Exploiting Online Locality and Reduction Parallelism for Sampled Dense Matrix Multiplication on GPUs

Zhongming Yu*, Guohao Dai*, Guyue Huang†, Yu Wang*, Huazhong Yang*
* Department of Electrical Engineering, Tsinghua University, Beijing, China
† Department of Electrical and Computer Engineering, UC Santa Barbara, California, US
yzm18@mails.tsinghua.edu.cn, daiguohao@mail.tsinghua.edu.cn, guyue@ucsb.edu, {yu-wang, yanghz}@tsinghua.edu.cn

***Abstract*—Sampled Dense-Dense Matrix Multiplication (SDDMM) is a core component of many machine learning systems. SDDMM exposes a substantial amount of parallelism that favors throughput-oriented architectures like the GPU. However, accelerating it on GPUs is challenging in two aspects: the poor memory access locality caused by the sparse sampling matrix with the poor parallelism caused by the dot-product reduction of vectors in two dense matrices.**

To address both challenges, we present PRedS to boost SDDMM efficiency with a suite of Parallel Reduction Scheduling optimizations. PRedS uses Vectorized Coarsen 1-Dimensional Tiling (VCT) to benefit the online locality of loading the dense matrix. PRedS uses Integrated Interleaving Reduction (IIR) to increase thread occupancy in the parallel reduction. PRedS also leverages Warp-Merged Tiling (WMT) to preserve occupancy and parallelism when reducing very long arrays. Enhanced with GPU-intrinsic vectorized memory loading, PRedS achieves a geometric speedup of 29.20× compared to the vendor library. PRedS achieves up to 8.31× speedup over state-of-the-art implementations on the SuiteSparse benchmark.

***Index Terms*—HPC, GPU, kernel optimization**

I. Introduction

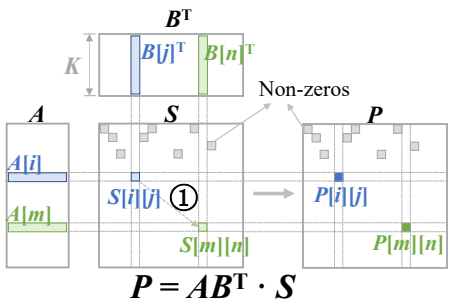
Sampled Dense-Dense Matrix Multiplication (SDDMM) performs a matrix multiplication between two dense matrices A and B, and sample the output with a sparse matrix S, yielding another sparse matrix P with the same non-zero locations of S, as shown in Fig. 1(a). SDDMM is implemented in many machine learning systems (e.g., recommendation [1]). It is used by many iterative matrix factorization algorithms such as Sparse Factor Analysis (SFA) [2], Latent Dirichlet Allocation (LDA) [3, 4] and Cyclic Coordinate Descent (CCD++) [5]. SDDMM is a core primitive in Tensor Algebra Compiler (TACO) [6]. It is also one of the core operators in Deep Graph Library (DGL) [7], a popular GNN systematic framework. Thus, speeding up SDDMM can potentially improve the performance of many of the algorithms mentioned before.

SDDMM exposes a substantial amount of parallelism because it needs to compute the dot product of every non-zero entry. It is also a bandwidth-bound kernel due to the substantial memory access of the dense matrices. Implementing SDDMM on GPUs with high performance has several challenges. Firstly, the access pattern of dense matrices A and B shows poor locality due to the sparsity of the sampling matrix S. As shown in Fig. 1(a), sparsely distributed elements in S cause irregular access to the dense matrices. Secondly, doing the multiple dot products together, as shown in Fig. 1(b), still lacks researched methodologies.

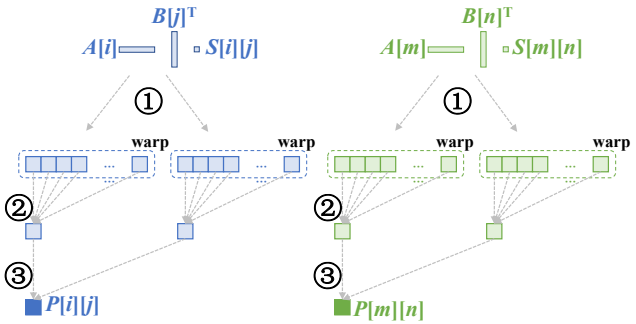
To tackle these problems, we propose PRedS, a high-performance SDDMM kernel with three dedicated schedule transformations detailed in Section III. Instead of using *offline* sparse format transformation, we show that the SDDMM performance can be greatly benefited by *online* hardware-aware, which is not conflict with offline preprocess. Schedule transformations (e.g., loop reorder, loop unroll, thread coarsening) are important techniques in parallel computing, extensively studied for dense matrix kernels [8], DNN kernels [9], image processing [10] and data processing [11]. We conduct extensive experiments on the 956 matrices from the SuiteSparse [12] benchmark to demonstrate the effectiveness of our techniques. The contributions of this paper are summarized below.

- To exploit the online locality without time-consuming preprocessing, we propose Vectorized Coarse 1-Dimensional Tiling (VCT) for SDDMM. VCT improves memory access locality without offline transformation into a non-standard sparse matrix format. Compared to baseline, VCT reduces the LLC miss bytes by up to 59%, and brings average 11.63×, 3.72× speedup on Tesla V100 and RTX 2080, respectively.
- To improve the parallelism for reduction operation, we propose Integrated Interleaving Reduction (IIR) and Warp-Merged Tiling (WMT) for SDDMM. IIR and WMT increase parallelism in the dot-product reduction through intra-warp and inter-warp workload merging, respectively. IIR brings 1.20×, 2.09× speedup on Tesla V100 and RTX 2080 on average for reduction dimension 32, respectively. WMT further improves the performance by 1.08∼1.16× in geometric mean for reduction dimension 256. To our best knowledge, this is the first work discussing the reduction scheduling for SDDMM.
- We present experimental results on 956 matrices of SuiteSparse [12] benchmark. PRedS achieves a geometric speedup of 29.20× compared to cuSPARSE [13]. PRedS surpasses the state-of-the-art SDDMM implementation ASpT [14] by average 1.03∼1.76×. Also, PRedS achieves up to 8.31× speedup compared to ASpT.

We organize the rest of the paper as follows: Section II elaborates on preliminaries on GPU architecture and relative programming techniques. Section III presents the three



(a)



(b)

Step	Challenge	Solution
① Sparse data loading	Poor locality and redundant data loading (Figure 2(a))	Vectorized Coarse 1-D Tiling (VCT, Section III-A, Figure 2(b))
② Intra-warp reduction	Low utilization of threads in a warp (Figure 3(a))	Integrated Interleaving Reduction (IIR, Section III-B, Figure 3(b))
③ Inter-warp reduction	Heavy inter-warp communication overhead (Figure 4(a))	Warp-Merging Tiling (WMT, Section III-C, Figure 4(b))

(c)

Fig. 1. The overview of the SDDMM operation on GPUs. (a) Calculating the result matrix P using two dense matrix A, B and a sparse matrix S. $P[i][j]$ equals to the dot product of $A[i]$ and $B[j]$ if $S[i][j]$ is not zero. (b) Three steps of implementing the SDDMM operation on GPUs. Step ①: Loading corresponding rows/columns in A and B according to the non-zero elements in the sparse matrix S. Step ②: Calculating the partial sum in a GPU warp, which is detailed in Section II-A. Step ③: Sum-up all partial sums in different warps of a same vector pair. (c) Challenges of each step in previous works, and the proposed solutions in our work. Corresponding sections and diagrams in this paper are labeled.

schedule-level designs adopted by PRedS. Section IV shows experimental results and we conclude in Section V.

II. Background

A. GPU Preliminary

1) Parallel Execution on GPUs: We use CUDA as our programming interface for NVIDIA GPUs. CUDA abstracts parallel programs into *blocks* of *threads*. The threads in the block are handed over to the same streaming multiprocessor (SM) in the GPU hardware for execution. The SM would execute threads in the Single Instruction, Multiple Thread (SIMT) model. In this model, the multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. [15] Threads in one threads block would map to warps according to thread ID. The warp scheduler picks an eligible warp and executes all threads in the warp. If any of the threads in the executing warp stalls, the warp scheduler would make it inactive. In addition, when the SM needs to execute one or more thread blocks, it follows the rule of the threads block scheduler. [16]

2) Memory Hierarchy on GPUs: In the CUDA programming model, the GPU memory system is composed of global memory and shared memory. The global memory is physically composed of the DRAM, the L2 cache shared by all SMs, and the L1 cache owned by each SM. [17] Specifically, data stored in the SM independent L1 cache would reduce redundant loading from DRAM.

3) Hardware Intrinsic: We leverage some hardware-intrinsic features in CUDA to accelerate SDDMM, such as vectorized memory access and warp-level primitives. With vectorized memory access, a thread could load up to 128 bits of memory in one instruction, a granularity appropriate for L1 data cache [18]. Vectorized memory reduces the total amount of instructions and improves bandwidth utilization [19]. With warp-level primitives, threads within a warp can process local variables in specific patterns efficiently [15]. By using these primitives, we could build a high-performance tree reduction to calculate dot production, on which we would elaborate in section 1 III-B.

B. Related Work

1) SDDMM on GPUs: Different from designing a high-perf accelerator [20], optimizing the SDDMM performance on GPU has to be aware of the hardware's characteristics. Previous studies have addressed the locality and parallelism issue by various methods. BIDMach [21] is a GPU-accelerated machine learning framework, providing the SDDMM kernel based on widely-used sparse matrix formats such as COO (Coordinate) or CSR (Compressed Sparse Row). It uses atomic operations to reduce results from different warps, leading to unsatisfactory performance. Some recent efforts focus on improving the performance of SDDMM by exploring different tiling methods. Nisa et al. [1] study how to apply different types of streaming to tile matrices by leveraging the shared memory capacity. Gale et al. [22] proposes new techniques to enable the use of vector instruction on misaligned memory addresses. Hong et al. [14] use the adaptive tiling method to identify condensed blocks of the sparse matrix, enabling fine-grained reuse of dense matrices tiles. Jiang et al. [23] improve the ASpT with a matrix row reorder algorithm to increase the occurrence of dense blocks.

However, these related works didn't fully release the potential of GPU's architecture while considering the SDDMM calculation features. [1] uses the matrix tiling method similar to that of GEMM [24]. [22] overlooks the reduction optimization brought by multiple dot products. Although implementations

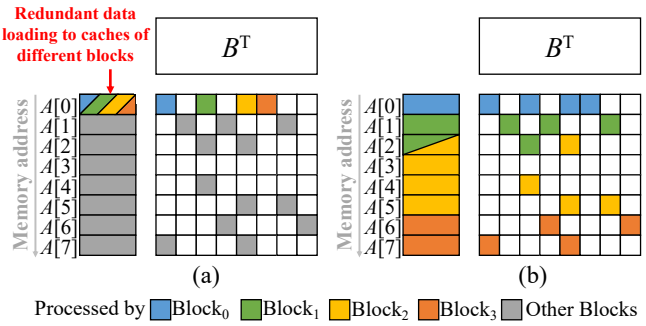


Fig. 2. Examples of data loading patterns by different blocks. (a) Four consecutive non-zero elements are processed by different blocks, leading to pool cache utilization. (b) Four consecutive non-zero elements are processed by a same block (Block_0), thus the corresponding row is only loaded by Block_0.

such as [14] and [23] surpass cuSPARSE in performance, they require preprocessing on the sparse matrix. Our work does not rely on *offline* preprocessing on the sparse matrix, but exploits hardware-aware scheduling to boost *online* performance, which are orthogonal to offline optimization techniques.

2) Parallel Reduction on GPUs: Reduction is common in all kinds of matrix multiplications. GPUs require enough parallelism to get high throughput, but it is hard to parallelize a reduction operation. Intuitively, the single output depends on all of the input elements. Previous work proposes algorithms and tuning techniques based on combinations of inner-thread loop-carried reduction, intra-warp reduction, and atomic operations in shared memory or global memory [25]. However, they couldn't fully adapt to SDDMM, which has the specialty to calculate a large number of dot products together.

III. PRedS Design

PRedS supports COO and CSR sparse formats, which adapts to formats in vendor libraries [13] and open-access toolkits [26]. They are also compatible with existing frameworks for data science [21] and graph learning [7]. COO represents a sparse matrix with three arrays: a row array, a column array, and a value array. The CSR format compressed the row array of COO to get the indices for only the first value of every row. [27] Row coordinates of CSR format could retrieve from the compressed pointer array by binary search. Note that the default triples in COO are in ascending order of rows and columns.

To accelerate SDDMM in parallel, a general organization assigns nonzeros to different blocks, in which threads are responsible for the parallel reduction. This section elaborates PRedS's key designs for SDDMM, which are Vectorized Coarsen 1-D Tiling (VCT) for better memory locality (Section III-A), Integrated Interleaving Reductions (IIR) (Section III-B) and Virtual Warp Reduction (VWR) (Section III-C) for efficient reduction.

A. Vectorized Coarsen 1-Dimensional Tiling

For data locality optimization, tiling has been widely used in many GPU-side algorithms, such as dense matrix-matrix multiplication [28] and sparse-dense matrix multiplication [14]. The effective use of data locality brings benefits like reducing global memory transactions and increasing the L1 cache hit rate, which exerts the high bandwidth advantage of the GPU memory. However, it is non-trivial to obtain a tiling method that is hardware-friendly without the overhead of preprocessing. Previous work has applied a 2-Dimensional block-wise tiling strategy to the sparse matrix [14], but it requires a dedicated sparse matrix representation. Instead of 2-Dimensional tiling, we use 1-Dimensional element-wise tiling, which does not require preprocessing and is adaptive to different matrix patterns or sizes. We introduced a Vectorized 1-Dimensional tiling method with the advantage of the spatial locality in the sparse matrix, noticing that consecutive nonzeros would locate in the same row.

There are two motivations for proposing 1-D tiling. Firstly, 1-D tiling utilizes the locality of row coordinates to reduce cache miss based on the global memory access mechanism within the warp [15]. As shown in Fig. 2, each block will request global memory and store related data in the L1 cache, which is locally exclusive in each Streaming Multiprocessor (SM). We introduce a hyper-parameter T_s, which represents the size of the tile allocated to each block. Finding an optimal T_s to utilize data locality in the sparse matrix is challenging. A larger T_s will reduce the data load of the DRAM and instead request data from the L1 cache. In other words, it makes data reuse through the L1 cache more effective. Secondly, 1-D tiling paves the way for vectorized memory loading, which could increase the utilization of memory bandwidth [19] and improved the efficiency of memory access by memory alignment.

Fig. 2 demonstrates four adjacent non-zeros elements in the same color map to one block, which explains the VCT method when the tile size equals four. When the number of elements in the sparse array is not divisible by the tile size, we need to reconsider the implementation of vectorized instructions. We use the residue handler to solve the residual elements using a non-vectorized method.

From many comparative experiments, we choose a $T_s = 16$ as a relatively good result obtained by various analysis indicators and benchmarks, to be detailed in Section IV-B as well.

B. Integrated Interleaving Reduction

Another problem we urgently need to solve is the low parallelism in reduction operation in previous work. The insight is to optimize multiple independent reduction operations cooperatively instead of focusing on a single reduction. Existing algorithms [1, 14] use the warp-level intrinsic functions provided by NVIDIA for the reduction in the dot-product, but dot-product for each output element is carried out independently, either sequentially (within a tile). We propose Integrated Interleaving Reduction (IIR), which makes a warp of threads merge different chunks in parallel possible.

To show the intra-warp reduction step, we take the case of K=32 as an example. As shown in Fig. 3(a), the traditional reduction method using the primitive `__shfl_down_sync`

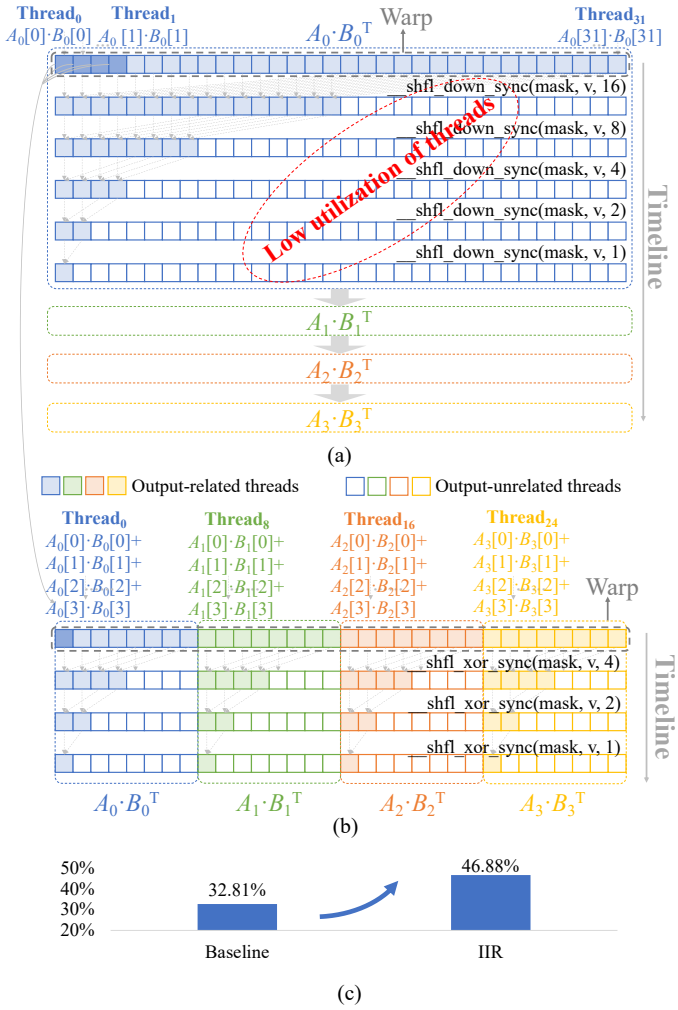


Fig. 3. Comparison of different reduction schemes in a GPU warp, using a warp with 32 threads and the reduction of vector $A_0 \sim A_3$ and $B_0 \sim B_3$ with K = 32 as the example. (a) The baseline reduction scheme. Each thread calculates the dot product of the corresponding element in A and B, then the results of 32 threads are reduced. $A_0 \sim A_3$ and $B_0 \sim B_3$ are processed by the warp sequentially. (b) The Integrated Interleaving Reduction (IIR) scheme. Each thread calculates the dot product of four element pairs in A and B. The dot of $A_0 \sim A_3$ with $B_0 \sim B_3$ are processed by the warp in parallel. (c) The comparison of thread occupancy (output-related threads) in two reduction schemes, IIR makes a better utilization of threads in a GPU warp during the intra-warp reduction step.

needs a warp of 32 threads. This method uses 32 threads to calculate the dot product, with only one value related to output in the end. This is also the baseline implementation of tree-based reduction. Unlike a generic parallel reduction, SDDMM is composed of a large number of independent reduction operations at the same time. Here we define the Shfl Coarsening Factor (SCF) as

$$SCF = number\_of\_output\_threads/warpSize.$$

to elaborate our idea. When using the *shfl* primitive model in Fig. 3(a), SCF is equal to 1/32, which means that the reduction uses 32 threads to output only one final result. Low SCF leads to an inefficient reduction in the warp. In this way, the proportion of threads used to output the dot product results is relatively low, which will make SDDMM inefficient.

Algorithm 1 SDDMM with IIR when $K = 32$

Input: S.row[nnz], S.col[nnz], S.val[nnz], A[M][K], B[N][K]
Output: P[nnz]

```
1:  if blockIdx.x < nnz/Ts then
2:      eid = blockIdx.x * (Ts/vec_size) + threadIdx.y * vec_size
3:      cid = threadIdx.x * SCF * warp_size
4:      /* Ready to load S */
5:      row[] = vec_load(S.row,eid)
6:      col[] = vec_load(S.col,eid)
7:      sum[] = vec_dot(A, B, S.row, S.col, cid)
8:      sum[] *= S.val[eid:eid+vec_size-1]
9:      /* Warp reduction */
10:     for stride = SCF*warp_size to 0 step stride/2 do
11:         sum[] = __shfl_down_sync(sum[], stride)
12:     end for
13:     /* Store */
14:     if threadIdx.x == 0 then
15:         P[eid:eid+vec_size-1] = sum[]
16:     end if
17: else
18:     residue_handler(P[eid:nnz-1], A, B, S)
19: end if
```

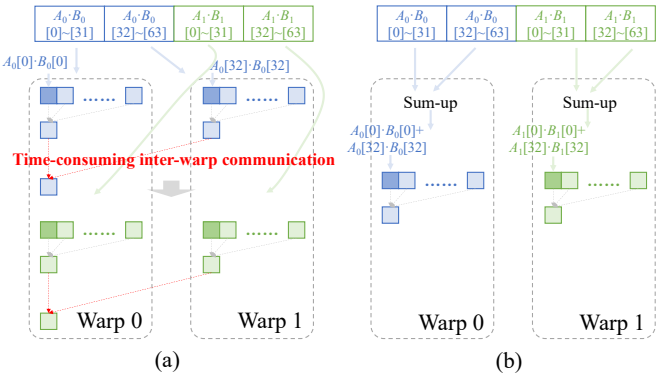


Fig. 4. Comparison of different reduction schemes using 2 warps with 32 threads each and 2 vector pairs $A_0 \sim A_1$ and $B_0 \sim B_1$ with K = 64 as the example. (a) The baseline reduction scheme. 64 threads in two warps calculates the dot product of the corresponding element in A and B, the reduction results of 2 warps are further reduced using inter-warps communication schemes (e.g., atomic add or shared memory). $A_0 \sim A_1$ and $B_0 \sim B_1$ are processed by the warp sequentially. (b) Warp-Merged Tiling (WMT) scheme. Each thread calculates the dot product of 2 element pairs in A and B. Thus, $A_0 \sim A_1$ and $B_0 \sim B_1$ are processed by the warp in parallel without inter-warps communication.

IIR method makes it possible for one *__shfl_down_sync* primitive to compute multiple vector dot products at the same time. As shown in Fig. 3(b), eight threads can solve a single 32-dimensional dot product reduction problem, which increased SCF by four times. Every eight threads of different colors will output a vector dot product at last. The small squares in the picture represent the intermediate value of the dot product of the vector elements.

We organize the thread model of IIR when $K = 32$ in the following form. A warp of 32 threads maps to a rectangular

Algorithm 2 WMT when K is larger than 32

Input: S.row[nnz],S.col[nnz],S.val[nnz],A[M][K],B[N][K]
Output: P[nnz]

```
1: if blockIdx.x < nnz/T_s then
2:    eid = blockIdx.x * (Ts/vec_size) + threadIdx.y * vec_size
3:    cid = threadIdx.x * SCF * warp_size
4:    sum[] = 0
5:    /* Ready to load S */
6:    row[] = Load(S.row,eid)
7:    col[] = Load(S.col,eid)
8:    /* Tiling in warp */
9:    for i = 0 to K * SCF/vec_size do
10:      sum[] = vec_dot(A, B, S.row, S.col , cid)
11:      cid += vec_size/SCF
12:   end for
13:   sum[] *= S.val[eid:eid+vec_size-1]
14:   for stride = SCF*warp_size to 0 step stride/2 do
15:      sum[] = __shfl_down_sync(sum[], stride)
16:   end for
17:   if threadIdx.x == 0 then
18:      P[eid:eid+vec_size-1] = sum[]
19:   end if
20: else
21:   residue_handler(P[eid:nnz-1], A, B, S)
22: end if
```

block pattern, where blockDim.x is eight, and blockDim.y is four. The calculation of these $T_s = 16$ nonzero elements can be divided into $T_s/vec\_size = 4$ groups, and each group will share the same reduction model as shown in Fig. 3(b).

To clearly describe our overall process, we show the pseudo-code of IIR under the $K = 32$ conditions in Algorithm 1. We exhibit the results of our experimental improvement in a subsequent experiment section.

C. Warp-Merged Tiling

To scale SDDMM to larger K, we implement the WMT method to tile the dense matrix into slices. 32 is one threshold because it exactly matches the number of threads in the warp. However, if K reaches a certain level, inter-warp communication will be inevitable. Previous work used shared memory or atomic operations[28, 29] for the inter-warp communication. However, atomic addition needs to ensure that the calculation process of each warp is the same, which will bring additional memory and instruction overhead. Shared memory would bring additional costs in dealing with synchronization. WMT overcomes these obstacles and brings several benefits.

As shown in Fig. 4, our idea is to divide K into multiple parts compatible with the warp. Algorithm 2 demonstrates our pseudo-code when K is more than 32.

Firstly, as shown in Algorithm 2, the sparse matrix data only needs to be loaded once in each warp, which reduces redundant memory transactions. Secondly, we move the *shfl* part and the multiplication part in IIR outside the loop, which

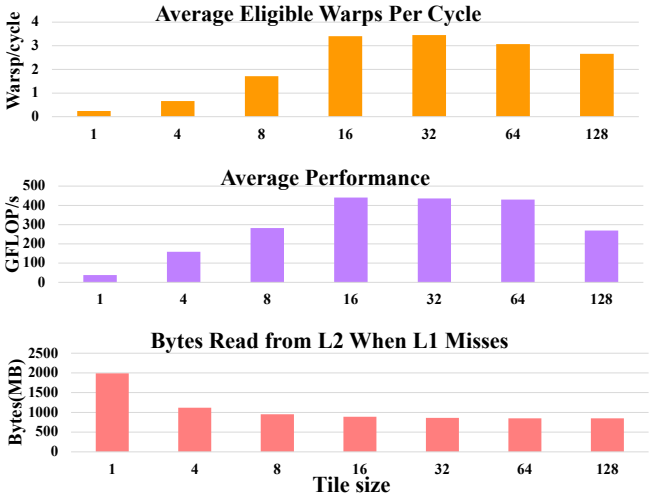


Fig. 5. Throughput performance (GFLOP/s), Bytes read from L2 for misses in L1, eligible warps per cycle under different slice sizes. To a large extent, these two metrics have a strong correlation. Experiment data is based on Tesla V100.

reduces the number of instructions and workload. Thus, we merge the calculation of multiple warps and cut the number of threads.

This method is an effective extension of both IIR and VCT, therefore inherits many of the previous parallel features.

IV. Evaluation and Analysis

We conduct extensive experiments on the SDDMM design. This section shows performance comparison with various SDDMM kernels and implementation on the ML framework.

A. Experiment Setup

1) Hardware Environments: In this section, we elaborates the experimental evaluation of the parallel reduction SDDMM on two different architectures:

- NVIDIA Tesla V100 (80 Volta SMs, 16GB global memory with bandwidth of 900 GB/s, 6MB L2 cache, 128KB L1/shared memory unified cache per each SM).
- NVIDIA RTX 2080 (72 Turing SMs, 8GB global memory, with bandwidth of 448 GB/s, 4M L2 cache and 96 KB L1/shared memory unified cache per each SM).

The code is compiled by NVCC 10.1 with the -O3 flag and ECC off. CUDA and cuSPARSE library [13] are both in version 10.1. We carry 100 tests to get the average execution time. In time statistics, we don't consider the data transfer time from CPU to GPU. We calculate the throughput by this formula: $2 * nnz * K/time$. Since the Turing architecture does not support the nvprof for operator analysis, we only implement our results on the Tesla V100 GPU.

2) Matrices Benchmark: We use data in the SuiteSparse collection [12], whose matrices come from different application domains. We selected 956 matrices from SuiteSparse with at least 10K rows and 100K non-zeros.

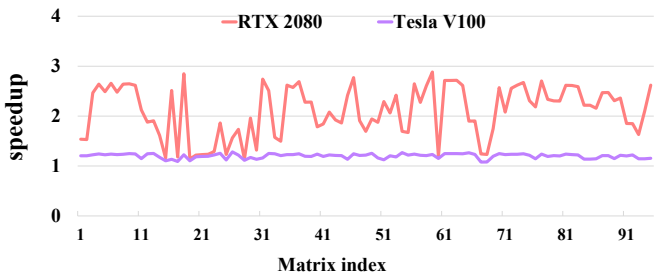


Fig. 6. The throughput performance (GFLOP/s) of the IIR method compared to only using VCT. Both kernels are tested under $T_s = 16$ condition. 95 matrices are randomly selected from SuiteSparse excluded those results which are out of memory.

3) Baseline: We compared our code with ASpT [14] and BIDMach [21], which represent high-performance SDDMM implementations. In addition, we selected related function from the NVIDIA cuSPARSE library [13] as our baseline.

- cuSPARSE Here we use a variant of SDDMM provided in cuSPARSE Library [13], which only samples the output of A multiplies B with the sparse matrix pattern and does not perform scaling calculations.
- BIDMach [21]: It is an efficient toolkit for machine learning with several parallel algorithms. It has built-in efficient kernels for SDDMM on GPU, which is the core of SFA [2] and LDA [3, 4] algorithms.
- Sputnik [22]: It provides SpMM and SDDMM kernels which are targeted specifically at deep learning applications on GPUs. It is based on CSR format and does not enforce any structure on the topology of nonzero values.
- ASpT [14]: It is a state-of-the-art tiling method that makes the SDDMM kernel of high performance. Although reordering is an important part of ASpT, we did not include preprocessing time in our evaluation. In their open-source code, we can only run SDDMM with K=32 and K=128.

B. Benefits of SDDMM Design

1) Benefits of Vectorized coarsen 1-D tiling: The purpose of VCT is to reduce cache miss and improve bandwidth efficiency. Different tile sizes have different effects. Here we demonstrate why we choose $T_s = 16$ in the following experiments. We compare the tile size T_s from 1, 4, 8, 16, 32, 64, 128 as representatives and use multiple indicators for evaluation. Our data is an average of over 956 matrices from the SuiteSparse dataset [12].

First, we analyze our kernels with different tile sizes by metric *l2_global_load_bytes* represents *Bytes read from L2 for misses in L1 for global loads*, which aims to prove the improvement of the L1 hit rate.

As shown in Fig. 5, the L1 cache miss drops when the tile size increases from 1 to 16. In theory, expanding the tile size will make more effective use of data locality and reduce cache misses. However, experiments have shown that continuing to increase the tile size will not achieve better performance.

We analyze this phenomenon from another metric called *eligible_warps_per_cycle*. It stands for the average number of warps that are eligible to issue per active cycle per SM [30]. This metric is strongly related to the whole performance. Note that this metric has a max limitation, which is bound to the number of warp schedulers. In architecture like volta, each SM has four warp schedulers.

Fig. 5 shows that eligible warps first go up to reach the max limit and then fall. As we introduced in Section II-A, the warp is the basic unit of parallel computing. The GPU effectively hides the stall caused by instructions or memory fetch through enough amount of eligible warps. Since there are limited workloads in each SM, the eligible warps are fewer in small tile size. However, mapping too many dot products to each SM would cause overload, reducing the number of eligible warps in each cycle. Also, as the tile size increases, the cost of processing residual elements will increase. Therefore, choosing $T_s = 16$ is a trade-off of multiple indicators.

2) Benefits of Interleaving Integrated Reduction: We introduce IIR to efficiently use reduction primitives and improve the SCF within a warp. We conducted comparative experiments of IIR and VCT based on $K = 32$. Fig. 6 shows the increase in throughput performance. Our result shows the geometric mean of speed up by IIR is 2.09× on RTX 2080, 1.20× on Tesla V100.

3) Benefits of Warp-merged Tiling: We propose the WMT to reduce the communication overhead between threads while merging the repetitive parts of the algorithm.

Machine	slowdown		speedup		
	<-10%	-10∼0%	0∼50%	50-100%	>1x
RTX 2080	3%	23.5%	67.7%	5.8%	0%
V100	4.4%	25.5%	67.0%	2.6%	0.5%

TABLE I
THIS TABLE SHOWS THE SPEEDUP DISTRIBUTION ON THE SUITESPARSE DATASET (956 MATRICES) WHEN COMPARING THE WMT METHOD WITH THE ATOMIC ADD METHOD. NOTE THAT THE ATOMIC ADD METHOD ALSO IMPLEMENT VCT AND IIR TO CONTROL VARIABLE.

Machine	RTX 2080		Tesla V100	
Baseline	cuSPARSE	BIDMach	cuSPARSE	BIDMach
K=32	29.20	5.20	16.11	3.32
K=64	17.46	4.27	15.92	3.96
K=128	11.51	4.57	12.38	2.76
K=256	10.52	2.56	13.42	4.13
K=512	10.91	2.45	15.86	2.56
K=1024	11.24	2.42	-	2.53

TABLE II
AVERAGE SPEEDUP AGAINST THE BASELINES WITHOUT PREPROCESSING (CUSPARSE [13] AND BIDMACH [21]).

As shown in Table I, we carry out comparative experiments under the condition of $K = 256$. The baseline is set up as an atomic add method across warps, with eight warps mapping to a thread block.

C. Overall Performance of SDDMM Kernel

As described in the background Section IV-A3, the first three kernels have no preprocessing format, while the latter one utilizes an optimized reordering format. Thus, we first compare kernels without preprocessing in the following part.

1) Comparison with non-preprocessed approaches: Fig. 7 shows the performance comparison of PRedS and non-

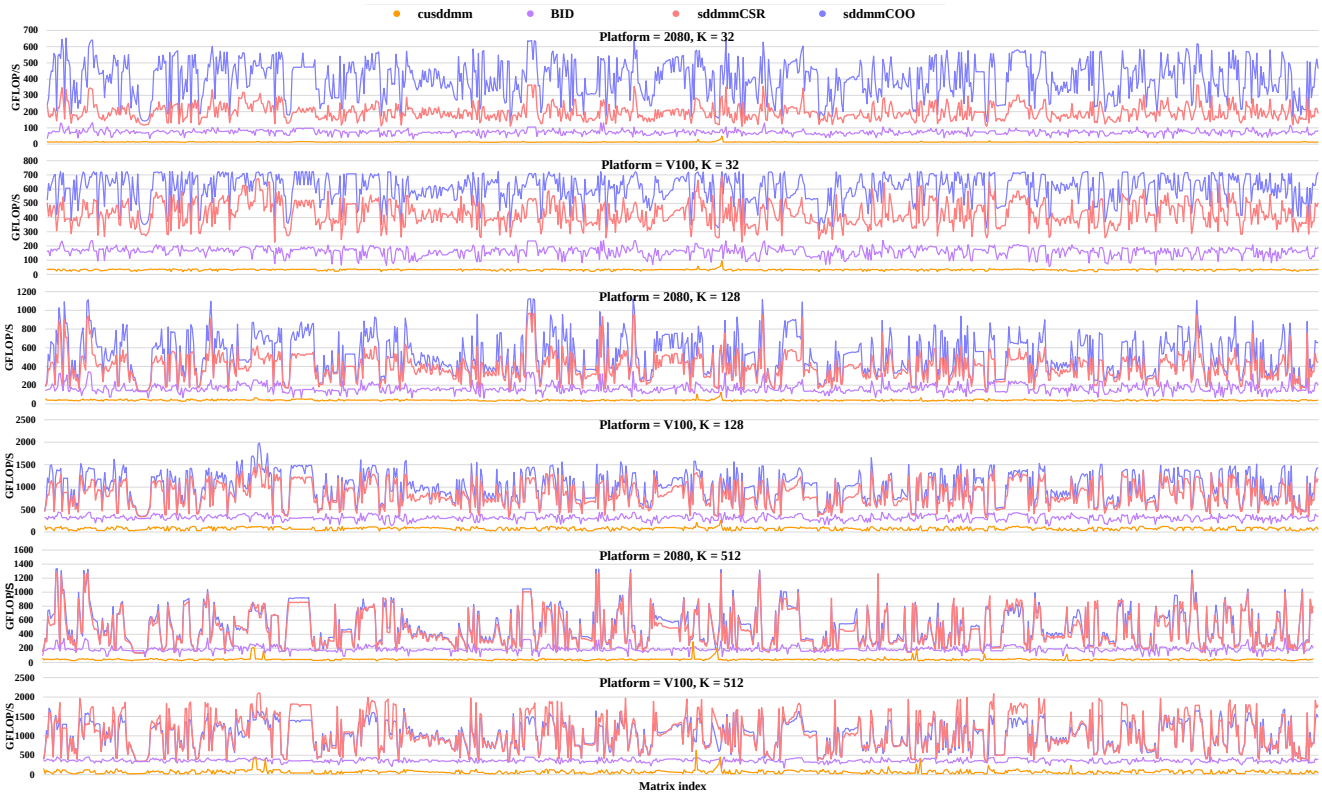


Fig. 7. SDDMM results compared with non-preprocessed kernels. We choose to display data with K equal to 32, 128, 512.

preprocessed baselines on the SuiteSparse dataset. To summarize, we list the average performance gain in Table II. PRedS achieves a geometric speedup of 29.20× and 5.20× compared to cuSPARSE and BIDMach on two GPUs. Our experiments with Sputnik [22] results are shown separately in Table III. The SDDMM kernel of Sputnik [22] is unusually slow when K is equal to 32 so that we have an average speedup of 167.7× compared with it. (test on machine Tesla V100).

Machine		speedup		
		<10x	10-100x	>100x
K=32	RTX 2080	3.6%	40.7%	55.7%
	V100	2.0%	31.9%	66.1%
K=1024	RTX 2080	54.5%	44.2%	1.3%
	V100	33.5%	61.9%	4.6%

TABLE III
SDDMM RELATIVE SPEEDUP PERCENTAGE COMPARED WITH SPUTNIK [22].

2) Comparison with preprocess-based approaches: Because ASpT [14] only provides K=32 and K=128, we show Table IV in these two cases. Note that ASpT relies on preprocessing of the sparse matrix. If we implement the SDDMM in the ML system, we must consider the overhead of preprocessing, which takes up nearly 50 percent of the computing time. However, in this work, we only calculate the computing time compared to ASpT. Thus, we could perform much better than ASpT when considering the system's requirements.

The table shows the average and maximum acceleration for ASpT's computing time. Our kernel performs competitively with the approach of ASpT, gaining up to 1.76x speedup on average without reordering the sparse matrix.

Machine	K=32		K=128	
	Average	Max	Average	Max
RTX 2080	1.76	7.48	1.03	5.17
Tesla V100	1.23	8.14	1.30	8.31

TABLE IV
SPEEDUP AGAINST ASPT [14]

D. End-to-End Performance for ML applications

Our kernel could accelerate machine learning applications with widely used formats that make it easy to implement in existing frameworks. As introduced in Section I, we embed PRedS in BIDMach. We test application speedup on LDA, SFA which involve SDDMM operations with matrices in Table V. The results of our embedded kernel are showed in Fig. 8.

Matrices	non-zeros	# rows	# columns
nytimes	69679427	102660	300000
nips	746316	12419	1500

TABLE V
THE TEST MATRICES IN THE MACHINE LEARNING ALGORITHM.

Although most of the time cost by data transmission, we still reduce the LDA application time max of 44.8%, with an average reduction of 19.8%. We reduce the SFA application time max of 30.1%, with an average of 16.1%.

V. CONCLUSION

In this paper, we propose an efficient SDDMM design on GPUs, PRedS considers requirements by Machine learning applications with no preprocessing. PRedS introduces three techniques: Vectorized Coarsen 1-D Tiling to reduce the

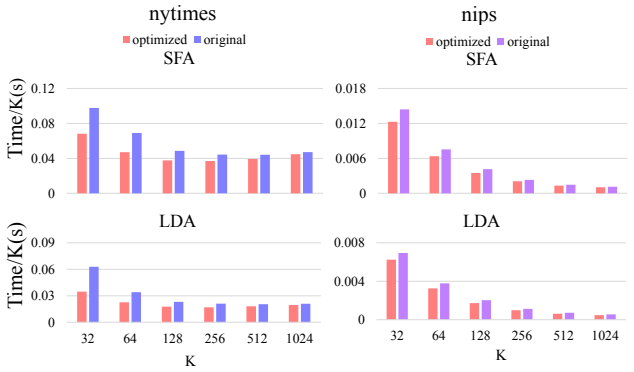


Fig. 8. Optimized method stands for the system embedded with our PRedS. Original method is the system with `__dds` kernel.[4] We record the total time spent by applications such as LDA, SFA. Experiments are carried out on RTX 2080. The Y-axis stands for the average running time of the application divided by K.

cache miss, Integrated Interleaving Reduction to improve inter-warp parallelism, and Warp Merged Tiling to reduce inter-warp communication overhead. PRedS achieves a geometric speedup of 29.20× compared to cuSPARSE. Also, PRedS achieves up to 8.31× speedup compared to the state-of-art implementation ASpT [14].

VI. Acknowledgement

This work is supported by National Natural Science Foundation of China (No. U19B2019, 61832007), China Postdoctoral Science Foundation (No. 2019M660641), Tsinghua EE Xilinx AI Research Fund, Beijing National Research Center for Information Science and Technology (BNRist), and Beijing Innovation Center for Future Chips. This work is also supported by Biren Technology. This work is included in the dgSPARSE project [31].

References

[1] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong, and P. Sadayappan. Sampled dense matrix multiplication for high-performance machine learning. In *HiPC*, pages 32–41, 2018.

[2] John Canny. Collaborative filtering with privacy. In *IEEE Symposium on Security and Privacy*, pages 45–57. IEEE, 2002.

[3] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[4] Huasha Zhao, Biye Jiang, John F Canny, and Bobby Jaros. Same but different: Fast and high quality gibbs parameter estimation. In *SIGKDD*, pages 1495–1502, 2015.

[5] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon. Parallel matrix factorization for recommender systems. *Knowledge and Information Systems*, 41(3):793–819, 2014.

[6] Fredrik Kjolstad, Stephen Chou, David Lugato, Shoaib Kamil, and Saman Amarasinghe. Taco: A tool to generate tensor algebra kernels. In *ASE*, pages 943–948. IEEE, 2017.

[7] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019.

[8] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *SC*, pages 1–11. IEEE, 2008.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, pages 578–594, 2018.

[10] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM Sigplan Notices*, 48(6):519–530, 2013.

[11] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel ir for high-performance gpu code generation. In *CGO*, pages 74–85. IEEE, 2017.

[12] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM TOMS*, 38(1):1–25, 2011.

[13] M Naumov, LS Chien, P Vandermersch, and U Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[14] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *PPoPP*, page 300–314, New York, NY, USA, 2019. Association for Computing Machinery.

[15] NVIDIA Corporation. Nvidia cuda c++ programming guide, 2021. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[16] Sreepathi Pai. How the fermi thread block scheduler works, 2014. https://www.cs.rochester.edu/~sree/fermi-tbs/fermi-tbs.html.

[17] NVIDIA Corporation. Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[18] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*, 2018.

[19] Justin Luitjens. Cuda pro tip: Increase performance with vectorized memory access, 2013. https://developer.nvidia.com/blog/cuda-pro-tip-increase-performance-with-vectorized-memory-access/.

[20] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. Extensor: An accelerator for sparse tensor algebra. In *MICRO*, MICRO '52, page 319–333, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Huasha Zhao and John F. Canny. *High Performance Machine Learning through Codesign and Rooflining*. PhD thesis, EECS Department, University of California, Berkeley, Sep 2014.

[22] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *SC*. IEEE Press, 2020.

[23] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus. In *PPoPP*, pages 376–388, 2020.

[24] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. Autotuning gemm kernels for the fermi gpu. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 2012.

[25] Walid Abdala Rfaei Jradi, Hugo Nascimento, and Wellington S Martins. A fast and generic gpu-based parallel reduction implementation. In *WSCAD*, pages 16–22. IEEE, 2018.

[26] Eric Jones, Travis Oliphant, Pearu Peterson, et al. Scipy: Open source scientific tools for python. 2001.

[27] Joseph L. Greathouse and Mayank Daga. Efficient sparse matrix-vector multiplication on gpus using the csr storage format. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 769–780, 2014.

[28] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.

[29] Justin Luitjens. Faster parallel reductions on kepler, 2014. https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/.

[30] NVIDIA Corporation. Nvidia cuda profiler user's guide, 2020. https://docs.nvidia.com/cuda/profiler-users-guide/.

[31] Guohao Dai, Guyue Huang, Zhongming Yu, Hengrui Zhang, Shang Yang, and Yu Wang. dgsparse: Easy and fast sparse graph processing, 2021. https://dgsparse.github.io/.