Dense **o**r **S**parse : Elastic SPMM Implementation for Optimal Big-Data Processing

Unho Choi, Kyungyong Lee, *Member, IEEE*

Abstract—Many real-world graph datasets can be represented using a sparse matrix format, and they are widely used for various big-data applications. The multiplication of two sparse matrices (SPMM) is a major kernel for various machine learning algorithms when using a sparsely expressed dataset. Apache Spark, a general-purpose big-data processing engine, includes the SPMM operation in its linear algebra package. The default Spark SPMM implementation, however, always converts a right sparse matrix to a dense format before performing multiplication, which can result in significant performance overhead for diverse SPMM scenarios. To address a limitation of the current Spark implementation, we describe an SPMM implementation that keeps the right matrix in a Compressed Sparse Column (CSC) format and propose an SPMM task latency prediction model based on a Deep Neural Network (DNN) architecture. Using the SPMM latency prediction model, we implement an elastic SPMM implementation recommendation service, which we name DoS (**D**ense **o**r **S**parse). The proposed DoS recommends an optimal SPMM implementation method of either transforming a right matrix to a dense format or keeping it as a sparse format during the multiplication. Through evaluation of the proposed system using a real-world graph reveals that the proposed service can improve the SPMM latency of default Spark implementation by 2.2 times while shortening the overall execution time.

Index Terms—Sparse matrix multiplication, Spark optimization, Optimal SPMM recommendation

1 Introduction

Analysis of large-scale datasets provides meaningful insights by uncovering hidden information from big data. In the data analysis, various machine learning algorithms are used for training a model on general big-data processing platforms, such as Apache Spark [1]. Though there exist various machine learning algorithms for different purposes, the multiplication of two sparse matrices, SPMM, is widely used as a core kernel operation [2], [3], [4], [5], [6]. To help efficiently build a data mining algorithm using big data, the linear algebra distributed matrix package [7] in Apache Spark provides APIs for SPMM implementation.

The native support of SPMM in Apache Spark greatly helps to implement various machine learning algorithms, but it suffers from poor performance due to its relatively simple implementation heuristic. The default implementation of Spark SPMM always transforms a right sparse matrix into a dense format before multiplication while keeping a left matrix in a sparse CSC format [8]. The advantage of statically transforming a right sparse matrix to a dense format before multiplication is its ease of implementation. Furthermore, when the density of the right matrix is high, which means that the right matrix has a large Number of Non-Zero (NNZ) elements and a low sparsity, the transformation to a dense format can improve SPMM execution performance because the dense formation allows for contiguous memory access during the computation. However, if the density of a right matrix is low, the overhead from the transformation can overwhelm the multiplication overhead. The performance variance is well represented in Figure 3.

- *Unho Choi and Kyungyong Lee are with the Department of Computer Science, Kookmin University, 77, Jeongneung-ro, Seongbuk-gu, Seoul, Republic of Korea. E-mail: officialunho@gmail.com, leeky@kookmin.ac.kr (Corresponding author: Kyungyong Lee)*

In order to provide an optimal SPMM implementation in a general purpose big-data processing platform, Apache Spark, we propose DoS, which stands for **D**ense **o**r **S**parse. In DoS, we first examine the performance variation of two SPMM implementations, transforming the right matrix to a dense format or keeping the right matrix in a sparse CSC format, with varying right matrix densities. We discovered cases where keeping a right matrix in a sparse format results in the better performance than transforming a right matrix to a dense format when conducting SPMM tasks. To be specific, as the sparsity of right matrix becomes higher, keeping a right matrix in a sparse CSC format is more advantageous. To take the performance advantage, we describe an implementation of SPMM while keeping a right matrix in a sparse CSC format. DoS proposes an SPMM execution latency prediction model based on two different implementation methods to recommend a better-performing implementation method. To build a model to predict SPMM execution latency, DoS first generates general and practical SPMM scenarios synthetically by referencing real-world graph datasets by applying Design-of-Experiments (DoE) algorithms [9]. DoS uses a DNN architecture to model non-linear and complex interactions among multiple input features to infer execution latency using the generated training dataset. DoS recommends an optimal implementation of SPMM based on the predicted latency, taking into account the dimension and density of the input matrices.

Through evaluation of DoS under practical experiment scenarios proves the validity of the proposed prediction model architectures and features. The evaluation using a real-world graph dataset reveals that DoS has a superb prediction accuracy of 93%. Using DoS's recommended SPMM implementation can reduce the SPMM execution time of the default Spark implementation, which always converts a right matrix to a dense format, by 2.2 times with negligible extra latency for prediction. The proposed DoS recommendation algorithm and implementation are publicly available, and the service is deployed, adopting a serverless architecture [10].

The major contributions of this paper are as follows

- uncovering inefficiencies of the default SPMM implementation of Apache Spark
- characterization of diverse SPMM tasks and proposing features to predict SPMM task latency
- proposing a synthetic training dataset generation heuristic using a real-world graph dataset
- building SPMM tasks latency prediction model using a DNN architecture
- implementing the optimal SPMM implementation recommendation as a service to enhance training latency of diverse machine learning jobs

2 Big-Data Analysis Using SPMM

SPMM performs multiplication of two sparse matrices, which we denote as L (left) and R (right), whose result is C, i.e., $L \times R = C$. To express the row and column index of a matrix, we use the notation of i and j as a subscript, respectively. For example, the value of the i-th row and j-th column of a left matrix is expressed as L_{ij}. A sparse matrix is denoted by the number of rows, columns, and NNZs, which are denoted as NR, NC, and NNZ, respectively. A subscript is used to specify the values of a specific matrix. For example, NR_L means the number of rows for a left matrix, and NNZ_R means the number of NNZs of a right matrix. The dimension of a matrix multiplication task is $(NR_L, NC_L) \times (NR_R, NC_R) = (NR_L, NC_R)$, where NC_L and NR_R should be the same.

The SPMM is a major computation kernel for various machine learning algorithms. The Multiple-Source Shortest Path (MSSP) algorithm [5], [4] finds shortest paths from multiple-source nodes to arbitrary destination nodes, which are expressed using a graph data structure. Because of the nature of sporadic connections among nodes in a real-world graph dataset [11], a sparse matrix for data representation is appropriate in the computation process. The sparse matrix representation of an input dataset graph is denoted as L, the left matrix in SPMM. With NR_L number of nodes, a square matrix of size $NR_L \times NR_L$ is built, where a node i, $0 \leq i \leq NR_L - 1$, is assigned a row and column index of i. An non-zero value of L_{ij} implies connection from a node i to j. To solve the MSSP problem of an input graph expressed using a sparse matrix, L, using SPMM, we define a right matrix, R, whose number of columns equals the number of source nodes to find destinations. In the first iteration, each source node is assigned a unique column to determine the number of hops for various destinations, and the value in the row index corresponding to the source node id is set to 1. The multiplication of L and R yields a matrix of size $NR_L \times NC_R$, and NNZ elements in each column of a result matrix represent the reachable nodes in the iteration from a source node.

As another example of using matrix multiplication for machine learning jobs, the dense matrix multiplication is a major computation kernel for various DNN implementations [12]. DeepCompression [13] was proposed using pruning, quantization, and coding to reduce the model size so that it could run on devices with limited capacity to mitigate model management and computation overheads. Weights in a DNN model become sparse owing to compression heuristics, and the dense matrix multiplication kernel is replaced with the SPMM. Mofrad et al. [6] proposed an optimization algorithm to conduct SPMM to train a sparse DNN model. Many algorithms are proposed to expedite Sparse DNN inference tasks [14], [15], [16]. The parallel minimum spanning forest computation algorithm requires sparse matrix multiplications [3].

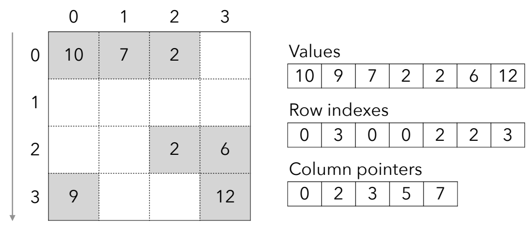


Fig. 1: An example of a sparse matrix in a CSC format

Apache Spark [1] is a general-purpose big-data processing engine that heavily relies on map and reduce primitives [17] to simplify complex distributed processing and programming. Compared to its predecessor MapReduce engine, Hadoop [18], Spark provides a way to utilize memory when necessary and outperforms Hadoop. In addition to the core Map and Reduce APIs, Spark provides implementations of various machine learning algorithms through Spark MLlib [19]. In the MLlib, distributed matrix computation APIs are provided [7]. In the implementation, users can decide how a matrix is stored in a distributed environment, block- or row-partitioned [20]. Users can also choose whether a matrix is stored in a distributed sparse or dense format. Users can save a sparse matrix in either coordinate (COO) or CSC format [8]. The CSC format represents a sparse matrix using three separate arrays, which store values, row indexes, and column pointers. Figure 1 shows an example of sparse matrix and its CSC format representation.

The figure shows a 4×4 matrix whose NNZ is 7. The *values* list is filled with non-zero matrix values in the column-major order. The *row indexes* list represents a row index of each value in the *values* list. The values in the *row indexes* and *values* list match by the index number. The *column pointers* list represents index numbers in the *values* and *row indexes* list in each column sequentially. In a zero-based index, a value in the $ColumnPointers[j]$ is the starting index and a value in the $ColumnPointers[j+1]$ is the ending index of column j, which is referenced in the *values* and *row indexes* list. We can make an inference how many entries exist in an arbitrary column j by $ColumnPointers[j+1] - ColumnPointers[j]$.

The Spark MLlib distributed matrix library stores a sparse matrix in a CSC format internally [20]. The library supports SPMMs using the CSC format sparse matrix. However, before performing the multiplication, the right sparse matrix must be transformed to a dense format. Algorithm 1

Algorithm 1 SPMM($C = L \times R$) implementation of Spark

```
Input: $L_{vals}, L_{row\_idxs}, L_{col\_ptrs}, R_{vals}$
Output: $C_{vals}$
 1: for $R_{col\_idx} = 0$ to $NC_R$ do
 2:     $rStart = R_{col\_idx} \times NC_L$
 3:     $cStart = R_{col\_idx} \times NR_L$
 4:     for $L_{col\_idx} = 0$ to $NC_L$ do
 5:         $idxStart = L_{col\_ptrs}[L_{col\_idx}]$
 6:         $idxEnd = L_{col\_ptrs}[L_{col\_idx} + 1]$
 7:         $curR = R_{vals}[rStart + L_{col\_idx}]$
 8:         for $lValIdx = idxStart$ to $idxEnd$ do
 9:             $cValIdx = cStart + L_{row\_idxs}[lValIdx]$
10:             $C_{vals}[cValIdx] \mathrel{+}= L_{vals}[lValIdx] \times curR$
11:         end for
12:     end for
13: end for
14: return $C_{vals}$
```

explains the SPMM implementation of Apache Spark.

The input to the function is the left sparse matrix, L, expressed in a CSC format with values, row indexes, and column pointers, which are notated as L_{vals}, L_{row_idxs}, and L_{col_ptrs}, respectively. Meanwhile, the right sparse matrix, R, is transformed to a dense one-dimensional vector format filled with values only, R_{vals}. The result from an SPMM task is expressed in a dense format using a one-dimensional vector and noted as C_{vals}. It iterates over the right matrix-vector, R_{vals}, per column (Line 1) and calculates the corresponding index of the right and result matrices in each column (Lines 2 and 3). It looks for the left matrix's column pointers list, L_{col_ptrs}, to determine the start and end column indexes (Lines 5 and 6). It stores the right matrix value to be multiplied in a variable using the current column index of the left matrix (Line 7). Iterating through the entries in a column of the left matrix, it calculates the location where the multiplication output is stored (Line 9) and multiplies the values of the left and right matrices to add them in the corresponding location in the result matrix (Line 10).

Figure 2 explains the default Apache Spark SPMM implementation presented in Algorithm 1 graphically. The left matrix is saved in a CSC format, while the right and result matrices are saved in a dense format with a one-dimensional vector. By finding a matching entry in the left matrix, the right and result matrices are accessed sequentially in column-major order.

The transformation of the right matrix into a dense format before multiplication can result in performance degradation. First, in general, the transformation of a sparse matrix to a dense format requires larger memory. Storing a right matrix of size $NR_R \times NC_R$ matrix in a dense format statically requires $NR_R \times NC_R \times 8$ bytes in Apache Spark assuming the *double* value representation using Scala language. Meanwhile, the memory size required to store a right sparse matrix in a CSC format is largely dependent on the NNZ_R. Storing NNZ_R values in the *double* type takes $NNZ_R \times 8$ bytes in Scala language, while storing the row indexes list requires $NNZ_R \times 4$ bytes. Storing the column pointers requires $(NC_R + 1) \times 4$ bytes [21]. Considering that $NR_R \times NC_R \gg NNZ_R$ generally stands for the most

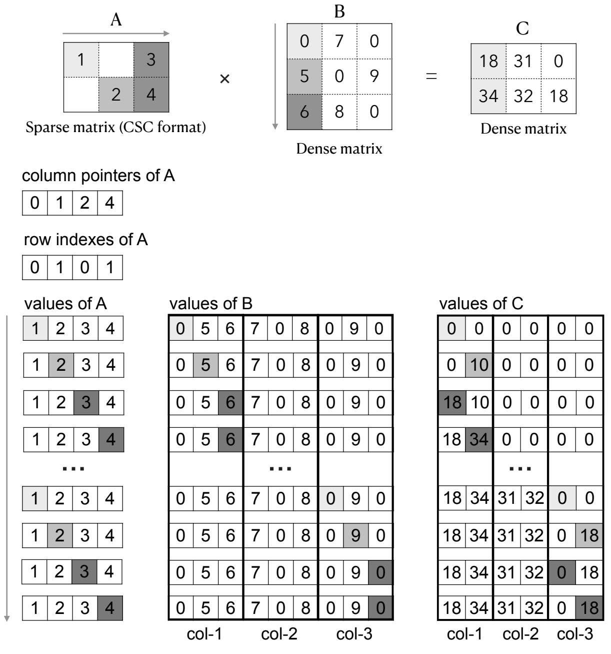
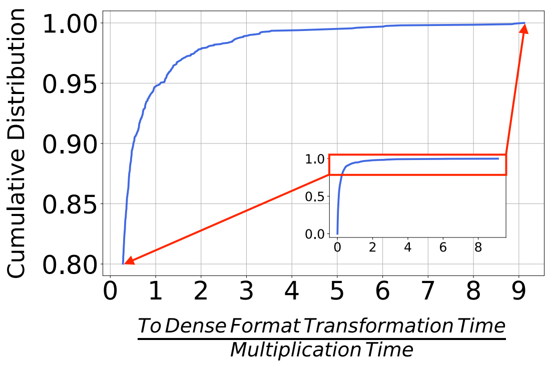


Fig. 2: SPMM implementation of Apache Spark. The right sparse matrix is transformed to a dense format before the multiplication.

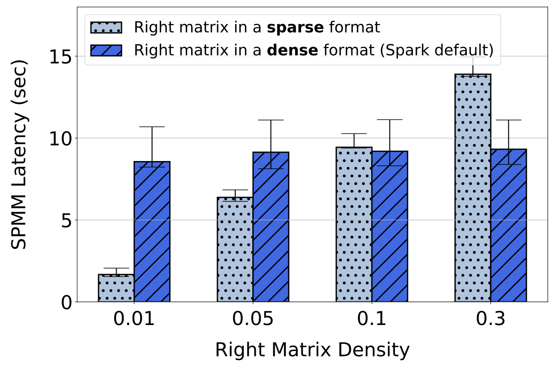
sparse dataset, storing in a dense format might consume significantly more memory. In addition to the large memory size requirement, the time to convert to a dense format may be greater than the actual matrix multiplication time.

Despite the aforementioned shortcomings of the current Spark SPMM implementation, transforming the right matrix to a dense format can be beneficial from the perspective of simplicity in the programming. In addition, when the density of the right matrix, which can be calculated as $\frac{NNZ_R}{NR_R \times NC_R}$, increases, accessing contiguous memory region in a *values* vector can enhance the total computation time. To characterize the performance variation of SPMM in the Apache Spark MLlib, we conducted SPMM experiments with varying right matrix densities and dimensions.

Figure 3a shows the Cumulative Distribution Function (CDF) of the ratio of the sparse to dense format transformation latency to the actual SPMM computation, which is shown in the horizontal axis. The larger horizontal axis value implies that format transformation overhead outweight the actual multiplication time. Using synthetically generated $1,300$ cases of various SPMM scenarios whose detail is presented in Section 3.2.1, we conducted SPMM tasks using Apache Spark MLlib implementation. We measured the latency to transform a CSC matrix to a dense format and the latency to complete a multiplication task presented in Algorithm 1 during the experiments. In the main part of the figure, we show the CDF of 20% cases of SPMM tasks with the highest transformation overhead ratio to better represent the transformation overhead. To find the CDF of the entire dataset, please refer to the sub-figure in the bottom right corner. As shown in the figure, 20% of SPMM tasks consumed at least 27% of time for dense format transformation. In the 10% of cases, the transformation took at least half



(a) Overhead of converting a sparse matrix to dense



(b) SPMM latency in a sparse and dense format

Fig. 3: Overhead of SPMM execution using a dense format

the latency to complete the multiplication, and about 5% of cases took more time for transformation than the actual multiplication. In the worst case, the transformation took nine times more than the actual computation time.

To present the performance of different SPMM implementations, we compare the response time of an SPMM task with the right matrix in a dense format (the default SPMM implementation provided by Apache Spark) and in a sparse CSC format (the algorithm that will be presented in Section 3.1) in Figure 3b. We varied the density of a right matrix which is shown in the horizontal axis and used a fixed sparse left matrix. The dotted bar on the left shows the response time when a right matrix is expressed in a sparse CSC format, and the bar with the upper-right diagonal pattern shows the latency when the right matrix is expressed in dense format. We can see that when the right matrix density is less than 0.1, the default Apache Spark SPMM implementation performs worse. The default Spark implementation is four times slower when the right matrix density is 0.01 than the implementation that keeps the right matrix in a sparse CSC format. However, as the density increases, we can see that converting the right matrix to a dense format improves performance.

In summary, as presented in Figure 3, the default Apache Spark SPMM implementation, which transforms a right sparse matrix to a dense format, imposes significant overhead during the storage format transformation. Despite the additional overhead, the transformation can result in better SPMM execution latency than performing a task in a sparse CSC format in some cases. To provide an optimal SPMM execution environment, it is critical to identify a better implementation method for different input dimensions. To achieve the goal, we propose an algorithm to detect a better SPMM method and implement it to the default Apache Spark MLlib to enable dynamic selection of SPMM implementation based on the input workload characteristics.

Algorithm 2 SPMM ($C = L \times R$) in Sparse Format

```
Input: $L_{vals}, L_{row\_idxs}, L_{col\_ptrs}, R_{vals}, R_{row\_idxs}, R_{col\_ptrs}$
Output: $C_{vals}, C_{row\_idxs}, C_{col\_ptrs}$
 1: $NNZ_C = CountNNZ(L, R)$
 2: $(C_{vals}, C_{row\_idxs}, C_{col\_ptrs}) = AllocateMem(NNZ_C)$
 3: for $R_{ci} = 0$ to $NC_R$ do
 4:     $resultSortedDict = \{\}$
 5:     for $R_{off} = R_{col\_ptrs}[R_{ci}]$ to $R_{col\_ptrs}[R_{ci}+1]$ do
 6:         $R_{ri} = R_{row\_idxs}[R_{off}]$
 7:         $R_{val} = R_{vals}[R_{off}]$
 8:         for $L_{off} = L_{col\_ptrs}[R_{ri}]$ to $L_{col\_ptrs}[R_{ri}+1]$ do
 9:             $L_{ri} = L_{row\_idxs}[L_{off}]$
10:             $L_{val} = L_{vals}[L_{off}]$
11:             $resultSortedDict[L_{ri}] += R_{val} \times L_{val}$
12:         end for
13:     end for
14:     $nnz = resultSortedDict.size()$
15:     $C_{col\_ptrs}[R_{ci}+1] = C_{col\_ptrs}[R_{ci}] + nnz$
16:     $C_{row\_idxs}.append(resultSortedDict.keys())$
17:     $C_{vals}.append(resultSortedDict.values())$
18: end for
19: return $C_{vals}, C_{row\_idxs}, C_{col\_ptrs}$
```

3 System Architecture of DoS

To improve SPMM task performance while taking advantage of both when a right matrix is expressed in a dense and sparse format, we first describe a memory-efficient SPMM implementation while keeping a right matrix in a sparse CSC format. With the sparse version of SPMM implementation, it is important to judge which type of right matrix storage format, either dense or sparse, would result in better performance. To achieve the goal, we propose a recommendation model to guide which implementation would result in better performance considering the SPMM task characteristics.

3.1 SPMM with Right Matrix in a Sparse Format

Transforming a right matrix to a dense format during the SPMM execution might result in significant additional overhead. To avoid such burden, an SPMM task can be completed while keeping the right matrix in a sparse CSC format. Algorithm 2 explains the overall process. The algorithm takes left and right matrices in a CSC format as input with its values, row indexes, and column pointers as a list. The output of the matrix is also returned as a CSC format. In Line 1, given left and right matrices, it counts the NNZ in a result matrix, which becomes NNZ_C. The *CountNNZ* method iterates the left and right matrices to find NNZ elements. The iteration process is similar to what is presented between Lines 3 and 13. In

Line 2, using the computed NNZ_C value, memory for variables to store output matrices is allocated. In the SPMM execution, we tried two memory allocation approaches: the static allocation by using the pre-computed NNZ_C and the dynamic allocation as needed. Each approach has trade-offs. The NNZ calculation and static allocation require an additional step to calculate the NNZ of a result matrix. However, after the calculation, no further memory re-allocation overhead is needed. Otherwise, the heuristic of dynamic result matrix memory allocation does not require additional step of NNZ_C calculation, but it can result in prohibitive overhead due to the memory reallocation. From the thorough empirical analysis of both approaches which is shown in Figure 10, we discovered that pre-computing NNZ_C and statically allocating memory results in 7.25% lower latency on average. The qualitative analysis about the implementation reveals that the static memory allocation is much simpler than the dynamic memory re-allocation, and we decided to adopt the static memory allocation heuristic. From Line 3, it iterates each column of a right matrix. In Line 4, a sorted key-value dictionary is created to store intermediate multiplication results. In the dictionary, the key indicates the row index of the result matrix, and the value indicates the intermediate aggregated multiplication result. Please note that the column index in the result matrix is R_{ci}. From Lines 5 to 7, it iterates elements in the corresponding right matrix column. Lines 8 to 11 iterate through the left matrix in the column-major order, looking for the corresponding left matrix entry for multiplication. The temporary multiplication result is stored in the *resultSortedDict* object and accumulated using a sum operator for the matching row index of a result matrix. After completing a right matrix column, the column pointer of a result matrix is updated (Line 15), and the row indexes and values are updated by appending results (Lines 16 and 17).

Figure 4 explains the SPMM implementation while keeping a right matrix in a sparse format. The CSC format is used to express the left, right, and result matrices. It looks for corresponding entries for multiplication in the left matrix by referencing the column pointers of a right matrix. Matching entries for multiplication between left and right matrices are colored the same. Following multiplication, a result matrix entry is updated with the row and column indexes determined by the left matrix row and right matrix column index, respectively.

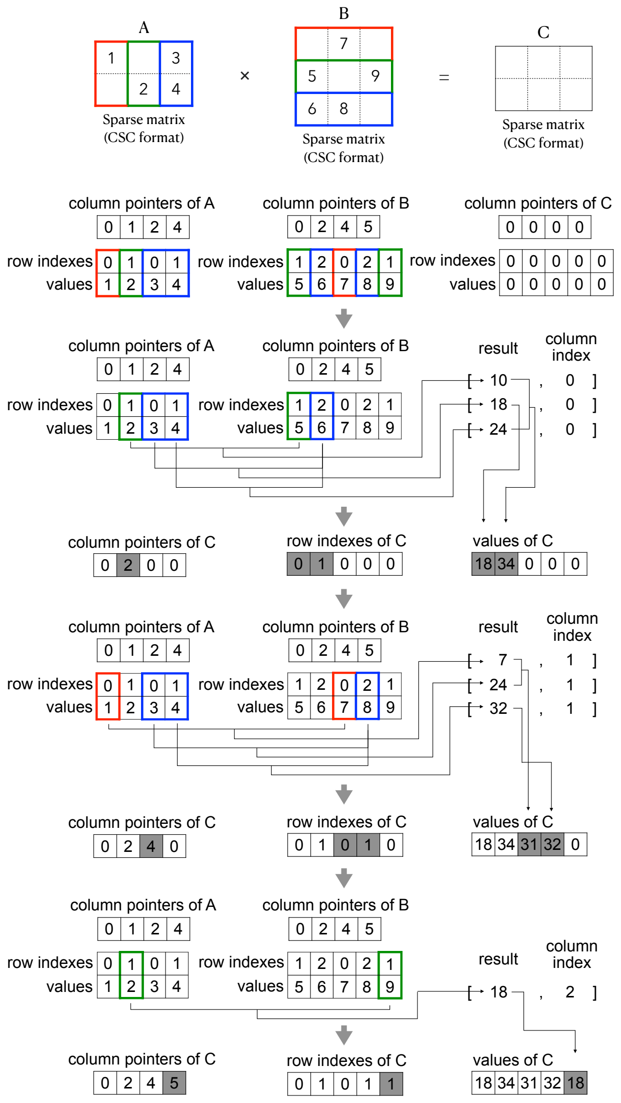


Fig. 4: Conducting SPMM while keeping two matrices in a sparse format

3.2 Recommending Optimal SPMM Implementation

In this section, we describe procedures to build SPMM latency prediction models.

3.2.1 Training Dataset Generation Using DoE

To build a model to predict SPMM latency of arbitrary size and density of matrices, it is important to generate a training dataset to represent various SPMM scenarios. There are many types of graph datasets that can be represented using a sparse matrix, such as SNAP [11], Graphalytics [22], and The University of Florida Sparse Matrix Collection [23]. Because of the large number of publicly available graph datasets and the duplicate data characteristics, performing SPMM operations for all available cases to generate training dataset is not ideal. DoS proposes a synthetic training dataset generation algorithm that uses Latin Hypercube Sampling (LHS) [24], filtering, and D-optimal [25], to build a general and high-accuracy prediction model with minimal offline experiments for train-dataset-generation.

SPMM Scenario Generation : The LHS [24] is a statistical algorithm used to design experimental scenarios by evenly distributing cases throughout the entire experimental spaces. Users can choose a probability density function to spread experiment cases when distributing them. To generate scenarios that follow a uniform distribution function for an experiment feature dimension of D and N number of cases, LHS partitions each feature dimension into N intervals of equal probability and selects a sample case independently. Finally, it shuffles a sample of each feature so that there is no correlation between the cases that were chosen. The output of the LHS algorithm ranges between 0.0 and 1.0, and users can multiply the maximum value in each experiment dimension to get final experimental scenarios.

SPMM tasks can be represented with a unique combination of NR_L, NC_L, and NC_R. Using the LHS algorithm with a uniform distribution function, we first generate a floating-point value in the range of 0.0 and 1.0 and multiply the maximum value in each dimension to the LHS-generated value to determine experimental cases. The maximum value in each dimension needs to be specified by considering the capability of an executor node in Spark that performs an SPMM task. In our prototype implementation and evaluation, we set the maximum dimension value as $150,000$, which is a reasonably large size to represent a sparse matrix presented in a block-partitioned way [20]. We generate $2,500,000$ distinct cases of NR_L, NC_L, and NC_R combinations as candidate experiment scenarios.

Other than the dimensions of left and right matrices in the SPMM, the NNZ and sparsity of input matrices are also important features to describe SPMM task characteristics. We simulate practical SPMM scenarios to represent a realistic density scenario of SPMM. Many machine learning jobs that use SPMM, such as MSSP [4], [5] and PageRank [2], construct a left matrix from an input graph and keep it fixed across multiple iterations of multiplications. Meanwhile, a right matrix is updated continuously through multiple iterations where a result matrix of iteration $i-1$ becomes a right matrix of iteration i.

To generate practical scenarios of a left matrix, L, and sparsity, we calculate the density of each node from a real graph dataset. To calculate per-node density, we count the NNZ of a node and divide it by the total number of nodes. Among many density values, we record six density values, which are the average density of all nodes, most $10,000$ density nodes, most $1,000$ nodes, most 100 nodes, most 10 nodes, and the most density node. We calculate the per-node density of the *DBLP, Amazon, Youtube, Orkut, and LiveJournal* datasets provided by Stanford SNAP [11], yielding a total of 30 left matrix density values. To reflect the workload characteristics of SPMM in machine learning jobs, we decided to empirically represent the sparsity of a right matrix, R, by increasing the value with a fixed interval. We use 16 right matrix density values for training: $0.0005, 0.001, 0.005, 0.01, 0.03, 0.05, 0.07, 0.1, 0.13, 0.15, 0.17, 0.2, 0.23, 0.25, 0.27$, and 0.3. To generate various SPMM task scenarios that are represented with a distinct combination of NR_L, NC_L, NC_R, NNZ_L, and NNZ_R, we randomly allocate left matrix densities from real graphs and the synthetically generated right matrix densities to the $2,500,000$ combinations of NR_L, NC_L, and NC_R.

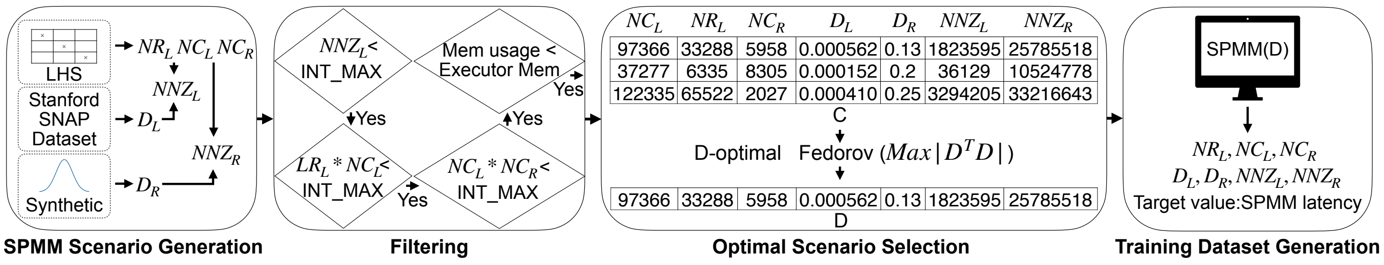
Filtering Executable SPMM Scenarios : All the generated $2,500,000$ SPMM cases cannot be executed on an executor node owing to the limited available memory size or various system limitations imposed by Apache Spark, and such inexecutable SPMM scenarios should be removed from offline experiments to avoid unnecessary cost. Apache Spark limits the NNZ of a matrix to the maximum 32-bit integer value [20]. Thus, an arbitrary matrix scenario is filtered out of a training dataset if its NNZ, the multiplication of the number of rows, columns, and density, is greater than $2,147,483,647$. The right matrix should be converted to a dense matrix in the current Spark SPMM implementation, and the multiplication result should also be stored in a dense format. Apache Spark limits the number of matrix entries to the maximum of 32-bit integer value [20]. Thus, an SPMM scenario whose $NR_R \times NC_R$ or $NR_L \times NC_R$ is larger than $2,147,483,647$ is also excluded from a experiment scenario.

Other than the Spark system limitation, a Spark executor's available memory size can decide whether an SPMM task can be completed. The estimation of real-time memory consumption of an SPMM task on Spark is challenging because of the uncertainty in the garbage collection timing of the underlying Spark engine, Java Virtual Machine (JVM) [26], and memory consumption from the Spark's core engine [27]. To avoid wasting offline experimental costs, all synthetically generated SPMM cases cannot be executed on any arbitrary Spark executor, and we must detect unexecutable SPMM cases due to memory constraints. It is a binary decision whether an SPMM task can meet a memory size constraint of an executor node, and a rough estimation of memory consumption is sufficient to select executable experiment cases given an executor memory size.

Because of the uncertainty of garbage collection timing of JVM and diverse sparse matrix characteristics, required memory size calculation based on left and right matrix dimensions might not be accurate, and we decided to build a non-linear model for memory consumption estimation using a tree-based regressor algorithm, XGBoost [28]. The dimensions of the left and right matrices, as well as the NNZ, are used as features in the required memory size estimation model. Similar to our approach, OFC [29] used a tree-based regressor model to estimate memory size consumption of cloud function runtimes and demonstrated reasonable accuracy, demonstrating the usefulness of memory consumption modeling. Using the predicted memory consumption, we filter out inexecutable cases in Spark executors that are used for the training dataset generation experiments.

Optimal SPMM Scenario Generation : From the $2,500,000$ synthetically generated SPMM scenarios using an LHS algorithm, we filter $877,290$ executable SPMM cases on an executor with a configured executor memory size of 32 GB, on which we will perform experiments. Performing offline experiments with all of the cases is prohibitively expensive and time-consuming. To build a general prediction model with a small number of training datasets, we must select representative and distinct SPMM cases from a pool of potential candidates. The D-optimal [25] algorithm can select a subset of experiment scenarios from many candidates while minimizing accuracy loss. The D-optimal algorithm represents all test case scenarios as a matrix $\mathcal{C}$, which becomes a candidate for the final experiment scenarios. Each row of the matrix $\mathcal{C}$ represents a unique experiment case, and each column represents a unique feature. In our experiment scenarios, a column represents the dimension of the left and right matrix, and the NNZ. The Fisher information matrix [30] of $\mathcal{C}$ is $\mathcal{C}^T\mathcal{C}$, which is the inverse of the covariance matrix of $\mathcal{C}$. The D-optimal algorithm selects a subset of rows from $\mathcal{C}$ and names a subset matrix as $\mathcal{D}$. The goal is to build $\mathcal{D}$ with the maximum determinant of the information matrix, which we denote as $|\mathcal{D}^T\mathcal{D}|$, with a minimal number of selected rows from a candidate matrix $\mathcal{C}$. Maximizing the determinant of an information matrix of $\mathcal{D}$ results in dispersing experiment cases as much as possible in the experiment region [9].

To build a D-optimal subset matrix, $\mathcal{D}$, out of a candidate experiment case matrix, $\mathcal{C}$, we apply the Fedorov exchange algorithm [31]. Given a randomly selected d experiment



NC_L	NR_L	NC_R	D_L	D_R	NNZ_L	NNZ_R
97366	33288	5958	0.000562	0.13	1823595	25785518
37277	6335	8305	0.000152	0.2	36129	10524778
122335	65522	2027	0.000410	0.25	3294205	33216643

97366	33288	5958	0.000562	0.13	1823595	25785518

Fig. 5: The procedure of generating training dataset to build a SPMM latency prediction model

cases from $\mathcal{C}$, the algorithm exchanges one of the rows in the $\mathcal{D}$ with a row in $\mathcal{C}$, which does not belong to $\mathcal{D}$, $(\mathcal{C} - \mathcal{D})$. Let us name the prior selection of experiment scenarios as $\mathcal{D}_{old}$ and name the experiment set after changing a row as $\mathcal{D}_{new}$. The determinant of the newly generated information matrix is expressed as follows.

$$|\mathcal{D}_{new}^T \mathcal{D}_{new}| = |\mathcal{D}_{old}^T \mathcal{D}_{old}| \times (1 + \Delta(d_i, d_j)) \qquad (1)$$

The d_i in Equation 1 means a row to be removed in the $\mathcal{D}_{old}$, and the d_j means a row to be added to the $\mathcal{D}_{new}$. The $\Delta(d_i, d_j)$ is defined as $Var(d_j) - [Var(d_i)Var(d_j) - Cov(d_i, d_j)^2] - Var(d_i)$. The basic idea of the Fedorov exchange algorithm is to calculate the delta of the determinant of information matrix for every possible pair of d_i and d_j. In every iteration, a pair with the greatest determinant increase is exchanged. This process is repeated until no pair of exchanges increases the determinant. We use Fedorov exchange in the train-dataset-generation using the R programming language's AlgDesign package [32]. We use the library to select the best $1,300$ training dataset cases from a pool of $877,290$ executable cases.

Conducting Experiments for Training Dataset Generation : Using the D-optimally selected scenarios, we conduct SPMM tasks in two different ways. One by using the default Apache Spark MLlib implementation which converts the right matrix to a dense format and another using the multiplication algorithm presented in Section 3.1 which keeps the right matrix in a sparse CSC format during the multiplication task. We measure latency to complete SPMM tasks with different implementations and use the latency as a target value in a prediction model.

Figure 5 explains the overall procedure of generating train datasets. As explained in this section, it is composed of *SPMM Scenario Generation*, *Filtering Executable Cases*, *Optimal Scenario Selection*, and *Conducting Experiments for Training Dataset Generation*. In the *SPMM Scenario Generation* step, we generate a comprehensive set of SPMM scenarios that can represent real-world applications. On an Apache Spark executor node, we use the generated scenarios to perform *Filtering* to remove inexecutable cases. The *Optimal Scenario Selection* step employs the D-optimal algorithm to cherry-pick scenarios to achieve maximum diversity with a subset of cases. The *Training Dataset Generation* step executes SPMM tasks with different implementations and records the execution latency to use as a model output using the D-optimally selected SPMM scenarios.

3.2.2 *SPMM Latency Prediction Modeling*

To build a model to predict SPMM execution latency of arbitrary scenarios, we must define features that become inputs for a prediction model. The features should represent characteristics of an SPMM task, and we propose to use $NR_L, NC_L, NC_R, NNZ_L, NNZ_R, D_L$, and D_R. D_L and D_R represent the sparse matrix density of the left and right matrices, respectively.

An SPMM latency predictor should model the non-linear interactions between the input features and the latency. To model the non-linearity, DoS employs a DNN [33] algorithm, which is commonly used to uncover complex relationships among input features to infer the target value. A DNN model architecture is typically composed of multiple layers, with the previous layer serving as an input to the next layer. Multiple nodes in an input and output layer are connected by an edge with a distinct weight value. DNN is similar to a Multi Layer Perceptron (MLP) [34] except the loss-minimization method. In a DNN model, weight values are adjusted through forward and backward propagation [35] in the direction of minimizing user-specified loss. In between layers, various activation functions [36], [37] can be inserted to normalize output values. Compared to a tree-based non-linear regressor [38], [28], a DNN model can better represent complex interactions among features, and it might not need specialized feature engineering.

When building a DNN model, various hyper-parameters, such as the number of hidden layers and nodes, network weight initialization, activation function, optimization algorithm, learning rate, and the number of epochs, should be decided by an algorithm developer. Properly setting the model's hyper-parameters during training is critical for improving prediction quality. To find optimal DNN hyper-parameters in DoS, we use a grid-search method [39], which finds an optimal parameter combination by automating the execution of trying every possible parameter combination. The grid-search method has a shortcoming of exhaustive search space, but it is guaranteed to find an optimal set of hyper-parameters.

We build an SPMM execution time prediction DNN model using a fully connected five hidden layers of 1024, 128, 64, 32, and 16. In the model, we perform hyper-parameter optimization for the activation function, weight initialization, optimization, and learning rate, which result in *Relu* [36], *normal* [40], *adagrad* [41], and 0.07, respectively. Model training runs for $1,000$ epochs, and early stopping [42] is adopted to prevent over-fitting.

DoS builds two independent models using the proposed DNN architecture: one is used to predict SPMM latency when a right matrix is represented in a dense format, which is the current Apache Spark default implementation, and the other implementation, described in Section 3.1, which keeps the right matrix in a sparse CSC format. When an arbitrary SPMM task is submitted, the proposed system predicts the SPMM execution time of two different implementations based on the input matrix expressions and selects the one with the shortest execution time.

3.2.3 Overhead From the Prediction

Despite the performance gain from the proposed system, overhead regarding the model training time, memory consumption for inference, and additional inference time might occur. To train a proposed prediction model, we used a server with two 3.3GHz CPU cores and 4GB memory (AWS *t2.medium* instance). The training process happens off-line, and it took about 127 seconds on average. Due to the nature of off-line training, it does not impact the SPMM task execution time, and users do not get latency disadvantage due to the model training.

Additional overhead during the inference step might impact end-user experiences. We measured additional memory consumed during inference, and it took about 154 MBytes while importing the model serving framework and loading the prediction model. Considering that Apache Spark executors generally equip few GBytes of memory, the additional memory overhead during inference is negligible. The additional inference time can directly influence the end-to-end SPMM task latency, and we thoroughly evaluate the latency impact and present the result in Section 5.3. In short, the additional latency incurred from the prediction is 63 milliseconds at most, and the average ratio of the additional prediction latency to the latency improvement while dynamically switching between two SPMM implementations is 8%. Based on the quantitative overhead analysis, we are confident that the proposed system is worth to be applied for real-world applications.

4 Application : SPMM Implementation Recommendation Service

To present the applicability of the proposed system, we build DoS on top of the Apache Spark MLlib library SPMM implementation. To minimize Spark source code change, we adopt the micro-service architecture [43] by implementing the proposed latency predictor as an independent service. Figure 6 shows the implemented system. The implemented system can be accessed via HTTP RESTful API [44]. We include an HTTP client library in the Apache Spark MLlib library SPMM method to invoke the service and determine the best SPMM implementation method.

To provide the prediction service, we use AWS Lambda to implement the service in a serverless way. AWS Lambda is a core component to make serverless computing feasible. Serverless computing relieves application developers of the burden of managing underlying resources required to maintain a highly available system, such as fault tolerance, scalability, and maintainability. A serverless architecture application is created by combining several fully managed cloud services in various domains, such as database, messaging, and API handling. Although fully managed cloud

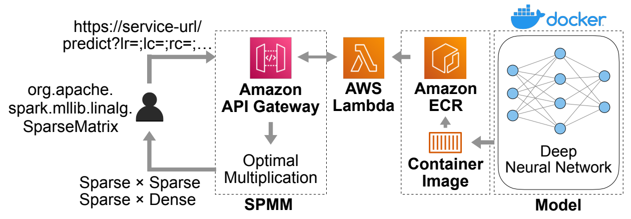


Fig. 6: The implementation of the proposed system on the SPMM code of the Apache Spark MLlib library

services reduce application developers' resource management overheads, they limit users' customized operations. AWS Lambda is a fully managed function runtime that allows to execute user-implemented source code. The service is termed as Function-as-a-Service and expands application scenarios in many fields [10], [45], [46].

In the DoS implementation, we serve the SPMM latency prediction model using AWS Lambda. Different from a server environment where a developer can customize system settings with a root privilege, AWS Lambda imposes lots of restrictions in the system configuration [47]. When serving a DNN prediction model, a large amount of local storage is required to keep the necessary libraries and model weights. However, AWS Lambda's local storage is limited to 512 MB, which is insufficient to store model weights and libraries. In our prototype implementation, we use TensorFlow [48] and SciKit-Learn [49], which consume about $2,035$ MB of local storage. There are a few ways to overcome the storage limitation, and one is by using a publicly accessible object storage service, such as Amazon S3, or by using a shared block file storage service using Network File System (NFS) protocol [50] and Amazon EFS [51], [52], which can be mounted to a Lambda function runtime. In another way, AWS supports executing a function runtime on a custom-built docker container image [53], which we adopt to implement the proposed system. To build a function runtime, we define a *Dockerfile*, which lets necessary libraries be added to a generated container image.

The image is stored in Amazon Elastic Container Registry (ECR), which is a fully managed container image storage service that can store, maintain, and distribute container images. In ECR, we specify a container image as a function runtime environment when creating a Lambda function. We use Amazon API Gateway to make the prediction service externally accessible, which provides an HTTP endpoint by passing query parameters. During the HTTP request invocation, $NR_L, NC_L, NC_R, D_L,$ and D_R, are passed as query parameters.

The SPMM latency prediction service implementation receives arguments via API Gateway, and a Lambda function predicts two latency values of an SPMM task when a right matrix is processed in a dense and sparse format. The predicted latencies are returned by API Gateway, and the SPMM module in the Spark MLlib library chooses a method with the lowest expected latency and performs SPMM accordingly. The overall procedure is shown in Figure 6, and the system is currently available as a web service[1].

1. http://dos.ddps.cloud

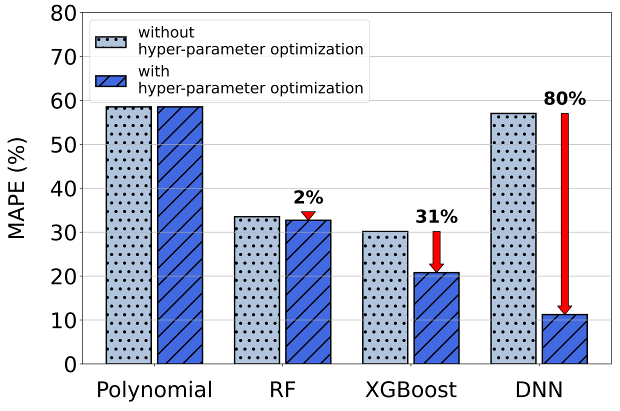
5 Evaluation

We conducted experiments thoroughly from various perspectives to evaluate the effectiveness and applicability of the proposed system, DoS. We execute SPMM tasks using Apache Spark 3.1.2, which is available via AWS Elastic Map Reduce (EMR) service. In creating an EMR cluster, we set one master with a *m5.xlarge* instance, which has 4 CPU cores and 16 GB RAM. For worker nodes, we have three instances of *m5.8xlarge* type, which has 32 CPU cores and 128 GB of memory. In building latency prediction models, we use TensorFlow 2.5.0, ScikitLearn 0.23.2, and XGBoost 1.3.3. For hyper-parameter optimization, we use the *GridSearch* method from ScikitLearn for DNN and Bayesian Optimization package for Random Forest and XGBoost. To generate diverse SPMM scenarios using the LHS algorithm, we use pyDOE 0.3.8 and the R package AlgDesign 1.2.0 to select optimal SPMM scenarios. We use Mean Absolute Percentage Error (MAPE) to assess evaluation accuracy, which is defined as the average ratio of absolute error to the true value. We also employ the Root-Mean-Square Error (RMSE). Lower values for MAPE and RMSE indicate better performance.

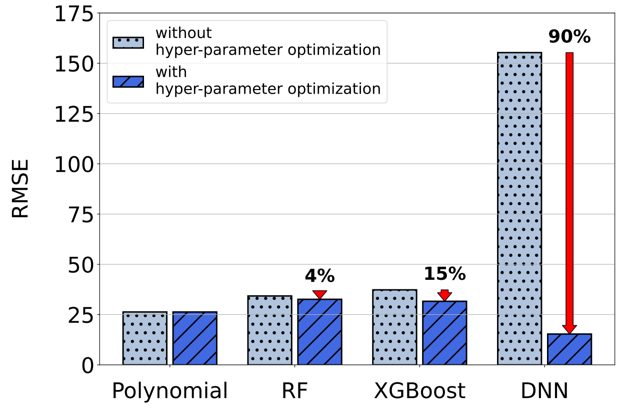
5.1 SPMM Latency Prediction Accuracy

We first evaluate the accuracy of SPMM execution latency prediction models. In the evaluation, we used an order-3 polynomial regressor [54], Random Forest (RF) [38], XGBoost [28], and DNN which we use in the proposed system, DoS. Figure 7 compares the prediction accuracy of various algorithms with and without applying hyper-parameter optimizations. The order-3 polynomial model is built by using an equation of $Y = \sum_{i=1}^{N}(\alpha_{i-3}X_i^3+\alpha_{i-2}X_i^2+\alpha_{i-1}X_i)+\alpha_0$. Y is the latency to predict, and X is the input features where the index i means each feature. In the prediction model training, it learns α to accurately predict Y from X. RF applies an ensemble of multiple decision-trees where each individual tree is built with different set of input dataset and features. It is known to lessen over-fitting which can happen in a single decision tree. XGBoost is a non-linear regression model based on trees that generates predictive models in the form of ensembles of weak predictive models, which are typically composed of decision trees. It iteratively generates a classifier based on the accuracy of the classifier generated in the previous step and ensembles those classifiers to generate a more accurate final model.

In the figure, the horizontal axis expresses different prediction model algorithms. Figure 7a shows the MAPE, and Figure 7b presents the RMSE in the vertical axis. The tree-based prediction models (RF and XGBoost) performs better than a DNN model without hyper-parameter optimization where we used the default setting for RF, XGBoost, and MLP implementation in the experiment packages. However, as we optimize hyper-parameters to improve accuracy, the DNN model's performance improves dramatically, outperforming the tree-based prediction models. After optimizing hyper-parameters, the MAPE of DNN improves from 57.02% to 11.24%. Among tree-based predictor models, XGBoost shows more improvement than RF with the hyper-parameter optimization. This phenomenon happens due to more complex process and a larger hyper-parameter search space of XGBoost. Our observation coincides with what is discovered in [55]. The polynomial regressor shows poor MAPE metric values comparing to the RMSE. Our investigation reveals that it failed to predict many SPMM cases with small latency values which can negatively impact MAPE even with small error values. Overall, the average MAPE improvement by using the hyper-parameter optimization of the DNN model is 80%, and the MAPE of the hyper-parameter-optimized DNN is 45% lower than the XGBoost algorithm, validating the modeling algorithm selection of DoS.



(a) MAPE of SPMM latency prediction models



(b) RMSE of SPMM latency prediction models

Fig. 7: MAPE and RMSE (lower is better) of latency prediction model when applying hyper-parameter optimization. The prediction accuracy of DNN becomes superior to other algorithms after applying the hyper-parameter optimization.

Next, we present the effectiveness of applying the D-optimal algorithm in Figure 8 compares various models' prediction accuracy, built with different numbers of training input datasets, which are shown in the horizontal axis. Among $1,300$ synthetically generated cases, we select a subset of cases at random and apply D-optimal. The MAPE value is shown in the vertical axis, the *Random* selection results in the dotted bar, and the *D-optimal (Proposed)* selection results in the upper-right diagonal line bar. We can see that using the D-optimal algorithm produces higher accuracy than does randomly selecting experiment cases. With a smaller number of train dataset cases, the performance gain

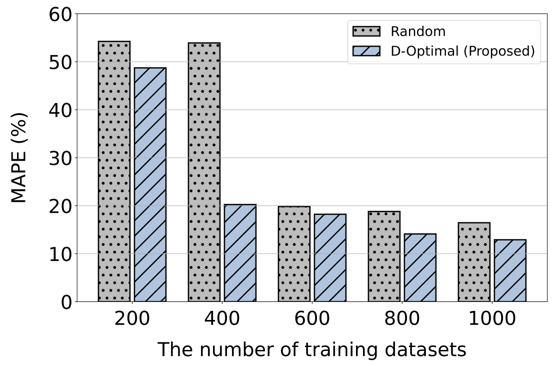


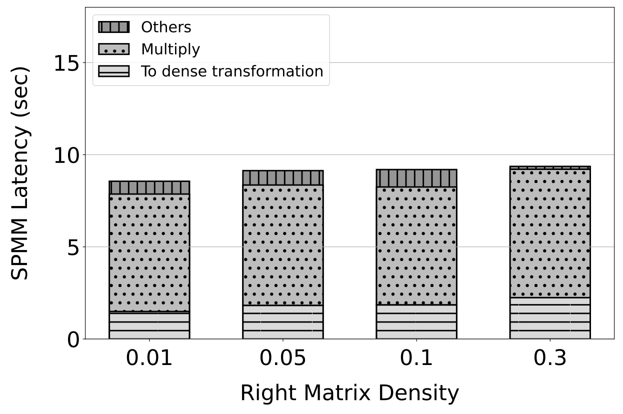
Fig. 8: The efficiency of the proposed workload generation algorithm which uses D-Optimal. Comparing to the randomly generated cases, the proposed algorithm shows 21.2% lower error rate.

is dramatic, demonstrating the effectiveness of the proposed approach. Overall, applying the D-optimal algorithm results in 21.2% better accuracy than the random selection with a same number of training dataset.

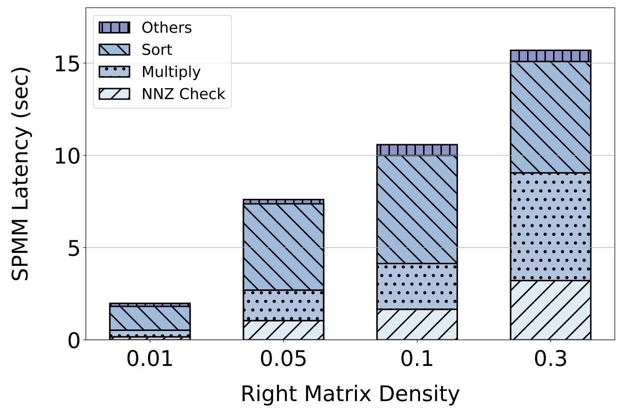
5.2 Recommending Optimal SPMM Implementation

To better understand the characteristics of a SPMM implementation which keeps a right matrix in a sparse CSC format (*Right Sparse*) and another implementation which transforms a right matrix to a dense format (*Right Dense*), Figure 9 presents the time-breakdown of two SPMM implementations under different right matrix densities. The dimension of the left matrix for the SPMM workloads in the experiment is $(10000, 30000)$, and the dimension of the right matrix is $(30000, 10000)$. We vary the density of the right matrix given the right matrix size.

Figure 9a presents the time-breakdown of *RightDense* implementation in a stacked bar format. The time required to convert a sparse matrix to a dense matrix is represented by the bottom part, which has a horizontal line pattern. The middle section with dot markers represents the multiplication time, while the top section shows the other time. We can see that the consumed time distribution is unaffected by the change in right matrix density shown on the horizontal axis, and the total SPMM latency is nearly constant. Figure 9b presents the time-breakdown of the *RightSparse* implementation. Different from the *RightDense* implementation, the *RightSparse* implementation does not perform a dense format transformation. Instead, it uses *Counting NNZ*, *Multiplication*, and *Sort* to perform multiplication while maintaining a correct matrix in a sparse format. In contrast to that in the *RightDense* implementation, we can see that the consumed time varies as the right matrix density varies because the overall overhead is proportional to the NNZ of a right matrix. Because of the latency difference, we can discover that when the right matrix is sparse, performing multiplication in the *RightSparse* implementation completes faster; when the right matrix density is 0.01, the *RightSparse* implementation is 4.3 times faster. The *RightDense* completes faster with a denser right matrix; when the right matrix density is 0.3, *RightDense* is 1.7 times faster.



(a) *RightDense* implementation



(b) *RightSparse* implementation

Fig. 9: Time breakdown of different SPMM implementations

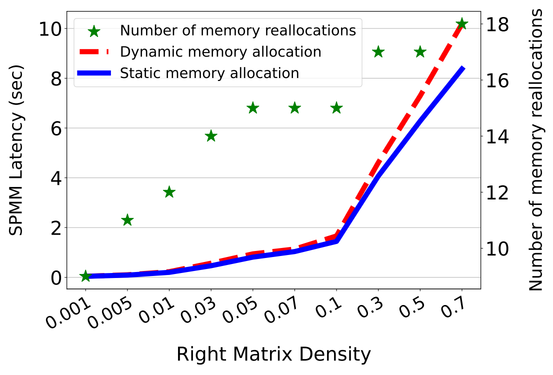


Fig. 10: The SPMM execution latency of two memory allocation methods of dynamic and static. For the dynamic allocation heuristic, it shows the number of reallocation due to insufficient initial memory allocation.

In the *RightSparse* implementation described in Algorithm 2, we chose to statically allocate an output matrix memory after computing the NNZ of an output matrix which can result in extra latency to complete a SPMM task. To validate the decision, we compare the performance of the static memory allocation and the dynamic strategy. In the dynamic memory allocation heuristic, it initially allocates 0.001% of output matrix dimensions $(NR_L \times NC_R)$ to store non-zero entries for an output matrix. When the current

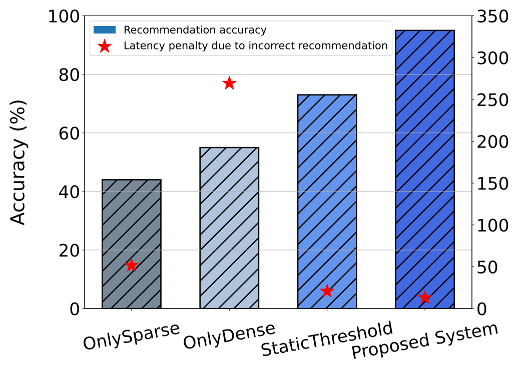


Fig. 11: SPMM implementation recommendation algorithm accuracy and latency penalty ratio due to false recommendations

memory is deficient to store further entries, it reallocates the output matrix memory doubling the current size. We tried other initial allocation sizes and reallocation policies, but they did not show noticeable performance differences.

Figure 10 compares the latency to complete SPMM tasks while conducting the MSSP algorithm using Amazon dataset [56]. During multiple iterations of the MSSP algorithm, the density of right matrix keeps increasing, and we show the right matrix density in the horizontal axis. The primary vertical axis shows the latency, and the blue solid line indicates the latency of static memory allocation while the red dotted line shows that of dynamic memory allocation. The secondary vertical axis presents the number of memory reallocations which happens only in the dynamic heuristic. When the sparsity of right matrix is low, both the NNZ calculation overhead of the static heuristic and the memory reallocation overhead of the dynamic heuristic are negligible, and they show very similar performance. As the density of right matrix increases, the memory reallocation overhead outweigh the NNZ calculation overhead, and the static memory allocation shows 7.25% better performance than the dynamic heuristic. Other than the quantitative superior performance, the implementation of the static allocation is much simpler than the dynamic heuristic which is beneficial regarding the operational perspective. The quantitative and qualitative comparison validate the choice of NNZ calculation followed by the static memory allocation for the *RightSparse* implementation.

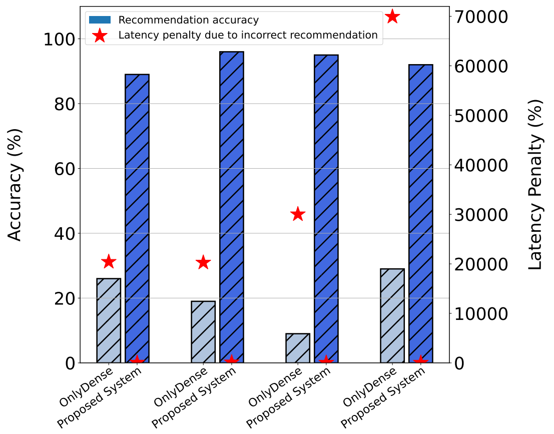
Figure 11 compares the accuracy to recommend a better-performing implementation method of *RightDense* and *RightSparse*. For the evaluation, we use the synthetically generated $1,300$ SPMM experiment cases that were presented in Section 3.2.1. The horizontal axis presents different implementation methods. The *OnlySparse* heuristic performs SPMM while always keeping the right matrix in a sparse CSC format. The *OnlyDense* heuristic executes SPMM while transforming the right matrix to a dense format. The default Apache Spark SPMM implementation adopts the *OnlyDense* implementation. The *StaticThreshold* heuristic selects the *RightSparse* or *RightDense* implementation by considering the density of an input right matrix, D_R. The *StaticThreshold* determines the static density value, and when the density of a right matrix is larger than the threshold value, it selects the *RightDense* implementation. If the density of a right matrix is less than the threshold value, the *RightSparse* implementation is chosen. In our empirical analysis, the threshold value between 0.04 and 0.14 showed the best performance, and we set the threshold as 0.1. The proposed system (*DoS*) employs the SPMM latency prediction model described in this paper. Following the prediction, it chooses an implementation method with the lowest expected latency. Please keep in mind that we separated training and test dataset exclusively with $1,300$ SPMM task scenarios when experimenting with DoS.

The primary vertical axis of Figure 11 shows the accuracy to recommend a faster SPMM implementation method whose value is expressed with bars. The secondary vertical axis shows the relative latency penalty expressed in the percentile unit due to the incorrect recommendation whose value is marked with red star markers. To calculate the relative latency penalty, for an arbitrary SPMM task, let us assume that the *RightDense* implementation takes 100 s and the *RightSparse* implementation takes 150 s. In the workload, the correct recommendation is the *RightDense* implementation. If the *RightDense* implementation is recommended, the penalty becomes zero. If the *RightSparse* implementation is recommended, the penalty is calculated as the absolute value of latency difference divided by the faster response time. In the example, the penalty is calculated as $\frac{150-100}{100}$, which is 50%, and counted towards penalty due to incorrect recommendation. After calculating the penalty ratio of SPMM tasks, we average the penalty ratio value of the false recommendations only.

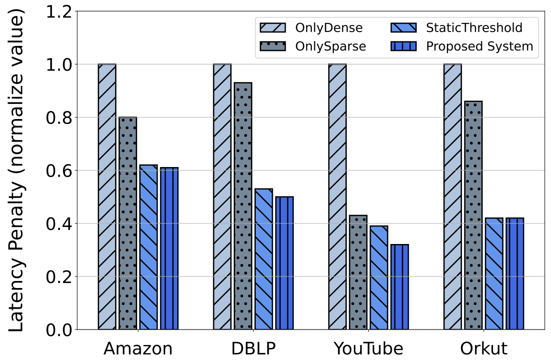
From the figure, we can observe that the recommendation accuracy of DoS is 95%, and it outperforms other methods whose prediction accuracy is $45\%, 55\%$, and 73% for *RightSparse*, *RightDense*, and *StaticThreshold*, respectively. From the latency penalty due to the false recommendation, we can discover that the default Spark implementation method, *OnlyDense*, has a large latency penalty of 269%. In the *RightDense* implementation, a static amount of overhead to transform a right matrix to a dense format always exists, and the overhead might overwhelm the actual multiplication time. The proposed DoS system has an average latency penalty of 13%, which is 20 times lower than the default Spark implementation. The lower penalty ratio of the DoS recommendation algorithm implies that, although DoS makes incorrect recommendations for 5% of the SPMM tasks in the experiments, the latency difference of false recommendations between *RightDense* and *RightSparse* is relatively small, and it shows minor additional latency. Comparing the recommendation accuracy of DoS with that of the *StaticThreshold*, the proposed *DoS* recommendation algorithm shows 30% accuracy improvement. With respect to the latency penalty ratio presented in Figure 11, *StaticThreshold* took 20.69% more time due to the false recommendation, which is 58.9% larger than that of DoS recommendation.

5.3 Optimal SPMM Recommendation Service

We evaluate the practicality of the proposed recommendation system while experimenting with real-world graph datasets, which are *Amazon, DBLP, Youtube*, and *Orkut* [56]. They are available in the SNAP repository [11]. The *Amazon*



(a) SPMM implementation recommendation algorithm accuracy and latency penalty ratio due to false recommendations in real-world graph datasets



(b) Performance improvement over the default Spark implementation

Fig. 12: Superb recommendation accuracy and performance gain by using DoS prediction algorithm over the default Spark implementation

dataset contains $334,863$ nodes and $925,872$ edges. The *DBLP* dataset contains $317,080$ nodes and $1,049,866$ edges. The *YouTube* dataset has $1,134,890$ nodes and $2,987,624$ edges. The *Orkut* dataset contains $3,072,441$ nodes and $117,185,083$ edges. They are encoded as a square matrix in CSC format. The real-world graph dataset is too large to process as a single block and should be divided into several blocks. We used the Spark distributed block matrix RDD representation [20] in the experiments and set the number of rows and columns of the left matrix block to $150,000$ and $100,000$, respectively. For the right matrix, we set the block size as $(100000, 500)$, where 500 is the maximally executable value with given memory size for the experiments. The experiments are conducted while conducting MSSP [5] in multiple iterations. To initialize the right matrix for the MSSP, we randomly select 500 nodes and set the corresponding index of the right matrix as 1 and all other values as 0. As with the MSSP algorithm, the right matrix becomes denser with each iteration. The minimum and maximum densities of the right matrix are 0.00000033 and 0.72875853, respectively. Many SPMM operations with different left and right matrix block densities are performed after partitioning the original input matrices. Each SPMM task on partitioned blocks is carried out independently using a recommended implementation method.

Figure 12 presents the SPMM implementation recommendation quality and performance gain in a realistic application scenario. Figure 12a shows the recommendation accuracy of *OnlyDense*, which is the Spark default SPMM implementation, and the proposed system, DoS. The secondary axis depicts the average performance penalty as a result of incorrect recommendations. The horizontal axis depicts the real-world graphs and SPMM implementation methods used in the experiments. We omit the results of *OnlySparse* and *StaticThreshold* because the algorithms show patterns similar to the results of other experiments.

Across all the practical scenarios, *OnlyDense*, which is the Spark default SPMM implementation, shows poor accuracy of 29% at most for the *Orkut* dataset. It implies that the default Spark SPMM implementation can perform very poorly when the right matrix density is low. Otherwise, the proposed recommendation algorithm of DoS shows superb prediction accuracy of 93% on average. Regarding the latency penalty due to incorrect recommendation, the *OnlyDense* implementation shows significant penalty because it performs poorly especially when the right matrix density is low. Regarding the DoS recommendation algorithm, for the 7% incorrect recommendations, it took 63.5% more time than the optimal implementation. Regarding the *StaticThreshold* algorithm, which is not shown in the figure, for 16% of false recommendations, it took 115% more time than the optimal implementation.

Figure 12b presents performance gain of various SPMM implementations over using the default Spark SPMM implementation, *OnlyDense*. Unlike the performance penalty metric, which only considers false recommendations, the metric in Figure 12b takes into account all cases of correct and incorrect recommendations. We can estimate the overall latency improvement over the default Spark SPMM implementation using the values presented. The Spark default implementation shows the worst performance, and we normalize the latency values to the *OnlyDense* implementation. For different workloads, which are presented in the horizontal axis, the performance of *OnlyDense*, *OnlySparse*, *StaticThreshold*, and *ProposedSystem* is shown in the order. The recommendation algorithm of the proposed DoS shows the most performance gain over the default Spark, and it is expected to improve the overall SPMM latency by 2.2 times.

Figure 13 presents the overhead incurred from the SPMM latency prediction. In our evaluation, the prediction took around 60 milliseconds on average. To quantitatively evaluate the additional overhead from inference, we calculate the ratio of the additional prediction time to the latency gain by choosing an optimal SPMM implementation, which is $\frac{InferenceTime}{|RightSparseLatency - RightDenseLatency|}$. The lower ratio value means the latency gain is much larger than the additional prediction time. Meanwhile, the ratio value larger than 1.0 implies that the additional time for prediction is larger than the performance improvement, and the proposed system can negatively impact the system. The negative impact can happen when the latency of *RightSparse* and *RightDense* is similar.

In Figure 13, we express the overhead ratio using a box-whisker plot. The vertical axis shows the overhead

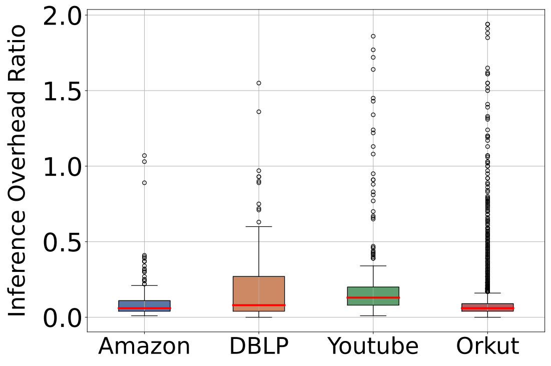
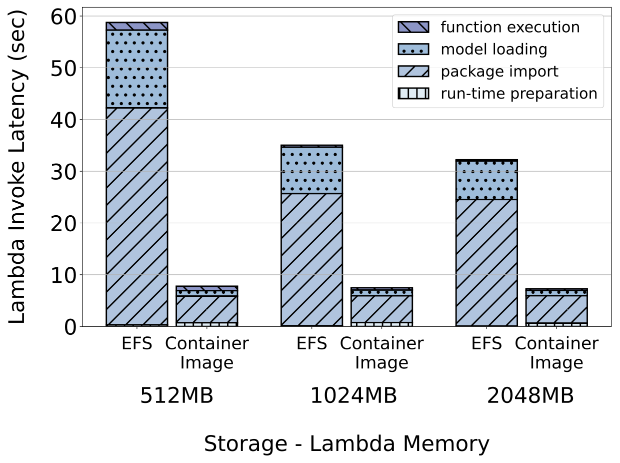
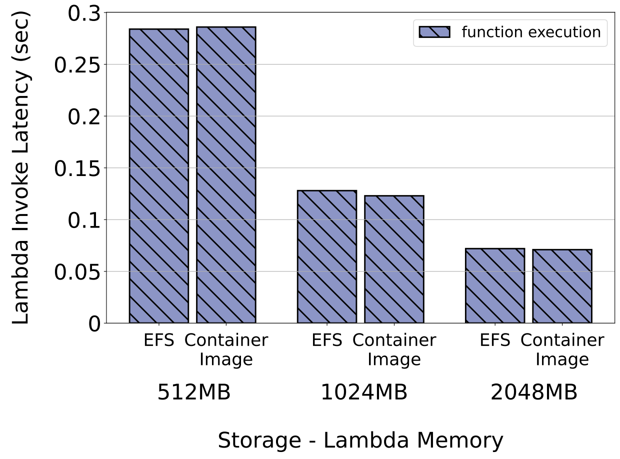


Fig. 13: The additional latency incurred from the SPMM execution latency prediction of DoS



(a) Lambda in a *Cold* runtime



(b) Lambda in a *Warm* runtime

Fig. 14: Time breakdown of deploying prediction model using a serverless architecture with different storage services

ratio value, and the horizontal axis shows different dataset used in the experiments. On average, the first quartile, the median, and the third quartile is $0.05, 0.08$, and 0.17, respectively. Among all the SPMM executions, 4% of cases have overhead ratio larger than 0.5, and 2% of cases have overhead ratio larger than 1.0. Despite of $10s$ of milliseconds of additional inference latency from the SPMM latency prediction, the performance improvement by choosing an optimal implementation method outweighs the inference overhead.

We implemented the proposed system applying a serverless architecture and using AWS Lambda, which has a primary limitation in the local storage size. To load necessary packages for running prediction models, we build two Lambda execution environments: one embedding packages in a container image and the other using NFS. To evaluate the implementation of the proposed service, Figure 14 presents how different storage solutions of Lambda impact the service latency. In the figure, the horizontal axis expresses the configured Lambda memory sizes from 512 to $2,048$ MB, which is known to impact CPU performance [47]. In each memory configuration, we show two different storage options: NFS service (*EFS*) and embedding packages in a container image (*Container Image*).

Figure 14a presents a consumed time-breakdown when a prediction model is served in a cold Lambda runtime. In a cold runtime status, a runtime should be prepared from the service provider, and the time is shown in the vertical line bar at the bottom of each configuration. After the runtime has been prepared, the necessary packages (right upper diagonal pattern portion) and the prediction model should be loaded (dotted portion). Following the completion of all prerequisite steps, the prediction function is executed, as shown in the left upper diagonal pattern. We can discover that the *package import* and *model loading* take the majority of the time when serving models in a cold runtime. The performance difference between *EFS* and *Container Image* is significant, especially when accessing storage services. The observation makes sense because packages and models should be loaded from storage services, and the NFS requires remote network access, which results in longer latency than that in a container image that contains necessary packages and models locally.

Figure 14b shows the latency to serve a prediction request when an execution runtime is in a warm status. In a warm runtime, a code execution environment including imported packages and loaded models is ready, and only the model inference step is required. Thus, Figure 14b shows only the function execution time, and there is no performance difference between *EFS* and *Container Image*. The inference time decreases linearly as the configured memory size increases, and users can set memory size considering the application's characteristics.

From the experimental result presented in Figure 14, embedding necessary files in a container image [53] is beneficial especially when a cold start happens. During a warm start, the underlying file storage service has no effect on overall latency, and users can choose an appropriate storage service based on application service patterns. In addition, using a provisioned concurrency [57], which prepares function runtimes in ready status, can be a good way to circumvent a cold start to provide consistent performance.

6 Related Work

Optimizing Matrix Multiplication Implementation: Yu et al. [58] proposed a system to minimize communication cost when conducting matrix multiplication with multiple worker nodes by predicting the intermediate data size. Considering the input matrix shapes, it generates multiple execution plans and chooses the best-performing one. HAMA [59] proposed an algorithm for efficient matrix multiplication using the Hadoop MapReduce engine [18], [17]. Marlin [60] proposes an algorithm to minimize shuffle and compute overhead by partitioning input matrices. Stark [61] proposes a distributed version of Strassen's matrix multiplication [62] using Apache Spark with a novel approach of creating a distributed recursion tree of computation. MPEC [12], [63] proposed a matrix multiplication performance estimation on various cloud computing resources. The aforementioned algorithms are primarily concerned with optimizing dense matrix multiplications, whereas sparse matrix multiplication tasks differ significantly in terms of the optimal execution environment and task characteristics.

With respect to multiplications of sparse matrix and vector, Xie et al. [64] proposed a new sparse-matrix-storage-format called Block COO+ to minimize shuffle overhead in the Spark. Dhandhania et al. [65] and Luo et al. [66] proposed an algorithm to select the optimal sparse data storage format for matrix multiplication. The preceding work on optimizing sparse matrix multiplication is useful for the proposed DoS algorithm. The DoS performance prediction model for dense and sparse matrix multiplication can be extended to other storage formats, which can improve the performance of sparse matrix multiplication on Spark even more. S-MPEC [67] proposed an algorithm to build an optimal SPMM execution environment on various cloud instances. Different from S-MPEC, DoS can recommend whether dense or sparse representation can result in better performance, while the S-MPEC assumes only the Spark default implementation, which conducts multiplication with the right matrix in a dense format.

TaskFlow [68] is a task scheduling system which consolidates heterogeneously dependent jobs. With the proposed static and dynamic tasking methods, conditional tasking, and concurrent CPU-GPU tasking heuristics, they could achieve latency and energy-efficient lightweight task graph computing system. TaskFlow can be applied for various tasks which can be expressed using a graph data structure including various machine learning algorithms. Considering that Apache Spark expresses jobs using a graph, the proposed algorithm from DoS is complementary to TaskFlow which can be used to detect optimal SPMM implementation method for machine learning jobs.

Hardware-Specific Matrix Multiplication Optimization: Using specialized hardware for SPMM can significantly improve performance, and optimizing SPMM using various hardware accelerators is a well addressed topic in literature [69], [70]. Lin et. al. [15] proposed a sparse DNN inference engine, SNIG, which avoids unnecessary computation from sparse elements. They proposed a decomposition strategy which enhances scheduling efficiencies between CPU and GPU. Xin et. al. [14] proposed several SPMM implementation optimization heuristics to enhance Sparse DNN inference, which includes using loop transformation primitives with a execution cost model on a GPU device. In addition, the authors proposed various GPU-specific optimization tactics. Bisson et. al. [16] presented an sparse DNN inference implementation on GPU devices to overcome the irregular memory access patterns of sparse DNN operations. Alperen et. al. [71] evaluated various sparse matrix computations on multi and many core CPU architectures. From thorough analysis, the authors proposed sparse solver optimization heuristics focusing on fewer cache misses. The aforementioned work and DoS have a common purpose of optimizing sparse dataset operations. However, the difference in the target hardware environments hinders applying the previous work to DoS. The optimization heuristics for specialized hardware and high-performance computing environments are not directly applicable to a general-purpose commodity hardware, such as CPU, which is widely adopted to build a big-data processing environment, such as Apache Spark.

Providing Optimal Data Processing Environment: As computing environment and system configurations become more complex, it is challenging for algorithm developers to build an optimally working environment. Ernest [72], Cherrypick [73], and PARIS [74] proposed algorithms to help build an optimal big-data processing environment on cloud computing resources with Apache Spark. They used down-sampling to reduce offline experiment overhead to cover diverse machine learning algorithms with large-scale datasets. Using the down-sampled dataset, Ernest adopted a non-negative linear equation [75], Cherrypick adopts Bayesian optimization [76] to find the next profiling cloud instance, and PARIS applied random forest [38]. OptEx [77] proposes an algorithm to predict Spark job completion time by considering the input dataset size and cluster configurations. The work is expected to help build a cost-effective Spark environment on cloud where users have flexibility in setting the execution environment. The aforementioned previous work focused on predicting performance of general machine learning algorithms on Apache Spark. However, MPEC [12] showed the performance characteristics of matrix multiplication on Spark is quite different from general machine learning jobs which do not use matrix multiplication as a core computation kernel. Considering the performance difference, SPMM task execution time prediction and providing optimal implementation should be addressed separately as presented in this work.

To help build an optimal deep learning training environment, Paleo [78] and MLPredict [79] proposed algorithms to infer the training time of DNN implementation by referencing the internal model architecture. This method is similar to the one used by DoS to predict the execution time of various SPMM tasks with different implementations. Though the algorithm details vary owing to different application domains, the publication of performance estimation under various configurations demonstrates the significance of the problem addressed in the paper.

Building Optimal Experiment Scenarios: SPMM task scenarios can be very diverse, and DoS adopts LHS, filtering, and a D-optimal algorithm to build SPMM training dataset generation. DoE [9] is widely adopted in relevant domains to minimize experiment cost. Ernest [72] adopted an A-optimal algorithm to select a subset of experiment scenarios. Bayesian optimization [76] helps select the next experiment scenario considering exploitation and exploration. Packing Light [80], like DoS, uses the LHS algorithm to generate random experiment scenarios and response surface design to aid in understanding how different workload characteristics change as the underlying hardware configurations change. Rodrigues et al. [81] discussed the interaction between hardware configurations and machine learning algorithm training on Apache Spark. To minimize experiment cost, the authors applied a randomized two-level fractional factorial design to filter out similar experiment scenarios.

7 Conclusion

SPMM is a core kernel operation for many machine learning algorithms. A general-purpose big-data processing framework, Apache Spark, supports SPMM operation in its linear algebra library. However, the default implementation of Spark SPMM has a major shortcoming as it always transforms a right sparse matrix to a dense format before conducting multiplication. To overcome this limitation, we proposed DoS, which predicts latency to complete various SPMM tasks and recommends an optimal SPMM implementation method. To accomplish this, we described an SPMM implementation method that maintains the right matrix in a sparse CSC format. We proposed a synthetic method of generating practical SPMM scenarios using statistics from real-world graph datasets using the LHS and D-optimal algorithms to build a latency prediction model. DoS created a prediction model with a DNN architecture using the training dataset. Thorough testing reveals that the DoS SPMM implementation recommendation module has excellent prediction accuracy and outperforms the default Spark SPMM implementation by 2.2 times. The proposed system is demonstrated by the real-world deployment of DoS using a serverless architecture with the current Apache Spark.

Acknowledgment

This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIP) under Grants NRF-2022R1A5A7000765 and NRF-2020R1A2C1102544, in part by the ICT R&D program of IITP under Grant 2017-0-00396, and in part by the SW Star Lab under Grant RS-2022-00144309.

References

[1] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[3] T. Baer, R. Kanakagiri, and E. Solomonik, "Parallel minimum spanning forest computation using sparse matrix kernels," 2021.

[4] S. Cabello, E. W. Chambers, and J. Erickson, "Multiple-source shortest paths in embedded graphs," *SIAM Journal on Computing*, vol. 42, no. 4, pp. 1542–1571, 2013. [Online]. Available: https://doi.org/10.1137/120864271

[5] P. N. Klein, "Multiple-source shortest paths in planar graphs," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, 2005, pp. 146–155.

[6] M. H. Mofrad, R. Melhem, Y. Ahmad, and M. Hammoud, "Multithreaded layer-wise training of sparse deep neural networks using compressed sparse column," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–6.

[7] R. Bosagh Zadeh *et al.*, "Matrix computations and optimization in apache spark," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 31–38. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939675

[8] D. Langr and P. Tvrdík, "Evaluation criteria for sparse matrix storage formats," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 428–440, 2016.

[9] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[10] J. M. Hellerstein *et al.*, "Serverless computing: One step forward, two steps back," in *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. [Online]. Available: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

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

[12] J. Kim, M. Son, and K. Lee, "Mpec: Distributed matrix multiplication performance modeling on a scale-out cloud environment for data mining jobs," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2019.

[13] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016.

[14] J. Xin *et al.*, "Fast sparse deep neural network inference with flexible spmm optimization space exploration," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.

[15] D.-L. Lin and T.-W. Huang, "A novel inference algorithm for large sparse neural network using task graph parallelism," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–7.

[16] M. Bisson and M. Fatica, "A gpu implementation of the sparse deep neural network graph challenge," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–8.

[17] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[18] A. S. Foundation, "Apache hadoop," 2004. [Online]. Available: http://hadoop.apache.org/

[19] X. Meng *et al.*, "Mllib: Machine learning in apache spark," *J. Mach. Learn. Res.*, vol. 17, no. 1, p. 1235–1241, Jan. 2016.

[20] A. Spark, "Apache spark mllib distributed matrix computation," https://github.com/apache/spark/tree/master/mllib/src/main/scala/org/apache/spark/mllib/linalg/distributed, 2021, [Online; accessed May. 2021.].

[21] D. Langr and I. Simecek, "Analysis of memory footprints of sparse matrices partitioned into uniformly-sized blocks," *Scalable Comput. Pract. Exp.*, vol. 19, no. 3, pp. 275–292, 2018. [Online]. Available: https://www.scpe.org/index.php/scpe/article/view/1358

[22] A. Iosup *et al.*, "Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms," *Proc. VLDB Endow.*, vol. 9, no. 13, p. 1317–1328, Sep. 2016. [Online]. Available: https://doi.org/10.14778/3007263.3007270

[23] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[24] A. Olsson, G. Sandberg, and O. Dahlblom, "On latin hypercube sampling for structural reliability analysis," vol. 25, no. 1, pp. 47–68, 2003. [Online]. Available: http://dx.doi.org/10.1016/S0167-4730(02)00039-5

[25] F. Triefenbach, "Design of experiments: The d-optimal approach and its implementation as a computer algorithm," 01 2008.

[26] P. Pufek, H. Grgić, and B. Mihaljević, "Analysis of garbage collection algorithms and memory management in java," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 1677–1682.

[27] M. Dessokey *et al.*, "Memory management approaches in apache spark: A review," in *International Conference on Advanced Intelligent Systems and Informatics*. Springer, 2020, pp. 394–403.

[28] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[29] D. Mvondo *et al.*, *OFC: An Opportunistic Caching System for FaaS Platforms*. New York, NY, USA: Association for Computing Machinery, 2021, p. 228–244. [Online]. Available: https://doi.org/10.1145/3447786.3456239

[30] A. Ly *et al.*, "A tutorial on fisher information," *Journal of Mathematical Psychology*, vol. 80, pp. 40 – 55, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022249617301396

[31] K. Ogungbenro, G. Graham, I. Gueorguieva, and L. Aarons, "The use of a modified fedorov exchange algorithm to optimise sampling times for population pharmacokinetic experiments," *Computer Methods and Programs in Biomedicine*, vol. 80, no. 2, pp. 115 – 125, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016926070500146X

[32] B. Wheeler, *AlgDesign: Algorithmic Experimental Design*, 2014, r package version 1.1-7.3. [Online]. Available: https://CRAN.R-project.org/package=AlgDesign

[33] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85 – 117, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608014002135

[34] M. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric Environment*, vol. 32, no. 14, pp. 2627–2636, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1352231097004470

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[36] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *CoRR*, vol. abs/1505.00853, 2015. [Online]. Available: http://arxiv.org/abs/1505.00853

[37] S. Narayan, "The generalized sigmoid activation function: Competitive supervised learning," *Information Sciences*, vol. 99, no. 1, pp. 69–82, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025596002009

[38] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[39] B. Shekar and G. Dagnew, "Grid search-based hyperparameter tuning and classification of microarray cancer data," in *2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP)*. IEEE, 2019, pp. 1–8.

[40] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[41] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011. [Online]. Available: http://jmlr.org/papers/v12/duchi11a.html

[42] L. Prechelt, *Early Stopping - But When?* Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 55–69. [Online]. Available: https://doi.org/10.1007/3-540-49430-8_3

[43] D. Namiot and M. sneps sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 2, pp. 24–27, 09 2014.

[44] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

[45] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, July 2019.

[46] J. Kim and K. Lee, "Practical cloud workloads for serverless faas," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: ACM, 2019.

[47] L. Wang *et al.*, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 133–146. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[48] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[49] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[50] R. Sandberg *et al.*, *Design and Implementation of the Sun Network Filesystem*. USA: Artech House, Inc., 1988, p. 379–390.

[51] A. W. is new., "Aws lambda support for amazon elastic file system now generally available." [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2020/06/aws-lambda-support-for-amazon-elastic-file-system-now-generally-/

[52] J. Choi and K. Lee, "Evaluation of network file system as a shared data storage in serverless computing," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, ser. WoSC'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 25–30. [Online]. Available: https://doi.org/10.1145/3429880.3430096

[53] A. Blog., "Aws lambda – container image support." [Online]. Available: https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/

[54] E. Ostertagová, "Modelling using polynomial regression," *Procedia Engineering*, vol. 48, pp. 500–506, 2012, modelling of Mechanical and Mechatronics Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877705812046085

[55] C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz, "A comparative analysis of gradient boosting algorithms," *Artificial Intelligence Review*, vol. 54, no. 3, pp. 1937–1967, aug 2020. [Online]. Available: https://doi.org/10.1007%2Fs10462-020-09896-5

[56] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, ser. MDS '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2350190.2350193

[57] A. Blog., "Aws lambda announces provisioned concurrency," last accessed April. 2020. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2019/12/aws-lambda-announces-provisioned-concurrency/

[58] Y. Yu *et al.*, "In-memory distributed matrix computation processing and optimization," in *ICDE*, April 2017, pp. 1047–1058.

[59] S. Seo *et al.*, "Hama: An efficient matrix computation with the mapreduce framework," in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, 2010, pp. 721–726.

[60] R. Gu *et al.*, "Efficient large scale distributed matrix computation with spark," in *2015 IEEE International Conference on Big Data (Big Data)*, Oct 2015, pp. 2327–2336.

[61] C. Misra, S. Bhattacharya, and S. K. Ghosh, "Stark: Fast and scalable strassen's matrix multiplication using apache spark," *IEEE Transactions on Big Data*, pp. 1–1, 2020.

[62] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, no. 4, p. 354–356, aug 1969. [Online]. Available: https://doi.org/10.1007/BF02165411

[63] M. Son and K. Lee, "Distributed matrix multiplication performance estimator for machine learning jobs in cloud computing," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 00, Jul 2018, pp. 638–645. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CLOUD.2018.00088

[64] K. Xie, C.-R. Lee, and F.-Y. Liu, "Performance optimization of spmv on spark," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 689–694.

[65] S. Dhandhania *et al.*, *Explaining the Performance of Supervised and Semi-Supervised Methods for Automated Sparse Matrix Format Selection*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458744.3474049

[66] S. Luo, D. Jankov, B. Yuan, and C. Jermaine, "Automatic optimization of matrix implementations for distributed machine learning and linear algebra," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD/PODS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1222–1234. [Online]. Available: https://doi.org/10.1145/3448016.3457317

[67] J. Park and K. Lee, "S-mpec: Sparse matrix multiplication performance estimator on a cloud environment," *Cluster Computing*, 2021. [Online]. Available: https://doi.org/10.1007/s10586-021-03287-3

[68] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 6, pp. 1303–1320, 2022.

[69] J. I. Aliaga *et al.*, "Compression and load balancing for efficient sparse matrix-vector product on multicore processors and graphics processing units," *Concurrency and Computation: Practice and Experience*, vol. n/a, no. n/a, p. e6515. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6515

[70] D. Baek *et al.*, "Innersp: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021, pp. 116–128.

[71] A. Alperen *et al.*, "An evaluation of task-parallel frameworks for sparse solvers on multicore and manycore cpu architectures," in *50th International Conference on Parallel Processing*, 2021, pp. 1–11.

[72] S. Venkataraman *et al.*, "Ernest: Efficient performance prediction for large-scale advanced analytics." in *NSDI*, 2016, pp. 363–378.

[73] O. Alipourfard *et al.*, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[74] N. J. Yadwadkar *et al.*, "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 452–465. [Online]. Available: http://doi.acm.org/10.1145/3127479.3131614

[75] D. Chen and R. J. Plemmons, "Nonnegativity constraints in numerical analysis," in *in A. Bultheel and R. Cools (Eds.), Symposium on the Birth of Numerical Analysis, World Scientific*. Press, 2009.

[76] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 2951–2959. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999325.2999464

[77] S. Sidhanta, W. Golab, and S. Mukhopadhyay, "Deadline-aware cost optimization for spark," *IEEE Transactions on Big Data*, vol. 7, no. 1, pp. 115–127, 2021.

[78] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proceedings of the International Conference on Learning Representations*, 2017.

[79] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 3873–3882.

[80] J. Duggan *et al.*, "Packing light: Portable workload performance prediction for the cloud," in *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*. IEEE, 2013, pp. 258–265.

[81] J. Rodrigues, G. Vasconcelos, and P. Maciel, "Screening hardware and volume factors in distributed machine learning algorithms on spark: A design of experiments (doe) based approach," *Computing*, vol. 103, 10 2021.

Unho Choi is an M.S. student in the Department of Computer Science at Kookmin University, South Korea. His current research topic covers large-scale distributed computing resource management, cloud computing, and machine learning. He received a bachelor degree in the department of Computer Science at Kookmin University.

Kyungyong Lee is an associate professor in the College of Computer Science at Kookmin University. His current research topic covers big data platforms, distributed and cloud computing. He received the Ph. D. degree in the Department of Electrical and Computer Engineering at the University of Florida.