] = i$
end for

6. MEMORY BANDWIDTH COMPARISON

This section will provide some insights on the amount of entries that need to be read from memory for the *GPU-Shared*, *GPU-Preprocessing* and *GPU-Convert* implementation. We first note that for the tiled implementations, they each need to read the coefficients of S K/T_k times along with their positions and that each entry in A is read exactly once, resulting in a cost of $nnz(S) * 3K/T_K + M * K = nnz(S) * 3K/T_k + |A|$.

- **GPU-Shared**: Each coefficient of B is read exactly once for a tile that covers it, which happens M/T_i times. Resulting in a total cost of $nnz(S) * 3K/T_k + |A| + N * K * M/T_i = nnz(S) * 3K/T_k + |A| + |A||B|/(T_i * K)$.

- **GPU-Preprocessing**: In the worst case, each non-zero entry in S is alone in the column of its tile, causing the whole matching column of B to be loaded for it, this has an overall cost of $nnz(S) * (K + 1)$, taking into account loading the pre-processed index. Therefore the overall number of entry reads will be $|A| + nnz(S)(1 + K + K/T_k)$
- **GPU-Convert**: For each non-zero entry, we must read an entire line from A, an entire column from B and its position, resulting in a cost of $nnz(S)*(2K+2)$. The pre-processing part causes $2nnz(S)$ additional reads. So the overall cost is $nnz(S) * (2K + 4)$.

From this analysis, we deduce that the GPU-Shared implementation should perform well when the density of the matrix is at least $1/(T_i * K)$. If the density gets lower, the tiling approach shows its limits.

7. GPU-DYNAMIC

State-of-the-art implementations typically use a single algorithm for both relatively sparse matrices and very sparse matrices. We think optimizing for both scenarios simultaneously is not the most effective approach and can even be counterproductive. While dense matrices can benefit from memory reuse techniques like tiling, very sparse matrices do not gain much from it and can even harm performance.

Therefore, we introduce *GPU-Dynamic*, which dynamically chooses a suitable algorithm at runtime based on simple and fast-to-calculate criteria, such as matrix density. *GPU-Dynamic* uses *GPU-Shared* if the density of the sparse matrix S is more than 0.01 %. Otherwise, it first converts the CSR input data to COO format if needed and then performs a straightforward SDDMM implementation as described for *GPU-Convert* in section 5 with both unrolling of loops and vectorized loads to speed it up.

GPU-Shared excels on matrices with a density of at least 0.01 % percent. For very sparse matrices, we discovered that converting to COO and distributing equal work among threads without potential memory reuse yielded the best results. Despite the basic implementation, it outperforms state-of-the-art implementations in very sparse matrices.

8. EXPERIMENTS

In this section we want to compare our implementation *GPU-Dynamic* to the well-established frameworks Torch [15] and DGL [7]. We will benchmark our implementation over a range of more than 20 sparse matrices S with different sparsity patterns and densities. All measurements have been run on the Swiss National Supercomputing Center (CSCS). We provide measurements for two different type of GPUs. The first one being an NVIDIA V100 GPU attached to PCIe based on the Volta architecture, having 32 GB of HBM2 device memory, 80 SMs each having 128 KB of L1 cache resulting in 5120 CUDA cores and a total of 6 MB of L2 cache. The second one being an NVIDIA A100 GPU attached to SXM based on the Ampere architecture, having 40 GB of HBM2 device memory, 108 SMs each having 192 KB of L1 cache resulting in 6912 CUDA cores and a total of 40 MB of L2 cache.

Benchmark setup. Our goal is to benchmark our implementations on a wide range of input matrices with different sparsity patterns and densities. We therefore divide them into three categories as follows: *Small matrices* (Small number of non-zeros), *Large relatively sparse matrices* (densities $\geq 0.1\%$) and *Large very sparse matrices* (densities $< 0.1\%$). All of them can be found in figures 4, 5 and 6 of the appendix. For a given sparse input matrix $S \in \mathbb{R}^{m \times n}$ and a given value k, we generate the dense matrices $A \in \mathbb{R}^{m \times k}$ and $B \in \mathbb{R}^{k \times n}$ uniformly at random. Our goal is to measure under which circumstances our implementation comes out ahead of the competition and especially when this might not be the case.

We divide a measurement into three different phases that we refer to as *Initialization*, *Computation* and *Cleanup* phase. The *Initialization* phase contains all the code for allocating memory on the GPU, transferring data from the CPU to the GPU and potential pre-processing depending on the implementation. The *Computation* phase contains the implementation of the SDDMM algorithm and finally the *Cleanup* phase retrieves the result from the GPU back to the CPU and contains any potential cleanup code of the corresponding implementation. In order to ensure robustness and correctness of our results, each measurement is repeated 20 times and has been validated by a suite of test cases using Google Test [8]. Our code is openly accessible on GitHub [17].

While the number of threads used on the GPU by the Torch and DGL implementations is determined by the given libraries, we ran our own implementations with different combinations of (thread blocks, threads per block) $\in$ [32, 2048] $\times$ [32, 1024] and choose the best one for the measurements below based on a simple ranking system.

Baselines - Torch & DGL. As the initial goal of this work was to beat the popular PyTorch framework we have decided to choose its C++ Torch backend as our baseline, meaning that all reported speedups below are with respect to Torch. We would like to point out that at the time of this work, the C++ Torch API is in beta state and therefore in further development. To our knowledge the library does not provide an explicit kernel implementation of the SDDMM algorithm as we have defined it in section 2. Instead we will use a combination of the **sparse_sampled_addmm_sparse_csr_cuda** kernel followed by an invocation of the **mul_out_sparse_csr**

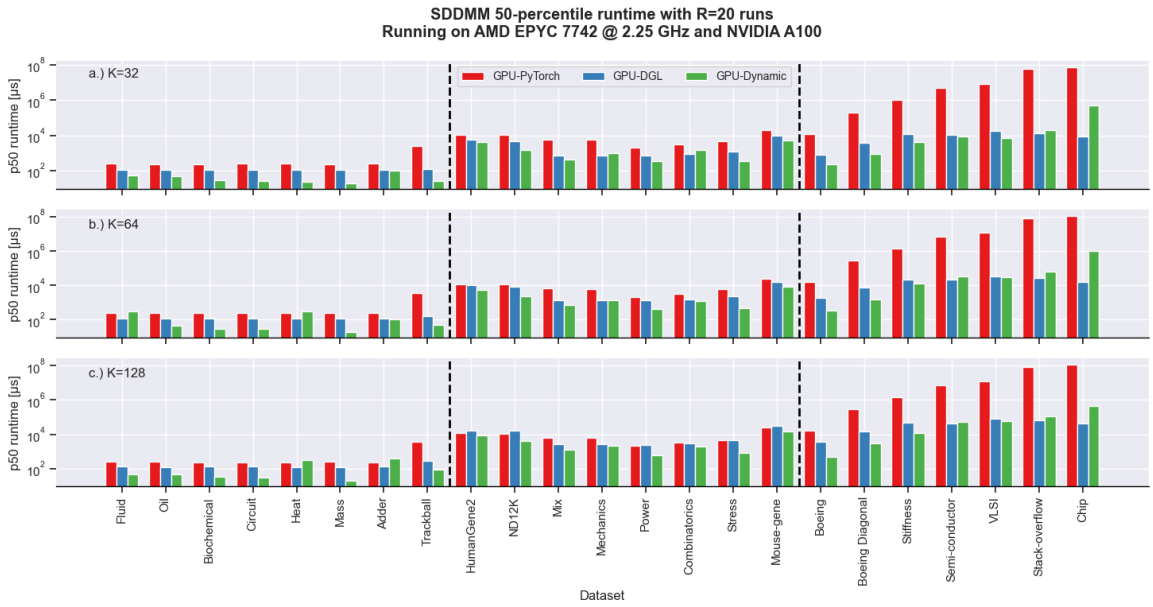

Fig. 2. Bar plots showing the 50th percentile computation runtimes over $R = 20$ runs of our implementation compared to the Torch and DGL implementations on the A100 GPU. (a) $K = 32$, (b) $K = 64$, (c) $K = 128$ where *Small matrices* (left), *Large relatively sparse matrices* (middle), *Large very sparse matrices* (right).

kernel both from the **at::native** namespace in order to scale the results. In addition, we would like to point out that the API currently only supports the Compressed Sparse Row (CSR) format for the sparse matrix S and that other formats like the Coordinate Format (COO) are not supported at this point. As a second way of comparing our implementation we chose the popular deep learning graph library DGL [7] which is designed to be framework agnostic. In the context of this work however, we use Torch as backend for DGL. It is to be mentioned though that DGL does not rely on a Torch based implementation for its SDDMM operation but instead implements a custom kernel for it which leads to DGLs performance being more competitive than Torchs.

Results on small matrices. We benchmark *GPU-Dynamic* against Torch & DGL using eight small matrices chosen from the sparse matrix dataset collection [18] from the University of Florida. *Small matrices* mean they have around 20,000 non-zero entries and fewer than 25,000 rows or columns and the sparsity ranges from 100% to 0.1% (more details can be found in the appendix in Table 1). The results for the 50th percentile over $R = 20$ runs is shown in Figure 2 on the left for different K. In all instances, *GPU-Dynamic* consistently beats Torch and is in most cases on par with DGL. However, for increasing K there are cases where DGL performs best.

Results on large, relatively sparse matrices. We repeat the same experiment for larger matrices. The next eight matrices have densities larger than 0.1% and contain between 2 and 30 million non-zero entries. *GPU-Dynamic* will use *GPU-Shared* as a backend for these matrices. As shown in Figure 2 in the middle, *GPU-Dynamic* performs very good over the different *Large, relatively sparse datasets* and values of K, beating Torch in all instances and beating or being on par with DGL.

Results on large, very sparse matrices. Finally we repeat the same measurements with eight matrices of similar size than in the previous section but with densities lower than 0.1%. *GPU-Dynamic* will use *GPU-Convert* as a backend for these matrices. The results can be seen in Figure 2 on the right. The runtime of *GPU-PyTorch* starts to blow up, meaning once again that *GPU-Dynamic* beats Torch in all instances and is in many cases on par with DGL. However, it must be noted that in the setting of *Large, very sparse matrices* DGL does beat *GPU-Dynamic* in some cases by a solid margin. We plan to investigate why and improve *GPU-Dynamic* accordingly as part of our future work discussed in more detail in section 10.

Roofline plots. In order to not only compare our implementations against existing state-of-the-art frameworks but also get a grasp of where they position themselves with respect to theoretical peak performance and peak memory bandwidth, Figure 3 shows the corresponding roofline plots. To do so we use theoretical single-precision peak performance numbers from the respective manuals [13, 14] which for the V100 lies at 14.13 [TFlops/s] and for the A100 at 19.49 [TFlops/s]. Similarly, for theoretical peak memory

bandwidth which lies at 897 [GB/s] for the V100 and at 1'555 [GB/s] for the A100. We are aware and want to draw attention to the fact that these numbers need to be taken with a grain of salt since they originate from a manual and are not practically measured. That said, the roofline plots show that most computations are substantially far away from the devices peak performance when looking at the log-log plot and the sheer scale of the y-axis. Nonetheless, for both GPUs we can show that some measurements are largely memory-bound and others are severely compute-bound but quite interestingly some are exactly balanced. We refer the reader to figure 9 of the appendix, which shows the roofline plot for the A100 GPU but this time grouped by the matrix categories mentioned earlier. One can see that *small matrices* and *large very sparse matrices* tend to be memory-bound. Unsurprisingly, for slightly denser matrices in the middle of Figure 9 we observe that computation, especially for larger K, starts to dominate.

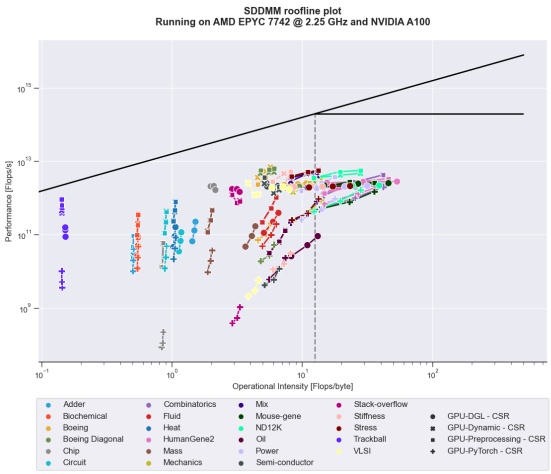

Fig. 3. Roofline plot on the A100 GPU over all implementations and all datasets where $K \in \{32, 64, 128\}$. *(Larger & extra plots can be found in the appendix in section 14.5).*

Pre-processing overhead. *GPU-Dynamic* does not perform any pre-processing which is a crucial advantage if sparsity patterns change between every execution but is something that can and should be exploited in cases where the sparsity pattern remains constant among many executions. In this common case, especially in the setting of ML oriented workloads, we presented *GPU-Preprocessing* earlier in section 4 which incurs a pre-processing overhead performed on the CPU of up to $6,4s$ in the worst case (stack-overflow dataset) but outperforms the competition in pure *computation time* in many cases such as in the cases shown in the speedup plots in the appendix in Figure 7.

9. RELATED WORK

Israt et al. [12] have a similar implementation to *GPU-Preprocessing* but in contrast to ours support multiple GPUs in parallel. They pre-compute some *actives rows* that the algorithm loads into memory before performing the SDDMM kernel, which as discussed multiple times by now, only pays out if the operation is repeated multiple times.

Hong et al. [9] address the challenge that reusing data with tiling is tricky due to the irregular and matrix dependent access patterns of sparse matrix multiplication. They therefore pre-arrange columns inside a matrix and divide the matrix into a first more dense region and a second more sparse region, resulting in a method the authors refer to as *adaptive tiling*. The dense region can be considered as performing simple SDDMM where techniques as tiling can improve performance while the sparse region can be considered as performing a product of a sparse and dense matrix (SpMM). The process of splitting the matrix and rearranging the columns incurs a pre-processing overhead which is only worth if the operation is repeated multiple times.

cuSPARSE [6] contains NVIDIAs implementation of the SDDMM kernel and is internally used by Torch. Unfortunately, cuSPARSE is a closed source library which makes it a black-box to us and means that we cannot state more about it.

10. FUTURE WORK

GPU-Dynamic is an algorithm paradigm which optimizes for two disjoint sets of matrices. It takes advantage of the fact that each algorithm can optimize for a specific configuration which enables optimizations otherwise not possible. At the moment, it is not clear how to decide the best number of disjoint sets. Also, figuring out when to use each algorithm is something we would like to address next. Another weakness of the current *GPU-Dynamic* implementation is that it runs only on a single GPU at the same time. Leveraging multiple GPUs is a programming task we leave for future work.

11. CONCLUSION

SDDMM is a way to multiply sparse matrices, making machine learning algorithms more efficient than current methods that use sparse BLAS2 SpMV primitives. It calculates the product of two dense matrices, but only where the sparse matrix has non-zero elements. Using dense implementations and filtering the results leads to lots of redundant computations with very sparse matrices, which is why we introduce *GPU-Dynamic*, an optimized implementation that chooses the fastest algorithm at runtime using simple and fast-to-decide criteria.

12. ACKNOWLEDGEMENTS

We acknowledge access to Ault at the Swiss National Supercomputing Centre, Switzerland under ETH Zurich's share with the project ID g34. We would also like to thank Nabil Abubaker for helping us throughout the entire project and provide us with valuable feedback.

13. REFERENCES

[1] Blas (basic linear algebra subprograms). Available at https://www.netlib.org/blas/.

[2] BLEI, D. M., NG, A. Y., AND JORDAN, M. I. Latent dirichlet allocation. *J. Mach. Learn. Res. 3*, null (mar 2003), 993–1022.

[3] CANNY, J. Collaborative filtering with privacy via factor analysis. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (New York, NY, USA, 2002), SIGIR '02, Association for Computing Machinery, p. 238–245.

[4] CANNY, J. Gap: A factor model for discrete data. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (New York, NY, USA, 2004), SIGIR '04, Association for Computing Machinery, p. 122–129.

[5] CANNY, J. F., AND ZHAO, H. Bidmach: Large-scale learning with zero memory allocation.

[6] cusparse, the cuda sparse matrix library. Available at https://docs.nvidia.com/cuda/cusparse/.

[7] Deep graph library: Easy deep learning on graphs. Available at https://www.dgl.ai/.

[8] Google test. Available at http://google.github.io/googletest/.

[9] HONG, C., SUKUMARAN-RAJAM, A., NISA, I., SINGH, K., AND SADAYAPPAN, P. 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, Association for Computing Machinery, p. 300–314.

[10] KOREN, Y., BELL, R., AND VOLINSKY, C. Matrix factorization techniques for recommender systems. *Computer 42*, 8 (2009), 30–37.

[11] NG, M. K., AND ZHU, Z. Sparse matrix computation for air quality forecast data assimilation. *Numerical Algorithms 80*, 3 (Mar 2019), 687–707.

[12] NISA, I., SUKUMARAN-RAJAM, A., KURT, S. E., HONG, C., AND SADAYAPPAN, P. Sampled dense matrix multiplication for high-performance machine learning. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)* (2018), pp. 32–41.

[13] Nvidia a100 manual. Available at https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[14] Nvidia v100 manual. Available at https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[15] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., DESMAISON, A., KOPF, A., YANG, E., DEVITO, Z., RAISON, M., TEJANI, A., CHILAMKURTHY, S., STEINER, B., FANG, L., BAI, J., AND CHINTALA, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32* (2019), H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., pp. 8024–8035.

[16] RAHMAN, M. K., SUJON, M. H., AND AZAD, A. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks, 2021.

[17] Sddmm github code repository. Available at https://github.com/francois141/dphpc.

[18] Suitesparse matrix collection. Available at https://sparse.tamu.edu/.

[19] TITSIAS, M. The infinite gamma-poisson feature model. In *Advances in Neural Information Processing Systems* (2007), J. Platt, D. Koller, Y. Singer, and S. Roweis, Eds., vol. 20, Curran Associates, Inc.

[20] ZHAO, H., JIANG, B., CANNY, J. F., AND JAROS, B. Same but different: Fast and high quality gibbs parameter estimation. KDD '15, Association for Computing Machinery, p. 1495–1502.

14. APPENDIX

14.1. Scatter plots - Small matrices

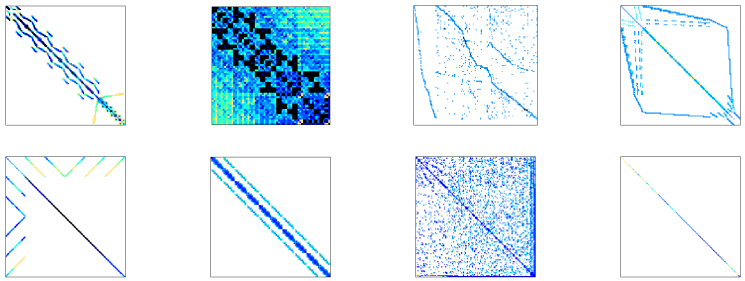

Fig. 4. (From top left to bottom right) Fluid, Oil, Biochemical, Circuit, Heat, Mass, Adder, Trackball

14.2. Scatter plots - Large, relatively sparse matrices

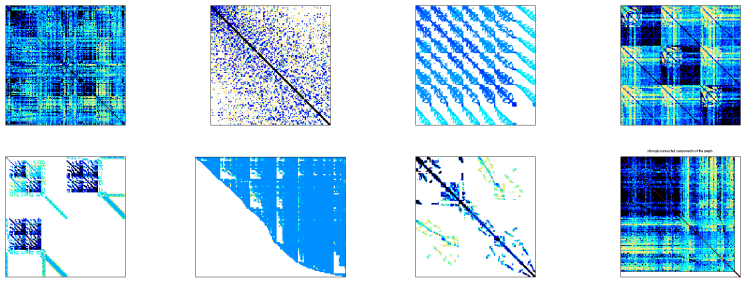

Fig. 5. (From top left to bottom right) HumanGene2, ND12K, Mix, Mecanics, Power, Combinatorics, Stress, Mouse

14.3. Scatter plots - Large, very sparse matrices

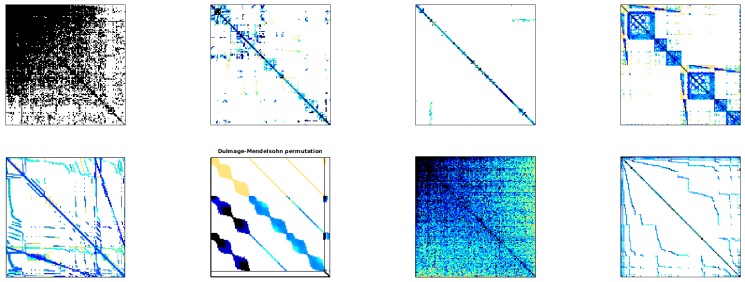

Fig. 6. (From top left to bottom right) EMail-Enron, Boeing, Boeing Diagonal, Stiffness, Semi-Conductor, VLSI, Stack-Overflow, Chip

14.4. Speedup plots

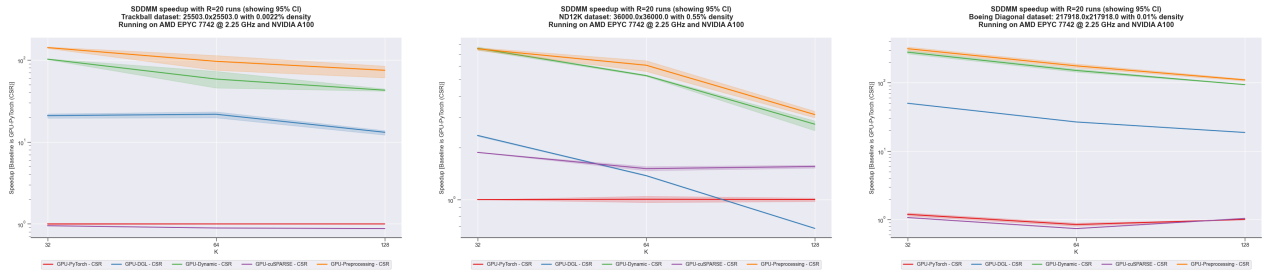

Fig. 7. Selection of speedup plots on the A100 GPU grouped by matrix categories: *Small matrix - Trackball* (left), *Large relatively sparse matrix - ND12K* (middle), *Large very sparse matrix - Boeing Diagonal* (right).

14.5. Roofline plots

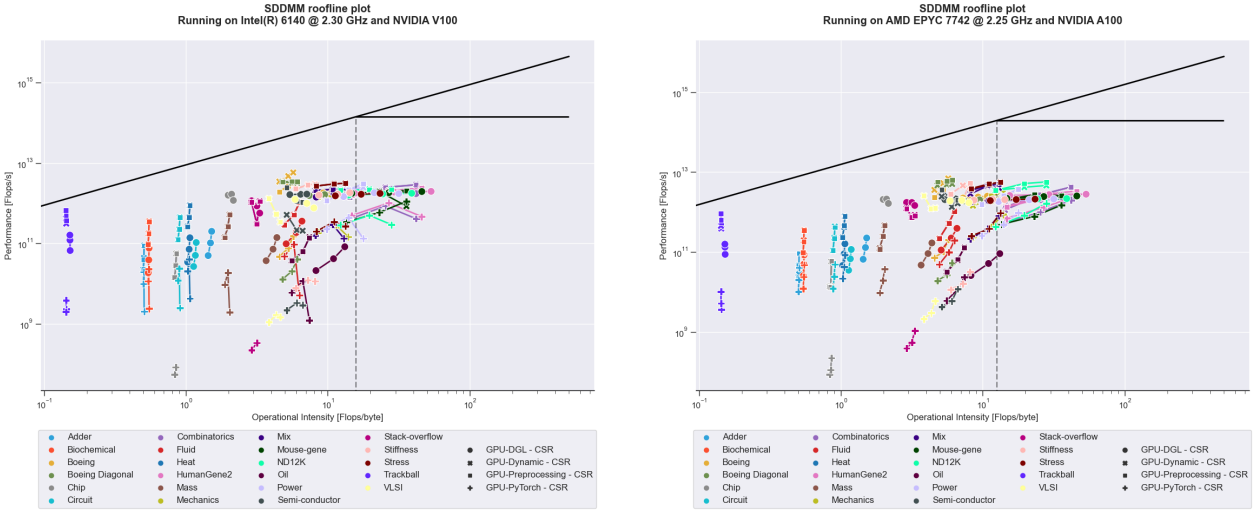

Fig. 8. Roofline plots on the V100 GPU (left) and A100 GPU (right) over all datasets and all implementations.

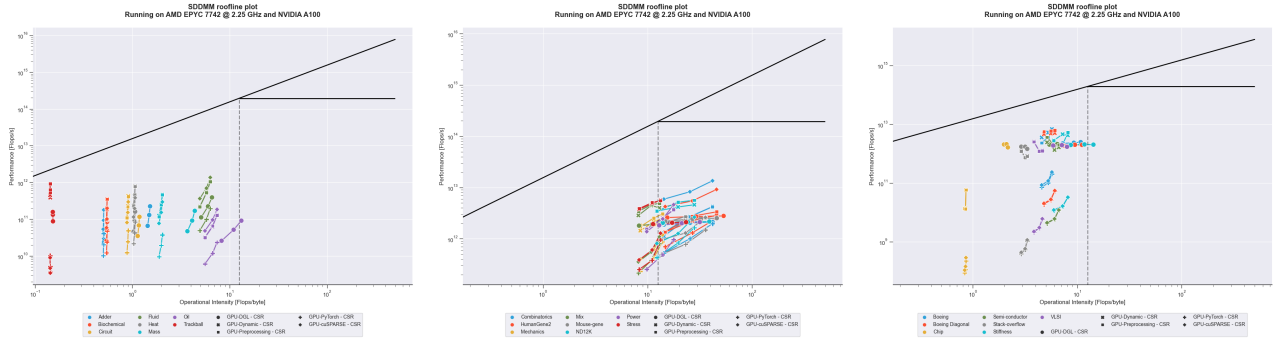

Fig. 9. Roofline plots on the A100 GPU over all implementations grouped by matrix categories: *Small matrices* (left), *Large relatively sparse matrices* (middle), *Large very sparse matrices* (right).

Dataset	Rows	Cols	Non-Zeros	Density
Fluid	656	656	18’964	4.4%
Oil	66	66	4’356	100%
Biochemical	1’922	1’922	4’335	0.1%
Circuit	1’220	1’220	5’860	0.39%
Heat	1’794	1’794	7’764	0.24%
Mass	420	420	7’860	4.45%
Adder	1’813	1’813	11’246	0.34%
Trackball	25’503	25’503	15’525	0.01%
HumanGene2	14’340	14’340	18’068’388	8.8%
ND12K	3’600	3’600	14’420’946	1%
Mix	29’957	29’957	1’990’919	0.22%
Mechanics	29’067	29’067	2’081’063	0.24%
Power	8’140	8’140	2’012’833	3.03%
Combinatorics	4’562	5’761	2’462’970	9.37%
Stress	25’710	25’710	3’749’582	0.56%
Mouse	45’101	45’101	28’967’291	1.42%
EMail-Enron	36’692	36’692	367’662	0.027%
Boeing	52’329	52’329	2’600’295	0.09%
Boeing Diagonal	217’918	217’918	11’524’432	0.02%
Stiffness	503’712	503’712	36’816’170	0.014%
Semi-Conductor	1’090’664	1’090’664	34’767’207	0.0029%
VLSI	1’453’908	1’453’908	37’475’646	0.0017%
Stack-Overflow	2’601’977	2’601’977	36’233’450	0.00053%
Chip	2’987’012	2’987’012	26’621’983	0.00029%

Table 1. Summary of the matrices used for measures. *Small matrices* (top), *Large relatively sparse matrices* (middle), *Large very sparse matrices* (bottom)