Sampled Dense Matrix Multiplication for High-Performance Machine Learning

Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, P. Sadayappan
Dept of Computer Science and Engineering
The Ohio State University
Columbus, OH
{nisa.1, sukumaranrajam.1, kurt.29, hong.589, sadayappan.1}@osu.edu

***Abstract*—Many machine learning methods involve iterative optimization and are amenable to a variety of alternate formulations. Many currently popular formulations for some machine learning methods based on core operations that essentially correspond to sparse matrix-vector products. A reformulation using sparse matrix-matrix products primitives can potentially enable significant performance enhancement.**

Sampled Dense-Dense Matrix Multiplication (SDDMM) is a primitive that has been shown to be usable as a core component in reformulations of many machine learning factor analysis algorithms such as Alternating Least Squares (ALS), Latent Dirichlet Allocation (LDA), Sparse Factor Analysis (SFA), and Gamma Poisson (GaP). It requires the computation of the product of two input dense matrices but only at locations of the result matrix corresponding to nonzero entries in a sparse third input matrix.

In this paper, we address the development of cuSDDMM, a multi-node GPU-accelerated implementation for SDDMM. We analyze the data reuse characteristics of SDDMM and develop a model-driven strategy for choice of tiling permutation and tile-size choice. cuSDDMM improves significantly (upto 4.6x) over the best currently available GPU implementation of SDDMM (in the BIDMach Machine Learning library).

***Index Terms*—SDDMM, GPU, Optimization, Sparse matrix**

I. Introduction

Machine Learning (ML) algorithms are becoming increasingly important in modeling tasks such as classification, clustering and pattern analysis. Factorization algorithms are a class of ML algorithms used for decomposing large sparse datasets as a product of smaller dense matrices. They are used in many applications, such as collaborative filtering. Topic modeling is used for document clustering and text classification.

Sparse factorization algorithms for machine learning use iterative optimization and are amenable to a variety of alternate formulations. These alternatives are typically chosen on the basis of computational complexity in terms of arithmetic operation costs. However, data movement costs are much more constraining than execution of elementary arithmetic operations on all current computing platforms, including clusters, multicore CPUS, GPUs, FPGAs, etc. The current situation in development of many ML algorithms is analogous to the early days in the development of efficient dense linear algebra algorithms, when it was recognized that wherever possible a reformulation using BLAS3 (dense matrix-matrix product) primitives offered significant performance benefits over a BLAS2 (dense matrix-vector product) or BLAS1 (dot product) operations. As we explain in greater detail in the next section, many currently popular formulations for some machine learning methods are based on core operations that essentially correspond to sparse BLAS2 (matrix-vector product) operations. A reformulation of the algorithms using sparse matrix-matrix product primitives can potentially enable significant performance improvement.

Sampled Dense-Dense Matrix Multiplication (SDDMM) is a kernel that can be used as a core operation in a formulation of factorization algorithms like Alternating Least Squares (ALS) [1], Sparse Factor Analysis (SFA) [2], and topic modeling algorithms like Latent Dirichlet Allocation (LDA)[3], [4] and Gamma Poisson (GaP) [5]. Recent work [6] has shown how SDDMM can be used to formulate applications like matrix factorization for recommender systems (ALS) and topic modeling (LDA). However, unlike primitives like Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN), which exhibit a regular data access pattern and have also been the subject of intense efforts to develop very high-performance implementations for GPUs, much less effort has so far been directed towards the optimization of the irregular-access SDDMM kernel for GPUs.

In contrast to the much studied problem of optimizing SpMV (Sparse Matrix Vector product) problem, which features one sparse matrix and a dense vector as input and a dense vector as output, SDDMM has one sparse and two dense matrices as input and a sparse matrix as output. Thus, there are more data accesses to consider in devising an efficient parallel implementation for GPUs. But unlike SpMV, which is severely memory bandwidth limited, SDDMM has much higher performance potential. With SpMV, at most two floating-point operations (one FMA) can be executed for each sparse matrix element brought in from global memory. But with SDDMM, each input sparse matrix element is multiplied by the dot-product of a vector each from the two dense input matrices, thereby significantly raising the roofline performance limit for SDDMM in comparison to SpMV.

Thus SDDMM has significantly higher performance potential than sparse BLAS2 operations like SpMV, and the availability of a high-performance SDDMM implementation can stimulate more efficient reformulations of other ML algorithms.

In this paper, we perform an in-depth analysis of alternate sparse-tiling strategies for SDDMM, considering loop permutation choices, data buffering choices, and impact of tile sizes. After elimination of many options on the basis of the analytical modeling, we devise two GPU implementations, each suitable under different fractional non-zero density in the sparse matrix.

This paper makes the following contributions:

978-1-5386-8386-6/18/$31.00 ©2018 IEEE
DOI 10.1109/HiPC.2018.00013

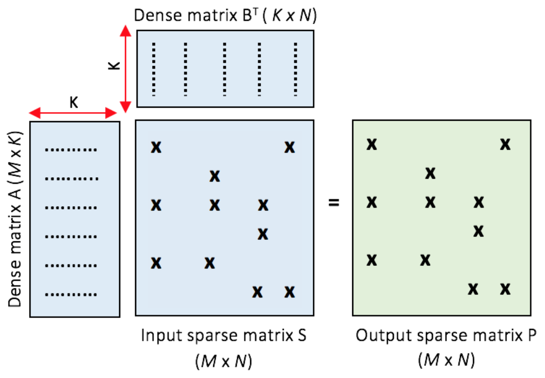

Dense matrix B^T (K x N)
K
K
Dense matrix A (M x K)
Input sparse matrix S (M x N)
=
Output sparse matrix P (M x N)

Fig. 1: SDDMM: Product of dense matrix A and B is accumulated at each non zero position of sparse matrix S to generate output sparse matrix P

- To the best of our knowledge, it presents the first detailed analysis and modeling of the performance implications of different choices for loop permutation and tile size choice for SDDMM.
- It develops an analytical model to determine the tile size based on the density of the input matrix and L2 cache capacity of the machine.
- It presents an experimental evaluation of a multi-GPU implementation (cuSDDMM) on a number of datasets, using model-predicted parameters as well as exhaustive tuning. It demonstrates significant performance improvement (up to 4.6x) for SDDMM over any existing alternative.

The rest of the paper is organized as follows: Sec. II elaborates on the potential for machine learning algorithms to be reformulated from a sparse BLAS2 core to use SDDMM. Section III presents an analysis of the data access pattern for SDDMM and discusses alternate tiling options. In Section IV, we present algorithmic details of cuSDDMM. Section V presents an experimental evaluation and compares the performance of cuSDDMM with an implementation of SDDMM in the machine learning library BIDMach [6]. Section VI presents related work and we conclude in section VII.

II. Background

SDDMM computes a dot product of two vectors at every non-zero position of a sampled matrix (e.g., S). These vectors are read from two dense matrices (e.g., A and B). The heights of A and B are defined by the height and width of S respectively, and the width by a user defined parameter K. For example, an SDDMM operation is illustrated in Figure 1 using input $A \in \mathbb{R}^{M\times K}$ and $B \in \mathbb{R}^{N\times K}$, $S \in \mathbb{R}^{M\times N}$ and output $P \in \mathbb{R}^{M\times N}$. SDDMM computes the dot product of the i^{th} row of A and j^{th} row of B for every $S_{(i,j)}$ position. A Hadamard product (element-wise multiplication) is then formed between the computed set of dot products and the sampled values of $S_{(i,j)}$. SDDMM computes:

$$P = A{*_S}B = (AB)\, o\, (S > 0) \tag{1}$$

$(S > 0)$ expresses the fact that AB is evaluated at the nonzero positions of S. The notations used in this paper are listed in Table I.

TABLE I: Notation

Name	Description
S	sparse input matrix of dimension: $M \times N$
A	dense input matrix of dimension: $M \times K$
B	dense input matrix of dimension: $N \times K$
P	sparse output matrix of dimension: $M \times N$
M	number of rows in S
N	number of columns in S
K	number of topics/features
nnz	number of non-zero element in S
$S_{(i,j)}$	non-zero element at (i, j) position in S
Ti	length of vertical tiles
Tj	length of horizontal tiles
Tk	length of K slices
$A * B$	matrix-matrix multiplication
AoB	element-wise multiplication
$A{*_S}B$	matrix-matrix multiplication at the non zero position of S

SDDMM can be performed using a dense-dense matrix multiplication (DGEMM) between A and B, followed by extraction of the sampled elements. Despite the availability of highly efficient DGEMM implementations, the excessive number of unnecessary computations make this an impractical alternative. By performing computations corresponding to only non-zero elements, the computational complexity can be reduced to $\mathcal{O}(K.nnz)$ from $\mathcal{O}(K.M.N)$.

SDDMM was first introduced as a *custom* kernel (non-standard matrix operations) in the BIDMach library by Zhao et al. [7]. BIDMach is a general purpose machine learning library targeting large-scale data, and high-performance implementations of factor analysis algorithms like ALS, SFA, LDA, and GaP have been developed using BIDMach functions. SDDMM is the computationally dominant kernel in these algorithms [7]. SDDMM has been used as a key kernel for optimizing several machine learning algorithms on parallel architectures [3], [8], [6].

A. *Reformulation of machine learning algorithms using SDDMM*

In this subsection, we elaborate on how the availability of the SDDMM kernel can enable reformulations of machine learning algorithms. We present non-SDDMM and SDDMM based formulations of matrix factorization using ALS [1], and demonstrate how the SDDMM based one has much higher potential operational intensity (OI). Currently, the non-SDDMM based algorithms are expressed using primitives that are not as efficient with respect to data movement.

Recommender systems predict a user's preference for items. One approach to recommender systems is matrix factorization. Cyclic Coordinate Descent (CCD++) [9], [10], an adaption of Alternating Least Squares (ALS) is a state-of-the-art technique for matrix factorization for recommender systems. Given a sparse ratings matrix $A \in \mathbb{R}^{M\times N}$, where M is the number of users and N is the number items, CCD++ iteratively updates two dense factor matrices $W \in \mathbb{R}^{M\times K}$ and $H \in \mathbb{R}^{N\times K}$ such that their product WH^T approximates A. W and H are the user and item matrices, respectively. K is the number of latent features.

CCD++ performs feature-wise update as shown in Alg. 1. One of the K features is selected for an update at a time and the values of the other features are treated as known and fixed. The columns of the item-feature (W) and user-feature (H) matrices corresponding to the selected feature are then

Algorithm 1: Sequential CCD++ Algorithm

```
   Input : $A, W, H, \lambda, K, T$
1  Initialize $R = A$ and $H = 0$
2  for iter = 0 to iterations do
3      for $t$ = 0 to $K$ do
4          $\hat{R}_{ij} = R_{ij} + w_{ti}h_{tj},\ \forall(i,j)\,\epsilon\,\Omega$
5          $v_j = h_t$
6          for inneriter = 0 to $T$ do
7              $u_i = \frac{\sum_{j\epsilon\Omega_i}\hat{R}_{ij}v_j}{\lambda+\sum_{j\epsilon\Omega_i}v_j^2},\ i = 1,...,m$
8              $v_j = \frac{\sum_{i\epsilon\Omega_j}\hat{R}_{ij}u_i}{\lambda+\sum_{j\epsilon\Omega_j}u_i^2},\ j = 1,...,n$
9          end
10         $w_t = u^*$ , $h_t = v^*$
11         $R_{ij} = \hat{R}_{ij} - u_i^* v_j^*,\ \forall(i,j)\,\epsilon\,\Omega$
12     end
13 end
```

updated. Each iteration of the t loop that runs over the K features requires a number of sparse BLAS2 operations: a sparse vector outer-products (lines 4 and 11), and two SpMV-like computations (lines 7 and 8) to update W and H. Each sparse BLAS2 operation requires a complete scan of the sparse matrix R, which represents the residual error matrix.

Next, we present SDDMM based computation in Equation 2 to perform the ALS computations, as previously formulated by Zhao et al. [7]. Equation 2 denotes the gradient update formula used in ALS. The right side of the equation involves an element-wise multiplication $(\beta^T *_S \gamma)$ (SDDMM) followed by a sparse-matrix dense-matrix multiplication(SpMM). λ and γ are the user and the item matrices. NVIDIA cuSPARSE library [11] provides an optimized SpMM implementation which can be used here. The lack of an optimized SDDMM function in libraries like cuSPARSE motivates the work in this paper.

$$M_i\gamma = \lambda w_\gamma o\gamma + \beta * (\beta^T *_S \gamma) \tag{2}$$

Another popular factor algorithm, Latent Dirichlet Allocation (LDA), is a statistical topic model originally proposed by Blei et al. [4]. Given a corpus of documents, LDA models latent topic distributions for each document and each word in the vocabulary based on variational inference algorithm. Each of the documents and words is parameterized as a random mixture over latent topics associated with multinomial distribution. Equation 3 denotes an SDDMM based expression to update the variational Dirichlet parameter γ for LDA. F is a variational parameter associated with a latent topic in a specific document. α and β are Dirichlet priors. The quotient in Equation 3 of S by $\beta^T *_S F$ involves an element-wise quotient followed by a sparse-dense matrix product (SpMM). The denominator of the update expression $\beta^T *_S F$ is an SDDMM operation.

$$\gamma = \alpha + Fo\left(\beta * \frac{S}{\beta^T *_S F}\right) \tag{3}$$

The above examples illustrate reformulations of machine learning algorithms in terms of the SDDMM primitive. Since SDDMM is a computationally dominant kernel (taking up to 65% of the total execution time), an optimized SDDMM kernel can aid in improving the performance of several ML algorithms. In the rest of this paper, we detail the development of cuSDDMM, a multi-GPU parallel implementation of SDDMM.

III. Algorithm Design and Analysis

A sequential SDDMM algorithm is presented in Algorithm 2, where the sparse matrices are represented in Compressed Sparse Row (CSR) format. The i-loop in line 2 iterates over the rows of S, and the j-loop in line 3 iterates over column indices of each row. The k-loop (line 4) computes the dot product of the vectors. Finally, the resulting value is scaled with the sampled value as shown in line 10. Unlike SpMV, in SDDMM, each non-zero element $S_{(i,j)}$ has a K-way reuse. Also, each element of A and B has a reuse factor of the number of non-zero elements in i^{th} row and j^{th} column of S respectively.

Algorithm 2: Sequential SDDMM

```
   input :  CSR S[M][N], float A[M][K], float B[N][K]
   output:  CSR P[M][N]
1                                                    ▷ Dot product
2  for i = 0 to M do
3      for j = S.rowptr[i] to S.rowptr[i+1] do
4          for k = 0 to K do
5              P.values[j] += A[i][k] * B[S.colidx[j]][k]
6          end
7      end
8  end
9                                                    ▷ Scaling
10 for i = 0 to S.rows do
11     for j = S.rowptr[i] to S.rowptr[i+1] do
12         P.values[j] *= S.values[j]
13     end
14 end
```

In this work, we focus on exploiting the maximum advantage of the reuse potential by maximizing data locality and reducing data movement.

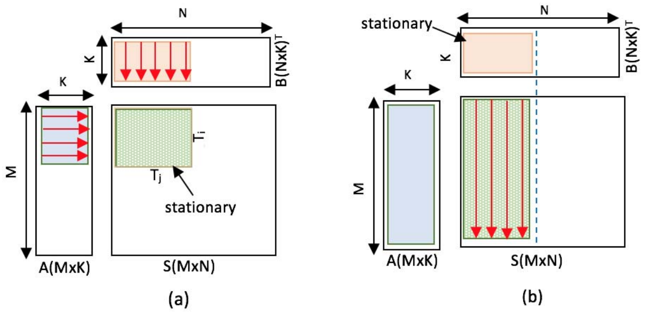


Fig. 2: Different types of streaming

Many dense matrix based algorithms such as Dense-Dense matrix multiplication (DGEMM) employ a streaming approach to reduce the volume of data movement. In the streaming approach, a slice of one of the three matrices is kept stationary in fast memory (cache, shared memory, or registers) and the corresponding slices of the other two are streamed in. The loop dimension that does not explicitly index the stationary matrix is chosen as the streaming dimension. For example, in Figure 2 (a), the sparse matrix S is kept stationary. S is indexed by dimensions i and j, and is independent of k. Hence, k is the streaming dimension. The streaming choices for SDDMM and their impact can be explained with the help of a tiled SDDMM algorithm as shown in Algorithm 3. In SDDMM, there are three choices (A, B, S or P) for the stationary matrix. In the tiled version, the streaming dimension can be represented by the innermost tile dimension.

Algorithm 3: Tiled SDDMM

```
input :  CSR S[M][N], float A[M][K], float B[N][K]
output:  CSR P[M][N]
                                                    ▷ Dot product
for ii = 0 to M step Ti do
    for jj = 0 to N step Tj do
        for kk = 0 to K step Tk do
            for i = Ti to min((ii + 1) * Ti, M) do
                for cur_elem ∈ s_tile[ii][jj] do
                    j = cur_elem.global_col
                    for k = kk to min((kk+1)* Tk, K) do
                        P.values[j] += A[i][k] * B[S.colidx[j]][k]
                    end
                end
            end
        end
    end
end
...
```

We first examine the trade-offs of selecting S as the stationary matrix and k as the streaming dimension. For simplicity of calculation, let us assume that $Ti = Tj = T$. A slice of S of size $T \times T$, is kept in the registers and the input dense matrices (A and B) are streamed along k (typically through shared memory). For each k, it forms an outer product of slice $1 \times T$ and $T \times 1$ from A and B, followed by a reduction in registers with previous outer products. Thus, the elements of S and P get full reuse, whereas A and B only get reuse within a block. Figure 2(a) illustrates the method.

To get the maximum benefit of the method, the slice of the result matrix should fit in registers. Also, the shared memory should be big enough to hold the required elements of A and B in $1 \times T$ and $T \times 1$ slices. The expected size of the result matrix for $T \times T$ region is $T^2\rho$ where ρ is the density of the result matrix. Due to register file size restrictions, $T^2\rho \leq Register\ Size$; hence, $T \leq \sqrt{\frac{Register\ Size}{\rho}}$. If there are empty rows or columns in $T \times T$ region corresponding row/column doesn't need to be loaded. However, for a single K, in the worst case, min(T,$T^2\rho$) data elements need to be loaded. In this scheme, 2.min(T,$T^2\rho$) data movement is needed for $2.T^2\rho$ operations. Hence, operational intensity (OI) or the number of operations per data movement is as follows: $OI = \frac{T^2\rho}{min(T,T^2\rho)} = max(\frac{T^2\rho}{T}, \frac{T^2\rho}{T^2\rho}) = max(T\rho, 1) \leq max(\rho\sqrt{\frac{Register\ Size}{\rho}}, 1) = max(\sqrt{(Register\ Size)\rho}, 1)$. Thus, OI is bound by $\sqrt{(Register\ Size)\rho}$.

The next streaming choice is choosing B as the stationary matrix (i as the streaming dimension (vertical streaming)). This scheme can be represented by ordering the tiling loops in Algorithm 3 as <Tj, Tk, Ti>. Figure 2(b) illustrates this approach. Column panels of size $Tj \times M$ are used to partition the sparse matrix. Each column panel is then swept along Ti and Tk. In this scheme, a tile of B is kept stationary, and A and S (and P) are streamed. Thus, B gets full reuse whereas reuse of A is limited to Tj and reuse of S is limited to Tk. Shared memory can be used to reduce the data movement costs. We can use shared memory to hold, i) both A and B elements, ii) A elements, and iii) B elements. Shared memory latency is approximately $100\times$ lower than global memory access and an order of magnitude lower than L2 cache access. With respect to latency, loading both matrices into shared memory is the preferred scheme. We call this method SM-SM (shared memory-shared memory) scheme. However, the limited amount of shared memory (64KB on Tesla P100) limits the tile sizes Ti and Tj. Note that, in this scheme, the volume of A slice ($Ti * Tk$) and B slice ($Tj * Tk$) should be less than the shared memory capacity. The latter limits the amount of work per thread block which could result in work starvation. Keeping only A elements in shared memory allows us to use larger blocks. The maximum length of Ti is limited by the capacity of the shared memory. To exploit the reuse of B, we can rely on L2 cache and tune Tj based on L2 cache capacity. We call it $SM - L2$ (shared memory-L2 cache) scheme.

The last streaming option (dual of vertical streaming) is to select A as the stationary matrix which corresponds to streaming along j (horizontal streaming). It can be represented by ordering the tiling loops in Algorithm 3 as <Ti, Tk, Tj>.

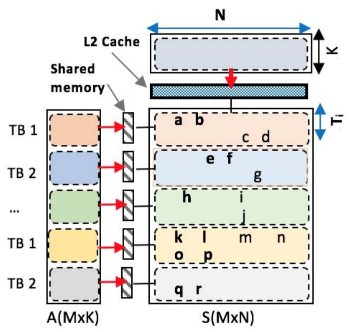
Now we discuss choosing the dense matrix (A or B) to stream along, based on data movement in the SM-L2 scheme. If $A(M \times K)$ is loaded into the shared memory, then $B(N \times K)$ relies on L2 cache (streaming along i). A is loaded into shared memory $\frac{N}{Tj}$ times, where $Tj = tilesize$. Both S and B are loaded only once. Each element in P is loaded and stored $\frac{c.K}{Tk}$ times, where Tk number of slices and c is a constant which depends on the representation of the sparse matrix (i.e. COO or CSR). We use COO storage format and it requires $3 \times nnz$ space to store tuple <row, col, val > of each non zero. Another possibility is to load dense matrix A to the L2 cache and dense matrix B to shared memory (streaming along j). The total number of DRAM transactions by keeping A or B in shared memory can be computed as:

$$\begin{aligned} DRAM\ Trans_{AinSh} &= \frac{MNK}{Tj} + NK + nnz + \frac{c.nnz.K}{Tk} \\ DRAM\ Trans_{BinSh} &= M.K + \frac{MNK}{Ti} + nnz + \frac{c.nnz.K}{Tk} \end{aligned} \tag{4}$$

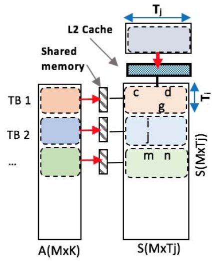
When $Ti = Tj$, the difference of data movement between these two cases is $K(M - N)$. Thus, we can conclude, selecting the smaller dimension to stream along will require less data movement, hence better performance.

The Operational Intensity (OI) can be computed as:

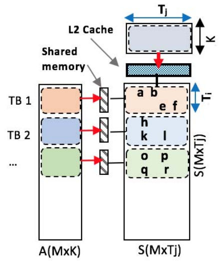
$$\begin{aligned} OI &= \frac{Number\ of\ Operations}{DRAM\ Transactions} \\ &= \frac{2K.nnz}{\left[\frac{M.N.K}{Tj} + N.K + nnz + \frac{c.nnz.K}{Tk}\right]} \\ &\approx \frac{2.nnz}{\left[\frac{M.N}{Tj} + \frac{c.nnz}{Tk}\right]} \end{aligned} \tag{5}$$



(a) Non tiled



(b) Tiled-1



(c) Tiled-2

Fig. 3: Tiled and non-tiled version of SM-L2 scheme. (a) Non-tiled (model derived $Tj >= N$): Matrix A is loaded into shared memory and matrix B relies on L2 cache for data reuse. (b) Tiled (model derived $Tj \approx N/2$): S and B are split into 2 tiles. Tile 1 loads corresponding *active* rows into shared memory and performs SDDMM (c) Tile 2 loads *active* rows into shared memory and performs SDDMM. Tile 1 and 2 are processed sequentially on a single GPU and in parallel on multi-GPU node

Using the optimal values of Tj and Tk explained in Section IV-A:

$$OI \approx \sqrt{\frac{(L2\ Size)\rho}{c}}$$

.

The OI for streaming along i or j is proportional to the L2 cache size and the OI for streaming along k is proportional to the register capacity. Since L2 size is significantly larger than register size, streaming along i or j rather than k is more efficient. We have empirically found that, matrices with more than 5% density usually have enough work to occupy a GPU, and can leverage faster memory access as well. For sparser matrices, the SM-L2 scheme is a better alternative. In the next section, we provide two such models based on the density of S.

IV. CuSDDMM

Based on the analysis presented in Section III, we derive two alternative SDDMM schemes, SM-L2 and SM-SM.

A. SM-L2 scheme

The thread idling or work starvation problem of the SM-SM scheme can be alleviated by increasing the tile size. Algorithm 5 describes the SM-L2 scheme. We assume A is the model chosen matrix to stream along for the rest of the explanation.

Algorithm 4: SDDMM using SM-L2 scheme CPU part

```
Input  : COO S[M][N], float A[M][K], float B[N][K], int K
Output: COO P[M][N]
                                          ▷ compute Ti,Tj and Tk
Tj = compute_tile_size_using_model(cache capacity, sparsity)
num_J_tiles = N/Tj
Tk = compute_k_slice_using_auto_tuning()
Ti = shared_mem_size/k_slice
num_Tks = K/Tk
for tile_id = 0 to num_J_tiles do
                       ▷ partition S and build set of active rows
    S_tile[M][Tj] = partition (S, Tj, tile_id)
    active_rows[] = collect rows with at least one non zero in
      S_tile[M][Tj]
    num_threadblocks = active_rows.size()/Ti
    for slice_id = 0 to num_Tks do
                                          ▷ launch GPU kernel
        SM-L2_GPU<<<num_threadblocks,
          512>>>(active_rows, S_tile[M][Tj], Tk )
    end
end
```

a) Select tile size for L2 - Tj: We model the selection of a suitable tile size depending on the cache capacity and density of the input matrix. To minimize the total number of DRAM transactions:

$$\begin{aligned}
&min\left[\frac{M.N.K}{Tj} + N.K + nnz + \frac{c.nnz.K}{Tk}\right] \\
&= N.K + nnz + M.N.K.min\left[\frac{1}{Tj} + \frac{c.\rho}{Tk}\right] \\
&\Leftrightarrow min\left[\frac{1}{Tj} + \frac{c.\rho}{Tk}\right] \geq 2\sqrt{\frac{c.\rho}{Tj.Tk}} \\
&= 2\sqrt{\frac{c.\rho}{L2\ size}} = constant
\end{aligned} \tag{6}$$

We get this minimum point where $\frac{1}{T}$ and $\frac{c.\rho}{Tk}$ are equal:

$$\frac{1}{Tj} = \frac{c.\rho}{Tk} \Leftrightarrow Tk = c.Tj.\rho \Leftrightarrow Tj.Tk = c.Tj^2.\rho$$

$$= L2\ size \Leftrightarrow Tj = \sqrt{\frac{L2\ size}{c.\rho}}$$

Hence, Tj is computed based on the size of the L2 cache of a GPU and density of the input matrix.

b) Selecting slice size Tk via auto-tuning: In this scheme, S needs to be loaded and written back $\frac{nnz}{Tk}$ times. $\frac{nnz}{Tk} = 1$ will require minimum read and write transaction which suggests using larger Tk. However, larger Tk makes the row panel height Tj smaller due to the shared memory space constraints. As a result, each thread block ends up having fewer work which can be as low as 1. This creates another critical issue, reduction of active columns. Active columns are the columns accessed by active thread blocks. As there is less number of rows now, chances of intra thread block cache reuse of a column are also decreased. On the other hand, if Tk is decreased, the height of the row panel is increased which potentially increases the chances of column uses. Under the circumstances, we adopt an auto-tuning approach to pick slice size Tk. The options for Tk are multiple of 32 (WARP size of GPU). Tj is selected from the model and execution is done for all possible slice to pick the combination of best Tj and Tk for a matrix. As LDA, ALS etc. are iterative algorithms, the time used to find Tk is negligible.

c) Fetching active rows: Real-word matrices often show power-law structures where many rows have very few elements. Tiling increases the chances of empty rows (rows with 0 elements) in a tile. For example, a row with one non-zero element will be active only in one tile. For the rest of the tiles, that row has no elements to process. Each thread block in a tile loads contiguous rows (even unused) of A to shared memory Even worse, this may limit the amount of available work at a given time step. In order to alleviate this, we maintain a list of active rows for each tile (column panel) and only the active rows are loaded into shared memory before processing. Line 10-15 in Algorithm 5 demonstrates the fetching of active rows of each tile to shared memory in an efficient and *coalesced* way. To ensure coalescing, we distribute threads in a WARP along K, which is the fastest varying index of A.

Algorithm 5: SM-L2_GPU()

```
Input  : COO S[M][Tj], active_rows, float A[M][K], float
         B[N][K], Tk, Ti
Output: COO P[M][Tj]
tID = thread ID
tile_no = TB_id = ThreadBlock ID
tile_start = find_starting_index(tile_no)
tile_end = find_ending_index(tile_no)
sh_actv_A[Ti][Tk]
wID = tID / WARP_SIZE
num_warps = TB_size/WARP_SIZE
t = tID % WARP_SIZE
                                        ▷ Load to shared mem
for i = wID to Ti step num_warps do
    active_rID = active_rows[ tile_no * Ti + i]
    for j = 0 to Tk step WARP_SIZE do
        sh_actv_A[i][t+j] = A[active_rID][t+j]
    end
end
syncthreads()
                        ▷ Thread-block cyclically processing non-zeores
for idx = tID/v_warp_size + tile_start to tile_end step
 TB_size/v_warp_size do
                                        ▷ Assignment of virtual warp
    laneid = tID % v_warp_size
    sh_rID = row_index[idx] - tile_no * Ti
                                ▷ loop unrolling and vectorization
    sum1[4] = sum2[4] = 0
    for t = laneid to Tk step v_warp_size do
        sum1[:] += sh_actv_A[sh_rID][t:t+3] *
         B[col_index[idx]][t:t+3]
        sum2[:] += sh_actv_A[sh_rID][t+4:t+7] *
         B[col_index[idx]][t+4:t+7]
    end
                                    ▷ reduction in a virtual warp
    for vws = v_warp_size/2 to 0 step vws/2 do
        sum1 += __shfl_xor(Σ(sum1), vws)
        sum2 += __shfl_xor(Σ(sum2), vws)
    end
    P[idx] = val[idx] * (sum1+sum2)
end
syncthreads()
```

d) Loop unrolling and vectorization: Although the proposed schemes achieve good reuse, the load transactions of A and B do not exploit available bandwidth provided by GPU for $DRAM\ transaction.$ Consider, a thread t is processing an element at $S_{(i,j)}$ (line 25 in Algorithm 5). At the first iteration, $B[i][0]$ is read and in the second iteration $B[i][1]$ is read, and so on. Each of these accesses requests 4-bytes of memory to be read (assuming the data type to be float). Each DRAM transaction is of width 32 bytes. Hence, by only requesting 4 bytes out of 32 possible bytes, the available bandwidth is not fully utilized. Instead of accessing a single element at a time, we can request four elements (e.g. $B[i][0:3]$) to be loaded at the same time. Similarly, elements of A are also read using vector loads. Correspondingly we reduce the number of inner loop iterations by a factor of 4. In addition to vector loads, we unroll the innermost loop to improve the amount of available Instruction Level Parallelism (ILP).

e) Use of virtual WARPs: In our implementation of the SM-L2 scheme, each thread block uses half of the shared memory available on an SM. Using full shared memory of an SM will reduce occupancy which may expose latency effects. From the shared memory perspective, since each thread block is using half the shared memory, only two thread blocks can be active simultaneously. In order to maximize occupancy, 1024 threads are assigned to each thread block. Note that, to achieve full occupancy, there should be 2048 active threads per SM. Due to the extreme sparse nature of the input matrix, the number of elements that can be processed simultaneously by a thread block may be less than 1024 which results in idle threads. If we assign more than one thread to process a single S element, we can achieve the required parallelism. However, the latter case requires reduction operations to combine contributions from multiple threads. The reduction operation can be efficiently done using warp shuffle instructions.

f) Load balance: In this work, we store S using a straightforward and popular sparse matrix storage format, COO. In COO, each non-zero is stored in a tuple format <row, column, value>. It requires more storage $(3 * nnz)$ compared to another popular format, compressed sparse row (CSR) $(M + 1) + 2 * nnz)$. In CSR, instead of repetitive row indices, only the start index of each row is stored. However, obtaining good load balance using CSR representation is a challenging task. The common practices to parallelize the CSR format is to assign a thread (CSR-scalar) or a warp (CSR-vector) or a virtual warp to process a row. However, none of these parallelization schemes is immune to row length variance of the input matrix and thus suffers from load imbalance. On the other hand, in COO format, a thread or a warp can independently work on each non zero. In order to alleviate the extra storage requirement by COO, we compress the global indices to their local indices (index value with respect to the beginning of the tile). Since local indices vary over a smaller range, fewer bits are used to represent them and thus reducing the final required storage. This technique helps to reduce the global memory traffic as well.

B. SM-SM scheme

In this subsection, we describe the SM-SM scheme which is targeted at sparse matrices with sufficient density (>5%). The objective of the SM-SM scheme is to eliminate the uncoalesced global memory accesses to both of the dense matrices (A and B) by loading them to the on-chip shared

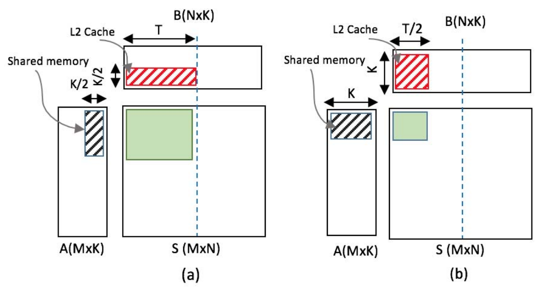


Fig. 4: Trade off between reuse and data movement of S caused by choosing different slice size

memory. The indices and values of S are accessed in a consecutive manner (each thread processes one non-zero) in global memory, which results in coalesced memory access. Coalesced memory accesses are optimized to use fewer number of global-memory transactions.

Figure 6 illustrates the data partitioning and processing techniques of the SM-SM scheme. S (and P) is partitioned into row panels of height Ti. The row panels are further sub-partitioned into tiles of width Tj. Each row panel is mapped to a thread block, and all the tiles in that row panel are processed sequentially. As shown in Figure 6, a thread block initially loads $(Ti \times K)$ elements of A into shared memory. Then for each tile within that row panel, $(Ti \times Tj)$ elements of B are also loaded to shared memory. A is loaded to shared memory once but B is loaded $\frac{M}{Ti}$ times. The streaming direction (A or B) is chosen from the model. The row panel height and tile width are chosen such that $(Ti + Tj) \times K$ fits in the shared memory. For example, using a Nvidia Tesla K80c GPU which has 48KB shared memory per SM, and using K=32, the maximum tile size for Ti = Tj is 96.

If the density of sparse matrix is high, both A and B will get good reuse and this scheme will perform well. However, keeping both A and B slices in shared memory limit the tile sizes along i and j (Ti and Tj). As Ti and Tj decreases, the number of sparse matrix elements in a tile also decreases. Since each tile is assigned to a thread block, if the tile does not contain enough work, many threads will be idle. This will adversely affect the performance. This model can substantially outperform SM-L2 scheme provided sufficient density ($\geq$5%). This pattern is shown in Figure 5 on synthetic matrices of dimension $100K \times 100K$ and $75K \times 75K$ matrices.

C. *Scalability of CuSDDM on multiple GPUs*

The capacity of DRAM memory on CPUs are usually much larger than the size of global memory on GPUs. The latest NVIDIA Tesla P100 only has 16 GB global memory, whereas the state of the art CPU like Intel Xeon E5-2697 has roughly 1.5 TB main memory. Often a single GPU's global memory is not sufficient to hold large-problem sizes (bigger matrices or larger values of K). This motivates the need for multi-GPU SDDMM solution. The single node SM-L2 scheme shown in Section IV-A, splits the input matrix into multiple tiles where each tile is processed sequentially. In the multinode scheme, we can launch the kernels in parallel across multiple machines and thus can process multiple tiles at the same time. However, dividing the entire columns equally across different nodes can result in significant load imbalance. Real-world datasets often follow power law distribution or non-structured patterns. For example, a straightforward two-way vertical split of NYTimes dataset causes 91% of the non zero elements to fall on one machine. This causes one machine to perform 14x slower than the other. To alleviate this problem, we follow a non-symmetric partitioning technique. S is split into multiple 1D tiles such that each partition has the similar amount of work. To develop this balanced scheme, we first compute the number of non-zero elements per column, and then the prefix sum of the latter is computed. The prefix sum array is then scanned to find partitions such that each partition has approximately nnz/P non-zeros where P is the number of processors (nodes). One of the dense matrices A or B is also partitioned across machines and the other one is shared across all nodes.

V. Experiments

In this section, we evaluate the performance of the proposed SM-L2 scheme against the SDDMM kernel of the BIDMach [12] library over a range of datasets. The Bag-of-Words datasets are from the UCI Machine Learning Repository [13], and the graph datasets are from SNAP [14] and GraphChallenge. Existing Bag-of-Words datasets have an average of 1.2% density. Thus, we don't show results for the SM-SM scheme (suitable for matrices with more than 5% density) for real-world datasets. In Figure 5, a comparison between the SM-SM and the SM-L2 scheme is shown for synthetic matrices of different range of density. The experiments are run on a Tesla P100 GPU from NVIDIA. It has 56 SMs, 64 cores/MP, 16 GB Global Memory, 1328 MHz clock frequency, and 4MB L2 cache. The CPU node used in this work is an Intel(R) Xeon(R) CPU E5-2680 V4 (28 cores).

A. *Benchmark*

In this work, we focus on optimizing SDDMM as a sparse linear algebra primitive that can be used in implementing many ML algorithms. Eigen [15], uBLAS [16] and the TACO-compiler [17] implement SDDMM on CPU, and BIDMach [12] on both CPU and GPU. TACO presents a novel compiler based method and generates an optimized kernel for CPU for a given tensor algebra expression in index notation. TACO outperforms Eigen and uBLAS by several orders of magnitude [17]. On an Intel Xeon with 28 cores using double precision, we found that SDDMM from TACO achieved around 5 GFLOPS, which is a couple of orders of magnitude slower than our GPU implementation of SDDMM. We therefore only present a comparison with the fastest alternative, the BIDMach GPU implementation of SDDMM.

BIDMach: BIDMach is a state-of-the-art toolkit for large-scale machine learning library. BIDMach has efficient CPU and GPU implementations of several machine learning algorithms like SVM (support vector machine) on sparse data, LDA, NMF (non-negative matrix factorization) with KL-divergence loss, ALS etc. SpMV, SpMM, and SDDMM primitives are the bottleneck kernels for these algorithms. Comparing to the performance of SDDMM using NVIDIA

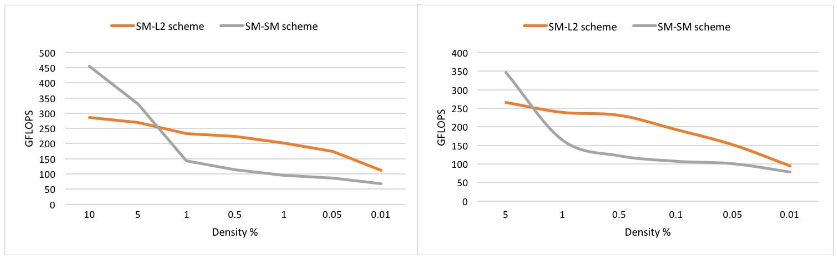


Fig. 5: GFLOPS achieved by using schemes based on SM-L2 and SM-SM on synthetic matrices with dimension of (a)75,000 x 75,000 and (b) 100,000 x 100,000 with different density%

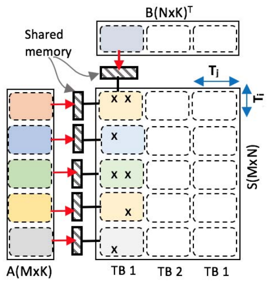


Fig. 6: SM-SM scheme: Each thread block loads a slice of A and B into GPU's shared memory according to its shared memory capacity.

TABLE II: Bag of Word and GraphChallenge Dataset

Datasets	M	N	#nnz	density %
KOS	3K	7K	353K	1.491
NIPS	2K	12K	746K	4.006
NYTimes	300K	103K	69M	0.226
cit-HepPh	35K	35K	422K	0.035
com-amazon	549K	549K	926K	0.000
com-dblp	426K	426K	1M	0.001
email-Enron	37K	37K	368K	0.027
facebook combined	4K	4K	88K	0.541
filter3D	106K	106K	3M	0.024
web-BerkStan	685K	685K	8M	0.002
web-Google	916K	916K	5M	0.001
web-NotreDame	326K	326K	1M	0.001
loc-gowalla edges	197K	197K	2M	0.005
mario002	390K	390K	2M	0.001
offshore	260K	260K	4M	0.006
patents main	241K	241K	561K	0.001
pdb1HYS	36K	36K	4M	0.328

sparse library, BIDMach's native SDDMM kernel achieves 3.7x performance improvement [12].

BIDMach's GPU based implementation follows a fine-grained parallelism scheme by assigning one thread to process one non-zero element of S. It also parallelizes over K by launching 2D thread blocks on thread level. Hence, BIDMach achieves almost perfect load balance and also high occupancy. However, it introduces reduction across threads inside a WARP to accumulate K products by each thread. Also, when K is greater than WARP size (32 for NVIDIA GPU), reductions across WARPs are required. These reductions are implemented using expensive atomic operations. In our methods, we avoid reductions across WARPs and allow reductions across threads within a WARP (when virtual WARPs are used). For example, if virtual WARP of size 16 is used and K is 32, 2 threads will process K computations and only one reduction will be required. CUDA provides a very efficient way ($shfl\_down()$) to process thread reduction.

BIDMach splits the matrices into small batches and launches a separate kernel for each batch. By default, it selects 1024 as the batch size. For sparse matrices, selecting batch size as small as 1024 affects the performance due to underutilization of GPU resources. We tune across batch sizes from 1024 to 50,000 and compare our results with the best performing BIDMach results across these batch sizes.

B. Evaluation

In this section, the performance of the SM-L2 scheme and BIDMach-GPU is compared. Single precision data type is used in both cases and we present achieved performance in GFLOPS for K=32, 128 and 512. Figure 7 shows the comparison between BIDMach, model-based cuSDDMM, and exhaustive search with cuSDDMM. With **Model**, we use predicted tile size Tj from the model and round it up to the nearest multiple of 5000. Slice size Tk is chosen via an auto-tuning approach. For example, for K=512, we have slice options of 32, 64, 128, 256 and 512. We run our program for all possible slices using the model selected tile size. The **Exhaustive** approach runs cuSDDMM for all possible combinations of slice sizes and tile sizes and selects the best $\{Tj,Tk\}$ pair.

Figure 7 shows that our scheme consistently outperforms BIDMach by a factor of 1 to 4.6. We achieve up to 403 and 414 GFLOPS by using the model predicted tile size and auto-tuned slice size for K=128 and K=512. BIDMach's performance improves with increasing K. Using K=512, it achieves 107 GFLOPS on average. With larger K, it achieves higher parallelism and occupancy. However, beyond a point, the reduction cost across threads and WARPs becomes high and the performance improvement saturates. In our work, we efficiently parallelize the work over K so that the workload and reduction cost is balanced. For the most commonly used bag of words dataset like NYTimes, our performance improvement for K=32, 128 and 512 is 5×, 3× and 2.6× respectively.

C. Impact of model derived tile size

Our analysis is based on the assumption that the size of ($Tk*Tj$) of B will stay in cache. Since all DRAM transactions are automatically cached in L2, loading elements of A to shared memory may evict B elements from cache. However, since data loaded into the shared memory (matrix A) are accessed only once from L2, we expect these elements to get evicted faster than the B elements (assuming LRU policy). In addition, the latency issues caused by occasional cache misses for B can be hidden with good occupancy/concurrency. Hence, in our scheme, we choose the size of logical shared memory and thread block to achieve full occupancy. Choosing $Tk*Tj$ to be lower than L2 cache size can help to increase the probability of B elements being served from the cache. However, since the loads for A are inversely proportional to Tj, the latter choice can result in an increase in the total number of

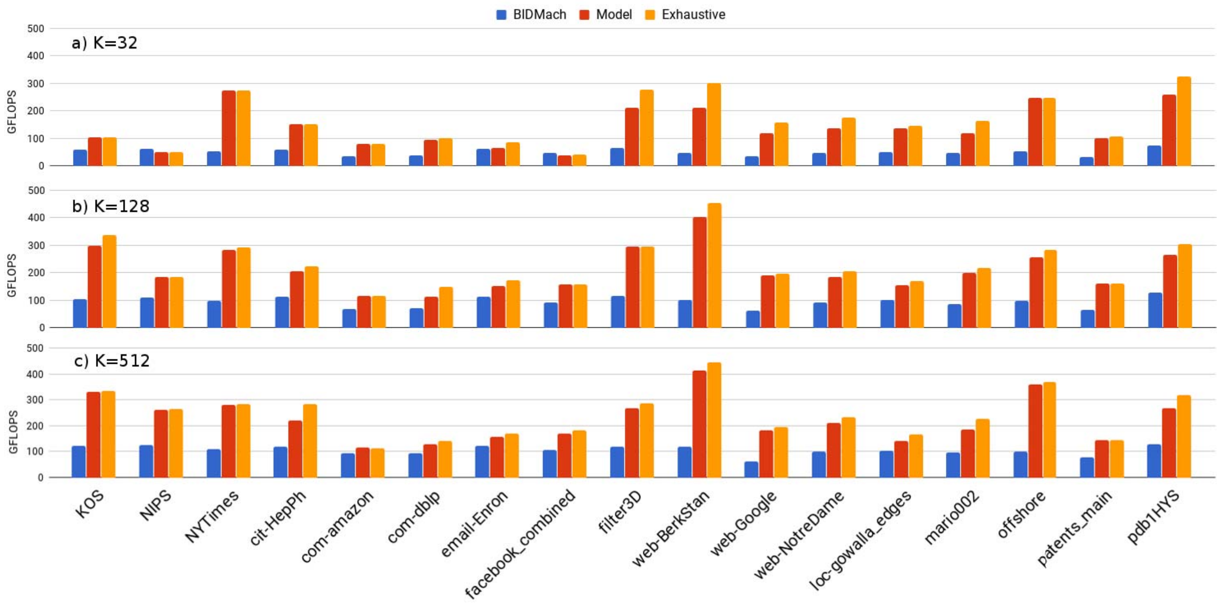


Fig. 7: Performance (GFLOPS) using a)K = 32, b)K = 128, and c)K=512 on Tesla P100 GPU

DRAM transactions. Table III shows the effect of different tile choices of Tj on total DRAM read transactions (measured using NVPROF) using NYTimes dataset. We calculate the theoretical data movement of A as ($number\ of\ tiles \times N \times K$), and data movement of S as $3 \times nnz$. As Tj decreases, the DRAM transactions increase.

TABLE III: Reduction in DRAM transactions with increasing number of tiles and shared load of A

Number of tiles	DRAM trans. (nvprof)	Theo. data mov. of A	Theo. data mov. of S	DRAM trans. by B
1	15,542M	307M	836M	14,399M
2	13,115M	614M	836M	11,664M
3	11,437M	921M	836M	9,679M
4	11,188M	1,228M	836M	9,123M
5	10,769M	1,536M	836M	8,396M
6	10,487M	1,843M	836M	7,807M
7	10,482M	2,150M	836M	7,495M
11	10,948M	3,379M	836M	6,733M
21	12,908M	6,451M	836M	5,621

D. Speedup of Multi-GPU scheme

Figure 8 shows the speedup of the multi-GPU implementation using the non-symmetric distribution scheme against the symmetric one. cuSDDMM achieves 3× and 10× speedup for the NYTimes dataset using 4 and 16 nodes respectively . The primary reason behind the weak scaling of other matrices is the insufficient amount of work to occupy a GPU. Table II, shows the characteristics of the input datasets. Real-world matrices often have power-law or clustered structures. Hence, some nodes inherently benefit from the high data reuse and some suffer from poor data locality. We extend our tiling techniques to address the problem. In Figure 8, we only use the datasets which have more than 1M nonzeroes in total.

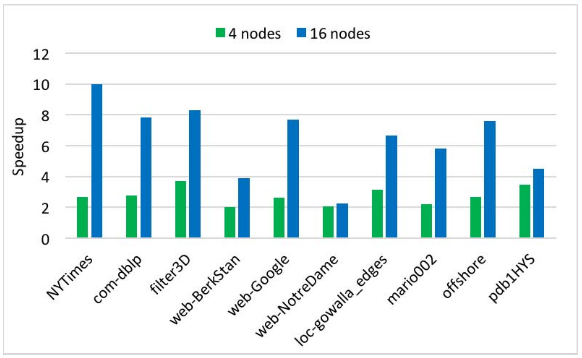


Fig. 8: Speedup achieved by using 4 and 16 Tesla P100 GPUs on using K=512 and single precision data type

TABLE IV: Achievable speedup by using cuSDDMM instead of default kernel from BIDMach

Dataset	SDDMM%	kernel speedup	App Speedup
Netflix	53.14	4.15	1.68
NYTimes	53.70	4.03	1.68
PubMed	42.52	4.64	1.50

E. Effectiveness of model

Figure 9 shows the loss of performance by using Tj and Tk from the model, compared to exhaustive search. For most of the cases, the model is able to predict a configuration with a performance loss under 20%, compared to exhaustive search.

F. Impact on ML applications

Table IV shows the potential application speedup by using cuSDDMM instead of the default kernel from BIDMach for the SFA application. Since we faced challenges in actually replacing BIDMach's SDDMM GPU kernel with cuSDDMM, we estimate application speedup by using the fraction of time

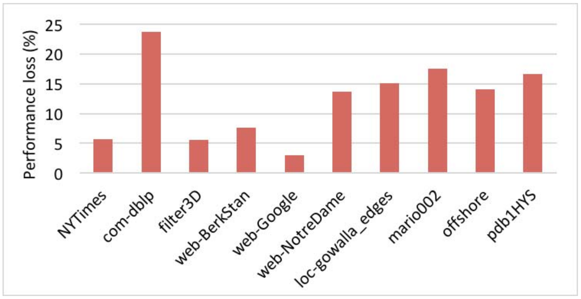


Fig. 9: Performance loss caused by using model predicted tile size (T) compared to best tile size via exhaustive search

spent on the SDMMM kernel and the speedup of cuSDDMM over BIDMach.

VI. Related Work

There has been a significant amount of research done in the past decade on improving the accuracy of collaborative filtering and topic modeling based ML algorithms. Their significance in application resulted in development of novel algorithms such as LDA [18], [4], Sparse Factor Analysis (SFA) [19], [2], Gamma Poisson (GaP) [20], [5], ALS [1] and so on. With the advent of many-core architectures, researchers have considered optimizing these algorithms on these architectures and also scaled them on distributed memory systems [21], [22], [23], [24], [25].

CuMF [26] presents such a matrix factorization library to solve ALS based MF on single and multiple GPUs. Other works [27] and [10] use SGD - Stochastic Gradient Descent and CCD++ - Cyclic Coordinate-based techniques on GPU to perform MF. Recently, Li et al. [28] proposed a novel LDA technique on GPUs, which maintains high accuracy as well as speed. Many of these algorithms like LDA, SFA etc. can be formulated using the SDDMM kernel and an optimized SDDMM kernel can aid in improving the performance of several ML algorithms. We show examples of such formulations in Section II. There have been several attempts to boost the performance of these algorithms by using a faster SDDMM kernel [7], [3], [6].

VII. Conclusion

SDDMM is a sparse matrix multiplication kernel that can be used to create more efficient formulations of several ML algorithms than existing formulations based on sparse BLAS2 SpMV primitives. Examples of ML algorithms for which SDDMM-based formulations exist are LDA, SFA, ALS etc. SDDMM requires the product of two dense matrices, but only at the position of non zero elements of a given sparse matrix. This paper presents the analysis, design, and efficient implementation of cuSDDMM, a multi-GPU parallel implementation for SDDMM. Experimental evaluation shows significant speedup over the existing frameworks. The performance improvement over a state of the art SDDMM GPU implementation ranges up to 4.6x.

Acknowledgments

We thank the reviewers for the valuable feedback and the Ohio Supercomputer Center for use of their GPU resources. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract D16PC00183, and the National Science Foundation (NSF) through awards 1404995, 1513120, and 1629548.

References

[1] Y. Koren, R. Bell, and C. Volinsky, “Matrix factorization techniques for recommender systems,” *Computer*, vol. 42, no. 8, 2009.
[2] J. Canny, “Collaborative filtering with privacy,” in *Security and Privacy*, 2002, pp. 45–57.
[3] H. Zhao, B. Jiang, J. F. Canny, and B. Jaros, “Same but different: Fast and high quality gibbs parameter estimation,” in *SIGKDD*, 2015, pp. 1495–1502.
[4] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.
[5] J. Canny, “Gap: a factor model for discrete data,” in *SIGIR*, 2004, pp. 122–129.
[6] J. Canny and H. Zhao, “Bidmach: Large-scale learning with zero memory allocation,” in *BigLearn workshop, NIPS*, 2013.
[7] ——, “Big data analytics with small footprint: Squaring the cloud,” in *SIGKDD*, 2013, pp. 95–103.
[8] J. Canny, H. Zhao, B. Jaros, Y. Chen, and J. Mao, “Machine learning at the limit,” in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 233–242.
[9] H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon, “Parallel matrix factorization for recommender systems,” *Knowledge and Information Systems*, vol. 41, no. 3, pp. 793–819, 2014.
[10] I. Nisa, A. Sukumaran-Rajam, R. Kunchum, and P. Sadayappan, “Parallel ccd++ on gpu for matrix factorization,” in *GPGPU*. ACM, 2017, pp. 73–83.
[11] C. NVIDIA, “Cusparse library,” *NVIDIA Corporation, Santa Clara, California*, 2014.
[12] H. Zhao, *High Performance Machine Learning through Codesign and Roofling*. University of California, Berkeley, 2014.
[13] M. Lichman, “UCI machine learning repository,” 2013. [Online]. Available: http://archive.ics.uci.edu/ml
[14] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” http://snap.stanford.edu/data, Jun. 2014.
[15] G. Guennebaud, B. Jacob *et al.*, “Eigen v3,” http://eigen.tuxfamily.org, 2010.
[16] J. Walter, M. Koch *et al.*, “ublas,” http://www.boost.org/doc/libs/1_66_-0/libs/numeric/ublas/doc/index.html, 2012.
[17] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, “The tensor algebra compiler,” *PACMPL*, vol. 1, no. OOPSLA, p. 77, 2017.
[18] D. M. Blei, A. Y. Ng, and M. I. Jordan, “Latent dirichlet allocation,” *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.
[19] A. S. Lan, A. E. Waters, C. Studer, and R. G. Baraniuk, “Sparse factor analysis for learning and content analytics,” *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1959–2008, Jan. 2014.
[20] M. K. Titsias, “The infinite gamma-poisson feature model,” in *NIPS*, 2008, pp. 1513–1520.
[21] H. Zhang, Z. Hu, J. Wei, P. Xie, G. Kim, Q. Ho, and E. P. Xing, “Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines,” *CoRR*, vol. abs/1512.06216, 2015.
[22] Y. You, A. Buluç, and J. Demmel, “Scaling deep learning on gpu and knights landing clusters,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 9.
[23] “Caffeonspark github project,” https://github.com/yahoo/CaffeOnSpark, accessed: 2017-01-23.
[24] “Paddlepaddle,” https://github.com/paddlepaddle/paddle, accessed: 2017-01-23.
[25] “Amazon dsstne github project,” https://github.com/amzn/amazon-dsstne, accessed: 2017-01-23.
[26] W. Tan, L. Cao, and L. Fong, “Faster and cheaper: Parallelizing large-scale matrix factorization on gpus,” in *HPDC*. ACM, 2016, pp. 219–230.
[27] X. Xie, W. Tan, L. L. Fong, and Y. Liang, “CuMF_SGD: Fast and scalable matrix factorization,” *CoRR*, vol. abs/1610.05838, 2016.
[28] K. Li, J. Chen, W. Chen, and J. Zhu, “Saberlda: Sparsity-aware learning of topic models on gpus,” in *ASPLOS*, 2017, pp. 497–509.