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 data sets 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:

- 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.

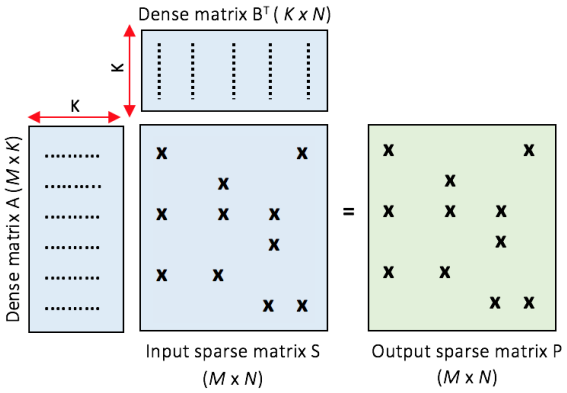


Fig. 1: SDDMM: Product of dense matrix A and B is accumulated at each non zero position of sparse matrix S to generate output sparse matrix P

- 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 4.7x) 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 **??** 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

A. Sampled Dense-Dense Matrix Multiplication (SDDMM)

SDDMM computes the dot product of two vectors $A(i,*)$ and $B(j,*)$ at each non-zero position (i,j) of the sparse matrix S. Each vector has K elements, where K is the number of features/topics. The resulting sparse matrix is then subjected to a Hadamard product (element-wise multiplication) with S. Figure 1 illustrates SDDMM.

SDDMM can be expressed as:

$$P = (A *_S B^T) o S \tag{1}$$

Where $A \epsilon \mathbb{R}^{M \times K}$ and $B \epsilon \mathbb{R}^{N \times K}$ are two dense matrices and $S \epsilon \mathbb{R}^{M \times N}$ is the *sampling* sparse matrix. The notations used in this paper are listed in Table I.

Algorithm 1 shows the pseudocode for the sequential SDDMM algorithm, where the sparse matrices are represented in Compressed Sparse Row (CSR) format. A GPU kernel for SDDMM was described by Zhao et al. [3] in the context of a reformulation of LDA. SDDMM can also be produced by 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 DGEMM an impractical alternative. By performing computations corresponding to only non-zero elements, the computational complexity can be reduced to $\mathcal{O}(K.nnz(S))$ from $\mathcal{O}(K.n^2)$.

TABLE I: Notation

Name	Description
S	sparse input matrix with dimension: $M \times N$
A	dense input matrix with dimension: $M \times K$
B	dense input matrix with dimension: $N \times K$
P	sparse input matrix with 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
T_i	length of vertical tiles
T_j	length of horizontal tiles
U	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

Algorithm 1: Sequential SDDMM

```
input : CSR S[M][N], float A[M][K], float B[N][K]
output: CSR P[M][N]
\\Sampled Dense-Dense multiplication
for i = 0 to M do
    for j = S.rowptr[i] to S.rowptr[i+1]-1 do
        for k = 0 to K-1 do
            P.values[j] += A[i][k] * B[S.colidx[j]][k]
\\Scaling
for i = 0 to S.rows-1 do
    for j = S.rowptr[i] to S.rowptr[i+1]-1 do
        P.values[j] *= S.values[j]
```

B. 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 that are currently expressed using primitives that are not as efficient with respect to data movement .

Algorithm 2: Sequential CCD++ Algorithm

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

Recommender systems seek to predict a user's preference for items. An approach to recommender systems is matrix factorization. Cyclic Coordinate Descent (CCD++) [7], [8], an adaption of Alternating Least Squares (ALS) is a state-of-the-art technique for matrix factorization. Given a sparse ratings matrix $A \epsilon \mathbb{R}^{M \times N}$, where M is the number of users and N is the number items, CCD++ iteratively updates two dense factor matrices $W \epsilon \mathbb{R}^{M \times K}$ and $H \epsilon \mathbb{R}^{N \times K}$ such that their

product WH^T approximates A well for the available ratings. W and H are the user and item matrix, respectively, and K is the number of latent features.

CCD++ performs feature-wise update as shown in Alg. 2, i.e. one of K features is selected for an update at a time and the values of the other features are treated as constants. The columns of the item-feature (W) and user-feature (H) matrices corresponding to the selected feature are then 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), 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.

Now, we show a reformulation of matrix factorization using ALS based on SDDMM which has a much higher potential operational intensity (OI) than sparse BLAS2 operations. Eq. 2 shows the gradient update formula used in ALS [1]. 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). NVIDIA cuSPARSE library [9] provides an optimized SpMM implementation which can be used here. The lack of optimized SDDMM operation motivates the work of this paper.

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

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. Gamma-Poisson (GaP) model developed by Canny et al. [5], is an alternative topic model to LDA. In GaP, a document-word matrix is approximately factorized into two matrices in order to infer word-topic and document-topic distributions.

Equation 3 denote the formula 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 Eq. 3 of S by $\beta^T *_S F$ involves an element-wise quotient followed by a Sparse Matrix Dense - Matrix Multiplication(SpMM). The denominator of the update formula $\beta^T *_S F$ 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 bottleneck 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

Algorithm 1 shows the sequential SDDMM algorithm where the sparse matrices are represented in Compressed Sparse Row (CSR) format. The i-loop in line 2 iterate over the rows of S and the j-loop in line 3 iterate over column indices of each row. The k-loop (line 4) computes the dot product and the resulting value is stored in the corresponding location of P. The nested loop in line 7 performs an element-wise product of sparse input and output matrix. Unlike SpMV, in SDDMM each non-zero element in S has a K-way reuse. Each element of A has a reuse factor equal to the average number of non-zero elements in rows of S, and each B element has a reuse factor equal to the average number of nonzero elements in columns of S. Our main goal in this work is to take advantage of the reuse potential by maximizing data locality and reducing data movement.

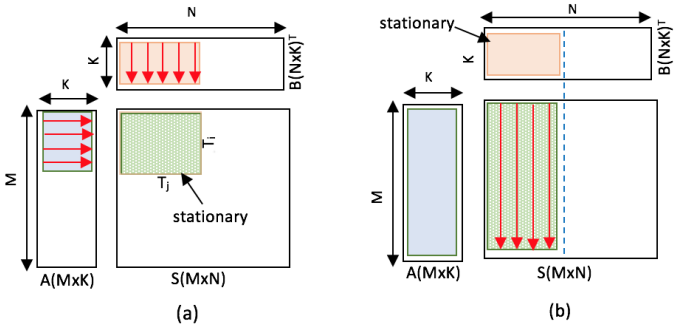


Fig. 2: Different types of streaming

Many dense matrix based algorithms such as Dense-Dense matrix multiplication (dgemm) employ a streaming approach to reduce the data movement volume. In the streaming approach, a slice of one of the three matrices is kept stationary in fast memory (cache, share-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.

Algorithm 3: Tiled SDDMM

```
   input : CSR S[M][N], float A[M][K], float B[N][K]
   output: CSR P[M][N]
1  \\Sampled Dense-Dense multiplication
2  for ii = 0 to M step Ti do
3      for jj = 0 to N step Tj do
4          for kk = 0 to K-1 step Tk do
5              for i = Ti to min((ii + 1) * Ti, M) do
6                  for cur_elem ∈ s_tile[ii][jj] do
7                      j = cur_elem.global_col
8                      for k = kk to min((kk+1)* Tk, K-1) do
9                          P.values[j] += A[i][k] * B[S.colidx[j]][k]
10 ...
```

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 $(A, B, S/P)$ for the stationary matrix. In the tiled version, the streaming dimension can be represented by the innermost tile dimension. Selecting S as the stationary matrix corresponds to choosing k as the streaming dimension. Figure 2 depicts this scheme. For simplicity of calculations, let us assume $Ti = Tj = T$. A slice of S of size $T \times T$ is kept in 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.

For streaming along k, the slice of the result matrix should fit in registers and shared memory should be big enough to hold the required elements of A/B in $1 \times T$ and $T \times 1$ slices. Since the result matrix is stored in registers, the expected size of the result matrix for $T \times T$ region is $T^2\rho$ where ρ is density of the result matrix. Due to register file size restriction, $T^2\rho \leq Register\ Size$; 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 . Operational intensity is number of operations per data movement and in this scheme 2.min(T,$T^2\rho$) data movement is needed for $2.T^2\rho$ operations. Hence, $OI = \frac{T^2\rho}{min(T,T^2\rho)} = max(\frac{T^2\rho}{T}, \frac{T^2\rho}{T^2\rho}) = max(T\rho, 1) \leq max(\rho\sqrt{\frac{Register\ Size}{\rho}}, 1) = max(\sqrt{(Register\ Size)\rho}, 1)$. Thus, OI bounded by $\sqrt{(Register\ Size)\rho}$.

The next streaming choice corresponds to 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 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 Tk. In this scheme, shared memory can be used to reduce the data movement costs. We can use shared memory to hold i) A elements or ii) B elements or iii) both A and B elements. Keeping A elements in shared memory reduces the global memory access cost of A by a factor of average number of elements in a row within a column panel. 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 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>. This horizontal streaming scheme is the dual of vertical streaming and has similar characteristics. The streaming dimension can be chosen based on the data movement volume, which can be estimated as follows. Consider the scenario where matrix $A(M \times K)$ is loaded into the shared memory and matrix $B(N \times K)$ into L2 cache (streaming along i). A has to be loaded into shared memory $\frac{N}{Tj}$ times, where Tj is the tile size. Both S and B will be loaded only once. Each sparse output matrix P element is loaded and stored $\frac{c.K}{Tk}$ times, where Tk is the 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. The total number of DRAM transactions can be computed as:

$$DRAM\ Transactions = \left[\frac{M.N.K}{Tj} + N.K + nnz + \frac{c.nnz.K}{Tk}\right]$$

Another possibility is to load dense matrix A to the L2 cache and dense matrix B to shared memory (streaming along j). The DRAM transactions for this scheme can be computed as:

$$DRAM\ Transactions = \left[M.K + \frac{M.N.K}{Ti} + nnz + \frac{c.nnz.K}{Tk}\right]$$

Assuming $Ti = Tj$, the difference of data movement between these two case 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) for streaming along I or J dimension can be computed as: $OI = \frac{Number\ of\ Operations}{DRAM\ Transactions}$

$$OI = \frac{2K.nnz}{\left[\frac{M.N.K}{Tj} + N.K + nnz + \frac{c.nnz.K}{Tk}\right]}$$

$$OI \approx \frac{2.nnz}{\left[\frac{M.N}{Tj} + \frac{c.nnz}{Tk}\right]}$$

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

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

.

The OI for streaming along i/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/j rather than K is more efficient.

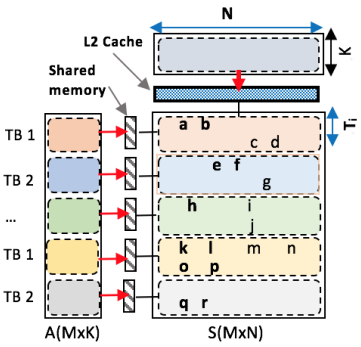
One option is to buffer both A and B in shared memory. In the rest of the paper, we refer to this as the $SM-SM$ (shared memory-shared memory) scheme. Since shared memory latency is approximately $100\times$ lower than global memory access and an order of magnitude lower than L2 cache access, with respect to latency SM-SM is the preferable 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 and this could result in work starvation. We have empirically found that matrices with more 5% density usually have enough work to occupy the GPU as well as can leverage faster memory access. For sparser matrices, the SM-L2 scheme is the better alternative. Loading only dense matrix to shared-memory allows us to choose bigger tiles. Thus, it trades off fast memory access of one dense matrix to avoid work starvation. In the next section, we provide two such models based on the density of S.

IV. CuSDDMM

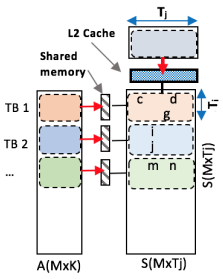
Based on the analysis presented in Section III we derive two alternative SDDMM schemes, SM-SM and SM-L2.

A. SM-L2 scheme

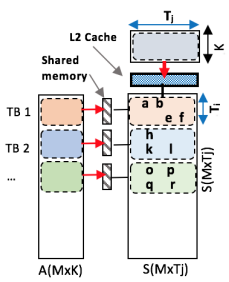
The thread idling or work starvation problem of SM-SM scheme can be alleviated by increasing the tile size. In Tesla K80c GPU, this scheme allows Tj (or Ti) to be as large as 192. Without loss of generality, assume that a slice of A is loaded into shared memory. If we stream along i in the vertical scheme, matrix B would get full reuse. This can be achieved



(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

as by keeping B in L2 cache which is shared across all SMs. The tile sizes Tj and Tk should be chosen such that $Ti * Tk$ will fit in L2. Thus in this scheme, reuse of A is equal to the average number of elements in a row with a column tile of size Tj, reuse of S is equal to K/Tk and B gets full reuse. Algorithm 5 describes our SM-L2 scheme. Now we provide details of each algorithm. We assume A is the model chosen matrix to stream along for the rest of the explanation.

Algorithm 4: SDDMM implementation using SM-L2 scheme (S, A, B, P)

```
   Input  :  COO S[M][N], float A[M][K], float B[N][K], int K
   Output:  COO P[M][N]
1  Tj = compute_tile_size_using_model(cache capacity, sparsity)
2  num_J_tiles = N/Tj
3  Tk = compute_k_slice_using_auto_tuning()
4  Ti = shared_mem_size/k_slice
5  num_Tks = K/Tk
6  for tile_id = 0 to num_J_tiles do
7      S_tile[M][Tj] = partition (S, Tj, tile_id)
8      active_rows[] = collect rows with at least one non zero in
       S_tile[M][Tj]
9      num_threadblocks = active_rows.size()/Ti
10     for slice_id = 0 to num_Tks do
11         SM-L2_GPU_kernel<<<num_threadblocks,
           512>>>(active_rows, S_tile[M][Tj], Tk )
```

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

$$min\left[\frac{M.N.K}{Tj} + N.K + nnz + \frac{c.nnz.K}{Tk}\right]$$

$$= N.K + nnz + M.N.K.min\left[\frac{1}{Tj} + \frac{c.\rho}{Tk}\right]$$

$$\Leftrightarrow min\left[\frac{1}{Tj} + \frac{c.\rho}{Tk}\right] \geq 2\sqrt{\frac{c.\rho}{Tj.Tk}}$$

$$= 2\sqrt{\frac{c.\rho}{L2\ size}} = constant$$

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

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

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

b) Selecting slice size: Tk: Sparse matrix 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 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 of A to shared memory before processing the S elements. This implies that even if a row has no elements to process, we will load the corresponding vector from A to shared memory which incurs unnecessary global memory traffic. 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 9-12 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_kernel (active_rows, S, Tk))

```
Input : COO S[M][Tj], active_rows, float A[M][K], float
        B[N][K], Tk, Ti
Output: COO P[M][Tj]
1  tile_no = threadblock_id
2  tile_start = find_starting_index(tile_no)
3  tile_end = find_ending_index(tile_no)
4  \\Each WARP cyclically loads Tk elements of a row from
   global memory to shared memory in a coalesced manner
5  shared_actv_A[Ti][Tk]
6  warp_id = thread_id / WARP_SIZE
7  num_warps = threadblock_size/WARP_SIZE
8  t = thread_id % WARP_SIZE
9  for i = warp_id to Ti step num_warps do
10 |  active_row_id = active_rows[ tile_no * Ti + i]
11 |  for j = 0 to Tk step WARP_SIZE do
12 |  |  shared_actv_A[i][t+j] = A[active_row_id][t+j]
13 __syncthreads()
14 \\Each Thread-block cyclically process all the non zeroes of a
   tile in a round robin fashion
15 for idx = thread_ID/v_warp_size + tile_start to tile_end step
   threadblock_size/v_warp_size do
16 |  \\Assign virtual warp to process each element idx
17 |  laneid = thread_id % v_warp_size
18 |  shared_row_id = row_index[idx] - tile_no * Ti
19 |  \\loop unrolling and vectorized shared and global memory
   |  access
20 |  sum1[4] = sum2[4] = 0
21 |  for t = laneid to Tk step v_warp_size do
22 |  |  sum1[:] += shared_actv_A[shared_row_id][t:t+3] *
   |  |  B[col_index[idx]][t:t+3]
23 |  |  sum2[:] += shared_actv_A[shared_row_id][t+4:t+7] *
   |  |  B[col_index[idx]][t+4:t+7]
24 |  \\reduction across the threads in a virtual warp
25 |  for vws = v_warp_size/2 to 0 step vws/2 do
26 |  |  sum1 += __shfl_xor(Σ(sum1), vws)
27 |  |  sum2 += __shfl_xor(Σ(sum2), vws)
28 |  P[idx] = val[idx] * (sum1+sum2)
29 __syncthreads()
```

d) Loop unrolling and vectorized data load from shared and global memory: Our scheme achieves good reuse from shared-memory (for A) and cache (for B). However, the load transactions of A and B do not exploit available bandwidth provided by $DRAM\ transaction$ of GPU. Consider a single S element $S[i][j]$ which is processed by a thread t. The innermost loop, line 21 in Algorithm 5 iterates over Tk. 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 SM-L2 scheme each thread block use half of 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 blocks. 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 required parallelism. However, the latter case requires reduction operation to combine contributions from multiple threads. The reduction operation can be efficiently done using warp shuffle instructions.

f) Load balance: The sparse matrix S is stored in COO format where each non-zero element is stored in a tuple format <row, column, value>. COO format requires more storage $(3 * nnz)$ compared to CSR format $(M + 1) + 2 * nnz)$. However, obtaining good load balance in CSR representation is a challenging task. Either a thread(CSR-scalar) or a warp(CSR-vector) or a virtual warp is assigned 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 elements and the entire work can be cyclically distributed across the threads or warp. In order to alleviate the extra storage requirement for the COO format, we compress the index values as follows. Instead of storing the global index values, we store the local index value (index value with respect to the beginning of the tile). Since local indices only vary over a smaller range, fewer bits can be used to represent them and thus reducing the required storage. This also helps to reduce the global memory traffic.

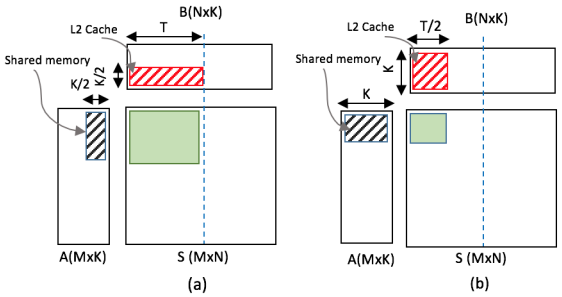


Fig. 4: Trade off between reuse and data movement of S matrix caused by choosing different slice size

B. SM-SM scheme

In this subsection, we describe our SM-SM scheme which is targeted at sparse matrices with sufficient density (>5%). The objective of SM-SM scheme is to eliminate uncoalesced global memory access for the dense matrices by loading them to the on-chip shared memory and then reusing it from shared memory. We use COO storage format to store the sparse matrices and parallelize over the non zero elements. Each thread accesses non zero elements of sparse matrices S and P from the global memory in a consecutive manner, which results in efficient coalesced global-memory accesses. Coalesed memory accesses result in a fewer number of global-memory transactions. The row and column indices are loaded through the read-only **texture memory**.

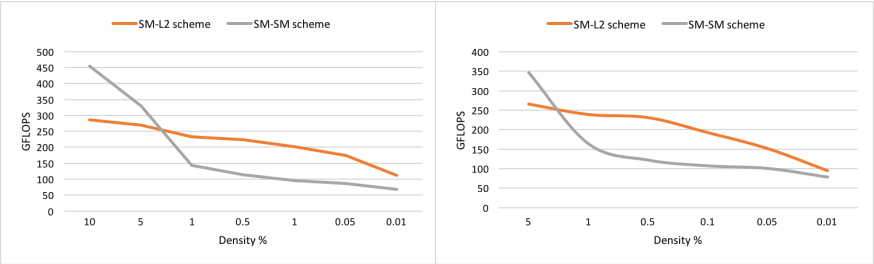


Fig. 5: GFLOPS achieved by using schemes based on SM-L2 and SM-SM on synthetic matrices with dimension of (a)75,000 x 75,000 and (b) 100,000 x 100,000 with different density%

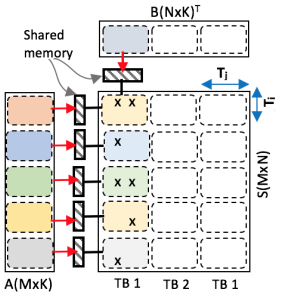


Fig. 6: SM-SM scheme: Each thread block loads a slice of A and B into GPU's shared memory according to it's shared memory capacity.

Figure 6 illustrates the data partitioning and processing techniques of SM-SM scheme. The sparse matrix S (and the output sparse matrix 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. All the tiles in a row panel are processed sequentially. A thread block initially loads a slice of A corresponding to the row-panel into shared-memory and then for each tile within the row panel, the corresponding slice of B is loaded to shared-memory. A is loaded to shared memory once but B is loaded $\frac{M}{Ti}$ times. Loading B once and streaming along A is similar. The streaming direction is chosen based from the model. The row panel height and tile width are chosen such that $(Ti+Tj)\times K$ fit in shared-memory. As an example consider Nvidia Tesla K80c GPU which has 48KB shared-memory per SM. For 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, and if the tile does not contain enough work, then many threads will be idle which 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 the dimension of 100Kx100K and 75K and 75K matrices.

C. Scalability of CuSDDM on multiple GPUs

The maximum amount of DRAM memory available for a CPU is much larger than the global memory capacity of a GPU. 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. Thus 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 and 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 nonstructured patterns which can lead to severe load imbalance. A straightforward 2 way vertical split of NYTimes dataset causes 91% of the non zero elements to fall on one machine. This results in one machine being 14x slower than the other. To alleviate this problem, we follow a non-symmetric partitioning technique. We split sparse matrix S into multiple 1D tiles such that each partition has the similar amount of work. First, we 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

sec:exp In this section, we evaluate our proposed SM-L2 scheme over a range of Bag of Words datasets from UCI Machine Learning Repository [10] and popular graph datasets from SNAP [11] and GraphChallenge datasets against BIDMach [12]. Existing Bag of word datasets have an average of 1.2% density. Thus, we don't show results for SM-SM scheme. However, as shown in Figure 5, with matrices of sufficient density SM-SM scheme will outperform the SM-L2 scheme. In this work we do not attempt to assess the performance of SDDMM-based ML algorithms against non-SDDMM based algorithms since that is very non-trivial and well beyond the scope of this paper.

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 [13], uBLAS [14] and the TACO-compiler [15] 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 [15]. On an Intel Xeon with 28 cores using double precision, TACO provides only up to 5 GFLOPS, which is a couple of order of magnitude slower than our GPU implementation. Thus, we don't show the comparison between our work and TACO.

BIDMach: BIDMach is a state-of-the-art toolkit for large-scale machine learning library. BIDMach has efficient CPU and GPU implementation of several machine learning algorithms like SVM (support vector machine) on sparse

TABLE II: Machine configuration

Machine	Resource	Details
1	GPU P100	Tesla P100 (56 SMs, 64 cores/MP, 16 GB Global Memory, 1328 MHz, 4MB L2cache)
2	CPU	Intel(R) Xeon(R) CPU E5-2680(28 core)

TABLE III: Bag of Word and GraphChallenges Dataset

Datasets	M	N	nnz
Enron	39,861	28,102	3,710,420
KOS	3,430	6,906	353,160
NIPS	1,500	12,419	746,316
NYTimes	300,000	102,660	69,679,427
PubMed	8,200,000	141,043	483,450,157
cit-HepPh	34,546	34,546	421,578
com-amazon	548,552	548,552	925,872
com-dblp	425,957	425,957	1,049,866
com-youtube	1,157,828	1,157,828	2,987,624
email-Enron	36,692	36,692	367,662
facebook_combined	4,039	4,039	88,234
filter3D	106,437	106,437	2,707,179
loc-gowalla_edges	196,591	196,591	1,900,654
mario002	389,874	389,874	2,101,242
offshore	259,789	259,789	4,242,673
patents_main	240,547	240,547	560,943
pdb1HYS	36,417	36,417	4,344,765
roadNet-CA	1,971,281	1,971,281	5,533,214
web-BerkStan	685,230	685,230	7,600,595
web-Google	916,428	916,428	5,105,039
web-NotreDame	325,729	325,729	1,497,134

data, LDA, NMF(nonnegative matrix factorization) with KL-divergence loss, ALS etc. SpMV, SpMM, and SDDMM primitives are the bottleneck kernels for these algorithms. BIDMach has similar performance for SDDMM as SpMM which is a 100-fold improvement over naive JAVA or C implementation. Comparing to the performance of SDDMM using NVIDIA sparse library, BIDMach native SDDMM kernel achieves 3.7x performance improvement [12].

BIDMach-GPU processes assigns a single thread to process each 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 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) reduction across WARPs is required and is implemented using expensive atomic operations. In our methods, we fully avoid reduction across WARPs and only allow reduction across thread within a WARP when virtual WARP is used. E.g. if virtual WARP of 16 is used, it means 2 thread will process K computations and only one reduction is required between the two threads. 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 picks 1024 as batch size. For sparse matrices, selecting batch size as small as 1024 affects the performance due to underutilization of GPU resources. We tune the batch size to 1024, 10,000 and 50,000 and compare our results with the best performing results chosen from these batch sizes.

B. Evaluation

In this section, the performance of our proposed SM-L2 scheme and BIDMach-GPU is compared. The hardware details are shown in Table II. Single precision data type is used in both cases and we present achieved GFLOPS for K=32, 128 and 512. Figure 7 shows the comparison of GFLOPS among BIDMach, model, and exhaustive approach. In **Model** approach, 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. E.g. 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 the program for all possible combinations of slice sizes and tile sizes and selects the best Tj and Tk pair.

Figure 7 shows that our scheme consistently outperforms BIDMach by a factor of $1\times$ to $4.6\times$. We achieve up to 414 GFLOPS by using our model predicted tile size and auto-tuned slice size for k =512. For k=128 and k=32, our best performance is 403 GFLOPS and 324 GFLOPS respectively. On the other hand, BIDMach's performance improves with increasing number of K. It achieves around up to 107 GFLOPS on average using K=512. This result is expected as BIDMach parallelize over the number of flops. With more number of K, it provides better parallelism and occupancy. However, after a threshold, the reduction cost across threads becomes high and the performance saturates. We efficiently parallelize the work over K so that work load and reduction cost is the balance. For the most commonly used bag of words dataset like NYTimes, our performance improvement for K=32, 128 and 512 is $5\times$, $3\times$ and $2.6\times$ respectively. Similarly, for PubMed performance improvement for K=32 and 128 is $5\times$ and $3\times$ respectively.

C. Effectiveness of model

Figure 8 shows the loss of performance for selecting Tj and Tk using our model instead of exhaustive search. It can be seen that, for most cases, our model is able to predict a configuration which achieves performance close to the best one selected using exhaustive search.

D. Impact of model derived tile size

Our analysis is based on the assumption that $Tk * Tj$ tile 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, assuming LRU policy, we expect these elements to get evicted faster than the B elements. 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 shared-memory size and thread block size such that we achieve full occupancy. Choosing $Tk*Tj$ to be lower than L2 cache size can help to improve the probability of B elements being served from cache. However, since the loads for A is inversely proportional to Tj, the latter can choice can result in increasing the total number of DRAM transactions. Table IV shows the effect of different tile choices of Tj on total DRAM read transactions (measured using NVPROF). We calculate theoretical data movement of A as $numberoftiles \times N \times K$ and data movement of S as $3 \times nnz$. As Tj decreases the data movement for B decreases, but the total DRAM transactions increases.

E. Speedup on Multi-GPU scheme

Figure 9 shows the speedup of our Multi GPU implementation using the non-symmetric scheme described in Subsection IV-C. cuSDDMM achieves almost linear speedup over NYTimes dataset for K values of 32, 64, 128, 256, 512 and

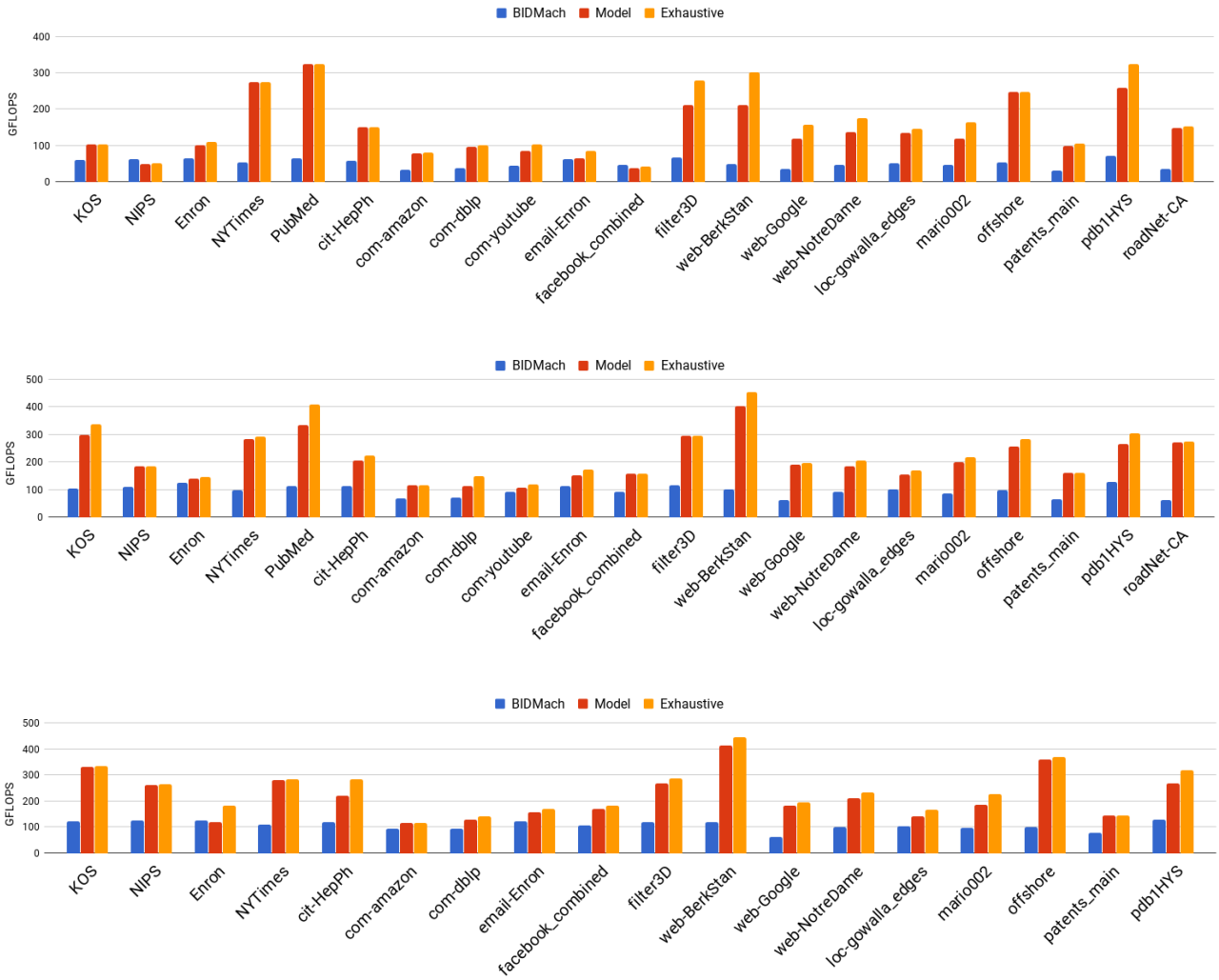


Fig. 7: GFLOPS with (a)K = 32, (b)K = 128 and (c)K = 512 on Tesla P100 GPU using single precision

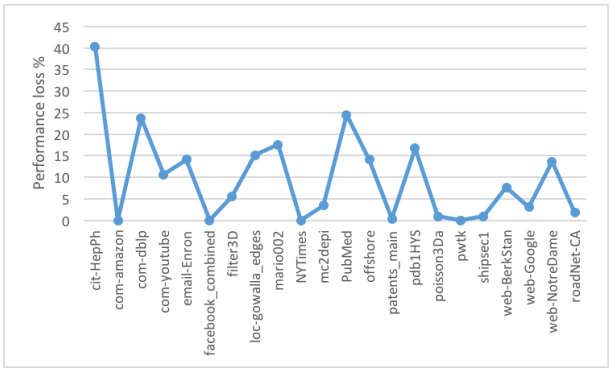


Fig. 8: Performance loss caused by using model predicted tile size (T) compared to best tile size via exhaustive search

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

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

1024. The primary reason behind the weak scaling of other matrices is the insufficient amount of work to occupy a GPU. Table III, 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.

F. Impact on ML applications

Table V shows the achievable application speedup by using cuSDDMM instead of default kernel from BIDMach. We use SFA algorithm to demonstrate our results in a total application. We estimated the application speedup by computing the

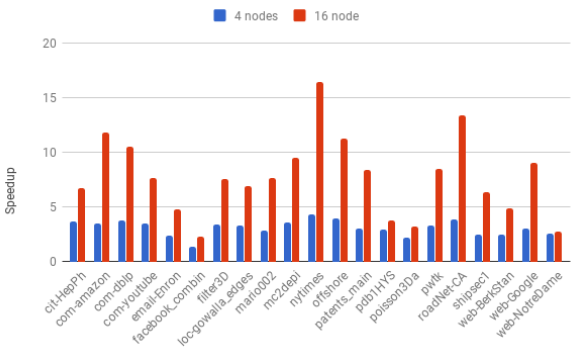


Fig. 9: Speedup achieved by using 4 and 16 Tesla P100 GPUs on using K=512 and single precision data type

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

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

fraction of time spend on 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 [16], [4], Sparse Factor Analysis (SFA) [17], [2], Gamma Poisson (GaP) [18], [5], ALS [1] and so on. With the advent of many-core architectures, researchers have considered optimizating these algorithms on these architectures and also scaled them on distributed memory systems [19], [20], [21], [22], [23].

CuMF [24] presents such a matrix factorization library to solve ALS based MF on a single and multiple GPUs. Other works such as [25] and [8] uses SGD - Stochastic Gradient Descent and CCD++ - Cyclic Coordinate-based techniques on GPU to perform MF. Recently, Li et al. [26] proposes a novel LDA technique on GPU which maintains high accuracy as well as speed. Many of these algorithms like LDA, SFA etc. can be formulated using SDDMM kernel and an optimized SDDMM kernel can aid in improving the performance of several ML algorithms. We show such formulation in Section II. There have been several attempts to boost the performance of these algorithms by using a faster SDDMM kernel [27], [3], [6].

VII. Conclusion

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

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. Proceedings. 2002 IEEE Symposium on*. IEEE, 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 *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 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 *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2004, pp. 122–129.

[6] J. Canny and H. Zhao, "Bidmach: Large-scale learning with zero memory allocation," in *BigLearn workshop, NIPS*, 2013.

[7] 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.

[8] I. Nisa, A. Sukumaran-Rajam, R. Kunchum, and P. Sadayappan, "Parallel ccd++ on gpu for matrix factorization," in *Proceedings of the General Purpose GPUs*. ACM, 2017, pp. 73–83.

[9] C. NVIDIA, "Cusparse library," *NVIDIA Corporation, Santa Clara, California*, 2014.

[10] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[11] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[12] H. Zhao, *High Performance Machine Learning through Codesign and Rooflining*. University of California, Berkeley, 2014.

[13] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[14] J. Walter, M. Koch *et al.*, "ublas," http://www.boost.org/doc/libs/1_66_-0/libs/numeric/ublas/doc/index.html, 2012.

[15] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," 2017.

[16] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.

[17] 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.

[18] M. K. Titsias, "The infinite gamma-poisson feature model," pp. 1513–1520, 2008.

[19] 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.

[20] Y. You, A. Buluc, and J. Demmel, "Scaling Deep Learning on GPU and Knights Landing clusters," *ArXiv e-prints*, Aug. 2017.

[21] "Caffeonspark github project," https://github.com/yahoo/CaffeOnSpark, accessed: 2017-01-23.

[22] "Paddlepaddle," https://github.com/paddlepaddle/paddle, accessed: 2017-01-23.

[23] "Amazon dsstne github project," https://github.com/amzn/amazon-dsstne, accessed: 2017-01-23.

[24] W. Tan, L. Cao, and L. Fong, "Faster and cheaper: Parallelizing large-scale matrix factorization on gpus," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 219–230.

[25] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "CuMF_SGD: Fast and scalable matrix factorization," *CoRR*, vol. abs/1610.05838, 2016.

[26] K. Li, J. Chen, W. Chen, and J. Zhu, "Saberlda: Sparsity-aware learning of topic models on gpus," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 497–509.

[27] J. Canny and H. Zhao, "Big data analytics with small footprint: Squaring the cloud," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 95–103.