FuseME: Distributed Matrix Computation Engine based on Cuboid-based Fused Operator and Plan Generation

Donghyoung Han
Korea Advanced Institute of Science and Technology, Republic of Korea
donghyoung.han@kaist.ac.kr

Jongwuk Lee
Sungkyunkwan University, Republic of Korea
jongwuklee@skku.edu

Min-Soo Kim*
Korea Advanced Institute of Science and Technology, Republic of Korea
minsoo.k@kaist.ac.kr

ABSTRACT

Operator fusion is essentially and widely used in a large number of matrix computation systems in science and industry. The existing distributed operator fusion methods focus on only either low communication cost with the risk of out of memory or large-scale processing with high communication cost. We propose a distributed elastic fused operator called Cuboid-based Fused Operator (CFO) that achieves both low communication cost and large-scale processing. We also propose a novel fusion plan generator called Cuboid-based Fusion plan Generator (CFG) that finds a fusion plan to fuse more operators including large-scale matrix multiplication. We implement a fast distributed matrix computation engine called FuseME by integrating both CFO and CFG seamlessly. FuseME outperforms the state-of-the-art systems including SystemDS by orders of magnitude.

CCS CONCEPTS

• **Information systems** → **Query operators**; *Query planning*.

KEYWORDS

Distributed data-parallel system, operator fusion, matrix operators

ACM Reference Format:
Donghyoung Han, Jongwuk Lee, and Min-Soo Kim. 2022. FuseME: Distributed Matrix Computation Engine based on Cuboid-based Fused Operator and Plan Generation. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517895

1 INTRODUCTION

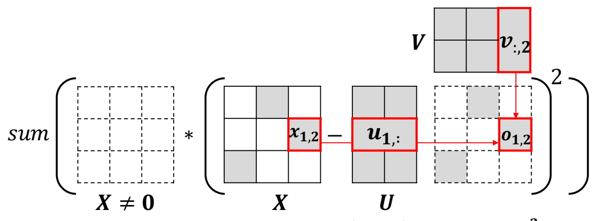
Matrix computation is essentially and widely used in a large number of applications in various fields such as database, machine learning, health, music, and games [2]. The applications include collaborative filtering, principle component analysis (PCA), singular value decomposition (SVD), Lower-Upper (LU) factorization, betweeness centrality, and deep neural networks. As the sizes of real matrix datasets are growing rapidly, fast and scalable matrix computation systems have become more important than ever before. For example, the sizes of Facebook's dataset are 100 billion ratings, more than a billion users, and millions of items [26].

For processing large-scale matrix computation in a fast and scalable way, a number of distributed matrix computation systems on top of MapReduce-based frameworks [6, 7, 14, 16, 18, 30, 32, 36, 37] have been proposed. For a matrix query, these systems generate and execute a query plan as Directed Acyclic Graph (DAG) of basic matrix operators. Some of those systems try to fuse a chain of basic operators into a single operator called a fused operator [7, 8, 12]. This operator fusion can reduce the amount of intermediate data by avoiding unnecessary materialization [24] and improve the query performance by eliminating unnecessary scans [3] and unnecessary computation through so-called sparsity exploitation [7]. There are also a lot of efforts to fuse basic matrix operators, such as element-wise multiplication and matrix multiplication [5, 8, 12, 15, 22, 25].

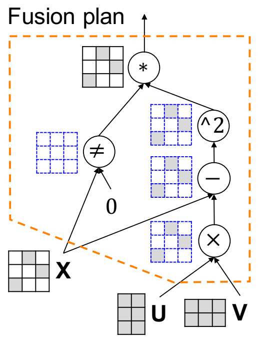
Alternating Least Squares (ALS) [7, 21, 31] is one of the matrix factorization methods. Figure 1(a) shows a motivating example of operator fusion for weighted squared loss $(X \neq 0) * (X - U \times V)^2$ of ALS [7], where $*$ is element-wise multiplication, $(X \neq 0)$ is non-zero elements of X, matrix X is sparse, and matrices U and V are dense. The gray cells mean non-zero elements, and the dotted cells indicate no materialization. The fused operator takes three matrices X, U, and V, and then, calculates the loss without the materialization of $(X \neq 0)$ and dotted cells of $U \times V$. It directly computes $x_{1,2} * (x_{1,2} - u_{1,:} \times v_{:,2})^2$ for $O_{1,2}$ without the materialization across operators and avoids the computation for most of the output elements since X is sparse.

SystemDS [6] is the state-of-the-art distributed matrix computation system. It uses GEN [8], which is the state-of-the-art template-based fusion plan generator. From a query plan in DAG, the fusion plan generator finds sub-DAGs called partial fusion plans. Figure 1(b) shows the fusion plan generated by GEN for the query in Figure 1(a). In the figure, the left matrix of each operator means its result matrix. For the query, GEN of SystemDS generates a single partial fusion plan in orange dotted, which is executed as a single fused operator. Although GEN can find a partial fusion plan to exploit sparsity well as in Figure 1(b), it tends to avoid including large-scale matrix multiplication in its fusion plan. Matrix multiplication is one of the operators having the highest communication cost and memory usage on the distributed environment [18], and so a fusion plan containing the large-scale one tends to fail. Figure 1(c) shows the fusion plan generated by GEN for the query $(X \times V^T * U) \div (V^T \times V \times U)$, which is the part of Gaussian Non-negative Matrix Factorization (GNMF) [28]. GEN fuses only two element-wise operators, i.e., $*$ and $\div$, for the query. The existing systems including SystemDS tend not to achieve the best possible

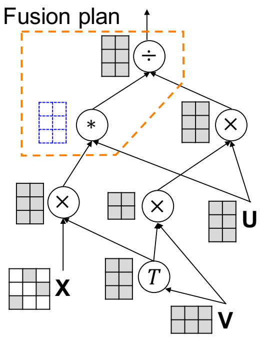
*Corresponding author.



(a) Fused operation of $sum((X \neq 0) * (X - U \times V)^2)$



(b) Fusion plan of $(X \neq 0) * (X - U \times V)^2$



(c) Fusion plan of $(X \times V^T * U) \div (V^T \times V \times U)$

Figure 1: Example of operator fusion.

performance by missing many opportunities of operator fusion including large-scale matrix multiplication.

To solve the above issues, we propose a new distributed fused operator, called Cuboid-based Fused Operator (CFO), and a new fusion plan generator, called Cuboid-based Fusion plan Generator (CFG). In terms of a fused operator, there are two kinds of operators: Broadcast-based Fused Operator (BFO) and Replication-based Fused Operator (RFO). The BFO broadcasts smaller matrices to all the tasks where a larger matrix is repartitioned. The RFO replicates the part of smaller matrices to all the tasks where a larger matrix is repartitioned. The BFO tends to fail due to exceeding the memory limit per task, while the RFO tends to be slow due to high communication overhead. Our proposed CFO achieves both low communication overhead and low memory usage per task by extending the concept of cuboid partitioning [18] from simple matrix multiplication to a complex arbitrary fused operator. The CFO performs partitioning elastically so as to achieve the best performance for given task memory, input matrices, and operators. In terms of a fusion plan generator, our proposed CFG can find a partial fusion plan containing large-scale matrix multiplication based on CFO, and moreover, find a much larger partial plan than the state-of-the-art plan generator GEN does. A larger partial plan means reducing unnecessary materialization and computation more. We implement the distributed matrix computation engine called *FuseME* by integrating our CFO and CFG seamlessly on top of Apache Spark [39]. Due to a new fused operator and plan generator, FuseME improves the performance of matrix queries compared to the state-of-the-art system, i.e., SystemDS, by orders of magnitude.

Our major contributions are summarized as follows:

- We propose a novel distributed fused operator, called Cuboid-based Fused Operator (CFO).
- We propose a novel fusion plan generator, called Cuboid-based Fusion plan Generator (CFG).
- We implement a distributed matrix computation engine FuseME based on CFO and CFG.
- We have demonstrated that our method improves the performance of the existing methods up to by 238× and 64× in terms of elapsed time and communication cost, respectively

The rest of this paper is organized as follows. Section 2 reviews operator fusion, fused operators, and cuboid-based matrix multiplication. We present our distributed fused operator, CFO, in Section 3, and our fusion plan generator, CFG, in Section 4. Section 5 presents the implementation of FuseME briefly, and Section 6 presents the results of the experimental evaluation. Finally, we discuss related work in Section 7 and conclude this paper in Section 8.

2 PRELIMINARIES

In this section, we explain operator fusion of matrix computation systems in Section 2.1 and the state-of-the-art distributed fused operators in Section 2.2. We also explain the cuboid-based matrix multiplication of DistME[18] in Section 2.3.

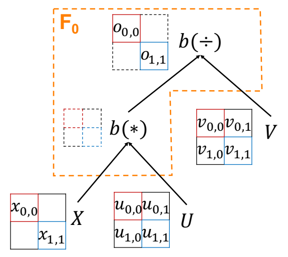
2.1 Operator Fusion

In general, matrix computation systems support five types of basic matrix operators [14]: binary, unary, binary aggregation, and unary aggregation, and reorganization.

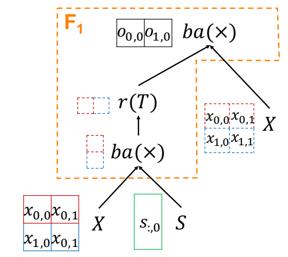
- **Unary** operator performs an element-wise operation (e.g., *log*, *sin*, and *pow*) for each element of an input matrix and returns the result of the operation as an output matrix, where both input and output matrices have the same dimensions.
- **Binary** operator takes two matrices as input and returns one matrix as output, where all the matrices of input and output have the same dimensions. This type performs an element-wise operation (e.g., $*, +, -$, and $\div$) for each pair of elements, $x_{i,j}$ and $y_{i,j}$, for two input matrices X and Y. If one input is just a scalar, it applies all elements of the other input matrix.
- **Unary aggregation** operator takes one matrix as input and returns a matrix, a vector, or a scalar as output depending on aggregation, and so the input and output have different dimensions in general. The examples are $sum()$, $rowSum()$, and $colSum()$. Given a matrix X of $I \times J$, $sum(X)$, $rowSum(X)$, and $colSum(X)$ perform $\sum_{0 \leq i < I, 0 \leq j < J} x_{i,j}$, $\sum_{0 \leq j < J} x_{i,j}$, and $\sum_{0 \leq i < I} x_{i,j}$, respectively.
- **Binary aggregation** operator takes two matrices having a common dimension as input and returns a matrix as output. Given X of $I \times K$ and Y of $K \times J$, it returns a matrix of $I \times J$ by performing an arithmetic operation (e.g., $*$) and aggregation operation (e.g., $\sum$ and $\prod$) on the common dimension K. Matrix multiplication belongs to this type.
- **Reorganization** operator takes one matrix and reorganizes the elements in the matrix. The transpose operator belongs to this type.

Many machine learning systems [1, 6–8, 12, 15, 22, 25, 37] support operator fusion to perform a complex of basic matrix operators as a single fused operator. These systems generate and execute a Direct Acyclic Graph (DAG) as a query plan. In a DAG, the leaf and root vertices are input and output matrices, respectively. Other vertices in a DAG are matrix operators, and the edge between two vertices is data flow. Operator fusion finds a sub-DAG as a fused operator. The query plan containing one or more such sub-DAGs is called a *fusion plan*. A sub-DAG is called a *partial fusion plan*. The search space for finding partial fusion plans in a DAG is exponential [8]. There are four types of operator fusions [8]: Cell, Row, Outer, and Multi-aggregation.

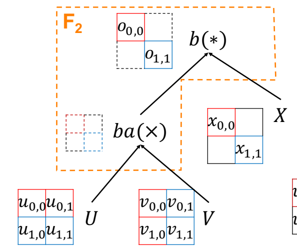
Cell fusion finds a partial fusion plan that contains consecutive unary or binary operators. Since the input and output matrices have the same dimensions in unary and binary operators, element-wise operations can be easily executed in a fused manner. Figure 2(a) shows an example of Cell fusion for $X * U \div V$, where a sparse matrix X has two non-zero elements, and two dense matrices U and V have 2×2 elements. The expression $X * U \div V$ is one of the matrix computation patterns used in the GNMF query [28]. The binary operators $b(*)$ and $b(\div)$ perform element-wise multiplication and division, respectively. Cell fusion finds F_0 as a partial fusion plan, which takes three input matrices X, U, and V and performs element-wise operations, i.e., $*$ and $\div$, in a fused manner. F_0 avoids the materialization of $X * U$ and directly results in the output matrix O for the entire expression $X * U \div V$.


F0
$o_{0,0}$ $o_{1,1}$
$b(\div)$
$b(*)$
$v_{0,0}$ $v_{0,1}$ $v_{1,0}$ $v_{1,1}$
V
$x_{0,0}$ $x_{1,1}$
X
$u_{0,0}$ $u_{0,1}$ $u_{1,0}$ $u_{1,1}$
U

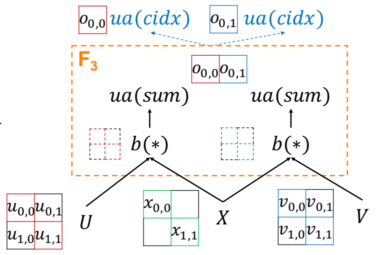
(a) Cell fusion ($X * U \div V$)


F1
$o_{0,0}$ $o_{1,0}$
$ba(\times)$
$r(T)$
$x_{0,0}$ $x_{0,1}$ $x_{1,0}$ $x_{1,1}$
X
$ba(\times)$
$x_{0,0}$ $x_{0,1}$ $x_{1,0}$ $x_{0,1}$
X
$s_{:,0}$
S

(b) Row fusion ($(X \times S)^T \times X$)


F2
$o_{0,0}$ $o_{1,1}$
$b(*)$
$ba(\times)$
$x_{0,0}$ $x_{1,1}$
X
$u_{0,0}$ $u_{0,1}$ $u_{1,0}$ $u_{1,1}$
U
$v_{0,0}$ $v_{0,1}$ $v_{1,0}$ $v_{1,1}$
V

(c) Outer fusion ($(U \times V) * X$)



(d) Multi-Aggregation ($sum(U * X), sum(X * V)$)

Figure 2: Examples of Operator fusion: Cell, Row, Outer, and Multi-aggregation.

Row fusion finds a partial fusion plan that reuses the rows of an input matrix to avoid the redundant scan of the input matrix and the materialization of intermediate matrices. The partial fusion plan may contain binary, unary, binary aggregation, and reorganization operators. Figure 2(b) shows an example of Row fusion for $(X \times S)^T \times X$, where S is 2×1. The expression $(X \times S)^T \times X$ is one of the matrix computation patterns used in principle analysis components (PCA) [9]. The F_1 contains three operators: two matrix multiplications $ba(x)$ and one transpose $r(T)$, where the rows of X are used twice, but scanned only once in the fused operator. In addition, F_1 avoids the materialization of $X \times S$.

Outer fusion fuses matrix multiplication and element-wise multiplication to avoid unnecessary calculations. The zero elements in an input matrix of element-wise multiplication do not need to be calculated in matrix multiplication as well. It is called sparsity exploitation [8]. Figure 2(c) shows an example of Outer fusion for $(U \times V) * X$, which is one of the matrix computation patterns used in loss computation such as weighted squared loss [7, 20, 21, 31] and generalized KL-divergence [27]. The F_2 includes matrix multiplication $ba(\times)$ and element-wise multiplication $b(*)$. It performs them only for non-zero elements of X. As a result, it avoid unnecessary calculations, i,e, $u_{0,:} \times v_{:,1}$ and $u_{1,:} \times v_{:,0}$, in $U \times V$.

Multi-aggregation fusion fuses multiple aggregation operators that have the same input matrix(es). In general, one matrix operator has one output. But, the fused operator found by Multi-aggregation fusion has more than two outputs. Figure 2(d) shows an example of Multi-aggregation fusion. The F_3 contains two binary operators $b(*)$ and two unary aggregation operators $ua(sum)$ and generates the output matrix O. $O_{0,0}$ stores the result of $sum(U * X)$, while $O_{0,1}$ stores the result of $sum(X * V)$. After F_3, the additional unary aggregation operator $ua(cidx)$ is called to pick a specific result in O, where $cidx$ is a column index (e.g., 0 or 1).

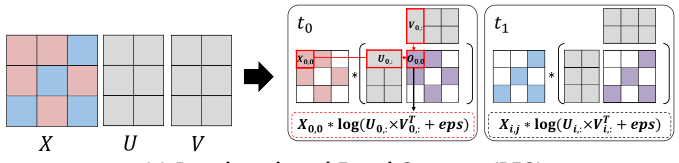
2.2 Distributed Fused Operators

Once distributed matrix computation systems [6, 7, 14, 18, 36, 37] finds partial fusion plans in a query DAG, as in Section 2.1, they generate a fusion plan containing the corresponding fused operators and execute the fusion plan in a distributed manner. They usually represent a matrix as a grid of fixed-sized *blocks* and use a block as a basic unit of matrix computation [6, 7, 10, 14, 17, 18, 33, 36, 37]. A block typically has the same width and height, e.g., 1000×1000. Without loss of generality, each distributed fused operator in the fusion plan is executed as the following three steps [8].

- ***Matrix consolidation*** step consolidates the input matrices required to each task. The input of a fused operator usually consists of a main matrix and side matrices. The main matrix is the one having the largest number of elements and repartitioned to the tasks in a cluster of machines. The side matrices are assigned to the tasks in a broadcast or replication manner, and so may have more than one replica in the cluster.
- ***Local operation*** step performs the fused operator using the blocks in a task and generates the result (intermediate blocks). The fused operator consists of multiple basic operators, and some of them are usually computed redundantly according to the replication of blocks of input matrices.
- ***Matrix aggregation*** step aggregates the intermediate blocks by shuffling them to generate the final result. This step is optional depending on the presence of the last aggregation operator in the fused operator.

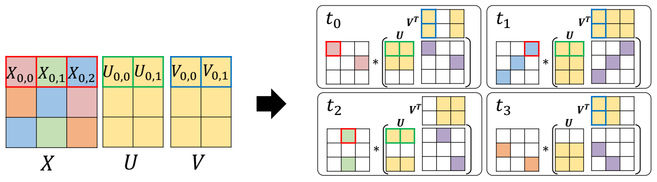
There are two kinds of distributed fused operators depending on the strategy of the matrix consolidation step [8]: Broadcast-based Fused Operator (BFO) and Replication-based Fused Operator (RFO). We explain each method in more detail. we denote the number of tasks in the cluster as T.

BFO broadcasts the side matrices to all the tasks where the main matrix is repartitioned. Figure 3(a) shows an example of BFO for the expression $O = X * log(U \times V^T + eps)$ in NMF [27], where X is 3×3 blocks, and both U and V are 3×2 blocks. The blocks



t_0
t_1
$X_{0,0} * \log(U_{0,:} \times V_{0,:}^T + eps)$
$X_{i,j} * \log(U_{i,:} \times V_{i,:}^T + eps)$
X
U
V

(a) Broadcast-based Fused Operator (BFO)



(b) Replication-based Fused Operator (RFO)

Figure 3: Example of distributed fused operators for $O = X * log(U \times V^T + eps)$.

of X in different colors indicate different partitions. The blocks of U and V are broadcasted to all tasks, t_0 and t_1. In the local operation step, for example, t_0 computes $X_{0,0} * log(U_{0,:} \times V_{0,:}^T + eps)$ and generates $O_{0,0}$. We assume that the main matrix X is $I \times J$, and the side matrices U and V are $I \times K$ and $J \times K$, respectively. Then, the memory usage per task is $\frac{|X|}{T} + |U| + |V|$ for input and $\frac{|O|}{T}$ for output. The communication cost, i.e., the amount of data transferred via network, is $|X| + T \cdot (|U| + |V|)$ in the matrix consolidation step. BFO executes the transpose of V (before performing matrix multiplication) two times in a cluster: one is in t_0 and the other is in t_1. In general, BFO executes the transpose T times.

RFO replicates the blocks of the side matrices by up to I or J times, where X is $I \times J$. Figure 3(b) shows an example of RFO for the same query. The red-lined blocks $\{X_{0,0}, X_{0,1}, X_{0,2}\}$ of X are repartitioned to t_0, t_1, and t_2. Thus, the green-lined blocks of U and the blue-lined blocks of V are replicated by up to 3 times to the same set of tasks. Then, the local operation step of t_0, t_1, and t_2 generate $O_{0,0}$, $O_{0,2}$, and $O_{0,1}$, respectively. The memory usage per task is $\frac{|X|}{T} + \frac{J \cdot |U|}{T} + \frac{I \cdot |V|}{T}$ for input and $\frac{|O|}{T}$ for output. The communication cost is $|X| + J \cdot |U| + I \cdot |V|$ in the matrix consolidation step. RFO executes the transpose of V three times on average (in general, I times on average).

Table 1 summarizes the communication cost, memory usage per task, and the maximum parallelism of BFO and RFO for $O = X * log(U \times V^T + eps)$. Our cuboid-based fused operator (CFO) will be explained in Section 3.

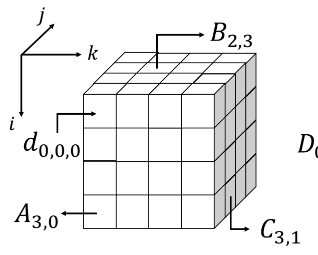
2.3 Cuboid-based Matrix Multiplication

The matrix multiplication of $C = A \times B$ can be computed as in Eq.(1), where $C_{i,j}$ $(0 \le i < I, 0 \le j < J)$ is a block of C.

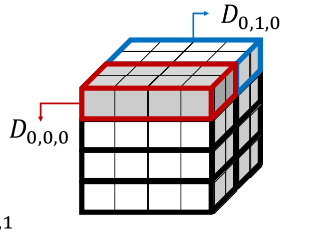
$$C_{i,j} = \sum_{0 \le k < K} A_{i,k} \cdot B_{k,j} = \sum_{0 \le k < K} C_{i,j}^k \qquad (1)$$

It can be represented as a 3-dimensional model [18], as in Figure 4(a), where $I = J = K = 4$. The ik-plane of the model indicates the matrix A, the kj-plane indicates B, and the ij-plane indicates C. The model space has a total of $I \cdot J \cdot K$ voxels. A *voxel* $d_{i,j,k}$ indicates that $A_{i,k} \cdot B_{k,j}$ generates a partial result of $C_{i,j}$, denoted by $C_{i,j}^k$.

Cuboid-based Matrix Multiplication (CuboidMM) [18] conceptually partitions the 3-dimensional model space into multiple cuboid-shaped chunks of voxels to reduce network communication. Its (P, Q, R)-cuboid partitioning partitions the whole space into $P \cdot Q \cdot R$ cuboids, where P, Q and R are the number of partitions on the i-axis, j-axis, and k-axis, respectively. Figure 4(b) shows an example of $(P = 4, Q = 2, R = 1)$-cuboid partitioning. Each cuboid consists of $\lceil \frac{I}{P} \rceil \times \lceil \frac{J}{Q} \rceil \times \lceil \frac{K}{R} \rceil$ voxels. $D_{p,q,r}$ indicates a specific cuboid, where $0 \le p < P$, $0 \le q < Q$, and $0 \le r < R$. CuboidMM determines (P, Q, R) as a small enough value for each cuboid to fit in the memory of a task, and at the same time, a large enough value for a cluster to fully exploit its parallelism, i.e., $P \cdot Q \cdot R \ge T$.



(a) 3-dimensional model



(b) (4,2,1)-cuboid partitioning

Figure 4: Example of Cuboid matrix multiplication.

CuboidMM performs the following three steps: matrix repartition, local multiplication, and matrix aggregation. In Figure 4(b), the matrix repartition step assigns each of eight cuboids to each task. A block of A is replicated to tasks by Q times, and a block of B is replicated to tasks by P times. The local multiplication step of a task performs matrix multiplication for the voxels in its cuboid and produces a partial result. For example, the cuboid $D_{0,0,0}$ produces the result of $C_{0,0}$ and $C_{0,1}$. The matrix aggregation step aggregates the partial results of R cuboids along the k-axis. In Figure 4(b), the matrix aggregation step does nothing because $R = 1$, i.e., the result of the local multiplication step is the final result.

3 CUBOID-BASED FUSION

In this section, we propose the model space for the cuboid-based fusion in Section 3.1 and propose the distributed Cuboid-based Fused Operator (CFO) in Section 3.2. Then, we present the cost-based optimization of CFO in Section 3.3.

3.1 3-Dimensional Model for Fused Operators

We propose a new 3-dimensional model space to represent an arbitrary fused operator that contains at least one matrix multiplication. The fused operators that do not contain matrix multiplication are relatively easy to find, and their costs in terms of computation and communication are relatively small. So, we focus on the ones that contain matrix multiplication in this study. We will explain partitioning the input matrices of a fused operator by using this model space in Section 3.2.

Figure 5(a) shows some basic matrix operators in Section 2.1 in our new model space. They are a unary operator $u(\hat{}2)$, a unary aggregation operator $ua(colSum)$, a binary operator $b(*)$, and a reorganization operator $r(T)$. All basic matrix operators, except binary aggregation, are represented along the same dimension. An operator and its operands are connected with a directed edge. For example, the binary operator $b(*)$ takes X and U and generates O.

Figures 5(b)-(e) show four types of operator fusions in Section 2.1 in our model space. Figure 5(b) shows Cell fusion of $X * U \div V$. It

Table 1: Comparison among distributed fusion methods for $O = X * log(U \times V^T + eps)$ ($P \leq I$, $Q \leq J$, and $R \leq K$).

Distributed fusion Methods	Communication cost	Memory usage per task	Maximum number of tasks	Redundant computation of the transpose of V
Broadcast-based fused operator (BFO)	$\lvert X\rvert + T\cdot(\lvert U\rvert+\lvert V\rvert)$	$\frac{\lvert X\rvert}{T} + \lvert U\rvert + \lvert V\rvert + \frac{\lvert O\rvert}{T}$	$I\cdot J$	T
Replication-based fused operator (RFO)	$\lvert X\rvert + J\cdot\lvert U\rvert + I\cdot\lvert V\rvert$	$\frac{\lvert X\rvert}{T} + \frac{J\cdot\lvert U\rvert}{T} + \frac{I\cdot\lvert V\rvert}{T} + \frac{\lvert O\rvert}{T}$	$I\cdot J$	I
Cuboid-based fused operator (CFO)	$R\cdot\lvert X\rvert + Q\cdot\lvert U\rvert + P\cdot\lvert V\rvert$	$\frac{R\cdot\lvert X\rvert}{T} + \frac{Q\cdot\lvert U\rvert}{T} + \frac{R\cdot\lvert V\rvert}{T} + \frac{\lvert O\rvert}{T}$	$I\cdot J\cdot K$	P

does not include binary aggregation, and so is represented along the same dimension. The dotted matrix between U and V is the output matrix of $b(*)$, which is actually not materialized. Figure 5(c) shows Row fusion of $(X \times S)^T \times X$. $X \times S$ is represented as a 3-dimensional model, since it is matrix multiplication. Its output matrix is not materialized, but fed into a transpose reorganization operator, $r(T)$ as input. The output of $r(T)$ is again fed into another matrix multiplication as input, without materialization. Figure 5(d) shows Outer fusion $(U\times V)*X$. The output matrix of $U\times V$ is fed into a binary operator $b(*)$ as input, without materialization. Figure 5(e) shows Multi-aggregation fusion $(sum(U * X), sum(X * V))$. The output matrices of two binary operators $b(*)$ are not materialized.

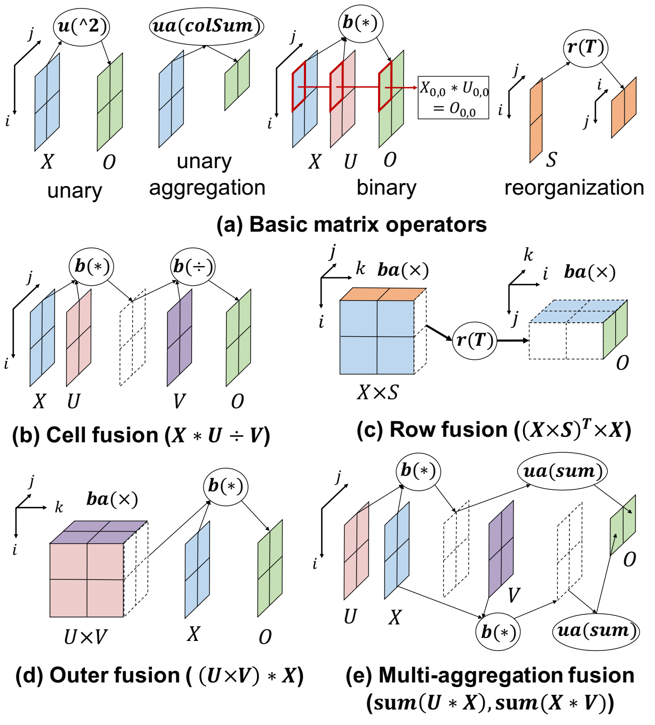
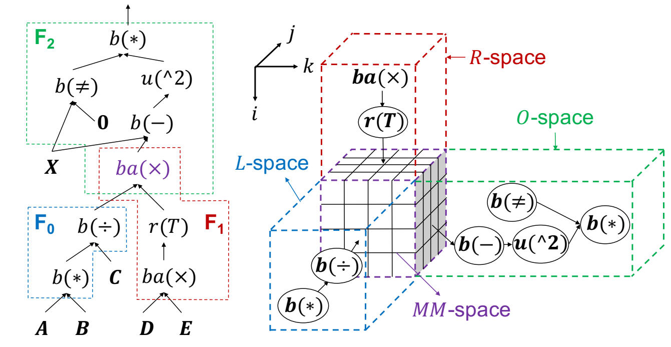

$u(\hat{}2)$
X O
unary
$ua(colSum)$
unary aggregation
$b(*)$
$X_{0,0} * U_{0,0} = O_{0,0}$
X U O
binary
$r(T)$
S
reorganization
(a) Basic matrix operators
$b(*)$ $b(\div)$
X U V O
(b) Cell fusion ($X * U \div V$)
$ba(\times)$ $r(T)$ $ba(\times)$
$X\times S$ O
(c) Row fusion ($(X\times S)^T \times X$)
$ba(\times)$ $b(*)$
$U\times V$ X O
(d) Outer fusion ($(U\times V) * X$)
$b(*)$ $ua(sum)$ $b(*)$ $ua(sum)$
U X V O
(e) Multi-aggregation fusion ($sum(U * X), sum(X * V)$)
i j k

Figure 5: Basic matrix operators and four types of operator fusions in our 3-dimensional model space.

The 3-dimensional model space for a fused operator containing matrix multiplication consists of four kinds of subspaces: L-space, R-space, O-space, and MM-space. Figure 6(a) shows an example DAG query plan, which include three partial fusion plans, F_0, F_1, and F_2. The types of operator fusions of F_0, F_1, and F_2 are Cell fusion, Row fusion, and Outer fusion, respectively. The entire query plan is represented as a single fused operator in our 3-dimensional model space, as in Figure 6(b). We denote the model space for main matrix multiplication in a query as MM-space. We assume that the center $ba(\times)$ in the DAG is the main matrix multiplication. Then, we denote the model space adjacent to the ik-plane of the MM-space as L-space. Likewise, we denote the model space adjacent to the jk-plane as R-space and that adjacent to the ij-plane as O-space. Here, $L-$, $R-$, and $O-$space means the left, right, and output space, respectively. The outputs of L-space and R-space become the input of MM-space, and the output of MM-space becomes the input of O-space.



(a) Example of query plan (b) Example of cuboid-based fusion

Figure 6: The 3-dimensional model for fused operators.

3.2 Cuboid-based Fused Operator (CFO)

In this section, we propose the Cuboid-based Fused Operator (CFO). Once the query plan generator described in Section 4 finds partial fusion plans, each partial fusion plan is executed by the CFO. The CFO is a physical operator and does not determine which operators it should fuse. The entire CFO is executed in a fused manner, i.e., do not generate intermediate matrices. The goal of the CFO is reducing communication cost as much as possible, and at the same time, exploiting the parallelism of a cluster as much as possible, for the memory limit of a task and the sizes of input matrices. Once the CFO partitions MM-space into (P, Q, R) cuboids, other three kinds of spaces, L-space, R-space, and O-space, are automatically partitioned according to the partitions of MM-space. Thus, the CFO partitions MM-space with considering the partitions of all other three spaces, so as to achieve the above goal.

The CFO performs (P, Q, R)-cuboid partitioning for MM-space. L-space, R-space, and O-space are partitioned by $(P, 1, R)$-, $(1, Q, R)$-, and $(P, Q, 1)$-cuboid partitioning, respectively. Figure 7(a) shows an example of $(P = 2, Q = 2, R = 2)$-cuboid partitioning for the model space in Figure 6(b). Each of L-, R-, and O-space has four partitions, while MM-space eight partitions. The first partitions of L-, R-, and O-space (in gray) are denoted as $L_{0,0,0}$, $R_{0,0,0}$, and $O_{0,0,0}$, respectively.

In general, a cuboid $D_{p,q,r}$ matches three cuboids $L_{p,0,r}$, $R_{0,q,r}$, and $O_{p,q,0}$, where $0 \le p < P$, $0 \le q < Q$, and $0 \le r < R$. We denote a combination of four cuboids, $D_{p,q,r}$, $L_{p,0,r}$, $R_{0,q,r}$, and $O_{p,q,0}$, as $P_{p,q,r}$. Each partitioned space of the CFO (e.g., $P_{0,0,0}$ in Figure 7(a)) is executed independently by a task.

Figures 7(b) and (c) show the details of L-space and R-space, which are partitioned into ($P = 2$, $Q = 1$, $R = 2$) and ($P = 1$, $Q = 2$, $R = 2$) cuboids, respectively. We rotate the original i-, j-, and k-axis for presentation of the details. In Figure 7(b), the input matrices A, B, and C are arranged like in Cell fusion of Figure 5(b), where the output matrix is not materialized since it is fed into matrix multiplication of MM-space. In Figure 7(c), the input matrices D and E are arranged like in Row fusion of Figure 5(c). The output of $D \times E$ is not materialized, and instead, after being transposed, it is fed into matrix multiplication of MM-space. Figure 7(d) shows the details of O-space, which contains four operators and two inputs, X and zero scalar. In Figure 7(d), all intermediate matrices are not materialized, and the final output matrix is materialized.

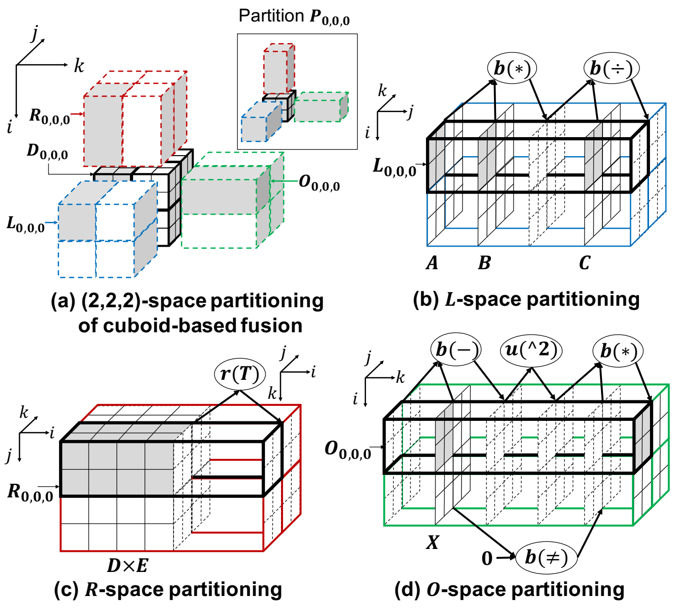


Figure 7: Example of space partitioning of the CFO.

Figure 8 shows the steps of cuboid-based fusion for $O = X * log(U \times V^T + eps)$, the same query in Section 2.2. In the figure, we just omit the transpose operator for V for simplicity. We assume there are three tasks, t_0, t_1, and t_2. The matrix consolidation step partitions the entire space into $\{P_{0,0,0}, P_{0,0,1}, P_{0,0,2}\}$, where $P_{0,0,0}$ consists of $D_{0,0,0}$ and $O_{0,0,0}$. The local operation step assigns each partition to each task. $P_{0,0,0}$ is processed by t_0, and $P_{0,0,2}$ is processed by t_2. We denote a single fused operation for a single output block as a kernel. In t_0, three kernels are executed since the partition $P_{0,0,0}$ has three output blocks $O_{0,0}$, $O_{0,1}$, and $O_{0,2}$. In each kernel, both input and output of $u(+)$ are not materialized, and the output of $u(log)$ (i.e., one input of $b(*)$) is also not materialized.

The bottom row in Table 1 summarizes the cost, memory usage, parallelism, and redundant computation of the CFO. The matrix U is replicated to tasks by Q times, the matrix V is replicated by P times, and the matrix X is replicated by R times. Thus, it reduces the

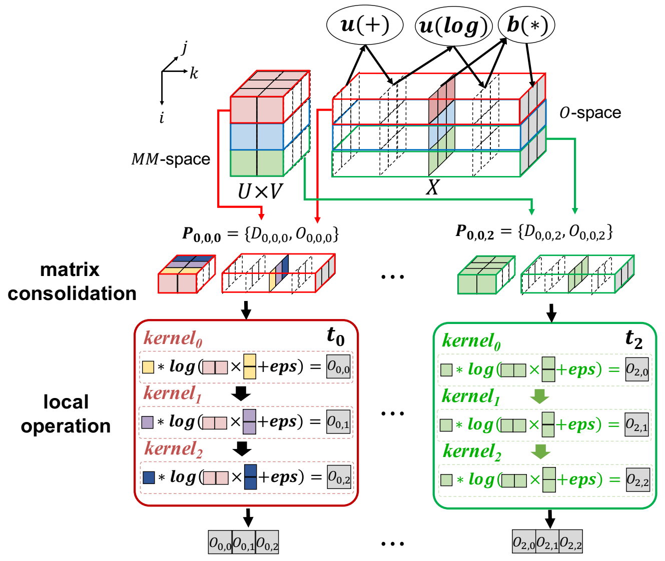

$u(+)$
$u(log)$
$b(*)$
j
k
i
O-space
MM-space
$U \times V$
X
$P_{0,0,0} = \{D_{0,0,0}, O_{0,0,0}\}$
$P_{0,0,2} = \{D_{0,0,2}, O_{0,0,2}\}$
matrix consolidation
local operation
$kernel_0$
t_0
$* \; log(\times +eps) = O_{0,0}$
$kernel_1$
$* \; log(\times +eps) = O_{0,1}$
$kernel_2$
$* \; log(\times +eps) = O_{0,2}$
t_2
$* \; log(\times +eps) = O_{2,0}$
$* \; log(\times +eps) = O_{2,1}$
$* \; log(\times +eps) = O_{2,2}$
$O_{0,0}$ $O_{0,1}$ $O_{0,2}$
$O_{2,0}$ $O_{2,1}$ $O_{2,2}$
…

Figure 8: Steps of cuboid-based fusion ($X * log(U \times V^T + eps)$).

communication cost for U and V compared with the RFO since $Q < J$ and $P < I$. Although the cost of the CFO rather increases for X by R times, our optimization method for (P, Q, R) in Section 3.3 tends to determine R as a value as small as possible. The memory usage per task is usually determined as a value smaller than the memory limit per task by adjusting (P, Q, R), and so out of memory (O.O.M.) does not occur in the CFO. In contrast, the BFO may have O.O.M. depending on the memory limit, since the memory usage per task is fixed. The number of maximum tasks (i.e., parallelism) of the CFO is higher than that of the BFO and RFO by K times. For the transpose of V, CFO executes the transpose three times in R-space because P=3. Since $P \le I$, and T is similar to $P \cdot Q \cdot R$, the degree of redundant computation of CFO is less than those of BFO and RFO.

Figure 9 shows theoretical communication cost and memory usage of BFO, RFO, and CFO while varying (P, Q, R). From Table 1, the BFO and RFO can be regarded as having ($P = T, Q = T, R = 1$) and ($P = I, Q = J, R = 1$), respectively. The BFO's memory usage is high although it has a low communication cost. The RFO has a high communication cost although its memory usage is low. Both methods have no control knob for the cost and memory usage. On the contrary, the CFO can determine the best parameters (P^*, Q^*, R^*) having the lowest communication cost within the memory budget per task.

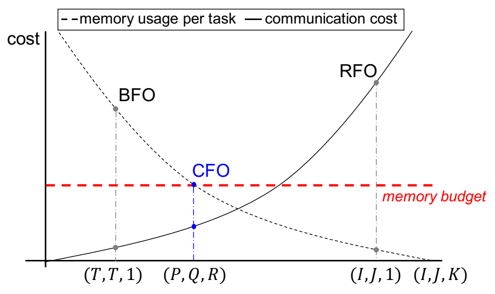


Figure 9: Comparison of theoretical cost and memory usage.

3.3 Parameter Optimization for CFO

In this section, we present a method of finding the optimal parameters (P^*, Q^*, R^*). We assume that we have a partial fusion plan F which will be executed as a single CFO. $F = (V, E)$ is a sub-DAG of a query plan in a DAG, where a vertex $v \in V$ is either a matrix operator or a matrix, and an edge $e \in E$ is matrix flow. We denote the main matrix multiplication as v_{mm} and the memory budget per task as θ_t. The vertex v_{mm} corresponds to MM-space in the CFO. We denote the peak network bandwidth as $\hat{B}^n$ (e.g., 1 Gbps) and the peak computation bandwidth as $\hat{B}^c$ (e.g., 500 GFLOPS). We use both bandwidths to normalize communication and computation costs.

We note that, in the CFO, not only data blocks are replicated via network, but also computations of matrix operators are repeated. We let the task processing a cuboid partition $P_{0,0,0}$ as $t(P_{0,0,0})$. Then, for example, in Figure 7, the four blocks of A, B, and C in $L_{0,0,0}$ are replicated to $t(P_{0,0,0})$ and $t(P_{0,1,0})$ (i.e., two times). In addition, the series of operators $b(*)$ and $b(\div)$ in $L_{0,0,0}$ is executed in both $t(P_{0,0,0})$ and $t(P_{0,1,0})$, that is, the computation is repeated two times in terms of a cluster.

The basic method of finding (P^*, Q^*, R^*) is picking the one having the minimum $Cost()$ and at the same time satisfying $MemEst(P, Q, R, F) \leq \theta_t$ in the search space of $(0 < P \leq I, 0 < Q \leq J, 0 < R \leq K)$. Here, $MemEst(P, Q, R, F)$ is the estimated memory usage per task when using (P, Q, R) for evaluating F. Hereafter, we simply denote (P, Q, R) as c, and so $MemEst(c, F)$ means $MemEst(P, Q, R, F)$. The function $Cost()$ is determined depending as the major cost between network communication cost $NetEst(c, F)$ and computation cost $ComEst(c, F)$, as in Eq.(2). It is reasonable to consider only the major cost because communication and computation can overlap[8]. The technique of overlapping computation and communication is possible because the basic unit of matrix computation is a block. For example, in Figure 3(b), we assume two tasks t_0 and t_1 run in the same worker node. Then, during t_0 computes $O_{0,0}$, the other task t_1 can read the necessary input blocks to compute $O_{0,2}$. Both $NetEst(c, F)$ and $ComEst(c, F)$ are the costs in terms of a cluster, and so normalized by $N \cdot \hat{B}^n$ and $N \cdot \hat{B}^c$, respectively, where N is the number of machines in a cluster.

$$Cost(c, F) = max(\frac{NetEst(c, F)}{N \cdot \hat{B}^n}, \frac{ComEst(c, F)}{N \cdot \hat{B}^c}) \tag{2}$$

We now present three estimations, $MemEst(c, F)$, $NetEst(c, F)$, and $ComEst(c, F)$, in detail. Algorithm 1 presents MemEst(c, F) that estimates the memory usage per task. It basically adds the costs in L-, R-, O-, and MM-space (Lines 2-9). Each of four spaces has its own cost. For example, in Figure 7(d), $O_{0,0,0}$ uses the memory of eight gray blocks in $t(P_{0,0,0})$. If either L-, R-, or O-space s includes matrix multiplication $v = ba(\times)$, we recursively call MemEst(c', v) since v may create another model space using v as v_{mm} in a confined space s (Line 5). In Line 4, c' is the parameter used for partitioning the space s. Specifically, the parameters for L-, R-, and O-space are $(P, 1, R)$, $(1, Q, R)$, and $(P, Q, 1)$, respectively. By passing c', MemEst(c', v) can know the confined space. If v is not a matrix multiplication operator in L-, R-, and O-space, or v is v_{mm} in MM-space, then we accumulate the memory size of v, i.e., Mem(v) (Lines 8-9). Here, we accumulate Mem(v) only when v is a materialized one (e.g., input or output matrix). For example, in Figure 7(d), the result of accumulation is eight blocks, since Mem$(X) = 4$, and Mem$(b(*)) = 4$. The function Mem() is in Eq.(3), where $size(v)$ is divided by either $P \cdot R$, $Q \cdot R$, or $P \cdot Q$, due to partitioning.

$$Mem(c = (P, Q, R), v) = \begin{cases} size(v) \cdot \frac{1}{P \cdot R}, & \text{if } v \in L\text{-space} \\ size(v) \cdot \frac{1}{Q \cdot R}, & \text{if } v \in R\text{-space} \\ size(v) \cdot \frac{1}{P \cdot Q}, & \text{if } v \in O\text{-space} \end{cases} \tag{3}$$

The algorithm for $NetEst(c, F)$ is similar to Algorithm 1, except Mem(v) being replaced with Net(v) in Eq.(4). For example, in Figure 7(b), the four gray blocks of A are replicated twice (i.e., $Q = 2$).

$$Net(c = (P, Q, R), v) = \begin{cases} Q \cdot size(v), & \text{if } v \in L\text{-space} \\ P \cdot size(v), & \text{if } v \in R\text{-space} \\ R \cdot size(v), & \text{if } v \in O\text{-space} \end{cases} \tag{4}$$

Likewise, the algorithm for $ComEst(c, F)$ is similar to Algorithm 1, except $isMaterialized(v)$ being replaced with $isOperator(v)$, and Mem(v) being replaced with Com(v) in Eq.(5), where $numOp(v)$ indicates the number of floating point operations to compute the operator v. As mentioned above, in L-, R-, and O-space, the computation of an operator is repeated by Q, P, and R times, respectively. However, in MM-space, matrix multiplication (v_{mm}) is computed only once.

$$Com(c = (P, Q, R), v) = \begin{cases} Q \cdot numOp(v), & \text{if } v \in L\text{-space} \\ P \cdot numOp(v), & \text{if } v \in R\text{-space} \\ R \cdot numOp(v), & \text{if } v \in O\text{-space} \\ numOp(v), & \text{if } v = v_{mm} \end{cases} \tag{5}$$

Finding the optimal parameters in an exhaustive manner may be time-consuming, since the size of search space $I \times J \times K$ increases exponentially as the sizes of input matrices increase. Thus, we use a simple pruning method that prunes a set of parameters $\{(P, Q, R)\}$ having a larger $Cost()$ in Eq.(2), compared to the current best parameters. For example, if the cost of $(P = 1, Q = 5, K = 5)$ is larger than the current best parameters, then we can prune a set of parameters $\{(P > 1, Q = 5, R = 5)\}$ because a larger P increases the cost in both $Net()$ and $Com()$. We also can prune a set of parameters $\{(P, Q, R)\}$ such that $P \times Q \times R < N \times T_c$, where T_c is the number of tasks per node. Such parameters do not fully exploit the parallelism of a given cluster. If $I \times J \times K < N \times T_c$, we set the parameters to

ALGORITHM 1: MemEst

```
Input: $c \in \{P, Q, R\}$, a partial fusion plan $F$
Output: $cost$
1  $cost \leftarrow 0$;
2  for $s \in \{L, R, O, MM\}$-space of $F$ do
3      if $s \neq MM \wedge v = ba(\times) \wedge v \in s$ then
4          $c' \leftarrow (P, 1, R)$ for $L$, $(1, Q, R)$ for $R$, $(P, Q, 1)$ for $O$;
5          $cost \leftarrow cost +$ MemEst$(c', v)$;
6      else
7          for $v \in s$ do
8              if $isMaterialized(v)$ then
9                  $cost \leftarrow cost +$ Mem$(v)$;
10 return $cost$;
```

the ones as large as possible. This pruning method is very efficient, and so can find the optimal parameters within a few milliseconds using a single thread for the search space of $I \times J \times K = 2M$.

4 CUBOID-BASED FUSION PLAN GENERATOR

The CFO executes all its basic operators in a fused manner. But, the CFO itself cannot determine its scope, i.e., cannot determine which set of basic operators should be fused into the CFO. Thus, we need a fusion plan generator to determine the scope of each CFO.

The existing systems [8, 37] tend to avoid the generation of a partial fusion plan that includes large-scale matrix multiplication. In particular, the state-of-the-art fusion plan generator, GEN [8] of SystemDS, generates a partial fusion plan that includes large-scale matrix multiplication only when sparsity exploitation is possible, as in Outer fusion. The main reason of avoiding such a partial fusion plan in the existing systems is that they have no control knob for communication cost and memory usage, as explained in Section 3.2. Without such a knob, a partial fusion plan including large-scale matrix multiplication tends to run into O.O.M. error. On the contrary, our system FuseME has a control knob for them, i.e., partitioning parameters (P, Q, R). So, it can freely generate a partial fusion plan including large-scale matrix multiplication.

In this section, we propose a Cuboid-based Fusion plan Generation (CFG) method for doing that. CFG consists of two phases: the exploration phase to find candidate plans and the exploitation phase to refine the candidate plans. We present the former in Section 4.1 and the latter in Section 4.2.

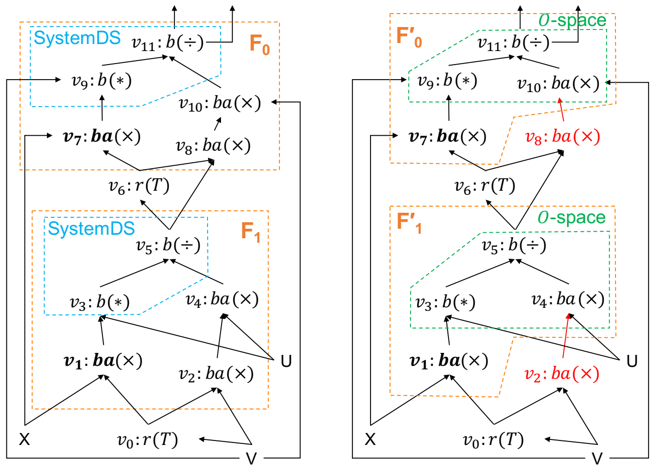
We explain our CFG method using the GNMF query, which approximates the two factor matrices U and V for the rating matrix X as in Eq.(6), where i is the iteration number, U and V are dense, and X is sparse.

$$U_{i+1} = \frac{U_i * (V_i^T \times X)}{V_i^T \times V_i \times U_i}, \; V_{i+1} = \frac{V_i * (X \times U_i^T)}{V_i \times U_i \times U_i^T} \tag{6}$$

4.1 Finding Candidate Partial Fusion Plans

The operator fusion can avoid the materialization of intermediate matrices by fusing multiple operators. However, it cannot fuse all kinds of operators. There are two kinds of operators that require the materialization of intermediate matrix and so cannot be fused together with other operators: (1) the operator having more than two outgoing edges in a DAG and (2) the unary aggregation operator requiring a shuffle. The output matrix of the former operator is called materialization point, because it should be materialized for multiple operators so as to take the matrix as input [8].

Figure 10(a) shows a query plan in a DAG for the GNMF query (i.e., Eq. 6), which consists of twelve operators $\{v_i | 0 \le i < 12\}$. Among them, four operators v_0, v_5, v_6, and v_{11}, have such materialization points. For the latter operator, if the input matrix is small, the operator may not require a shuffle, and so can be fused. However, if the input matrix is too large, then the operator should materialize its partial results and shuffle them. Obviously, its final result is also materialized. For example, if $colSum(X)$ is added in Figure 7(d), the partial result of $colSum(X)$ in each task should be materialized and shuffled to get the final result of $colSum(X)$. For the binary aggregation operator (e.g., $ba(\times)$), it is fused as a main operator (MM-space) in our FuseME.



(a) Candidate partial fusion plans (b) Determining partial fusion plans

Figure 10: Results of two phases of CFG for GNMF.

We call the above two kinds of operators as the termination operator. The CFG method finds candidate partial fusion plans such that a partial fusion plan does not include any termination operator, except for the top operator. That is, a partial fusion plan can have a termination operator only as the top (root) operator. It starts with a matrix multiplication operator as an initial candidate plan and extends it by fusing its adjacent operators. In a DAG, an operator usually has two kinds of adjacent operators: outgoing (i.e., parent) adjacent operators and incoming (i.e., child) adjacent operators.

Figure 10(a) shows a fusion plan generated by the exploration phase of our CFG for GNMF, which contains two candidate partial fusion plans, F_0 and F_1. We note that SystemDS fuses only two operators v_3 and v_5 in F_1 (in blue dotted). We assume that CFG starts one of matrix multiplications, v_1 (i.e., initial F_1). It has two adjacent operators v_3 and v_0. F_1 is extended to include v_3, i.e, $F_1 = \{v_1, v_3\}$, since v_3 is not a termination operator. But, F_1 does not include v_0, which is a termination operator. Now, F_1 has an adjacent operator v_5, which is a termination operator, but can be the top operator of F_1. Thus, F_1 becomes $\{v_1, v_3, v_5\}$ Next, F_1 is further extended to include v_4 and then v_2, and so becomes $\{v_1, v_2, v_3, v_4, v_5\}$. Since there is no additional operator that can be fused, the CFG method returns F_1 as a candidate plan. Next, we assume that CFG starts with v_8 as a seed of another candidate plan F_0. Similar to F_1, F_0 is extended to include v_{10}, v_{11}, v_9, and v_7, but does not include v_6 since it is a termination operator that cannot be the top operator. As a result, CFG finds two candidate partial fusion plans F_0 and F_1.

Algorithm 2 presents the exploration phase to find candidate partial fusion plans. In the algorithm, the function $adjacent(F, top)$ (Lines 8 and 18) returns a set of adjacent operators of the partial fusion plan F, excluding the outgoing adjacent operators if the flag top is true.

4.2 Refining Partial Fusion Plans

Each candidate partial fusion plan obtained by Algorithm 2 may be too large in terms of the memory limit θ_t, or rather less efficient than two or more smaller partial fusion plans split from the original plan. In order to solve these issues, the exploitation phase of CFG refines each candidate plan.

ALGORITHM 2: ExplorationPhase

```
Input: $G$: query plan in a DAG
Output: $\mathcal{F}$: candidate partial fusion plans
1  $\mathcal{F} \leftarrow \emptyset$;
2  $W \leftarrow$ all operators $\{v_i\}$ in $G$;          /* W: workload */
3  while $ba(\times)$ exists in $W$ do
4      $v_m \leftarrow$ any $ba(\times)$ in $W$;
5      $W \leftarrow W - v_m$;
6      $F \leftarrow \{v_m\}$;
7      $top \leftarrow false$;          /* flag of reaching the top */
8      $ADJ \leftarrow adjacent(F, top) \cap W$;
9      while $ADJ \neq \emptyset$ do
10         for $v_i \in ADJ$ do
11             if $isTermination(v_i) = false$ then
12                 $F \leftarrow F \cup \{v_i\}$;
13             else
14                 if $isOutgoing(v_i) = true$ then
15                     $F \leftarrow F \cup \{v_i\}$;
16                     $top \leftarrow true$;
17             $W \leftarrow W - v_i$;
18         $ADJ \leftarrow adjacent(F, top) \cap W$;
19     $\mathcal{F} \leftarrow \mathcal{F} \cup \{F\}$;
20 return $\mathcal{F}$;
```

Algorithm 3 presents the phase. F may have multiple matrix multiplications, and the one is determined as the main matrix multiplication v_m (Line 3). Then, we find the optimal parameters and calculate the cost of F, as explained in Section 3.3 (Lines 4-5). If F is too large and so cannot fit in θ_t, then (P^*, Q^*, R^*) are determined as (I, J, K), and *cost* becomes an infinite value. Next, we find all matrix multiplication operators in F, except v_m, as a set of splitting points SP (Line 6). All operators $SP = \{v_i\}$ are sorted by the minimum distance (i.e., number of hops) between v_i and v_m in descending order. For example, in Figure 10(b), the distance between v_1 and v_4 is three. We try to first split the most distant v_i from F since such v_i tends to cause the highest cost in F (Lines 8-16), which will be explained in more detail later. When splitting v_i from F, if v_i has its descendent operators in F, the operators are also split together with v_i becoming F_i. After F is split into F_m and F_i, we find their optimal parameters and calculate their costs (Lines 10-13). Then, if the sum of costs of F_m and F_i is smaller than the original cost, we decide to split them actually (Lines 14-16).

Figure 11 shows the model space of F_1 in Figure 10(a). We assume that v_1 is the main matrix multiplication, $(P = 2, Q = 3, R = 2)$, X is 4, U is 2×3, and V is 4×2. L-space and R-space of v_1 are V^T and X, respectively. O-space of v_1 contains $\{v_2, v_3, v_4, v_5\}$. Among them, v_2 and v_4 form cuboids in the O-space, since they are matrix multiplication. v_4 has its own L-,R-, and O-space recursively. L-space and R-space of v_4 are v_2 and U, respectively. Since v_1 is partitioned with $(P = 2, Q = 3, R = 2)$, O-space of v_1 is partitioned with $(P = 2, Q = 3, R = 1)$, and L-space of v_4 is again partitioned with $(P = 2, Q = 1, R = 1)$, as explained in Section 3.3. The red boxes show a partition $P_{0,0,0}$ which will be assigned to a task. There are a total of $2 \times 3 \times 2 = 12$ tasks for F_1 due to $(P = 2, Q = 3, R = 2)$. In this situation, twelve input blocks of v_2 in $P_{0,0,0}$, i.e., eight blocks of V^T and four blocks of V, must be replicated to six tasks, since $Q \times R = 2 \times 3 = 6$. On the contrary, the input blocks of v_4 in $P_{0,0,0}$, i.e., , six blocks of U, must be replicated only to two tasks, since $R = 2$. Thus, the more distant matrix multiplication from v_1, i.e., v_2, tends to cause the higher cost than the less distant one from v_1, i.e., v_4. Figure 10(b) shows the situation where F_1 is split into F_1' and v_2, and similarly F_0 is split into F_0' and v_8. Thus, the outputs of v_2 and v_8 are materialized and fed into F_1' and F_0', respectively.

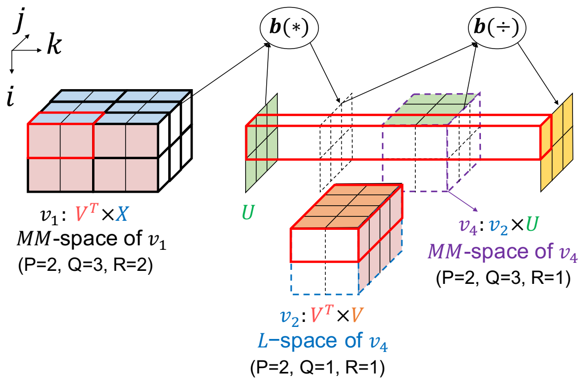


Figure 11: Model space of F_1 in Figure 10(a).

5 IMPLEMENTATION

In this section, we briefly explain the implementation of FuseME. We implement FuseME on top of Apache Spark, implement the fusion plan generator by extending SystemML [7, 14], and implement the fused operators by extending DistME [18]. As a result, our system allows users to describe their matrix queries (e.g., GNMF) using Scala API and Declarative Machine Language of SystemML. From the user query, the CFG of FuseME generates a fusion plan and selects our CFO fused operators as physical operators.

ALGORITHM 3: ExploitationPhase

```
Input: $G$: query plan in a DAG,
       $\mathcal{F}$: candidate partial fusion plans,
       $\theta_t$: the memory budget per task
Output: $\mathcal{F}^*$: final partial fusion plans
1  $\mathcal{F}^* \leftarrow \emptyset$;
2  for $F \in \mathcal{F}$ do
3      $v_m \leftarrow ba(\times)$ in $F$ having the largest $(I \times J \times K)$;
4      $(P^*, Q^*, R^*) \leftarrow$ the optimal parameters for $F$;
5      $cost \leftarrow Cost(P^*, Q^*, R^*, F)$;
6      $SP \leftarrow$ all $ba(\times)$ in $F$, except $v_m$; /* $SP = \{v_i\}$ */
7      desc. sort $SP$ by the min. distance between $v_i$ and $v_m$;
8      for $v_i \in SP$ do
9          $\{F_m, F_i\} \leftarrow$ split $F$ by $v_m$ and $v_i$;
10         $(P^*_m, Q^*_m, R^*_m) \leftarrow$ the optimal parameters for $F_m$;
11         $cost_m \leftarrow Cost(P^*_m, Q^*_m, R^*_m, F_m)$;
12         $(P^*_i, Q^*_i, R^*_i) \leftarrow$ the optimal parameters for $F_i$;
13         $cost_i \leftarrow Cost(P^*_i, Q^*_i, R^*_i, F_i)$;
14         if $cost > cost_m + cost_i$ then
15             $\mathcal{F} \leftarrow \mathcal{F} \cup F_i$;
16             $F \leftarrow F_m$;
17     $\mathcal{F}^* \leftarrow \mathcal{F}^* \cup F$;
18 return $\mathcal{F}^*$
```

A block of a matrix is implemented using RDD (Resilient Distributed Datasets) [38], 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 DenseMatrix or SparseMatrix class. FuseME supports a number of matrix operators, such as element-wise, matrix multiplication, and transpose, which are implemented using breeze[1] libraries of Scala. We implement them based on the transformation operations of RDD, i.e., map, groupByKey, Cogroup, and reduceByKey. For the row, column, and grid partitioning schemes of FuseME, we extend the RDD partitioner class. We use the parquet format for reading and writing the matrix data with HDFS. We note that our FuseME does not consider data distribution of matrices as the existing systems including SystemDS, DMac, and DistME do not consider it and so cannot support sparse big matrices.

6 EXPERIMENTAL EVALUATION

In this section, we present experimental results in four categories.

- We compare the CFO with the existing BFO and RFO of SystemDS [6] and DistME [18], in terms of the elapsed times and communication cost (i.e., amount of transferred data in the matrix consolidation and matrix aggregation steps). We also vary the number of nodes.
- We check whether the P, Q, and R parameters determined by the optimization in Section 3.3 can achieve the best performance for the CFO. We also compare the pruning method with the exhaustive method proposed in DistME [18].
- We compare the performance of the fusion plan of our FuseME with that of the state-of-the art systems, SystemDS [6], MatFast [37], and DistME [18], in terms of the elapsed times. We use GNMF [28] as a query. We include DistME although it does not support operator fusion, since it is the fastest among the systems not supporting operator fusion.
- We compare the performance of a deep learning query, AutoEncoder [4], of FuseME with those of SystemDS [6] and Tensorflow [1].

6.1 Experimental Setup

Datasets: For experiments, we use both real and synthetic datasets. For real datasets, we use MovieLens [19] for a small size, Netflix [41] for a medium size, and YahooMusic[2] for a large size. Table 2 summarizes the statistics of three datasets. We mainly 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 SystemDS [6] and DistME [18]. The density 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.

Table 2: Statistics of real datasets.

dataset	size of matrix X (users × items)	# of non-zeros of X (rating type: double)
MovieLens	283,228 × 58,098	27,753,444
Netflix	480,189 × 17,770	100,480,507
YahooMusic	1,823,179 × 136,736	717,872,016

[1]https://github.com/scalanlp/breeze

[2]https://webscope.sandbox.yahoo.com/catalog.php?datatype=r

Systems compared: We compare our FuseME with SystemDS, MatFast, DistME, and TensorFlow. We use the original codes for SystemDS[3], MatFast[4], DistME (from the authors), and TensorFlow 2.3.0. SystemDS supports the state-of-the-art fusion method GEN [8, 12]. MatFast uses a simple folded operator that fuses consecutive element-wise operators. DistME has CPU and GPU versions. We use its CPU version for fair comparison. SystemDS, DistME and FuseME all are implemented on top of Apache Spark [39]. TensorFlow uses TensorFlow XLA [15] to generate a fusion plan of the DAG.

H/W and S/W setting: We conduct all the experiments on the same cluster of one coordinator node and eight worker nodes. All nodes are connected via 1 Gbps Ethernet. Each node is equipped with an eight-core 3.6GHz CPU, 128 GB main memory, 10 TB HDD for Spark and HDFS. In terms of software, we use Ubuntu 18.04.3 LTS, Spark 2.4.5, Hadoop 2.7.2, and TensorFlow 2.3.0. We set the number of tasks per node to 12 ($T_c = 12$), and so the memory budget per task is $\theta_t = 10$ GB. We use the block size of 1000×1000 in all experiments, which is the default size in other systems such as MatFast and SystemDS. We set the number of instances per node to 12 for TensorFlow.

6.2 Comparison of Distributed Fused Operators

We compare the performances of the BFO, RFO, CFO, and DistME using synthetic matrices for the query $X * log(U \times V^T + esp)$, where X is sparse, and both U and V are dense. Here, X has $I \times J$ blocks, U has $I \times K$, and V has $J \times K$. We use three types of datasets, as in [17, 18]: matrices varying two large dimensions ($I = J > K$), matrices varying a common dimension ($K \leq I = J$), and matrices varying the density ($0 \leq |X| \leq I \times J$). Both BFO and RFO are evaluated on SystemDS, and the CFO is evaluated on FuseME. We note that SystemDS uses only either BFO or RFO, which is determined depending on the input matrices. It uses the BFO, if the number of partitions of X is smaller than I or J. Otherwise, it uses the RFO.

We denote the BFO and RFO of SystemDS by SystemDS(B) and SystemDS(R), respectively. Table 3 summarizes the sizes of three types of datasets used, the density of X used, and the optimal parameters (P^*, Q^*, R^*) used in the CFO of FuseME. The optimal parameters are automatically determined as in Section 3.3. Figures 12(a), (b), and (c) show the elapsed times of the BFO, RFO, CFO, and DistME for three types of datasets, and Figures 12(e), (f), and (g) show their corresponding communication costs. In the figures, T.O. means time out (longer than 12 hours).

Matrices varying two large dimensions: Figures 12(a) and (e) show that the CFO of FuseME significantly outperforms the BFO of SystemDS and DistME in terms of both elapsed times and communication cost. In this experiment, U and V are tall matrices (i.e., $I > K$ and $J > K$). SystemDS uses the BFO because the number of partitions of X is smaller than I and J due to the low density of X (i.e., 0.001). The BFO times out when $n = 750K$. The improvement of the CFO compared to the BFO becomes more marked as n gets larger. In details, in terms of elapsed times, the improvement is 21×, 85×, and 238×, when n is 100 K, 250 K, and 500 K, respectively. In terms of communication cost, the improvement is 3.9×, 17.1×, and 64×, when n is 100 K, 250 K, and 500 K, respectively. FuseME

[3]https://github.com/apache/systemds

[4]https://github.com/purduedb/MatRel

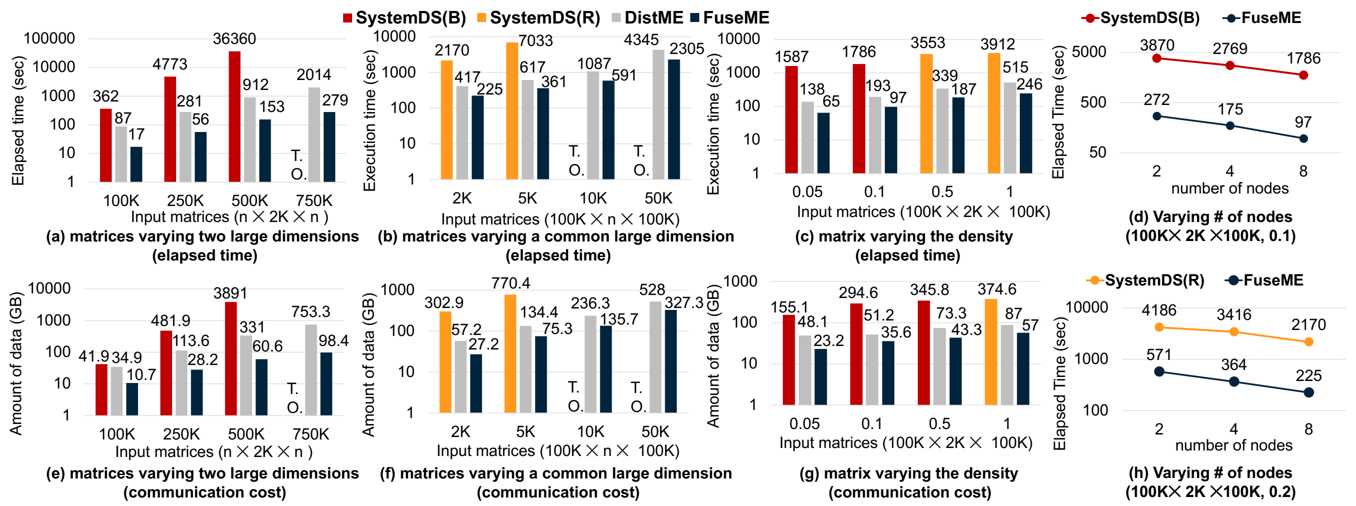


Figure 12: Performance comparison among distributed fused operators: BFO (SystemDS(B)), RFO (SystemDS(R)), and CFO (FuseME). DistME is compared as the fastest one among the systems not supporting operator fusion.

Table 3: Statistics of synthetic datasets (K: thousand).

type	size of n	density of X	(P^*, Q^*, R^*)
Matrices	$100K$	0.001	(8,6,2)
varying two	$250K$	0.001	(8,6,2)
large dimensions	$500K$	0.001	(8,6,2)
$(n \times 2K \times n)$	$750K$	0.001	(8,6,2)
Matrices	$2K$	0.2	(12,8,1)
varying a common	$5K$	0.2	(8,6,2)
large dimension	$10K$	0.2	(6,4,4)
$(100K \times n \times 100K)$	$50K$	0.2	(4,3,8)
Matrices	$100K$	0.05	(8,6,2)
varying	$100K$	0.1	(8,6,2)
the density	$100K$	0.5	(12,8,1)
$(n \times 2\,K \times n)$	$100K$	1.0	(12,8,1)

significantly outperforms DistME in terms of both elapsed time and communication cost. It is because FuseME gains the advantages of sparsity exploitation and no materialization of intermediate matrices due to operator fusion.

Matrices varying a common large dimension: Figures 12(b) and (f) show similar tendencies with Figures 12(a) and (e). SystemDS uses the RFO because the number of partitions of X is larger than I and J due to the high density of X (i.e., 0.2). The RFO times out when n is 10 K and 50 K.

Matrices varying the density: Figures 12(c) and (g) show that the CFO still significantly outperforms the BFO, RFO, and DistME in terms of both elapsed times and communication cost. SystemDS uses the BFO when X has a relatively low density (i.e., 0.05, and 0.1), and uses the RFO when X has a relatively high density (i.e., 0.5 and 1.0). Both elapsed time and communication cost in this dataset increase less rapidly compared to the above two types of datasets, since the density increases in this dataset, while the dimension size increases in the above two datasets.

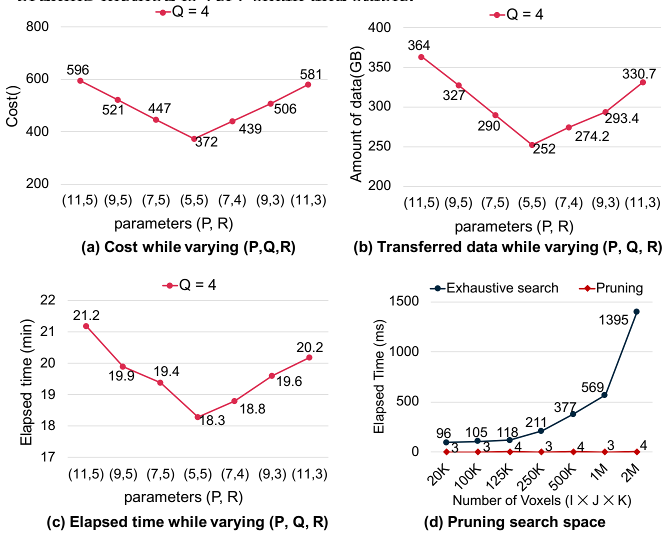
Varying the number of nodes: Figures 12(d) and (h) show the performance of BFO, RFO, and CFO using the dataset of 100 K × 2 K × 100 K in Table 3 while varying the number of nodes. BFO is used by SystemDS in Figure 12(d) since the density is 0.1, while RFO is used in Figure 12(h) since the density is 0.2. In the figures, CFO consistently outperforms both BFO and RFO. When the number of nodes is two, BFO and RFO still have a relatively high communication cost of 294.6 GB and 302.9 GB, respectively. On the contrary, CFO has a relatively small cost of 16.7 GB. As the number of nodes increases, the elapsed times of all three methods decrease due to utilizing more number of tasks, and the performance gap between CFO and other methods slightly gets larger.

Overall analysis: In Figure 12, the bigger difference on elapsed time than on communication cost is mainly due to (1) the difference in the number of tasks and (2) the issue of Apache Spark. First, in Figure 12(a), the size of X is small in byte due to its low density, and so X is repartitioned into a small number of partitions (e.g., 13 for $n = 100\,K$). Thus, BFO does not utilize the maximum number of tasks, while RFO and CFO utilize it. Second, as the communication cost increases (e.g., about 3.9 TB when n=500 K in Figure 12(e)), Apache Spark tends to occupy CPU cores more longer time for data shuffling, which makes the tasks take longer time to compute their partitions.

We note that the test query in this section is simple, and so the plan generator is not used in both SystemDS and FuseME. The entire query is executed as a single fused operator in both systems. In Figure 12(c), CFO significantly outperforms BFO and RFO, which shows the importance of CFO itself.

6.3 Optimization of (P,Q,R) Parameters

Figures 13(a)-(c) show $Cost()$ in Eq.(2), the amount of transferred data, and the elapsed times, while varying (P, Q, R) in the CFO for the same query with Section 6.2 on the matrices of $1\,M \times 5\,K \times 1\,M$. In Figure 13(a), the optimization method in Section 3.3 determines $(P^* = 5, Q^* = 4, R^* = 5)$ as the optimal parameters. Our cluster used in the experiments has a relatively low network bandwidth of 1 Gbps and a relatively high computational bandwidth of 546 GFLOPS. Thus, $Comm()$ in Eq.(4) has a major impact on determining the optimal parameters. In Figures 13(b)-(c), the optimal parameters actually achieve the minimum amount of transferred data and the minimum elapsed time. Figure 13(d) shows the elapsed times of finding the optimal parameters of the exhaustive and pruning methods. As the size of search space of $I \times J \times K$ increases exponentially, the elapsed time of the exhaustive method also increases exponentially. In contrast, the elapsed time of the proposed pruning method is very small and stable.



(a) Cost while varying (P,Q,R) (b) Transferred data while varying (P, Q, R)

(c) Elapsed time while varying (P, Q, R) (d) Pruning search space

Figure 13: Optimization of (P, Q, R) parameters.

6.4 Comparison of Fusion Plans

We compare the performance of the fusion plans of SystemDS [6], MatFast [37], DistME [18], and FuseME for the GNMF query in Eq.(6). We use real datasets in Table 2. As described in Eq.(6), GNMF factorizes a given rating matrix X (i.e., $users \times items$) into two factor matrices V (i.e., $users \times factor$) and U (i.e., $factor \times items$) such that $X \simeq VU$. Each element of the rating matrix X, i.e., $x_{i,j}$, means the rating of the i-th user for the j-th item. If the i-th user has never used the j-th item, $x_{i,j}$ in X is zero. After executing GNMF, we can find the predicted rating matrix VU, which may have a non-zero value $(vu)_{i,j}$. The recommendation system can recommend the j-th item to the i-th user if $(vu)_{i,j}$ is a high rating value.

Figures 14(a)-(c) shows the accumulated elapsed times for MovieLens, Netflix and YahooMusic, when the factor dimension $k = 200$. In Figure 14(a), FuseME outperforms MatFast, SystemDS, and DistME by 7.37×, 2.93×, and 2.18×, respectively. Although DistME cannot use operator fusion, it outperforms both SystemDS and MatFast. It shows that cuboid-based partitioning may be more important than operator fusion for performance. FuseME shows the best performance since it not only uses cuboid-based fused operator (CFO) but also fuses many operators into a single fused operator. It can fuse much more operators than SystemDS and MatFast, as shown in Figure 10. SystemDS and MatFast fuses only two element-wise operators, $*$ and $/$, for the same query. Compared to DistME, FuseME computes multiple matrix multiplications in a fused operator using a single shuffle, while DistME has to compute each of them using a shuffle. In Figure 14(b), FuseME outperforms MatFast, SystemDS, and DistME by 8.68×, 3.62×, and 2.45×, respectively.

Figures 14(e)-(g) shows the results when the factor dimension $k = 1000$, which are similar to Figures 14(a)-(c). In Figure 14(g), MatFast fails due to O.O.M. The performance gap gets larger as the factor dimension k increases. In Figure 14(g), FuseME outperforms SystemDS and DistME by 6.48× and 2.69×, respectively, while it outperforms SystemDS and DistME by 3.12× and 1.96×, respectively in Figure 14(c).

Figures 14(d) and (h) show the amount of the data shuffled at each iteration. FuseME significantly reduces the amount of data compared with MatFast, SystemDS, and DistME due to its CFO and fusion plan. For YahooMusic when $k = 200$, it reduces communication overhead of MatFast, SystemDS, and DistME by 59.8×, 23.9×, and 7.9×, respectively.

We note that the performance improvement of FuseME is bigger for the simple query in Section 6.2 than for the complex query (i.e., GNMF) in Section 6.4. The simple query is relatively communication intensive since it includes only a single matrix multiplication. The complex query is relatively computation intensive since it includes four matrix multiplications. The communication cost is largely dependent on (P, Q, R) as in Eq.(4). In contrast, the computation cost is less dependent on (P, Q, R) as in Eq.(5). Thus, our method tends to be more effective for relatively communication-intensive queries.

6.5 Comparison of Deep Learning Workload

We compare the performance of the fusion plans of SystemDS [6], TensorFlow [1], and FuseME for the AutoEncoder query [4]. We measure the elapsed time of one epoch. We use dense synthetic datasets in [8] for input matrix data. In detail, the input matrix is *the number of inputs* × *features*, and we vary both *the number of inputs* and *features* from $n = 1\,\text{K}$ to $n = 100\,\text{K}$. In general, deep learning queries including AutoEncoder is processed in a *batch*. We follow the architecture of AutoEncoder[5] used in SystemDS, where the encoder (or the decoder) consists of two fully connected hidden layers. Thus, the encoder has two weight matrices, $W1$ ($h1\times$ *features*) and $W2$ ($h2 \times h1$), where $h1$ and $h2$ are the numbers of hidden neurons of the first and the second layers, respectively. Likewise, the decoder has two weight matrices, $W3$ ($h1 \times h2$) and $W4$ (*features* $\times\, h1$).

Figure 15(a) shows the elapsed times for varying the size of input matrix. For $n = 10\,\text{K}$, FuseME outperforms SystemDS and TensorFlow by 6.05× and 3.32×, respectively. In Figure 15(b), by using the smaller *batch*, the elapsed times of all systems increase since the number of steps of updating gradients for one epoch also increases. In Figure 15(c), as *batch* increases, SystemDS fails due to O.O.M. In Figure 15(d), FuseME outperforms TensorFlow by 3.32× for $h1 = 500$ and $h2 = 2$ and by by 8.77× for $h1 = 5000$ and $h2 = 20$.

7 RELATED WORK

There have been proposed a number of distributed matrix computation systems. They can be classified into two categories: the systems based on MapReduce and the ones not based on MapReduce. The former systems include SystemML [7, 14], DMac [36], DistME [18], Cumulon [23], MatFast [37], and SystemDS [6]. SystemML [7, 14] translates new algorithms written using an R-like high-level declarative interface into a DAG of Spark jobs and rewrites the DAG to replace patterns with hand-coded distributed fused operators.

[5]https://github.com/apache/systemds/scripts/builtin/autoencoder_2layer.dml

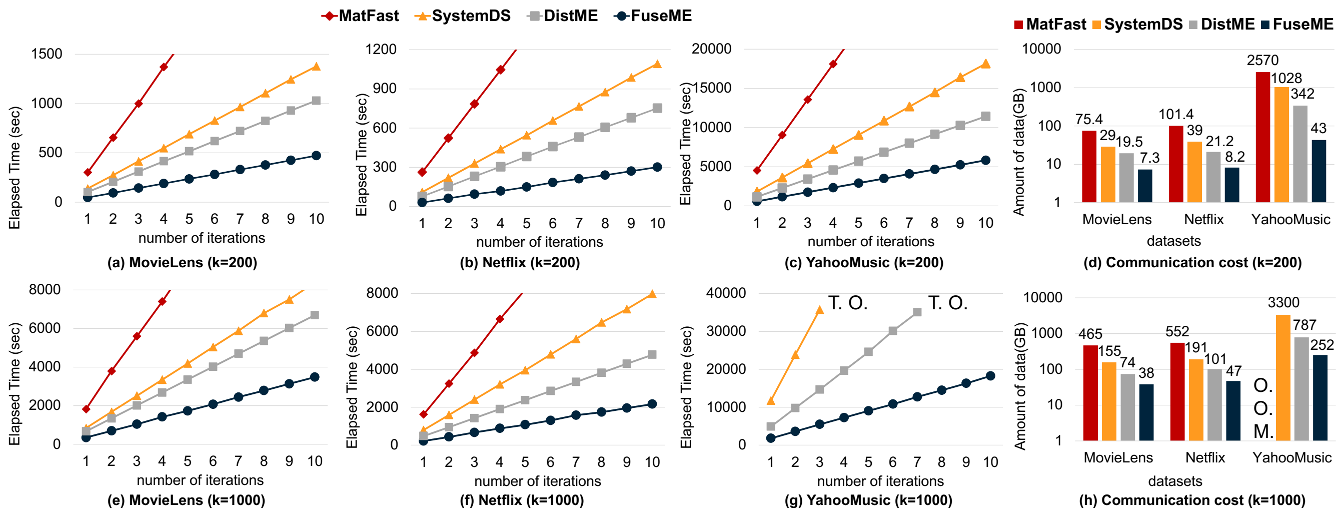


Figure 14: Performance comparison of GNMF.

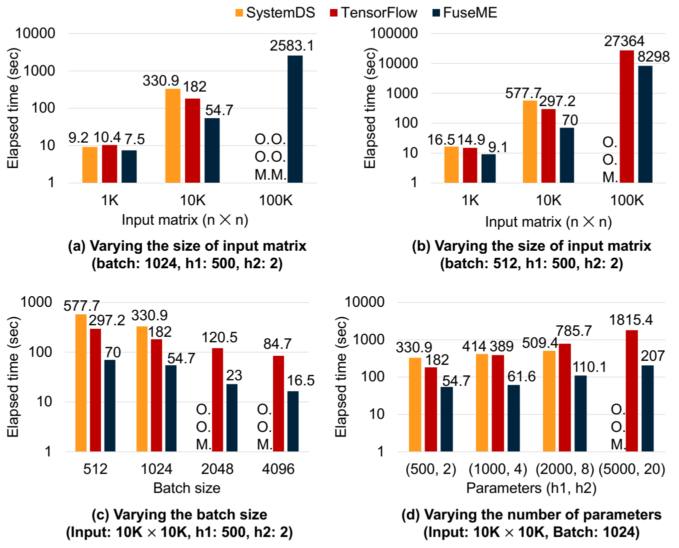


Figure 15: Performance comparison of AutoEncoder.

DMac [36] exploits matrix dependencies for a complex matrix query like GNMF to reduce the overall communication overhead. DistME [18] supports the CuboidMM for large-scale distributed matrix multiplication to achieve the lowest communication cost for a given memory limit per task. However, both DMac and DistME do not support operator fusion. MatFast [37] stores the output matrix of an operator using the partitioning scheme (e.g., row, column) so as to reduce the communication cost for the next operator in the query plan. Cumulon [23] uses a query optimization technique to estimate monetary cost under the time constraints for matrix-based algorithms and supports generic masked fused operators to exploit sparsity. SystemDS [6] provides a template-based fusion plan generator for automatic operator fusion instead of hard-coded fused operators. It however has a relatively high communication cost since it uses either BFO or RFO as a distributed fused operator and tends to avoid generating a fusion plan containing large-scale matrix multiplication.

The latter systems include TensorFlow XLA [1, 15], Julia [5, 25], MATLAB [29], and TC [35]. These systems provide practical applications using operator fusion with code generation, but those rely on the fusion plans generated by manual declaration or heuristics. Recent works [11, 13, 34, 40] consider operator fusion running on heterogeneous hardware such as GPU and FPGA.

8 CONCLUSIONS

In this paper, we have proposed a distributed fused operator called CFO that performs the optimal cuboid partitioning elastically for memory limit of a task, sizes of input matrices, computational power, and communication speed. We also have proposed a novel fusion plan generator called CFG, which can find a large fusion plan that contains even large-scale matrix multiplications. We implement FuseME integrating our CFO and CFG seamlessly on top of Apache Spark. Through extensive experiments, we have demonstrated that our CFO improves the elapsed time by up to 238× and reduces the communication cost by up to 64× compared to the existing methods. We have also shown that FuseME outperforms the state-of-the-art systems including SystemDS by orders of magnitude. As future work, we will extend our FuseME to exploit GPU acceleration and achieve a better load balancing by considering differences in sparsities of cuboids, which may further improve the performance. We will also try to re-implement FuseME based on Message Passing Interface (MPI), which may be able to reduce the number of replications of the same cuboid and so 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 (No. 2017R1E1A1A0107763 0), and Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2019-0-01267, GPU-based Ultrafast Multi-type Graph Database Engine SW).

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *OSDI*. 265–283.

[2] 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.

[3] A. Ashari, S. Tatikonda, M. Boehm, B. Reinwald, K. Campbell, J. Keenleyside, and P. Sadayappan. 2015. On optimizing machine learning workloads via kernel fusion. *SIGPLAN* 50, 8 (2015), 173–182.

[4] P. Baldi. 2012. Autoencoders, unsupervised learning, and deep architectures. In *ICML workshop on unsupervised and transfer learning*. JMLR Workshop and Conference Proceedings, 37–49.

[5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98.

[6] M. Boehm, I. Antonov, S. Baunsgaard, M. Dokter, R. Ginthör, K. Innerebner, F. Klezin, S. Lindstaedt, A. Phani, B. Rath, et al. 2019. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. *arXiv preprint arXiv:1909.02976* (2019).

[7] 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.

[8] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *VLDB* 11, 12 (2018), 1755–1768.

[9] A. Bose, V. Kalantzis, E.-M. Kontopoulou, M. Elkady, P. Paschou, and P. Drineas. 2019. TeraPCA: a fast and scalable software package to study genetic variation in tera-scale genotypes. *Bioinformatics* 35, 19 (2019), 3679–3683.

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

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799* 11 (2018), 20.

[12] T. Elgamal, S. Luo, M. Boehm, A. V. Evfimievski, S. Tatikonda, B. Reinwald, and P. Sen. 2017. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning.

[13] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. 2018. Pipelined query processing in coprocessor environments. In *SIGMOD*. 1603–1618.

[14] 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.

[15] Google. 2017. *TensorFlow XLA (Accelerated Linear Algebra)*. Retrieved August 2, 2020 from tensorflow.org/performance/xla

[16] 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.

[17] 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.

[18] D. Han, Y.-M. Nam, J. Lee, K. Park, H. Kim, and M.-S. Kim. 2019. DistME: A Fast and Elastic Distributed Matrix Computation Engine using GPUs. In *SIGMOD*. 759–774.

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

[20] X. He, H. Zhang, M.-Y. Kan, and T.-S. Chua. 2016. Fast matrix factorization for online recommendation with implicit feedback. In *SIGIR*. 549–558.

[21] Y. Hu, Y. Koren, and C. Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *ICDM*. Ieee, 263–272.

[22] B. Huang, S. Babu, and J. Yang. 2013. Cumulon: optimizing statistical data analysis in the cloud. In *SIGMOD*. 1–12.

[23] Botong Huang, Shivnath Babu, and Jun Yang. 2013. Cumulon: Optimizing statistical data analysis in the cloud. In *SIGMOD*. 1–12.

[24] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. 2015. Resource elasticity for large-scale machine learning. In *SIGMOD*. 137–152.

[25] S. G. Johnson. 2017. *More Dots: Syntactic Loop Fusion in Julia*. Retrieved August 2, 2020 from julialang.org/blog/2017/01/moredots

[26] M. Kabiljo and A. Ilic. 2015. *Recommending items to more than a billion people*. Retrieved August 2, 2020 from https://code.fb.com/core-data/recommending-items-to-more-than-a-billion-people

[27] D. D. Lee and H. S. Seung. 1999. Learning the parts of objects by non-negative matrix factorization. *Nature* 401, 6755 (1999), 788–791.

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

[29] MathWorks. 2019. *GPU Coder: Generate CUDA code for NVIDIA GPUs*. Retrieved June 2, 2021 from mathworks.com/products/gpu-coder

[30] 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.

[31] R. Pan, Y. Zhou, B. Cao, N. N. Liu, R. Lukose, M. Scholz, and Q. Yang. 2008. One-class collaborative filtering. In *ICDM*. IEEE, 502–511.

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

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

[34] Arvind K Sujeeth, HyoukJoong Lee, Kevin J Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: an implicitly parallel domain-specific language for machine learning. In *ICML*.

[35] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

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

[37] 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.

[38] 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.

[39] 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.

[40] K. Zhang, J. Hu, B. He, and B. Hua. 2017. DIDO: dynamic pipelines for in-memory key-value stores on coupled CPU-GPU architectures. In *ICDE*. IEEE, 671–682.

[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.