DistME: A Fast and Elastic Distributed Matrix Computation Engine using GPUs

Donghyoung Han
DGIST, Republic of Korea
icedrak@dgist.ac.kr

Yoon-Min Nam
DGIST, Republic of Korea
ronymin@dgist.ac.kr

Jihye Lee
DGIST, Republic of Korea
jh_lee@dgist.ac.kr

Kyongseok Park
KISTI, Republic of Korea
gspark@kisti.re.kr

Hyunwoo Kim
KISTI, Republic of Korea
pardess@kisti.re.kr

Min-Soo Kim*
DGIST, Republic of Korea
mskim@dgist.ac.kr

ABSTRACT

Matrix computation, in particular, matrix multiplication is time-consuming, but essentially and widely used in a large number of applications in science and industry. The existing distributed matrix multiplication methods only focus on either low communication cost (i.e., high performance) with the risk of out of memory or large-scale processing with high communication overhead. We propose a distributed elastic matrix multiplication method called CuboidMM that achieves both high performance and large-scale processing. We also propose a GPU acceleration method that can be combined with CuboidMM. CuboidMM partitions matrices into cuboids for optimizing the network communication cost with considering memory usage per task, and the GPU acceleration method partitions a cuboid into subcuboids for optimizing the PCI-E communication cost with considering GPU memory usage. We implement a fast and elastic matrix computation engine called DistME by integrating CuboidMM with GPU acceleration on top of Apache Spark. Through extensive experiments, we have demonstrated that CuboidMM and DistME significantly outperform the state-of-the-art methods and systems, respectively, in terms of both performance and data size.

*Corresponding author.

KEYWORDS

Matrix multiplication, Distributed data-parallel system, GPU computation

ACM Reference Format:
Donghyoung Han, Yoon-Min Nam, Jihye Lee, Kyongseok Park, Hyunwoo Kim, and Min-Soo Kim. 2019. DistME: A Fast and Elastic Distributed Matrix Computation Engine using GPUs. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3299869.3319865

1 INTRODUCTION

Matrix computation is essentially and widely used in a large number of applications in various fields such as database, machine learning, health, music, and games [4]. The applications in the machine learning field include collaborative filtering, Cholesky factorization, singular value decomposition (SVD), LU factorization, betweenness centrality, and deep neural network. As the sizes of real matrix dataset are growing rapidly, fast and scalable matrix computation systems have become more important than ever before. For example, for collaborative filtering, the sizes of the Netflix competition dataset [41] are 100 million ratings, 480,000 users, and 17,770 items, and those of Facebook's dataset are 100 billion ratings, more than a billion users, and millions of items [21].

Matrix computation, in particular, matrix multiplication is time-consuming due to its high computational complexity of $O(N^3)$ when input matrices are of $N \times N$. In order to process large-scale matrix computation in a fast and scalable way, a number of distributed matrix computation systemson top of MapReduce-based frameworks such as SystemML [6, 18], Marlin [19], Mahout [29], DMac [37], and MatFast [38] have been proposed.

For large-scale matrices, these systems perform matrix multiplication as the following three steps: (1) repartitioning input matrices among tasks (matrix repartition); (2) performing local matrix multiplication within each task (local multiplication); (3) aggregating the intermediate results of local matrix multiplication by shuffling them (matrix aggregation). For fast matrix multiplication, they usually focus on reducing the communication overhead occurred in the matrix repartition and aggregation steps since the total number of low-level multiplication operations is the same regardless of a method used [37, 38].

For the matrix repartition and aggregation steps, the existing systems have proposed or used the following three methods: Broadcast Matrix Multiplication (BMM) [6, 18, 37, 38], Cross Product-based Matrix Multiplication (CPMM) [6, 18, 37, 38], and Replication-based Matrix Multiplication (RMM) [6, 18, 26]. The BMM method broadcasts a smaller input matrix to all tasks, and the CPMM method performs multiple outer products between both input matrices and aggregates the results of outer products. Thus, BMM has relatively high communication overhead in the matrix repartition step, and CPMM has relatively high communication overhead in the matrix aggregation step. Both methods are usually faster than the other method, RMM, but tend to fail for large-scale matrices due to their excessive memory usage per task. The RMM method repartitions input matrices into much smaller-size units called *blocks*, and so, can process large-scale matrix multiplication without out of memory error. However, it tends to have much higher communication overhead than the other methods.

As explained above, the existing systems have drawbacks of either risk of out of memory due to high memory usage per task (BMM and CPMM) or degradation of performance due to high communication cost (RMM). In addition, they have another drawback of not exploiting hardware acceleration of modern processors (e.g., GPUs) that can significantly improve the performance of the local multiplication step. For low-level matrix multiplication, the existing systems including SystemML [6, 18], DMac [37], and MatFast [38] use CPU-based libraries such as LAPACK [1], ATLAS [35], and Intel MKL [32]. If we could use GPU-based libraries such as cuBLAS [14] (for dense matrix) and cuSPARSE [27] (for sparse matrix), the performance of the local multiplication step would be significantly improved. However, it is nontrivial to design such a method (or system) since it requires taking the characteristics of both distributed systems and GPUs into account.

To alleviate the above drawbacks, we propose a distributed elastic matrix multiplication method called *CuboidMM* that can achieve both low communication cost and low memory usage per task. The CuboidMM method partitions input matrices into multiple pieces called *cuboids* that have the optimal sizes in terms of communication cost and memory usage per task. In fact, it is a generalization of the existing three methods. The optimal size of cuboid varies depending on the sizes of input matrices and system resources available. As a result, CuboidMM outperforms all the existing methods, BMM, CPMM, and RMM, in terms of both the elapsed time and the maximum sizes of matrices that can be computed without failure. In addition, we propose a GPU acceleration method of matrix multiplication that can be seamlessly combined with CuboidMM. It partitions each cuboid into multiple *subcuboids* that have the optimal sizes in terms of both the PCI-E communication cost between main memory and GPU memory and the GPU memory usage per task.

We implement a fast and elastic matrix computation engine called *DistME* by integrating our proposed CuboidMM and GPU acceleration method seamlessly on top of Spark [40] distributed data-parallel framework. DistME improves the performance of all three steps of distributed matrix multiplication compared with the existing systems, in particular, the matrix repartition and aggregation steps due to CuboidMM, and the local multiplication step due to GPU acceleration. It also can handle not only matrix multiplication, but also a complex query like matrix factorization. Through extensive experiments using both real and synthetic data, we have demonstrated that CuboidMM improves the elapsed time up to by 3.92 times and reduces the communication cost up to by 60.39 times compared with the existing methods. We also have shown that DistME significantly outperforms the state-of-the-art systems in terms of both performance and data size that can be processed.

Our major contributions are summarized as follows:

- We propose an elastic method, CuboidMM that can process distributed matrix multiplication in an optimal manner in terms of network communication cost and memory usage per task.
- We propose a GPU acceleration method for the local matrix multiplication that can be integrated with CuboidMM in an optimal manner in terms of PCI-E communication cost and GPU memory usage per task.
- We implement a matrix engine DistME on top of Spark that improves the performance of all three steps of distributed matrix multiplication with CuboidMM and GPU acceleration.
- Through extensive experiments, we have demonstrated that CuboidMM improves the performance of the existing methods up to by 60.39 times, and DistME significantly outperforms the state-of-the-art systems.

The rest of this paper is organized as follows. Section 2 reviews the existing methods for distributed matrix multiplication. In Section 3, we present the CuboidMM method for optimizing the distributed matrix multiplication. In Section 4, we present the GPU computation method that can be combined with CuboidMM for accelerating matrix multiplication. Section 5 presents the implementation of DistME briefly, and Section 6 presents the results of the experimental evaluation.

Table 1: Summary of symbols.

Symbol	Description
A, B, C	input and output matrices
$\|A\|$	size of matrix A (number of elements of A)
I, J, K	number of blocks on the i-, j-, and k-axis
P, Q, R	number of partitions on the i-, j-, and k-axis
P_2, Q_2, R_2	number of subpartitions on the i-,j-,k-axis
$A_{i,k}$	block of A located at (i, k)
$v_{i,j,k}$	voxel located at (i, j, k)
$D_{p,q,r}$	cuboid located at (p, q, r)
S_{p_2,q_2,r_2}	subcuboid located at (p_2, q_2, r_2)
T	number of all tasks
T_c	number of concurrent tasks per cluster node
M	number of cluster nodes

Finally, we discuss related work in Section 7 and conclude this paper in Section 8.

2 PRELIMINARIES

In this section, we explain the matrix partitioning schemes in Section 2.1 and the existing distributed matrix multiplication methods in Section 2.2. We summarize the symbols used in this paper in Table 1.

2.1 Matrix Partitioning Schemes

To process large-scale matrices efficiently, most of distributed matrix computation systems represent a matrix as a grid of fixed-sized *blocks* and use a block as a basic unit of matrix computation [6, 8, 18, 19, 31, 37, 38]. Here, a block typically has the same width and height, i.e., 1000×1000. Each block can be represented either in a dense format or sparse formats such as Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR) [15].

In general, distributed matrix computation systems need to split the matrices into a number of *partitions* and assign the partitions to the tasks running on the cores of the cluster for parallel computation. There are the following representative partitioning schemes: *Row*, *Column*, *Hash*, and *Grid* partitioning schemes. Figure 1 shows an example of partitioning a matrix A of 4×4 blocks into four tasks according to each partitioning scheme, where the blocks of the same color belong to the same partition. We consider each block has its own index $A_{i,j}$ in a matrix A where i is a row index, and j is a column index, as in Figure 1(a).

- **Row/Column** partitioning schemes [37, 38] distribute the blocks in the same row/column block index to the same task, as shown in Figures 1(a) and (b).
- **Hash** partitioning scheme [6, 18] determines the task for each block by using a hash function, as shown in Figure 1(c). The hash function allows the blocks to be evenly distributed among the tasks.
- **Grid** partitioning scheme [9] divides a matrix into a number of grids of $\alpha \times \beta$ blocks, where α is the number of blocks in the row, and β is the number of blocks in the column. Figure 1(d) shows an example of 2×2 grid partitioning of the matrix A.

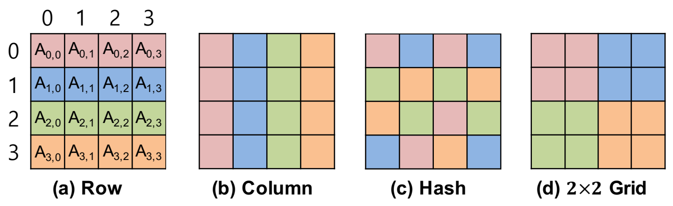


Figure 1: Example of partitioning schemes for the matrix A.

The above partitioning schemes divide a matrix into disjoint partitions, each of which is copied to a single task for computation. However, matrix multiplication of $A \times B$ requires either A or B to be copied to all tasks, i.e., *broadcasted* in many cases, which will be explained in Section 2.2.

2.2 Distributed Matrix Multiplication

In matrix multiplication of $C = A \times B$, each block of $C_{i,j}$ ($0 \le i < I$, $0 \le j < J$) can be computed as in Eq.(1) where the input matrices A and B have $I \times K$, $K \times J$ blocks, respectively, and the output matrix C has $I \times J$ blocks.

$$C_{i,j} = \sum_{0 \le k < K} A_{i,k} \cdot B_{k,j} \tag{1}$$

The matrix multiplication is often represented as a 3-dimensional model having i-axis, j-axis, and k-axis where $0 \le i < I$, $0 \le j < J$, and $0 \le k < K$. The ik-plane of the model indicates the matrix A, the kj-plane of the model the matrix B, and the ij-plane of the model the matrix C. The areas of A, B, and C are $I \times K$ blocks, $K \times J$ blocks, and $I \times J$ blocks, respectively. Thus, the volume of the model becomes $I \times J \times K$ *voxels*, each of which means a computational unit of matrix multiplication, i.e., computing an *intermediate block* of $C_{i,j}$ by multiplication between two blocks $A_{i,k} \cdot B_{k,j}$. We denote the intermediate block, i.e., the result of $A_{i,k} \cdot B_{k,j}$ for a specific k, by $C^k_{i,j}$. We also denote the index of a specific voxel by $v_{i,j,k}$ in the model. Figure 2(a) shows a 3-dimensional model for $4 \times 4 \times 4$ voxels, where the computation in Eq.(1) corresponds to an array of voxels on the k-axis, i.e., $[v_{i,j,0}, \cdots, v_{i,j,K-1}]$.

Without loss of generality, distributed matrix multiplication is performed as the following three steps [19].

- ***Matrix repartition*** step: the input matrices are repartitioned or broadcasted to the tasks of a distributed system.

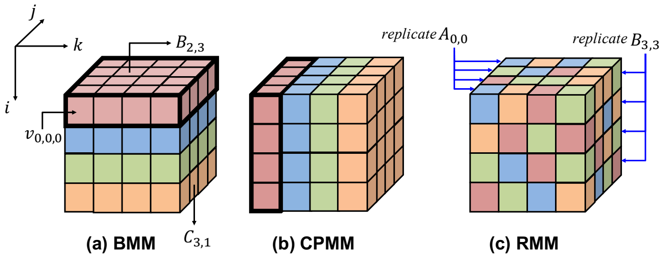


Figure 2: The 3-dimensional model for matrix multiplication.

- ***Local multiplication*** step: in each task, the input matrix blocks are multiplied to generate the intermediate blocks of the output matrix.
- ***Matrix aggregation*** step: the intermediate blocks are shuffled to generate the final output matrix (this step is optional depending on the strategy of the matrix repartition step).

The existing methods for distributed matrix multiplication can be categorized into the following three groups depending on the strategy of the matrix repartition step: Broadcast Matrix Multiplication (*BMM*) [6, 18, 37, 38], Cross Product-based Matrix Multiplication (*CPMM*) [6, 18, 37, 38], and Replication-based Matrix Multiplication (*RMM*) [6, 18, 26]. We explain each method in more detail.

2.2.1 ***BMM.*** The matrix repartition step of BMM partitions the input matrix A to each task according to the row partitioning scheme in Section 2.1 and broadcasting the input matrix B to all the tasks, if the matrix B is smaller than the matrix A. Figure 2(a) shows the BMM method when both A and B are of 4×4 blocks, and there are four tasks. In the figure, the voxels in different colors indicate the computation in different tasks. For example, the first task in red takes the first row of A (i.e., four blocks in red on the ik-plane) and the entire B (i.e., 16 blocks in red on the kj-plane) as input. Then, the BMM method performs the local multiplication step between both inputs and generates four blocks $C_{0,0}$, $C_{0,1}$, $C_{0,2}$, and $C_{0,3}$ as output. Each of these four output blocks is computed according to Eq.(1). These blocks are not intermediate blocks, but the final blocks for the output matrix C, and so, the BMM method does not require the matrix aggregation step.

The BMM method is similar to the broadcast join of a distributed DBMS [5]. Most of distributed matrix multiplication systems including SystemML [6, 18], DMac [37], and MatFast [38] use BMM as a default matrix multiplication method for small matrices. We let the number of tasks be T. The memory usage per task becomes $\frac{|A|}{T} + |B|$ for input and $\frac{|C|}{T}$ for output. Here, $|A|$ indicates the size of the matrix A in terms of the number of elements. The communication cost, i.e., the amount of data transferred via the network, becomes $|A| + T \cdot |B|$ in the matrix repartition step and zero in the matrix aggregation step.

2.2.2 ***CPMM.*** The matrix repartition step of CPMM partitions the input matrix A to each task according to the column partitioning scheme and partitions the input matrix B to each task according to the row partitioning scheme. Figure 2(b) shows the CPMM method. For example, the first task in red takes the first column of A (i.e., $A_{0,0}$, $A_{1,0}$, $A_{2,0}$, and $A_{3,0}$) and the first row of B (i.e., $B_{0,0}$, $B_{0,1}$, $B_{0,2}$, and $B_{0,3}$) as input. Then, the CPMM method performs the local multiplication step between both inputs, which is actually an outer product at the block level, and so, generates 16 intermediate blocks $\{C^k_{i,j} | 0 \le i < 4, 0 \le j < 4, k = 0\}$ as output. Since these blocks are not final blocks of the output matrix C, the CPMM method performs the matrix aggregation step where each four of them are copied to the same task among four tasks.

The CPMM method is used in MatFast [38], DMac [37], and SystemML [6, 18]. The memory usage per task becomes $\frac{|A|}{T} + \frac{|B|}{T}$ for input and $|C|$ at most for output. The communication cost in the matrix repartition step is $|A| + |B|$, and that in the matrix aggregation step is $T \cdot |C|$ at most. The actual cost of the matrix aggregation step depends on the sparsity of the intermediate blocks of C generated by the local multiplication step. The actual cost may be lower than $T \cdot |C|$, but most of distributed matrix computation systems including SystemML [6, 18] and DMac [37] use the worst-case complexity for estimating the sparsity of intermediate blocks, and so, we also use the worst-case complexity.

2.2.3 ***RMM.*** In the above methods, a task may fail due to its high memory usage per task, i.e., $\frac{|A|}{T} + |B| + \frac{|C|}{T}$ for BMM and $\frac{|A|}{T} + \frac{|B|}{T} + |C|$ for CPMM, as the size of either B or C is very large. Even though we increase the number of tasks, the problem cannot be solved due to the limit on the number of tasks in those methods. In principle, the maximum number of possible tasks in BMM becomes I when broadcasting the matrix B, and that in CPMM becomes K. The RMM method can be used for solving this problem with more communication cost.

The matrix repartition step of RMM replicates every A's blocks J times and every B's blocks I times and shuffles them using the index of the corresponding voxel as a key. For example, in Figure 2(c), the block $A_{0,0}$ is replicated $J = 4$ times and shuffled as $\langle (i = 0, j = 0, k = 0), A_{0,0} \rangle$, $\langle (0, 1, 0), A_{0,0} \rangle$, $\langle (0, 2, 0), A_{0,0} \rangle$, and $\langle (0, 3, 0), A_{0,0} \rangle$. In $\langle \cdot, \cdot \rangle$, the former indicates a key, and the latter a value. Thus, the RMM method uses the hash partitioning scheme in Section 2.1 for all the replicated blocks of A and B. After all replicated blocks of A and B are shuffled, a task takes a set of A block and B block pairs having the same key. For example, a task takes a pair of $A_{0,0}$ and $B_{0,0}$ having the same key $(0, 0, 0)$. Then, the task performs the local multiplication step for each pair of blocks to generate the intermediate blocks including $C^0_{0,0}$. In the matrix aggregation step, the RMM method shuffles these intermediate blocks such that the ones having the same i and j indices are gathered for computing $C_{i,j}$.

The RMM method is used in SystemML [6, 18] and Spark MLlib [26] when processing large-scale matrix multiplication. The communication cost in the matrix repartition step is $J \cdot |A| + I \cdot |B|$, and that in the matrix aggregation step is $K \cdot |C|$ at most. Different from BMM and CPMM, the RMM method can distribute the workload to up to $I \cdot J \cdot K$ tasks since the number of all replicated blocks of A (or B) is $I \cdot J \cdot K$. Here, the memory usage per task becomes $\frac{J \cdot |A|}{T} + \frac{I \cdot |B|}{T}$ for input and $\frac{K \cdot |C|}{T}$ for output.

2.2.4 ***Comparison.*** Table 2 summarizes the communication cost, memory usage per task, and maximum parallelism of the BMM, CPMM, and RMM methods. Since the total number of multiplication operations is the same regardless of BMM, CPMM, or RMM, the performance of those methods mainly depends on the communication cost [37, 38]. Thus, BMM (or CPMM) tends to be faster than RMM due to its smaller communication cost, as long as $|B|$ (or $|C|$) fits in the memory of a task. In contrast, RMM has much better scalability than BMM and CPMM in terms of the number of tasks and can process without out of memory even in the case when $|B|$ (or $|C|$) cannot fit in the memory of a task. Our CuboidMM will be explained in Section 3.

3 CUBOID MATRIX MULTIPLICATION

In this section, we propose the Cuboid Matrix Multiplication (CuboidMM) method that pursues both the high performance of BMM or CPMM and the scalability of RMM in terms of data size. We present the concept and steps of CuboidMM method in Section 3.1 and the optimization of parameters used in CuboidMM in Section 3.2.

3.1 (P, Q, R)-Cuboid partitioning

The RMM method in Section 2.2.3 achieves good scalability in terms of the size of the matrix by using the smallest unit in the 3-dimensional model, i.e., a voxel, as a unit for workload distribution in the matrix repartition step. The size of a voxel, i.e., the sum of the sizes of a single block of A, a single block of B, and a resulting single block of C, is small enough to be processed in a single task without a lack of memory. Such a small size, however, causes a large amount of communication overhead due to the replication of every block of A, B, and C matrices J, I, and, K times, respectively, where J, I, and K can be large (e.g., 100,000).

The CuboidMM method conceptually partitions the 3-dimensional model space into multiple cuboid-shaped chunks of voxels such that the size of each cuboid becomes the biggest one that can fit in the memory of a task. Here, the size of a cuboid means the sum of the sizes of the blocks of A, B, and C in the cuboid. For partitioning, we use three parameters P, Q, and R that mean the number of partitions on the i-axis, j-axis, and k-axis, respectively. This partitioning scheme makes a total of $P \cdot Q \cdot R$ cuboids, and so, we denote it by (P, Q, R)-cuboid partitioning. In fact, the CuboidMM method performs the grid partitioning scheme in Section 2.1 for each of the input matrices A and B. The parameters should satisfy the condition that $0 < P \leq I$, $0 < Q \leq J$, and $0 < R \leq K$. Figure 3(a) shows an example of $(P = 2, Q = 2, R = 2)$-cuboid partitioning for matrix multiplication where A is of 4×8 blocks, B is of 8×6 blocks, and C is of 4×6 blocks. In general, each cuboid consists of $\lceil \frac{I}{P} \rceil \times \lceil \frac{J}{Q} \rceil \times \lceil \frac{K}{R} \rceil$ voxels. For instance, a cuboid in Figure 3(a) consists of $2 \times 3 \times 4$ voxels. We denote the index of a specific cuboid by $D_{p,q,r}$ where $0 \leq p < P$, $0 \leq q < Q$, and $0 \leq r < R$. For instance, the index of the first cuboid in gray is $D_{0,0,0}$.

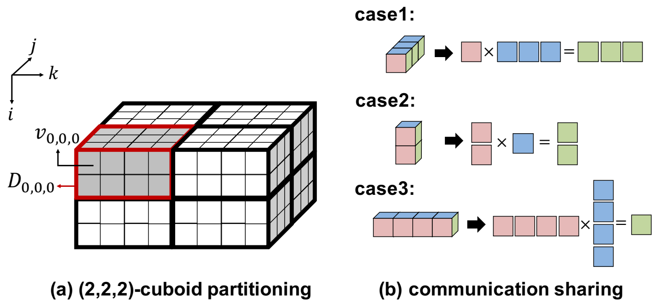


Figure 3: Example of (P, Q, R)-cuboid partitioning.

The CuboidMM method can significantly reduce the communication cost occurred in the RMM method by sharing network communication among consecutive voxels in the 3-dimensional model. Figure 3(b) shows three cases that reduce the communication cost by using $(2, 2, 2)$-cuboid partitioning of Figure 3(a). In case 1, computation of three consecutive voxels on the j-axis requires replicating each block of A only once instead of three times, as long as the cuboid is processed in a single task. Likewise, in case 2, the computation of two consecutive voxels on the i-axis requires replicating each block of B only once instead of two times. Finally, in case 3, the computation of four consecutive voxels on the k-axis avoids shuffling four intermediate blocks of C. Thus, cases 1 and 2 can reduce the communication cost in the matrix repartition step, while case 3 can reduce that in the matrix aggregation step. More specifically, CuboidMM reduces the communication cost compared with RMM up to by $\frac{J}{Q}$ times

Table 2: Comparison among matrix multiplication methods ($|A| > |B|$, $P \leq I$, $Q \leq J$, and $R \leq K$).

methods	communication cost		memory usage	maximum number
	matrix repartition	matrix aggregation	per task	of tasks
BMM	$\|A\| + T \cdot \|B\|$	-	$\frac{\|A\|}{T} + \|B\| + \frac{\|C\|}{T}$	I
CPMM	$\|A\| + \|B\|$	$T \cdot \|C\|$	$\frac{\|A\|}{T} + \frac{\|B\|}{T} + \|C\|$	K
RMM	$J \cdot \|A\| + I \cdot \|B\|$	$K \cdot \|C\|$	$\frac{J \cdot \|A\|}{T} + \frac{I \cdot \|B\|}{T} + \frac{K \cdot \|C\|}{T}$	$I \cdot J \cdot K$
CuboidMM	$Q \cdot \|A\| + P \cdot \|B\|$	$R \cdot \|C\|$	$\frac{Q \cdot \|A\|}{T} + \frac{P \cdot \|B\|}{T} + \frac{R \cdot \|C\|}{T}$	$I \cdot J \cdot K$

for A, by $\frac{I}{P}$ times for B, and by $\frac{K}{R}$ times for C. We note that the RMM method cannot reduce the communication cost when using the same number of tasks with CuboidMM (i.e., the same number of voxels per task) because a task in RMM processes non-consecutive voxels due to the hash partitioning scheme. The bottom row in Table 2 summarizes the cost of CuboidMM. Since a single cuboid is processed by a single task, the memory usage per task of CuboidMM is equal to the size of a cuboid in Table 2.

In fact, CuboidMM is a generalization of the existing three methods, BMM, CPMM, and RMM, and so, can perform matrix multiplication like either BMM, CPMM, or RMM by changing the parameters P, Q, and R. For example, in Figure 3(a), CuboidMM using $(4, 1, 1)$-cuboid partitioning works like BMM, that using $(1, 1, 8)$-cuboid partitioning works like CPMM, and that using $(4, 6, 8)$-cuboid partitioning works like RMM.

Figure 4 shows the steps of distributed matrix multiplication of CuboidMM for the example in Figure 3(a). We assume the number of tasks $T = 8$. In the matrix repartition step, each of $P \cdot Q \cdot R = 8$ cuboids is assigned to each task. In the local multiplication step, a total of eight tasks $\{t_0, \cdots, t_7\}$ perform their own workload that multiplies 2×4 blocks of A by 4×3 blocks of B and produces 2×3 intermediate blocks of C. This step is accelerated by GPU computation, which will be presented in Section 4. In the matrix aggregation step, the intermediate blocks of C from a pair of tasks $\langle t_o, t_{o+1} \rangle$ $(0 \leq o < 4)$ are aggregated to obtain the final output blocks of C. In general, the intermediate blocks from R tasks, e.g., $\{t_o, \cdots, t_{o+R-1}\}$ $(0 \leq o < P \cdot Q)$ should be aggregated for the final blocks.

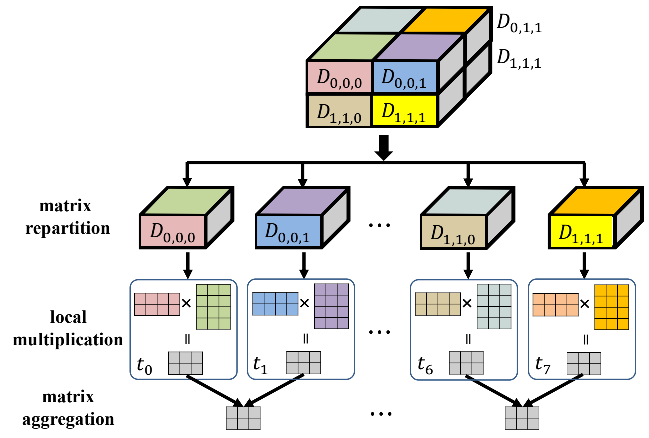


Figure 4: Steps of CuboidMM using $(2, 2, 2)$-cuboid partitioning.

3.2 Optimization of CuboidMM

For maximizing the performance of CuboidMM, it is important to find the best parameters P^*, Q^*, and R^* that can reduce the communication cost as much as possible. We assume that every task can use the same amount of memory, θ_t, which is typically equal to the amount of main memory divided by the number of concurrent tasks per node, T_c. Then, our optimization problem can be formulated as in Eq.(2), where $Mem(P, Q, R)$ is a function of memory usage per task when using the parameters P, Q, and R, and $Cost(P, Q, R)$ a function of communication cost for the parameters.

$$(P^*, Q^*, R^*) = \underset{c \in \{(P,Q,R) | Mem(P,Q,R) \leq \theta_t\}}{\text{argmin}} Cost(c) \tag{2}$$

The function $Mem()$ can be defined as in Eq.(3), where the terms $\frac{|A|}{P \cdot R}$, $\frac{|B|}{R \cdot Q}$, and $\frac{|C|}{P \cdot Q}$ indicate the average numbers of elements per cuboid in the matrices A, B, and C, respectively. Here, we know both $|A|$ and $|B|$ and estimate $|C|$ as a fully dense matrix as mentioned in Section 2.2.2.

$$Mem(P, Q, R) = \frac{|A|}{P \cdot R} + \frac{|B|}{R \cdot Q} + \frac{|C|}{P \cdot Q} \tag{3}$$

The function $Cost()$ can be defined as in Eq.(4), where the terms $Q \cdot |A| + P \cdot |B|$ indicate the amount of replicated data of A and B in the matrix repartition step, and the term $R \cdot |C|$ indicates the amount of shuffled data of C in the matrix aggregation step.

$$Cost(P, Q, R) = Q \cdot |A| + P \cdot |B| + R \cdot |C| \tag{4}$$

We find the optimal parameters (P^*, Q^*, R^*) using exhaustive search. Here, the execution time is almost negligible due to their limited search space. Although the matrices are very large in terms of the number of elements, the search space of $I \times J \times K$ is usually not so large, since I, J, and K are the numbers of blocks. For example, in our experiments, when both A and B matrices are $100,000 \times 100,000$, and block size is 1000×1000, determination of the optimal parameters takes only 0.3 seconds using a single thread.

If the sizes of input matrices are relatively small, the matrices may be fit in the available memory of a single or few tasks. In this case, if we determine P, Q, and R such that $P \times Q \times R < M \times T_c$, where M is the number of cluster nodes, we cannot fully exploit the parallelism of a distributed system. Thus, we prune the parameters such that $P \times Q \times R < M \times T_c$ from the search space when solving Eq.(2). In the exceptional case where $I \times J \times K < M \times T_c$, we determine the parameters as $P^* = I$, $Q^* = J$, and $R^* = K$ for exploiting the parallelism as much as possible, which actually works like the RMM method.

4 ACCELERATION OF CUBOIDMM USING GPUS

In this section, we propose a method that can accelerate distributed matrix multiplication by exploiting GPUs. We present the concept of subcuboid partitioning in Section 4.1 and the optimization of parameters for subcuboid partitioning in Section 4.2. Then, we present the streaming of subcuboids to GPUs in Section 4.3 and its algorithm in Section 4.4.

4.1 Subcuboid partitioning

The CuboidMM method in Section 3 partitions the entire 3-dimensional model space into multiple cuboids such that each cuboid can fit in a task memory of the capacity θ_t. There are typically multiple tasks running in a single machine. We assume that each machine can be equipped with multiple GPUs for acceleration of matrix multiplication in general, but we explain our method with a single GPU in this paper for simplicity. GPU computation is typically done by the following three steps: (1) copying input data from main memory to GPU (device) memory; (2) executing a kernel function; (3) copying output data back from GPU memory to main memory. The size of the GPU memory is usually much smaller than that of main memory. Thus, multiple tasks that run on a machine and try to use the same GPU simultaneously can lead to a serious shortage of working memory for each task in the GPU memory [7]. We denote the capacity of GPU memory per task by θ_g, which is usually smaller than θ_t. For example, when six tasks are concurrently running on a machine equipped with 64 GB main memory and a GPU of 12 GB device memory, θ_g is only 2 GB, whereas θ_t is about 10 GB.

In order to solve the shortage problem of the GPU memory, we further partition each cuboid into multiple *subcuboids* such that each subcuboid can fit in the GPU memory space of the capacity θ_g. We use the same partitioning scheme with CuboidMM for partitioning a cuboid into subcuboids. For this subcuboid partitioning, we use three parameters P_2, Q_2, and R_2 that mean the numbers of partitions in a single cuboid on the i-axis, j-axis, and k-axis, respectively. Thus, it makes a total of $P_2 \cdot Q_2 \cdot R_2$ subcuboids per cuboid, and so, we denote it by (P_2, Q_2, R_2)-subcuboid partitioning. Figure 5(a) shows an example of $(1, 1, 2)$-subcuboid partitioning for two cuboids $D_{0,0,0}$ and $D_{0,0,1}$ in Figure 4. Here, each cuboid is partitioned into two subcuboids $\{S_{0,0,0}, S_{0,0,1}\}$, each of which consists of $2 \times 3 \times 2$ voxels.

Since all the subcuboids of a task cannot fit in the GPU memory space for the task at the same time, the subcuboids are copied to and processed in the GPU sequentially. We denote in-GPU processing of a single subcuboid by an *iteration*. For example, in Figure 5(a), in the task t_0, a subcuboid $S_{0,0,0}$ is processed as $iteration_0$, and then, the other subcuboid $S_{0,0,1}$ is processed as $iteration_1$. Likewise, in task t_1, two subcuboids $\{S_{0,0,0}, S_{0,0,1}\}$ are processed as $iteration_0$ and $iteration_1$, respectively. We assume two tasks t_0 and t_1 are running on the same machine. Then, both $iteration_0$ of t_0 and $iteration_0$ of t_1 are executed concurrently in the GPU. After these are done, the succeeding $iteration_1$s are executed concurrently. The concurrent execution of multiple iterations in our DistME system is mainly implemented using CUDA Multiple Process Service (MPS) [13], which allows multiple processes to execute their kernel functions concurrently.

In many cases, a cuboid is partitioned into multiple subcuboids along the k-axis as in Figure 5(a), which will be explained in Section 4.2 in detail. Thus, the intermediate blocks of the output matrix C can be efficiently aggregated during processing all the subcuboids by keeping them in the same buffer in GPU memory. We denote the intermediate blocks of C computed by $iteration_n$ of the task t_m by $C^{m,n}$. Then, the task t_m aggregates $\{C^{m,0}, \cdots, C^{m,R_2-1}\}$ into C^m, where R_2 is the number of subcuboids along the k-axis, and C^m the result of aggregation by t_m. For example, in Figure 5(a), $C^{0,0}$ and $C^{0,1}$ are aggregated in the GPU by the task t_0, and $C^{1,0}$ and $C^{1,1}$ are aggregated in the GPU by the task t_1. The result of t_0, i.e., C^0, and that of t_1, i.e., C^1 are further aggregated by the matrix aggregation step in Figure 4.

4.2 Optimization of subcuboids for GPU

For maximizing the performance of processing a cuboid using GPU, it is important to find the best parameters P_2^*, Q_2^*, and R_2^* that reduce the communication cost between main memory and GPU memory as much as possible. The bandwidth of PCI-E bus between main memory and GPU is usually up to 16 GB/s and tends to become a performance bottleneck as the computational power of GPU increases. Since the total number of multiplication operations for processing a cuboid in GPU is the same regardless of the result of subcuboid partitioning, we focus on reducing the PCI-E

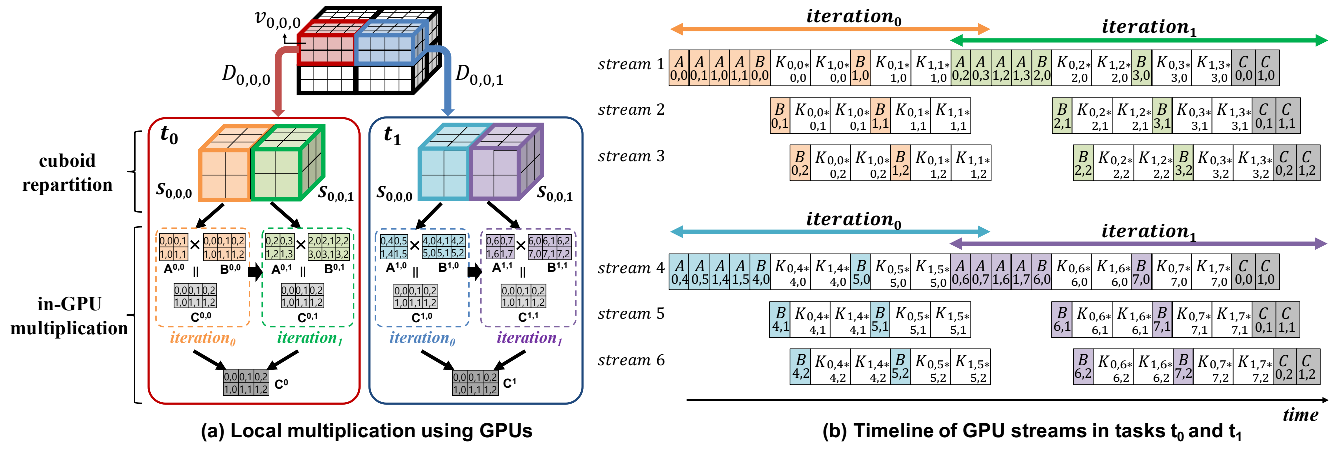


Figure 5: Acceleration of the local multiplication step using GPUs ($T = 8$, $(1, 1, 2)$-subcuboid partitioning).

communication cost, as we focus on reducing the network communication cost in Section 3.2. The optimization problem for subcuboid partitioning can be formulated as in Eq.(5).

$$(P_2^*, Q_2^*, R_2^*) = \underset{c \in \{(P_2,Q_2,R_2) | Mem^m(P_2,Q_2,R_2) \le \theta_g\}}{\text{argmin}} Cost^m(c) \quad (5)$$

The function $Mem^m()$ is basically the same function with $Mem()$ in Section 3.2 except that $Mem^m()$ considers the sizes of A and B (denoted by A^m and B^m, respectively) within the given cuboid processed by the task t_m, instead of those of the entire A and B. Different tasks process different parts of A or B, which have different sizes and sparsity.

The function $Cost^m()$ can be defined as in Eq.(6), which is slightly different from the function $Cost()$ in Section 3.2. The last term in $Cost^m()$ has no multiplication of R_2, while that in $Cost()$ is $R \cdot |C|$. If a subcuboid satisfies the condition $Mem^m(P_2, Q_2, R_2) \le \theta_g$ in Eq.(5), A^m, B^m, and C^m all can fit in the GPU memory space for t_m, and so, C^m can obviously fit in. That means we can keep and aggregate the intermediate blocks for C^m in the GPU memory without communication between main memory and GPU memory. Thus, we omit R_2 in Eq.(6). As a result, the optimization of Eq.(5) tends to produce $(1, 1, R_2)$-subcuboid partitioning. If a subcuboid does not satisfy the condition $Mem^m(P_2, Q_2, R_2) \le \theta_g$ due to its large size, in particular, due to the size of C^m, larger parameters of $P_2 > 1$ and $Q_2 > 1$ are picked from the search space so as to decrease the size of C^m. In that case, these parameters are still the ones that minimize $Cost^m()$ among all possible valid parameters. We note that a different task can use different subcuboid partitioning depending on the size and sparsity of its corresponding cuboid.

$$Cost^m(P_2, Q_2, R_2) = Q_2 \cdot |A^m| + P_2 \cdot |B^m| + |C^m| \quad (6)$$

4.3 GPU streaming of subcuboids

The naive method of processing each subcuboid using the GPU in a task t_m would be (1) copying an entire subcuboid from main memory to GPU memory (H2D copy), (2) executing kernel functions for multiplication while updating C^m, and (3) copying C^m from GPU memory to main memory (D2H copy) at the last iteration. However, execution of kernel functions in this method cannot start until the parts of A^m and B^m at $iteration_n$ (denoted by $A^{m,n}$ and $B^{m,n}$, respectively) are completely copied to GPU memory. For example, in task t_0 in Figure 5(a), $C^{0,0}$ can be calculated by calling kernel functions only after four blocks of $A^{0,0}$ and six blocks of $B^{0,0}$ are prepared in GPU memory. In order to improve the performance of the naive method, our DistME system exploits the asynchronous GPU streams (e.g., CUDA Streams), which could hide some memory access latency between GPU and main memory [22].

At each $iteration_n$, our strategy copies the smaller one between $A^{m,n}$ and $B^{m,n}$ as a chunk (H2D copy) and then copies the other bigger one in a block-by-block fashion (H2D copy) using multiple GPU streams while updating C^m. Figure 5(b) shows the timeline of six GPU streams used in the tasks t_0 and t_1 for processing two cuboids $D_{0,0,0}$ and $D_{0,0,1}$ in Figure 5(a). Here, streams 1, 2, and 3 are used by t_0, and streams 4, 5, and 6 used by t_1. We denote calling the kernel function that performs matrix multiplication between a single A block ($A_{i,k}$) and a single B block ($B_{j,k}$) by $K_{i,k*k,j}$. In stream 1, the task t_0 calls the kernel function two times, $K_{0,0*0,0}$ and $K_{1,0*0,0}$, after copying the $B_{0,0}$ block, where the first function call is for $A_{0,0} \times B_{0,0}$, and the second function call for $A_{1,0} \times B_{0,0}$. The task t_0 does similar jobs in streams 2 and 3. Here, H2D copies of these streams cannot overlap with each other since the current GPU architecture does not support it.

We note that each task in our strategy copies the set of B blocks used for updating the same C block using the same GPU stream for more efficient aggregation of C blocks. For example, in Figure 5(b), the task t_0 uses the same stream 1 for copying $B_{0,0}$, $B_{1,0}$, $B_{2,0}$, and $B_{3,0}$ (H2D copy) since it can update $C_{0,0}$ and $C_{1,0}$ consistently. After the last iteration finishes all its executions of the kernel function, it copies them back to main memory (D2H copy) in the same GPU stream.

4.4 Algorithm

Algorithm 1 presents the pseudo-code of the local multiplication step processed by a task t_m using GPU. The task first finds (P_2^*, Q_2^*, R_2^*) through the optimization in Section 4.2 and performs subcuboid partitioning. Here, for simplicity, we consider the size of a subcuboid as $I' \times J' \times K'$ and denote A-, B-, and C-side of the subcuboid by A', B', and C', respectively. Then, the task creates J' streams and allocates the buffers for A', B', and C', i.e., `BufA`, `BufB`, and `BufC`, in GPU. After the initialization, the task copies A' of the subcuboid to GPU memory, if $A' < B'$. Then, it performs asynchronous copying each block of B' to GPU memory followed by calling a kernel function by I' times consecutively. Here, the kernel function is for multiplication between a pair of blocks, i.e., $A_{i,k}$ and $B_{k,j}$. In particular, we use `cublasDgemm()` for dense matrices and `cusparseDcsrmm()` for sparse matrices, as the kernel function. There is usually a limitation on the number of concurrent streams per GPU (e.g., 32). The task may create and use more GPU streams than the limitation (i.e., $J' > 32$). Then, these streams are arranged and executed by the GPU scheduler. If the subcuboid just completed is the last one on the k-axis (Line 19), the task copies the updated C' in GPU memory back to main memory.

ALGORITHM 1: Local matrix multiplication in t_m

```
Input: $D_{p,q,r}$, /*a cuboid*/
       $\theta_g$, /*the capacity of memory on GPU */
1  /* initialization */
2  $(P_2^*, Q_2^*, R_2^*) = \text{argmin}_{c \in \{(P_2,Q_2,R_2) | Mem^m(P_2,Q_2,R_2) \le \theta_g\}} Cost^m(c)$
3  $S \leftarrow (P_2^*, Q_2^*, R_2^*)$-subcuboid partitioning of $D_{p,q,r}$;
4  sort subcuboids $S$ by $(p_2, q_2, r_2)$;
5  $(I', J', K') \leftarrow$ dimension of a subcuboid;
6  create $J'$ GPU streams;
7  allocate BufA, BufB, and BufC in GPU memory;
8  /* processing subcuboids on GPU */
9  for $n \leftarrow 0$ to $P_2^* \cdot Q_2^* \cdot R_2^*$ do
10     $S_{p_2,q_2,r_2} \leftarrow n$-the subcuboid in $S$
11     $A', B', C' \leftarrow A-, B-, C$-side of $S_{p_2,q_2,r_2}$;
12     copy $A'$ to BufA in GPU;
13     for $(k, j) \leftarrow 0$ to $(K', J')$ do
14         async-copy $B_{k,j}$ to BufB using $j$-th stream;
15         for $i \leftarrow 0$ to $I'$ do
16             call $K_{i,k*k,j}(A_{i,k}, B_{k,j})$ using $j$-th stream;
17         end
18     end
19     if $r_2 = R_2^* - 1$ then
20         copy $C'$ in BufC to main memory;
21     end
22 end
```

5 IMPLEMENTATION OF DISTME

In this section, we briefly explain the implementation of DistME. We implement DistME on top of Spark, and so, it allows users to describe their matrix computation queries (e.g., GNMF) using Scala API. From the query described by users, DitsME generates a kind of physical plan that can be executed in either CPU or GPU. Here, we implement the plan generator by extending SparkSQL [3], which approach is also used in MatFast [38]. For the GPU computation in the plan, we use Jcuda [36]. DistME exploits the data serialization and deserialization of SparkSQL to reduce the amount of shuffled data.

A block of matrices is implemented using RDD (Resilient Distributed Datasets) [39], in particular, using a record of RDD, where a key is the row and column indices (e.g., i and k) of the block, and a value is either our DenseMatrix class or SparseMatrix class. DistME supports a number of matrix operators such as element-wise, matrix multiplication, and transpose. We implement them based on the transformation operations of RDD, i.e., map, groupByKey, Cogroup, and reduceByKey. For the local multiplication step of DistME, we use the cuBLAS and cuSPASE libraries for GPU computation. For the Row, Column, and Grid partitioning schemes of DistME (in Section 2.1), we extend the RDD partitioner class. We use the parquet format for reading and writing the matrix data with HDFS.

6 EXPERIMENTAL EVALUATION

In this section, we present experimental results in four categories. First, we compare the CuboidMM method with the existing methods, BMM, CPMM, and RMM, in terms of the elapsed times and communication cost (i.e., amount of transferred data in the matrix repartition and aggregation steps). We also check the P, Q, and R parameters determined by the optimization in Section 3.2 can achieve the best performance in CuboidMM. Second, we evaluate the performance of the DistME system compared with the state-of-the-art systems, SystemML [6, 18], and MatFast [38] in terms of the elapsed times. Third, we evaluate the performance of matrix factorization, in particular, Gaussian Non-Negative Matrix Factorization [23] (GNMF) of DistME, compared with that of the state-of-the-art systems, SystemML [6, 18], MatFast [38],

Table 3: Statistics of real datasets.

dataset	ratings	users	items
MovieLens	27,753,444	283,228	58,098
Netflix	100,480,507	480,189	17,770
YahooMusic	717,872,016	1,823,179	136,736

and DMac [37]. Fourth, we compare DistME with two well-known distributed matrix computation systems in the HPC area, ScaLAPACK [12] and SciDB [8, 31].

6.1 Experimental setup

Datasets: For experiments, we use both real and synthetic datasets. For real datasets, we use MovieLens [20] for small size, Netflix [41] for medium size, and YahooMusic[1] for large size. Table 3 summarizes the statistics of three datasets. We use those datasets for evaluating the performance of GNMF. For synthetic datasets, we generate matrices that have randomly and uniformly distributed non-zero elements as in SystemML [6, 18]. The input matrices A and B are of $I \times K$ and $K \times J$, respectively, where K becomes the common dimension of A and B. We generate three types of synthetic datasets, which are also used in [19]: two general matrices ($I = K = J$), two matrices with a common large dimension ($K > I = J$), and two matrices with two large dimensions ($I = J > K$). The sparsity of matrices are in the range of 0.0 to 1.0 and vary depending on the experiment, where 1.0 means a fully dense matrix.

Systems compared: We compare our DistME with SystemML, MatFast, DMac, ScaLAPACK, and SciDB. We use the original codes for SystemML[2] and MatFast[3]. There are two versions of MatFast, naive and optimization, where we use the former version since the latter version is not available. Since the current SystemML and MatFast do not support GPU-based matrix multiplication in a distributed environment, we modify both SystemML and MatFast so as to support GPU-based matrix multiplication by implementing GPU-based matrix multiplication kernel functions based on cuBLAS and cuSPARSE as in our DistME. We denote GPU-versions of SystemML and MatFast by SystemML(G) and MatFast(G), respectively.

For DMac, we cannot find the code available, and so, implement it in the same code optimization level with DistME. We note that the above four systems and our DistME all are implemented on top of Spark [40]. We also evaluate the performance of DistME compared with the open-source library ScaLAPACK [12] and the array database SciDB [8, 31].

H/W and S/W setting: We conduct all the experiments on the same cluster of one master node and nine slave nodes.

[1]https://webscope.sandbox.yahoo.com/catalog.php?datatype=r
[2]https://github.com/apache/systemml/tree/branch-1.0.0
[3]https://github.com/yuyongyang800/SparkDistributedMatrix

Table 4: Sizes of input matrices and the optimal parameters of CuboidMM (K: thousand, M: million).

type	sizes of input matrices	parameters (P^*, Q^*, R^*)
two general matrices $(N \times N \times N)$	$70K \times 70K \times 70K$	(4,7,4)
	$80K \times 80K \times 80K$	(6,7,4)
	$90K \times 90K \times 90K$	(10,5,5)
	$100K \times 100K \times 100K$	(7,9,5)
two matrices with a common large dimension $(10\,K \times N \times 10\,K)$	$10K \times 100K \times 10K$	(1,1,9)
	$10K \times 500K \times 10K$	(1,1,18)
	$10K \times 1M \times 10K$	(1,1,36)
	$10K \times 5M \times 10K$	(1,1,176)
two matrices with two large dimensions $(N \times 1\,K \times N)$	$100K \times 1K \times 100K$	(9, 10, 1)
	$250K \times 1K \times 250K$	(8, 13, 1)
	$500K \times 1K \times 500K$	(17, 24, 1)
	$750K \times 1K \times 750K$	(26, 35, 1)

All nodes are connected via 10 Gbps Ethernet. Each node is equipped with a six-core 3.5 GHz CPU, 64 GB main memory, 500 GB SSD for Spark, 4 TB HDD for HDFS, and a single NVIDIA GTX 1080 Ti GPU having 11 GB device memory. In terms of software, we use CentOS 6.6, Spark 2.1.0, Hadoop 2.7.2, CUDA 8.0, ScaLAPACK 2.0 with MPICH 3.2, and SciDB 18.1. We set the number of tasks per node to 10 ($T_c = 10$), and so, set $\theta_t = 6\,GB$ and $\theta_g = 1\,GB$. We use the block size of 1000×1000 in all experiments, which is the default size in other systems such as MatFast [38] and SystemML [6, 18].

6.2 Performance of CuboidMM

We compare the performances of BMM, CPMM, RMM, and CuboidMM using large-scale synthetic dense matrices (sparsity is 0.5). We note that all four methods in this experiment are evaluated on DistME and so exploit GPU computation, where RMM cannot perform cuboid-level GPU computation, but simple block-level GPU computation due to its hash partitioning. Since our (P, Q, R)-cuboid partitioning is a generalization of the existing methods, we can evaluate the performance of BMM, CPMM, and RMM by changing the parameters P, Q, and R as explained in Section 3.1. Here, CuboidMM uses the optimal parameters P^*, Q^*, and R^*. Table 4 summarizes the sizes of three types of datasets used and the optimal parameters (P^*, Q^*, R^*) used in CuboidMM. The optimal parameters are automatically determined as in Section 3.2.

We set T to the maximum number of tasks for the BMM and CPMM methods, i.e., $T = I$ for BMM, and $T = K$ for CPMM, to obtain their maximum performances, and at the same time, to avoid out of memory. Likewise, we set $T = I \cdot J$ for RMM, which is the best setting in terms of the aggregation performance of intermediate blocks of the output matrix. The setting of $T = I \cdot J \cdot K$ for RMM incurs some errors due to too many tasks in Spark. In our CuboidMM, T is automatically determined as $P^* \cdot Q^* \cdot R^*$. Figures 6(a), (b), and (c) show the elapsed times of four methods for three types of datasets, and Figures 6(d), (e), and (f) show their corresponding communication costs. In the figures, O.O.M. means out of memory, and T.O. means time out (longer than 4,000 seconds).

Two general matrices: Figures 6(a) and (d) shows that CuboidMM significantly outperforms all other methods in terms of both elapsed times and communication cost. We note that Y-axis is in log-scale. Among the existing methods, CPMM and BMM show better performance than RMM as mentioned in Section 2.2.4. The BMM method fails due to O.O.M. when N is larger than $80\,K$.

Compared with RMM, the proposed CuboidMM improves the elapsed time by 3.86 times for $N = 70\,K$, 4.80 times for $80\,K$, 5.34 times for $90\,K$, and 6.11 times for $100\,K$. There is a similar tendency when comparing CuboidMM with CPMM. That is, the improvement of CuboidMM compared with the existing methods becomes more marked as the matrix sizes get larger. The gap between CuboidMM and the other methods is bigger in terms of communication cost. When $N = 100\,K$, CuboidMM reduces the amount of transferred data by 8.17 times compared with CPMM and 19.46 times compared with RMM.

Two matrices with a common large dimension: In this experiment, the matrix A is fat (i.e., $K > I$), while the matrix B is tall (i.e., $K > J$). Figures 6(b) and (e) shows similar tendencies with Figures 6(a) and (d). CuboidMM still significantly outperforms all other methods in terms of both measures, and BMM fails due to O.O.M. when N is larger than $500\,K$.

We note that CuboidMM is much faster than CPMM, and at the same time, reduces the amount of transfer compared with CPMM, although $P^* = 1$ and $Q^* = 1$ as CPMM in Table 4. It is natural that $P^* = 1$ and $Q^* = 1$ in CuboidMM because the corresponding 3-dimensional model has a very long dimension along the k-axis. In the figures, when $N = 5\,M$, CuboidMM improves the elapsed time by 3.92 times and reduces the communication cost by 60.39 times compared with the second best method, CPMM. This is mainly due to the difference in the numbers of partitions on the k-axis. CPMM uses K partitions, while CuboidMM uses R^* partitions ($R^* < K$). For example, when $N = 5\,M$ (i.e., $N_b = 5000$), K is 5000, but R^* is just 176 in Table 4. A larger number of partitions incurs a larger amount of data transferred in the matrix aggregation step.

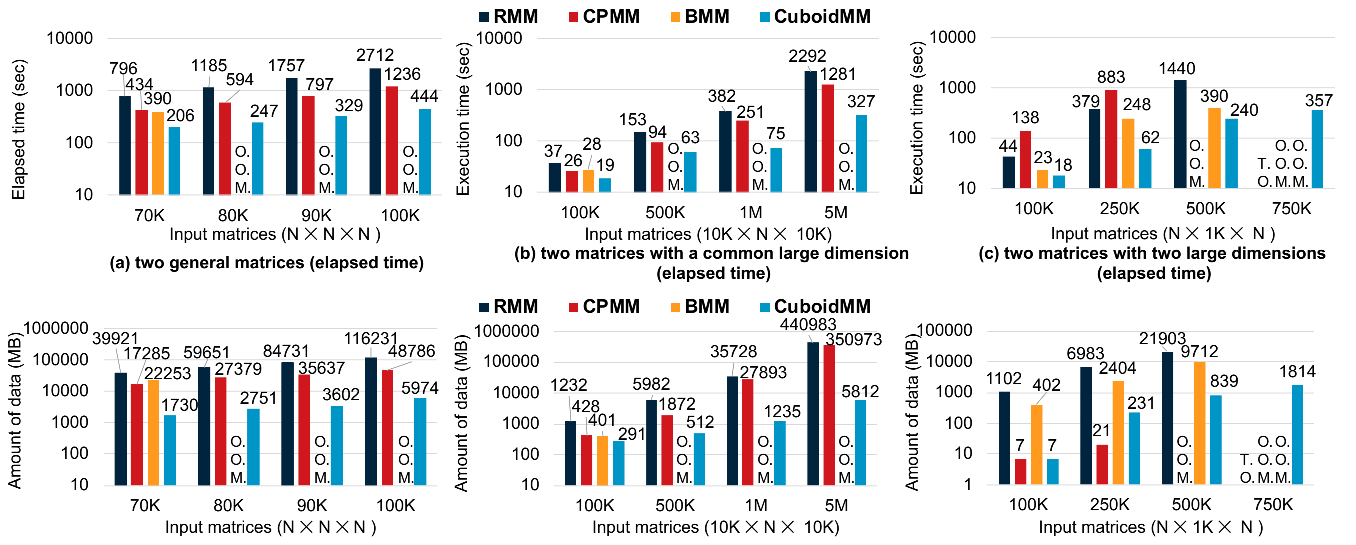
Two matrices with two large dimensions: In this experiment, the matrix A is tall (i.e., $K < I$), while the matrix B is fat (i.e., $K < J$). Figures 6(c) and (f) shows that only CuboidMM can process the matrix multiplication of $750\,K \times 1\,K \times 750\,K$. For that size, both CPMM and BMM fail due to O.O.M. and RMM times out (i.e., longer than 4000 seconds). In particular, CPMM fails due to O.O.M. even for the case of $N = 500\,K$. It is because the amount of intermediate output blocks per task, i.e., $|C|$, increases rapidly as N increases for this type of dataset. In the figures, when $N = 500\,K$, CuboidMM improves the elapsed time by 1.63 times and reduces the communication cost by 11.58 times compared with the second best method, BMM.

6.3 Performance of DistME

We compare the performances of SystemML, MatFast, and DistME using three different synthetic datasets. We denote DistME with and without the GPU method in Section 4 by DistME(G) and DistME(C), respectively. We note that Y-axis in Figures 7(a)-(d) is in log-scale. In the figures, E.D.C. means exceeding the disk capacity, where the size of the intermediate data becomes larger than the total amount of hard disk capacity in the cluster (> 36TB), and so the execution fails.

Figure 7(a) shows the elapsed times when using the dataset of the type "two general matrices" of dense matrices. For this dataset, MatFast and SystemML use CPMM for all N values. For $N = 30K$, DistME(C) is 3.1 and 1.62 times faster than MatFast(C) and SystemML(C), respectively. The performance gap gets larger as the data size increases. For example, when $N = 40K$, DistME(C) is 2.54 times faster than SystemML(C), where MatFast(C) is O.O.M. This performance gap is mainly due to both lower communication overhead and higher maximum parallelism of DistME(C) using CuboidMM. The left three bars in Figure 7(e) show the time ratio of MatFast(C), SystemML(C), and DistME(C), where the communication overhead in the matrix repartition and aggregation steps of DistME(C) is much lower than those of the other methods. In addition, as shown in Table 2, CuboidMM used in DistME(C) has much higher maximum parallelism than CPMM used in SystemML(C). For example, when $N = 40K$, SystemML(C) executes only 40 concurrent tasks among 90 possible ones, while DistME(C) executes all 90 concurrent tasks. In Figure 7(a), MatFast(G), SystemML(G), and DistME(G) improve the performance by 3.8, 2.39, and 5.59 times compared with MatFast(C), SystemML(C), and DistME(C), respectively. The improvement of DistME(G) is larger than those of MatFast(G) and SystemML(G) due to its GPU acceleration method in Section 4.

Figure 7(b) shows the result for the dataset of the type "two matrices with a common large dimension" of dense matrices. This dataset represents the case where the convolution layer in deep learning is calculated by matrix multiplication [11]. This type requires a larger amount of computation, and at the same time, incurs a larger amount of intermediate data in order to generate a single result block than other types. In the experiments, both MatFast and SystemML choose CPMM since the size of the output matrix is smaller than



(a) two general matrices (elapsed time) (b) two matrices with a common large dimension (elapsed time) (c) two matrices with two large dimensions (elapsed time)

(d) two general matrices (communication cost) (e) two matrices with a common large dimension (communication cost) (f) two matrices with two large dimensions (communication cost)

Figure 6: Performance comparison among BMM, CPMM, RMM, and CuboidMM.

that of input matrices. DistME(C) outperforms both MatFast(C) and SystemML(C), and DistME(G) outperforms both MatFast(G) and SystemML(G). The gap among GPU-based systems is larger than that among CPU-based systems since the type of dataset is computationally intensive, and so, reducing communication overhead becomes more important. For example, when $N = 5\,M$, DistME(C) outperforms SystemML(C) by 1.26 times, whereas DistME(G) outperforms SystemML(G) by 2.18 times. This type of dataset incurs a tremendous amount of intermediate data as N increases, and thus, SystemML and MatFast fail due to E.D.C. when $N = 20\,M$, where the amount of intermediate data exceeds 36 TB. In contrast, DistME incurs only 1.5 TB intermediate data due to its CuboidMM.

Figure 7(c) shows the result for the dataset of the type "two matrices with two large dimensions" of dense matrices. This dataset represents the case of multiplying two factor dense matrices in matrix factorization. In this experiment, MatFast uses CPMM, while SystemML uses RMM. Since the size of the result matrix is larger than those of input matrices, i.e., $|C|$ is very large, MatFast using CPMM fails due to O.O.M. for all data sizes used. SystemML using RMM has no problem of O.O.M., but when $N = 1.5\,M$ and $N = 2\,M$, it fails due to E.D.C. When $N = 1\,M$, DistME(C) and DistME(G) outperform SystemML(C) and SystemML(G) by 4.92 and 6.63 times, respectively.

Figure 7(d) shows the elapsed times of the multiplication between one large sparse matrix and one small dense matrix while varying the sparsity of the large matrix. This dataset represents the case of multiplying a large rating sparse matrix with a small dense factor matrix in matrix factorization. For this dataset, both MatFast and SystemML use CPMM. Although one of the input matrices of this dataset is not dense, but sparse, the shape of the dataset is similar with that of the dataset used in Figure 7(b), i.e., a large common dimension, and so, performance tendencies in both figures are somewhat similar with other.

Figures 7(e) and (f) show that our DistME significantly reduces communication overhead compared with MatFast and SystemML due to its cuboid partitioning for all kinds of datasets. For example, when $1M \times 1K \times 1M$, DistME shuffles 3.18 times smaller data than SystemML. Figure 7(g) shows the GPU core utilization of MatFast(G), SystemML(G), and DistME(G). Here, the y-axis means the average of GPU core utilization in the local multiplication step, which is measured by the NVIDIA's monitoring tool, nvidia-smi. In the figure, DistME(G) achieves better utilization for both dense and sparse matrices due to its GPU acceleration method seamlessly combined with CuboidMM.

6.4 Performance of GNMF

We compare the performances of a total of seven systems for the GNMF query on three real datasets: MovieLens, Netflix, and YahooMusic. The GNMF query requires to perform a number of iterations to find two factor matrices W and H for a given V, and we perform the query up to ten iterations. In Figures 8(a)-(c), We set the factor dimension to 200 as in MatFast [38] and DMac [37]. Figure 8(a) shows the accumulated execution times for MovieLens (small dataset). In the figure, DistME(G) outperforms all the other systems, in particular, MatFast(G) and SystemML(G) by 1.56 and 1.2 times, respectively. Figure 8(b) shows the results

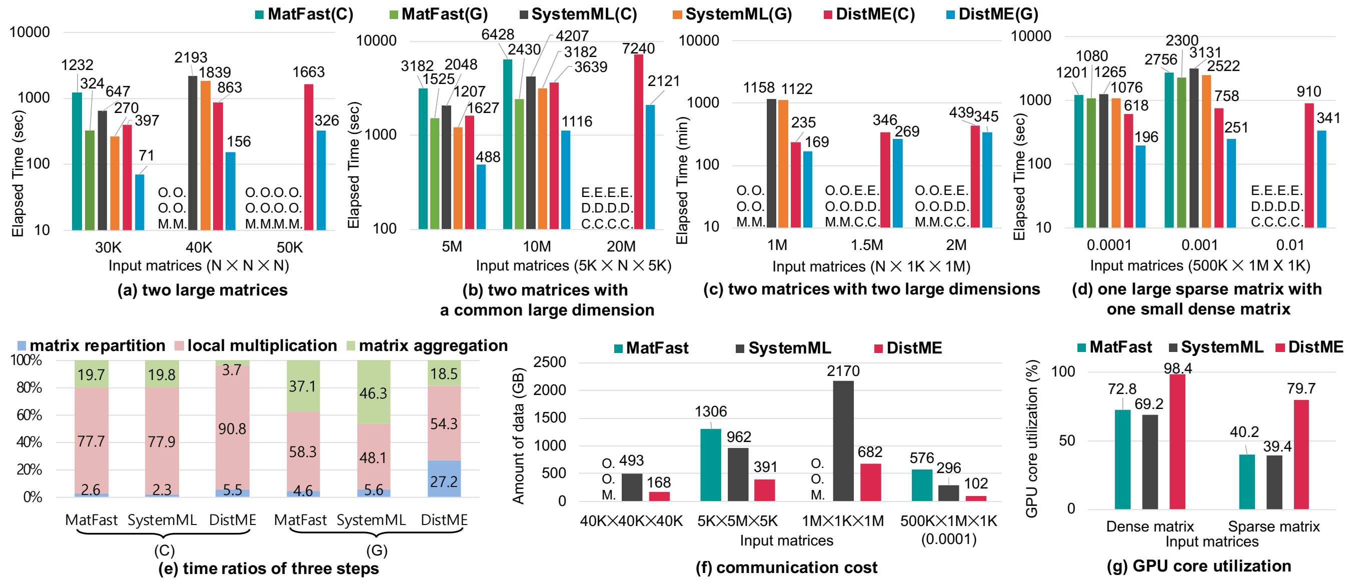


Figure 7: Comparison among MatFast(C), MatFast(G), SystemML(C), SystemML(G), DistME(C), and DistME(G).

for Netflix (medium dataset). DistME(G) outperforms MatFast(G) and SystemML(G) by 3.5 and 1.7 times, respectively. We note that the performance gap gets larger as the data size increases. Figure 8(c) shows the results for YahooMusic (large dataset). DistME(G) outperforms MatFast(G) and SystemML(G) by 3.45 and 1.92 times, respectively. Except for DistME(G), SystemML(G) shows the fastest performance. Among CPU-based systems, DistME(C) shows the fastest performance for Netflix and YahooMusic, while SystemML(C) shows the fastest performance for MovieLens.

Figure 8(d) presents the execution times while varying the factor dimension in YahooMusic. When the factor dimension is larger than 500, MatFast fails due to O.O.M. DistME(G) outperforms SystemML(G) by 3.88 times when the factor dimension is 1000. We note that the performance gap between DistME(C) and SystemML(G) gets smaller as the factor dimension increases because SystemML generates a larger amount of intermediate data than DistME. We also note that matrix multiplication is very time-consuming, and so, takes 81% of a total elapsed time for processing the GNMF query (SystemML(C), YahooMusic).

6.5 Comparison with Systems in HPC

We compare the performance of DistME with SciDB [8, 31] and ScaLAPACK [12]. For fair comparison, we use DistME(C) instead of DistME(G). ScaLAPACK is a highly tuned library for distributed linear algebra routines using MPI communication. SciDB is an open-source data management system for large-scale array data that can support matrix operations. SciDB provides linear algebra operators wrapping ScaLAPACK. We launch ten processes per node for ScaLAPACK, SciDB, and DistME(C).

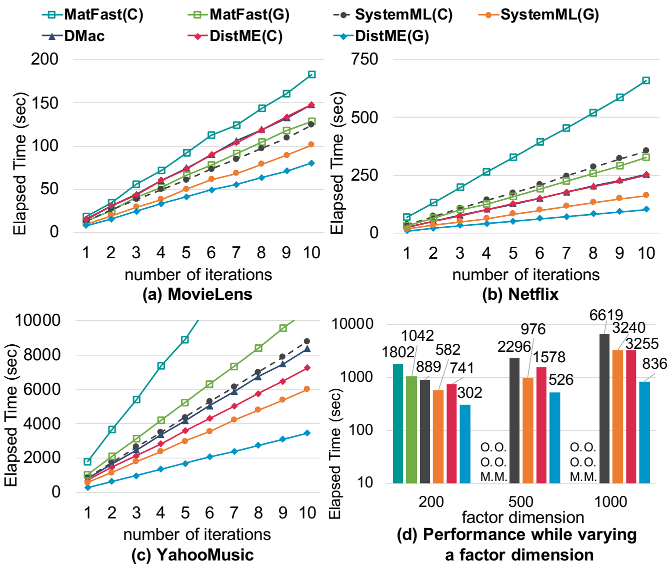


Figure 8: Performance comparison for GNMF.

Table 5 shows the elapsed times of matrix multiplication for three types of dense matrices: $N \times N \times N$, $5K \times N \times 5K$, and $N \times 1K \times N$. In all experiments, ScaLAPACK shows a better performance than SciDB. For the first type $N \times N \times N$, DistME(C) shows a worse performance compared with ScaLAPACK and SciDB for a small data ($N = 10K$), but outperforms both ScaLAPACK and SciDB for a large data ($N = 50K$). The reason why DistME(C) is faster than ScaLAPACK for a large data is that the matrix multiplication method used in ScaLAPACK incurs a large amount of communication overhead across processes [16, 19]. In particular, the communication overhead in ScaLAPACK becomes severe

Table 5: Comparison with ScaLAPACK and SciDB.

type	N	ScaLAPACK	SciDB	DistME(C)
$N \times N \times N$	10K	31s	33s	42s
	50K	1865s	1998s	1663s
$5K \times N \times 5K$	1M	995s	1069s	326s
	5M	70m	O.O.M.	27m
$N \times 1K \times N$	100K	248s	332s	122s
	500K	O.O.M.	O.O.M.	57m

when dealing with a common large dimension [16, 19]. For the second type $5K \times N \times 5K$, which has a common large dimension, DistME(C) outperforms SciDB and ScaLAPACK by 3.28 and 3.05 times, respectively, when $N = 1\,M$. Different from ScaLAPACK, DistME(C) performs distributed matrix multiplication in an optimal manner in terms of network communication cost and memory usage per task due to CuboidMM. For the third type $N \times 1K \times N$, only DistME(C) can perform matrix multiplication when $N = 500\,K$. SciDB and ScaLAPACK fail due to O.O.M. In fact, they easily fail for large-scale matrix multiplication since they keep all blocks of a local matrix as a single array in main memory.

7 RELATED WORK

Matrix multiplication methods: SUMMA [34] is a distributed matrix multiplication method widely used in the High Performance Computing (HPC) environment. It partitions the 3-dimensional model as CuboidMM performs $(1, Q, R)$-cuboid partitioning. If we let the number of machines be M, SUMMA performs partitioning such that $P \times Q$ is equal to $T_c \times M$. It is known that this approach has relatively a low communication cost in the matrix repartition step, but relatively a high communication cost in the matrix aggregation step [19]. It is implemented in a library for the HPC environment, called ScaLAPACK [12].

The method used in Marlin [19], called CRMM, is the same as the RMM method in principle. However, instead of using "physical" blocks as they are, CRMM forms bigger "logical" blocks by the shuffle and performs matrix multiplication using those bigger blocks so as to reduce the communication cost. In Table 2, the communication cost of the RMM method decreases when I, J, and K become smaller by using bigger blocks. However, those bigger blocks are cubes, and so, cannot achieve the minimum communication cost as cuboids of CuboidMM can. In addition, the shuffle step for forming bigger blocks increases the communication cost.

Matrix computation systems: There have been proposed a number of large-scale matrix computation systems. They can be classified into two categories: the systems based on MapReduce and the systems not based on MapReduce. The former systems include SystemML [6, 18], DMac [37], Mahout [29], HAMA [30], MatFast [38], and SimSQL [17, 24, 25]. HAMA and Mahout focus on providing the libraries for machine learning based on matrix computations, which are implemented using MapReduce. They provide a number of machine learning algorithms, but it is difficult for users to implement a new algorithm since MapReduce is a very low-level interface. SystemML [6, 18] allows users to write a new algorithm using an R-like high-level declarative interface, which is translated into a series of MapReduce or Spark jobs. However, it has relatively a large communication cost since it uses either BMM, CPMM, or RMM as a distributed matrix multiplication method. Both DMac [37] and MatFast [38] exploit matrix dependencies for a complex query like GNMF to reduce the overall communication overhead. They store an output matrix using the partitioning scheme (e.g., Row, Column) that can reduce the communication cost in the matrix repartition step for the next matrix operator in the plan of the query. SimSQL supports linear algebra computation through a SQL-like declarative language, called BUDS, based on Apache Hadoop [2]. SimSQL uses either BMM or CPMM as a matrix multiplication method.

The latter systems include SciDB [8, 31] and MORPHEUS [10, 33]. SciDB focuses on managing multidimensional array data. For distributed matrix multiplication, it uses the ScaLAPACK [12] library. Both SciDB and ScaLAPACK utilizes MPI communication. SciDB may have extra communication overhead before matrix multiplication since the input matrices should be repartitioned depending on the partitioning scheme required by ScaLAPACK. MORPHEUS automatically factorizes ML algorithms to linear algebra operators and then executes the operators over the platforms that can support linear algebra operators such as R [28].

8 CONCLUSIONS

In this paper, we have proposed a distributed matrix multiplication method called CuboidMM that performs the optimal cuboid partitioning for given input matrices elastically. CuboidMM can achieve the lowest communication cost with a given constraint on memory usage per task. As a result, it significantly outperforms the existing methods, BMM, CPMM, and RMM, in terms of both performance and scalability. We also have proposed a GPU acceleration method of matrix multiplication that can be seamlessly combined with CuboidMM. We have implemented a matrix computation system called DistME by integrating CuboidMM and the GPU acceleration method on top of Spark. It significantly outperforms the existing systems such as SystemML, MatFast, and DMac in terms of both performance and scalability. As future work, we will extend our GPU acceleration method to exploit multiple GPUs per node and to achieve a better load balancing by considering differences in sparsities of cuboids, which may further improve the performance.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2018R1A5A1060031), Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (2017R1E1A1A01077630), and Institute for Information communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (R7124-16-0004, Development of Intelligent Interaction Technology Based on Context Awareness and Human Intention Understanding).

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, et al. 1999. *LAPACK Users' guide.* SIAM.

[2] Apache. 2011. Hadoop. Retrieved May 2, 2018 from http://hadoop.apache.org/

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD.* ACM, 1383–1394.

[4] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, et al. 2009. A view of the parallel computing landscape. *Commun. ACM* 52, 10 (2009), 56–67.

[5] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. 2010. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD.* ACM, 975–986.

[6] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, et al. 2016. SystemML: Declarative machine learning on spark. *VLDB* 9, 13 (2016), 1425–1436.

[7] A. Brodtkorb, T. R Hagen, and Martin L Sæ. 2013. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *J. Parallel and Distrib. Comput.* 73, 1 (2013), 4–13.

[8] P. G. Brown. 2010. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD.* ACM, 963–968.

[9] U. Catalyurek and C. Aykanat. 2001. A hypergraph-partitioning approach for coarse-grain decomposition. In *SC.* ACM, 28–28.

[10] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. 2017. Towards linear algebra over normalized data. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1214–1225.

[11] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[12] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the.* IEEE, 120–127.

[13] NVIDIA Corporation. 2012. Nvidia Multi-Process Service. Retrieved May 2, 2018 from https://docs.nvidia.com/deploy/mps/index.html

[14] NVIDIA Corporation. 2015. cuBLAS. Retrieved May 2, 2018 from https://docs.nvidia.com/cuda/cublas/index.html

[15] T. A. Davis. 2006. *Direct methods for sparse linear systems.* Vol. 2. Siam.

[16] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger. 2013. Communication-optimal parallel recursive rectangular matrix multiplication. In *IPDPS.* IEEE, 261–272.

[17] Z. J. Gao, S. Luo, L. L. Perez, and C. Jermaine. 2017. The BUDS Language for Distributed Bayesian Machine Learning. In *ICDE.* ACM, 961–976.

[18] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. 2011. SystemML: Declarative machine learning on MapReduce. In *ICDE.* IEEE, 231–242.

[19] R. Gu, Y. Tang, C. Tian, H. Zhou, G. Li, X. Zheng, and Y. Huang. 2017. Improving Execution Concurrency of Large-Scale Matrix Multiplication on Distributed Data-Parallel Platforms. *TPDS* 28, 9 (2017), 2539–2552.

[20] F. M. Harper and J. A. Konstan. 2016. The movielens datasets: History and context. *ACM TIIS* 5, 4 (2016), 19.

[21] M. Kabiljo and A. Ilic. 2015. Recommending items to more than a billion people. Retrieved May 2, 2018 from https://code.fb.com/core-data/recommending-items-to-more-than-a-billion-people

[22] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim. 2016. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *SIGMOD.* ACM, 447–461.

[23] D. D. Lee and H. S. Seung. 2001. Algorithms for non-negative matrix factorization. In *NIPS.* 556–562.

[24] S. Luo, Z. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. 2018. Scalable linear algebra on a relational database system. *IEEE TKDE* (2018).

[25] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. 2017. Scalable Linear Algebra on a Relational Database System. In *ICDE.* IEEE, 523–534.

[26] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, et al. 2016. Mllib: Machine learning in apache spark. *JMLR* 17, 1 (2016), 1235–1241.

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

[28] R Core Team. 2014. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/

[29] S. Schelter, A. Palumbo, S. Quinn, S. Marthi, and A. Musselman. 2016. Samsara: Declarative machine learning on distributed dataflow systems. In *NIPS Workshop MLSystems.*

[30] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. 2010. Hama: An efficient matrix computation with the mapreduce framework. In *CloudCom.* IEEE, 721–726.

[31] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. 2011. The architecture of SciDB. In *SSDBM.* Springer, 1–16.

[32] MKL Development Team. 2015. Intel math kernel library developer reference. Retrieved Oct, 2018 from https://software.intel.com/en-us/articles/mkl-reference-manual

[33] A. Thomas and A. Kumar. 2018. A comparative evaluation of systems for scalable linear algebra-based analytics. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2168–2182.

[34] R. A. Van De Geijn and J. Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.

[35] R. C. Whaley and J. J. Dongarra. 1998. Automatically tuned linear algebra software. In *SC.* IEEE, 38–38.

[36] Y. Yan, M. Grossman, and V. Sarkar. 2009. JCUDA: A programmer-friendly interface for accelerating Java programs with CUDA. In *Euro-Par.* Springer, 887–899.

[37] L. Yu, Y. Shao, and B. Cui. 2015. Exploiting matrix dependency for efficient distributed matrix computation. In *SIGMOD.* ACM, 93–105.

[38] Y. Yu, M. Tang, W. G. Aref, Q. M. Malluhi, M. M. Abbas, and M. Ouzzani. 2017. In-Memory Distributed Matrix Computation Processing and Optimization. In *ICDE.* IEEE, 1047–1058.

[39] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI.* USENIX Association, 2–2.

[40] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. 2016. Apache

spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[41] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. 2008. Large-scale parallel collaborative filtering for the netflix prize. In *ICAAM*. Springer, 337–348.

A THE GNMF QUERY

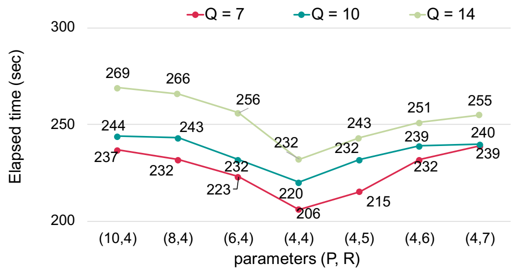
The GNMF query approximates the two factor matrices H and W for a given matrix V as follows:

$$H_{i+1} = \frac{H_i * (W_i^T \times V)}{W_i^T \times W_i \times H_i}, \; W_{i+1} = \frac{W_i * (V \times H_i^T)}{W_i \times H_i \times H_i^T} \quad (7)$$

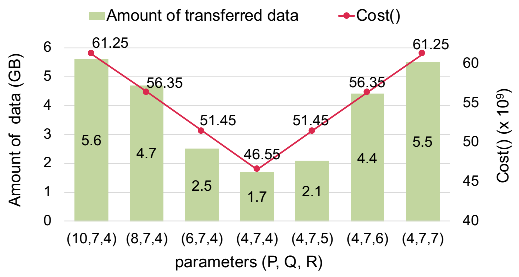
where i is the iteration number, $*$ is element-wise multiplication, $\times$ is matrix multiplication, and W^T means the transpose of W. The initial H and W are generated randomly and are expressed as H_0 and W_0. We use the same query plan with DMac for the GNMF query.

B OPTIMIZATION OF (P,Q,R)

Figure 9 shows the elapsed times and amount of transferred data while varying (P, Q, R) in CuboidMM for the first dataset in Table 4, i.e., two general matrices of $70K \times 70K \times 70K$. CubiodMM determines the optimal parameters as $(P^* = 4, Q^* = 7, R^* = 4)$ for the dataset. Figures 9(a) and (b) show that using (P^*, Q^*, R^*) achieves the minimum elapsed time and the minimum amount of transferred data, respectively. In Figure 9(b), the actual communication cost (in green bars) and $Cost()$ (in red curve) should be the same theoretically, but they are slightly different, due to the serialization and deserialization used during data shuffle.



(a) Elapsed times while varying (P,Q,R)



(b) Communication cost while varying (P,Q,R)

Figure 9: Optimization of (P,Q,R).