A COMMON SUBEXPRESSION ELIMINATION-BASED COMPRESSION METHOD FOR THE CONSTANT MATRIX MULTIPLICATION

by

Emre Bilgili

B.S., Computer Engineering, Boğaziçi University, 2019

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2022

ACKNOWLEDGEMENTS

This thesis is supported by TUBITAK BIDEB 2210-A program.

ABSTRACT

A COMMON SUBEXPRESSION ELIMINATION-BASED COMPRESSION METHOD FOR THE CONSTANT MATRIX MULTIPLICATION

The execution time, resource and energy costs of deep learning applications become much more important as their popularity grows. The Constant Matrix Multiplication has been studied for a long time and takes place in deep learning applications. Reducing the computation cost of those applications is a highly active research topic. The weights are pruned or quantized while satisfying the desired accuracy requirement. The pruned matrices are compressed into one-dimensional arrays without data loss. Matrix multiplication is performed by processing those arrays without decompression. Processing one-dimensional arrays to perform matrix multiplication is deployed on various hardware platforms that employ Central Processing Unit, Graphics Processor Unit and Field-Programmable Gate Array. The deployments can also be supported with common subexpression elimination methods to reduce the number of multiplications, additions and storage size. However, the state-of-the-art methods do not scale well for the large constant matrices as they reach hours for extracting common subexpressions in a 200×200 matrix. In this thesis, a random search-based common subexpression elimination method is constructed to reduce the run-time of the algorithm. The algorithm produces an adder tree for a 1000×1000 matrix in a minute. The Compressed Sparse Row format is extended to build a one-dimensional compression notation for the proposed method. Simulations for a single-core embedded system show that the latency is reduced by 80% for a given 100×100 matrix compared to the state-of-the-art methods. The storage size of the sparse matrices is also reduced by more than half in the experiments compared to the Compressed Sparse Row format.

ÖZET

SABİT MATRİS ÇARPIMI İÇİN ORTAK ALT İFADE ELEME TABANLI BİR SIKIŞTIRMA YÖNTEMİ

Derin öğrenme uygulamalarının çalışma süresi, kaynak kullanımı ve enerji maliyeti bu uygulamalar arttıkça daha önemli bir duruma gelmiştir. Uzun bir süredir çalışılan sabit matris çarpımı, derin öğrenmede de kullanılmaktadır. Bu uygulamaların hesaplama maliyetinin düşürülmesi yaygın bir araştırma konusudur. Ağırlıklar istenen doğruluk oranı gözetilerek budanır veya nicelendirilir. Budanan matrisler veri kaybı olmadan tek boyutlu dizilerin içine sıkıştırılır. Matris çarpımı bu dizileri geri açmadan işleyerek gerçekleştirilir. Matris çarpımını gerçekleştirmek için tek boyutlu dizilerin işlenmesi Merkezi İşlem Birimi, Görüntü İşlem Birimi ve Alanda Uyarlanabilir Kapı Dizini çalıştıran çeşitli donanımlara uygulanır. Bu uygulamalar toplama ve çarpma sayıları ile depolama boyutunu düşürmek için ortak alt ifade eleme yöntemleri ile de desteklenebilir. Ancak, son yöntemler büyük sabit matrisler için ölçeklenebilir değildir çünkü hesaplama süreleri 200×200 boyutlu bir matris için saatleri bulmaktadır. Bu tezde algoritmanın hesaplama süresini azaltmak için rastgele arama tabanlı bir ortak alt ifade eleme yöntemi inşa edilmiştir. Bu algoritma 1000×1000 boyutlu bir matris için toplama ağacını bir dakikada üretmektedir. Önerilen yönteme uygun bir sıkıştırma gösterimi Sıkıştırılmış Seyrek Satır biçimini genişleterek geliştirilmiştir. Tek çekirdekli gömülü sistem simülasyonları, 100×100 boyutlu bir matris çarpımı için işlem süresinin son yöntemlere göre %80 azaldığını göstermektedir. Deneylerde seyrek matrislerin depolama boyutu da Sıkıştırılmış Seyrek Satır biçimiyle elde edilenin yarısından azdır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS

M	The column number of the matrix
N	The row number of the matrix
$\mathbf{T}$	A constant matrix
$\mathbf{v}$	An input vector
$\mathbf{y}$	A result vector

LIST OF ACRONYMS/ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
At	Attempt Number
CDF	Cumulative Distribution Function
CEA	Copy Elements Array
CESA	Copy Elements Separator Array
CPA	Copy Pairs Array
CPSA	Copy Pairs Separator Array
CMM	Constant Matrix Multiplication
CPU	Central Processing Unit
CSC	Compressed Sparse Column
CSE	Common Subexpression Elimination
CSR	Compressed Sparse Row
ECU	Element Copy Unit
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
It	Iteration Number
IVU	Intermediate Values Unit
MRA	Multiplication Result Array
NNZ	Number of Non-zero
NZR	Non-zero Ratio
PCU	Pair Copy Unit
UEA	Unique Elements Array
UESA	Unique Elements Separator Array
UV	Unique Values

1. INTRODUCTION

Most deep learning applications perform a series of Constant Matrix Multiplication (CMM). Various improvements have been developed to speed up the process and reduce power consumption [1]. For example, the constant matrices are pruned at the cost of some accuracy loss. Then, the resulting sparse matrices are compressed to eliminate the process with zero operands. The applied compression format is implemented on customizable hardware Field-Programmable Gate Arrays (FPGA) and low-end devices [2, 3]. In addition, the matrix is quantized to process on smaller execution units [4].

Common subexpression elimination (CSE) methods modify the statements that process the one-dimensional arrays to perform matrix multiplication. They are upgraded to integrate the common subexpressions. However, the modified statements may not fit the target hardware properly. For example, the Graphics Processing Unit (GPU) is specialized in performing two-dimensional matrix multiplication. It outperforms the Central Processing Unit (CPU) and FPGA in terms of latency of the matrix multiplication. It becomes the top choice to deploy a deep learning model when the price and power costs are affordable. Yet, it shows a weak performance in matrix multiplication with Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats. Therefore, deploying a CSE method on a GPU is not a common practice. In addition, the efficiency of the CSE methods depend on the similarity of the coefficients. Pruning and quantization operations tend to increase the similarity, but they also cause an accuracy reduction in many models. If the accuracy requirement of the application is not allowed to be reduced, those operations are not applied by a significant amount. As a result, the latency and accuracy requirements limit the usability of the CSE methods.

The quantization operation is also performed to reduce the number of distinct elements in deep learning applications [5]. The number of duplicate elements increases

in this way. Hence, a CSE algorithm is applied to remove the redundant operations [6]. Besides, the matrices are pruned to reduce the number of multiplications and additions. The sparse matrices are recorded into several one-dimensional arrays, and matrix multiplication is performed by processing the arrays [7]. A CSE method connected with a compression and matrix multiplication format is aimed in this thesis.

This thesis constructs a lossless compression method that includes a reuse technique while ignoring zeros. The matrix-vector multiplication is discussed in two steps. Firstly, the vector is element-wise multiplied by the matrix. Secondly, the rows of the resulting matrix are accumulated to produce the result vector. A CSE technique is constructed for both steps. A portion of the multiplications and additions are replaced with the copy operations. Afterwards, a compression format is provided to realize the eliminated subexpressions by the reuse operations.

The gem5 simulation tool is used in the experiments [8]. The hardware of a low-cost device is imitated because the deep learning applications are deployed on the low-end devices to reduce the price and power consumption. Sample matrices are constructed to measure the latency, storage size, and the number of additions and multiplications. The proposed method, the base implementation of the matrix multiplication and two state-of-the-art methods [9] and [10] are compared in terms of latency and the number of additions. Plus, the proposed method is compared with the CSR format in terms of the storage size. Only the proposed method reduces the number of multiplications. So, it is not compared with other techniques in terms of the number of multiplications.

The thesis is organized as follows. Model restructuring, pruning, quantization and compression operations in the field of study are mentioned in the next chapter. The compared methods are briefly explained with an illustrative example in Chapter 3. Proposed search algorithm and compression format are constructed with an illustrative example in Chapter 4. Experiments are performed, and the results are commented on in Chapter 5. The thesis is concluded in the last chapter.

2. RELATED WORK

The studies to improve deep learning applications can be grouped into two categories. One group implies increasing the accuracy. The models are enlarged, and more complex functional models are introduced [11]. The models are upgraded to improve the accuracy, but the upgrades increase the costs of the applications at the same time. For this reason, the metrics are built to predict the costs of the models and frameworks [12, 13]. The layer-specific features of a deep learning model are analyzed. The additional cost caused by adjusting a layer and adding a layer is measured. In addition, the effects of the hyper-parameters on the accuracy are investigated with various data sets. The results are aimed to use in future application developments as training and testing the models requires a significant amount of time [14].

The studies in the second group aim to reduce the implementation costs of the trained models. The latency, power consumption and size of an inference model are minimized while the accuracy is kept above a certain threshold. The solutions for reducing the costs may be gathered under four topics [15, 16]. Firstly, the models are analyzed to be reshaped. The large layers are approximately compressed or decomposed into a set of small layers to remove the redundant operations and reduce the size [17,18]. The compression and decomposition ratio is tuned by the change in accuracy because the accuracy may decrease significantly due to data loss. Moreover, each problem requires a different model. Unfortunately, reshaping the weights depends on the target model. A generic framework is introduced in [19] to produce a model-independent solution. The purpose of the framework is to build sparse deep learning models from scratch. Pruning is not required after the training operation as the density of the weights is adjusted while building the matrices. The framework can be applied to produce different models.

Secondly, the weights are pruned. The matrix elements are set to zero during the training step or after the weights are produced [20, 21]. The density of the matrices

is reduced by a set of particular ratios. The accuracy for each ratio is calculated, and the most suitable one is picked. The coefficients are selected in a structural or non-structural manner. Each coefficient is individually considered in the non-structural pruning. The coefficients with less impact on the accuracy are explored and eliminated. The coefficients are considered group by group in structural pruning. The matrices are divided into rectangular areas. If an area is selected to be pruned, its elements are set to zero. The size of the rectangular areas and prune ratio are determined according to the target hardware and compression method. The purpose of the structural pruning is loading and processing the matrices block by block. Pruning eliminates some blocks in structural pruning and some elements in non-structural pruning to reduce the latency and storage size.

Both pruning methods are compared in [22] in terms of storage size and latency. It shows that structural pruning outperforms non-structural pruning when applied with the compression and data placement methods. An example deployment on FPGA is studied for structural pruning in [23]. A compression method is built for the non-zero blocks. The blocks are distributed into the Block RAMs for parallel processing. The results show that latency and power consumption are reduced against the non-structural pruning and sequential implementation of the structural pruning.

Thirdly, the weights are quantized. The data type of the weights is changed to reduce the storage and computation costs [24]. If the hardware contains the processing unit for the target data type, the latency is reduced. Otherwise, the quantization operation still continues to be beneficial as the size of the weights is reduced [25]. For example, FPGA contains arithmetic units for 16-bit half-precision floating-point and arbitrary precision fixed-point data types. 32-bit single-precision floating-point coefficients can be mapped to shorter data types. The target bit size is determined by the target accuracy. The design is tested under a set of word lengths, and the accuracy is obtained for each configuration. The most appropriate one is deployed. Note that the quantization operation can sometimes slightly increase the accuracy [26]. Hence, pruning and quantization are also used to improve accuracy.

The weights may be quantized to a small set of values. For example, the coefficients are mapped to powers-of-two in [27]. Each coefficient contains a single 1 in its binary representation. The storage cost is significantly reduced, while the data loss does not cause a valuable reduction in accuracy in some models. In addition, the hardware efficiency of the shift operation on an FPGA is analyzed in [28]. The coefficients are not restricted with powers-of-two. The input matrix values are converted to the fixed-point data type. So, the multiplication operation is replaced with a series of shift and addition operations. Its overhead is traded with a low power option. The results imply a significant gain in the computation cost.

The fourth topic includes processing compressed matrices. The weights are pruned to reduce the density. Those sparse matrices are compressed to ignore zero elements [29]. The non-zero elements are saved into several one-dimensional arrays. The data is not lost during compression, and its two-dimensional form can be reconstructed. In addition, the matrix multiplication is performed without decompression. Processing the one-dimensional arrays produces the result vector. In this way, the matrices are kept in the compressed form while storing and executing [30]. Decompressing the matrices is not required during the run-time of the device. CSR and CSC formats are two examples of processing one-dimensional arrays to perform matrix multiplication. The non-zeros are row-wise or column-wise ordered, respectively. The values and indices of the coefficients are recorded into two one-dimensional arrays. Then, the end position of each row or column is recorded into the third array, respectively. So, the matrix is displayed with three one-dimensional arrays. They are deployed on various hardware platforms such as CPU, FPGA and GPU [31–33].

All of those improvements provide crucial benefits for the edge devices and mobile phones [34, 35]. The limited power delivery and low storage size form the main issues for those devices. The model size and power consumption need to be reduced below a certain level to perform the inference operation properly [36, 37]. Therefore, the sparse matrix-vector multiplication is studied to be accelerated with the guidance of those improvements [38]. In addition, it can be supported with CSE methods. They explore

and eliminate the redundant operations in matrix multiplication. An adder tree is produced to show the operation sequence. A notation is provided to save the adder tree in the storage and execute it in the hardware.

The efficiency of the CSE methods can be increased with quantization. The number of non-zero elements stays the same, but the number of distinct elements is reduced. So, the occurrence rate of an element is increased. CSE methods utilize the similarity between the duplicate elements. A quantized network example is studied in [39]. The coefficients of the matrices are mapped to -1 and 1. The output of each layer is mapped -1 and 1 in the run-time. Zeros are also considered positive or negative values. So, the negative values are represented with 0, and positive values are represented with 1. The binary multiplications are performed with bit-wise operations. The models are deployed on CPU, GPU, FPGA and Application-Specific Integrated Circuit (ASIC) for the latency comparison. ASIC implementation takes the first place, and the proposed FPGA architecture takes the second place. FPGA seems to utilize bit-wise operations much more than GPU and CPU.

Two CSE algorithms are introduced in [9] and [10] for matrices that contain only $0s$ and $1s$. So, the multiplication operation is not considered. The methods use the same notation to express subexpressions. A pair of a row and column is appended to the matrix for each common subexpression. A huge matrix is produced to form the adder tree. Proposed in [9] searches the size-of-two common subexpressions and concatenate them to form longer ones. Proposed in [10] searches the longest common subexpressions. The search procedure of each method is explained in the next chapter on an example.

Both methods are compared in [40] to run an inference model on an FPGA. The value set is extended to -1, 0 and 1 for the experiments. So, the matrix multiplication is performed with the addition and subtraction units. The results show that eliminating the longest subexpression misses the shorter subexpressions in [10]. Proposed in [9] explores both shorter and longer subexpressions. In this way, it reduces the number

of additions more than proposed in [10]. In addition, proposed in [41] is used in the comparison. It considers fixed-point and integer coefficients in their Canonical Signed Digit (CSD) forms. Multiplication operations are replaced with a series of additions, subtractions and shift operations. The method produces an adder tree with a depth-first search algorithm on the CSD bits. It produces better results than proposed algorithms in [10] and [9] for 3×3 matrices which is the first layer. However, it does not scale for the larger matrices due to its longer run-time for the first layer. For this reason, it is not used for the next layers.

Algorithms introduced in [10] and [9] also contain two major problems. The first problem is long search time for the large matrices. The run-time of the algorithms reach approximately an hour for 100×100 matrices and nine hours for 200×200 matrices. The second problem is the notation they use to imply the subexpressions. The additional rows and columns include many zeros. So, the algorithms take an input matrix and produce a larger matrix by the notation. The storage size increases due to the lack of a compression method. In this thesis, a heuristic algorithm is proposed by sacrificing some of the addition eliminations to produce the results for 1000×1000 matrices in a shorter time. Additionally, a one-dimensional compression format which does not record the zero elements is provided. It compresses experimented matrices more than CSR and CSC formats.

3. THE STATE-OF-THE-ART METHODS

Two CSE algorithms are introduced in [9] and [10] using the same notation. They are explained on an illustrative example. Let $\mathbf{T}$ be a 4×5 matrix and $\mathbf{v}$ be a 5×1 vector. The matrix multiplication $\mathbf{Tv}$ can be written as

$$\mathbf{Tv} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \end{bmatrix} . \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix} = \begin{bmatrix} v_0 + v_2 + v_3 + v_4 \\ v_1 + v_2 + v_3 + v_4 \\ v_0 + v_3 \\ v_0 + v_2 + v_3 + v_4 \end{bmatrix} . \tag{3.1}$$

The coefficients are selected as ones and zeros because both methods focus on only binary values. They follow a procedure to record the common subexpressions. The input vector consists of M values from v_0 to v_{M-1} and the result vector consists of N values from y_0 to y_{N-1}. A common subexpression is picked according to the applied algorithm. Then, it is extracted from the rows and appended as a new row at the bottom of the matrix. Plus, a new column is appended to the right of the matrix. In that column, the coefficients in the extracted rows are set to one. v_M is appended to both the input vector and result vector.

3.1. Explanation of the Proposed Method in Hsiao et al. [9]

This method searches size-of-two common subexpressions. The longer subexpressions are iteratively obtained from size-of-two subexpressions. The most occurred size-of-two subexpression is processed at each iteration. The common subexpression list of $\mathbf{T}$ is prepared as follows:

- $v_0 + v_2$ occurs twice.
- $v_0 + v_3$ occurs three times.
- $v_0 + v_4$ occurs twice.
- $v_2 + v_3$ occurs three times.

- $v_2 + v_4$ occurs three times.
- $v_3 + v_4$ occurs three times.

Four size-of-two expressions occur three times. The first one of the most occurred subexpressions, $v_0 + v_3$, is picked. Then, the matrix, input vector and result vector are updated. Let $\mathbf{T}_r$ and $\mathbf{v}_r$ denote the resulting matrix and input vector for the r'th iteration. The result of the first iteration is shown as

$$\mathbf{T}_1\mathbf{v}_1 = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ v_5 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} = \begin{bmatrix} v_2 + v_4 + v_5 \\ v_1 + v_2 + v_3 + v_4 \\ v_5 \\ v_2 + v_4 + v_5 \\ v_0 + v_3 \end{bmatrix}, \tag{3.2}$$

where $v_0 + v_3$ is replaced with v_5. Extracting a subexpression may affect the other subexpressions. For this reason, the subexpression list is recalculated as follows:

- $v_2 + v_4$ occurs three times.
- $v_2 + v_5$ occurs twice.
- $v_4 + v_5$ occurs twice.

$v_2 + v_4$ is extracted from the matrix as it occurs the most. The result of the second iteration is shown as

$$\mathbf{T}_2\mathbf{v}_2 = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ v_5 \\ v_6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{bmatrix} = \begin{bmatrix} v_5 + v_6 \\ v_1 + v_3 + v_6 \\ v_5 \\ v_5 + v_6 \\ v_0 + v_3 \\ v_2 + v_4 \end{bmatrix}, \tag{3.3}$$

where $v_2 + v_4$ is replaced with v_6.

The remaining subexpression is listed as follows:

- $v_5 + v_6$ occurs twice.

The last common subexpression is extracted, and the result of the third iteration is shown as

$$\mathbf{T}_3\mathbf{v}_3 = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} . \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} = \begin{bmatrix} v_7 \\ v_1 + v_3 + v_6 \\ v_5 \\ v_7 \\ v_0 + v_3 \\ v_2 + v_4 \\ v_5 + v_6 \end{bmatrix}, \tag{3.4}$$

where $v_5 + v_6$ is replaced with v_7. The iterations finish as $\mathbf{T}_3$ does not contain any common subexpression. It is the result of the method [9].

3.2. Explanation of the Proposed Method in Wu et.al. [10]

This method picks two rows of the matrix and notes the longest subexpression. All row pairs are checked, and the longest subexpression is processed in each iteration. The common subexpression list of $\mathbf{T}$ in Equation (3.1) is prepared as follows:

- Row 0 and 1 includes $v_2 + v_3 + v_4$ by three elements.
- Row 0 and 2 includes $v_0 + v_3$ by two elements.
- Row 0 and 3 includes $v_0 + v_2 + v_3 + v_4$ by four elements.
- Row 1 and 3 includes $v_2 + v_3 + v_4$ by three elements.
- Row 2 and 3 includes $v_0 + v_3$ by two elements.

The longest subexpression is $v_0 + v_2 + v_3 + v_4$ and occurs in the row 0 and 3. It is appended to the matrix as a new row and column pair. Let $\mathbf{T}_r$ and $\mathbf{v}_r$ denote the

resulting matrix and input vector for the r'th iteration. The result of the first iteration is shown as

$$\mathbf{T}_1\mathbf{v}_1 = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ v_5 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix} = \begin{bmatrix} v_5 \\ v_1 + v_2 + v_3 + v_4 \\ v_0 + v_3 \\ v_5 \\ v_0 + v_2 + v_3 + v_4 \end{bmatrix}, \tag{3.5}$$

where $v_0 + v_2 + v_3 + v_4$ is replaced with v_5. The subexpression list is updated as follows:

- Row 1 and 4 includes $v_2 + v_3 + v_4$ by three elements.
- Row 2 and 4 includes $v_0 + v_3$ by two elements.

The subexpression $v_2 + v_3 + v_4$ is picked and extracted as being the longest one. The result of the second iteration is shown as

$$\mathbf{T}_2\mathbf{v}_2 = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ v_5 \\ v_6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{bmatrix} = \begin{bmatrix} v_5 \\ v_1 + v_6 \\ v_0 + v_3 \\ v_5 \\ v_0 + v_6 \\ v_2 + v_3 + v_4 \end{bmatrix}, \tag{3.6}$$

where $v_2 + v_3 + v_4$ is replaced with v_6. The iterations end as the $\mathbf{T}_2$ does not contain any common subexpression. It is the result of the method [10].

4. METHOD

Let $\mathbf{T}$ be an $N \times M$ constant matrix where each column is accessed as $\mathbf{t}_j$. Let $\mathbf{v}$ represent an input vector, and v_j is one of its entries. Then a matrix-vector multiplication can be handled as

$$\mathbf{Tv} = \mathbf{y} = \mathbf{t}_0 v_0 + \mathbf{t}_1 v_1 + ... + \mathbf{t}_j v_j + ... + \mathbf{t}_{M-1} v_{M-1}. \tag{4.1}$$

This approach enables matrix-vector multiplication in two steps. In the first step, each entry multiplies the related column. In the second step, the multiplied columns are added row-by-row. Hence, Equation (4.1) reduces the number of multiplications when a column contains a constant more than once.

In a matrix, there are $M.N$ entries. Since only non-zero elements contribute to the computation, the upper bound on the number of computations is determined by the number of non-zero elements (NNZ). Then, the sparsity of a matrix can be determined as

$$\text{NZR} = \frac{\text{NNZ}}{M.N}. \tag{4.2}$$

It is obvious that sparsity increases as the non-zero ratio (NZR) reduces. Sparse matrix dense vector multiplications can be realized with fewer operations.

Computations can be reduced further if all of the non-zero elements can be represented by only a few different numbers. For example, a matrix that contains only -1, 0 and 1 strips off the multiplications. A matrix of size 1000×1000 with $4-$bit fixed point entries requires at most 16000 multiplications instead of one million if the computation is carried out as shown in Equation (4.1). Without loss of generality, we can claim that the number of unique values (UV) in a matrix is essential in reducing the number of computations, regardless of the data format used in representing these values. Thus, only letters are used in the representation of UV throughout the illustrative examples in this chapter.

Given a constant matrix, NNZ and UV may be reduced through the quantization and pruning operations to reach better results regarding the number of additions, multiplications and compression size. The data loss may cause an error in the result vector. For this reason, the developer should decide the portions for the pruning and quantization by analyzing the benefits and accuracy loss. The proposed method does not provide a recommendation about the portions. It accepts any constant matrix with or without pruning and quantization.

The proposed method consists of CSE and compression steps. The number of multiplications and additions is reduced in the CSE step. The constant matrix is compressed into several one-dimensional arrays in the compression step. Processing those arrays produces the result vector without data loss. Both steps are explained with an illustrative example.

4.1. Common Subexpression Elimination

An example 8×8 constant matrix with 0.75 NZR and 3 UV is constructed to be used in the explanations as

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} a & b & 0 & c & b & a & c & 0 \\ 0 & a & a & b & b & c & 0 & c \\ c & b & 0 & 0 & c & a & b & a \\ c & 0 & b & a & 0 & b & b & b \\ b & c & a & b & 0 & 0 & a & c \\ c & 0 & c & c & a & c & b & 0 \\ 0 & c & a & a & b & 0 & c & b \\ a & b & a & 0 & b & a & 0 & c \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix}$$

$$= \begin{bmatrix} av_0 \\ 0v_0 \\ cv_0 \\ cv_0 \\ bv_0 \\ cv_0 \\ 0v_0 \\ av_0 \end{bmatrix} + \begin{bmatrix} bv_1 \\ av_1 \\ bv_1 \\ 0v_1 \\ cv_1 \\ 0v_1 \\ cv_1 \\ bv_1 \end{bmatrix} + \begin{bmatrix} 0v_2 \\ av_2 \\ 0v_2 \\ bv_2 \\ av_2 \\ cv_2 \\ av_2 \\ av_2 \end{bmatrix} + \begin{bmatrix} cv_3 \\ bv_3 \\ 0v_3 \\ av_3 \\ bv_3 \\ cv_3 \\ av_3 \\ 0v_3 \end{bmatrix} + \begin{bmatrix} av_4 \\ bv_4 \\ cv_4 \\ 0v_4 \\ 0v_4 \\ av_4 \\ bv_4 \\ bv_4 \end{bmatrix} + \begin{bmatrix} av_5 \\ cv_5 \\ av_5 \\ bv_5 \\ 0v_5 \\ cv_5 \\ 0v_5 \\ av_5 \end{bmatrix} + \begin{bmatrix} cv_6 \\ 0v_6 \\ bv_6 \\ bv_6 \\ av_6 \\ bv_6 \\ av_6 \\ 0v_6 \end{bmatrix} + \begin{bmatrix} 0v_7 \\ cv_7 \\ av_7 \\ bv_7 \\ cv_7 \\ 0v_7 \\ bv_7 \\ cv_7 \end{bmatrix}, \tag{4.3}$$

where $U = \{\text{a, b, c}\}$ represent the set of values.

Let $mul_{u_x,i}$ be defined as

$$mul_{u_x,i} = u_x v_i, \tag{4.4}$$

where u_x is a unique value in U and v_i is an entry at the input vector v. The main idea of the first step in CSE is to replace the multiplication operation with the copy operation in the first step. In the illustrative example of Equation (4.3), cv_0 appears three times: third, fourth and sixth rows. In other words, there are three occurrences of

$mul_{c,0}$ according to Equation (4.4). Instead of calculating $mul_{c,0}$ three times, it can be calculated once, and its result can be reused. In this way, two multiplication operations are eliminated in the first column. Other columns also include multiple occurrences for reuse. Though $\mathbf{T}$ contains 48 non-zero elements, only 24 multiplication operations are sufficient to build the result vector.

Row-wise additions can be reduced similarly. In the proposed approach, two-element common subexpressions are sought. Let $\mathbf{t}_i$ and $\mathbf{t}_j$ be two selected columns from matrix $\mathbf{T}$. The elements at the k'th row of these columns can be accessed as $t_{k,i}$ and $t_{k,j}$. Element-wise addition at these rows, $add_{k,ij}$ can be defined as

$$add_{k,ij} = t_{k,i}v_i + t_{k,j}v_j. \tag{4.5}$$

If there exists another row l such that $add_{k,ij} = add_{l,ij}$, then one of the additions can be eliminated since the result of the first addition can be used in the other. Assume that there are $z_{k,ij}$ occurrences of $add_{k,ij}$ in the $(\mathbf{t}_i, \mathbf{t}_j)$ pair. Then, the number of addition eliminations due to $add_{k,ij}$ can be computed as $z_{k,ij} - 1$. Thus, a solution that maximizes the total gain should be sought as

$$\begin{aligned} gain &= \max \sum_{\forall(\mathbf{t}_i,\mathbf{t}_j)} \sum_{k} (z_{k,ij} - 1) \\ &\text{such that} \\ &(\mathbf{t}_i, \mathbf{t}_j) \cap (\mathbf{t}_m, \mathbf{t}_n) = \emptyset \qquad\qquad i \neq j \neq m \neq n \\ &\bigcup_{i \neq j} (\mathbf{t}_i, \mathbf{t}_j) = \mathbf{T}. \end{aligned} \tag{4.6}$$

Two-element common subexpressions are sought in the second step iteratively. Each iteration consists of an initial phase and an improvement phase. The matrix is partitioned into $\frac{M}{2}$ pairs in the initial phase. Some of the pairs are untied, and the remaining columns are re-matched in the improvement phase. All common subexpressions that maximize $gain$ are extracted from the matrix at the end of the current iteration. The remaining matrix is processed through the same procedure in the next iteration. The user decides the number of iterations. Once all iterations end, the CSE step terminates. Two consecutive iterations are illustrated in the rest of this section.

The columns of the illustrative matrix are randomly picked and paired as $(\mathbf{t}_0, \mathbf{t}_6)$, $(\mathbf{t}_1, \mathbf{t}_4)$, $(\mathbf{t}_2, \mathbf{t}_5)$ and $(\mathbf{t}_3, \mathbf{t}_7)$ at the initial phase. The common subexpressions of the pairs are listed as follows:

- $(\mathbf{t}_0, \mathbf{t}_6)$: $cv_0 + bv_6$ occurs three times.
- $(\mathbf{t}_1, \mathbf{t}_4)$: This pair does not contain a common subexpression.
- $(\mathbf{t}_2, \mathbf{t}_5)$: This pair does not contain a common subexpression.
- $(\mathbf{t}_3, \mathbf{t}_7)$: $av_3 + bv_7$ occurs twice and $bv_3 + cv_7$ occurs twice.

Then, the gain becomes

$$gain^{1,0} = (3-1) + 0 + 0 + [(2-1) + (2-1)] = 4 \tag{4.7}$$

according to the selected pair order. In this equation, $gain^{1,0}$ represents the gain achieved at the initial attempt of the first iteration.

The improvement phase consists of a series of improvement attempts. Two columns from different pairs are randomly picked and temporarily exchanged in a single attempt. The number of addition eliminations is calculated for the new order. If the current gain is higher than the previous one, the replacement is accepted, and the improvement attempt becomes successful. Otherwise, the replacement is reverted, and the improvement attempt becomes unsuccessful. The first improvement attempt on the selected pair order $(\mathbf{t}_0, \mathbf{t}_6)$, $(\mathbf{t}_1, \mathbf{t}_4)$, $(\mathbf{t}_2, \mathbf{t}_5)$ and $(\mathbf{t}_3, \mathbf{t}_7)$ temporarily exchanges $\mathbf{t}_1$ and $\mathbf{t}_6$. The common subexpressions, according to the new order, are listed as follows:

- $(\mathbf{t}_0, \mathbf{t}_1)$: $av_0 + bv_1$ occurs twice.
- $(\mathbf{t}_4, \mathbf{t}_6)$: This pair does not contain a common subexpression.
- $(\mathbf{t}_2, \mathbf{t}_5)$: This pair does not contain a common subexpression.
- $(\mathbf{t}_3, \mathbf{t}_7)$: $av_3 + bv_7$ occurs twice and $bv_3 + cv_7$ occurs twice.

The gain becomes

$$gain^{1,1} = (2-1) + (2-1) + (2-1) = 3 \tag{4.8}$$

according to the new pair order. The exchange attempt of $\mathbf{t}_1$ and $\mathbf{t}_6$ is rejected as $gain^{1,1}$ is less than $gain^{1,0}$.

In the next attempt, $\mathbf{t}_4$ and $\mathbf{t}_5$ are randomly picked and temporarily exchanged on the pair order $(\mathbf{t}_0, \mathbf{t}_6)$, $(\mathbf{t}_1, \mathbf{t}_4)$, $(\mathbf{t}_2, \mathbf{t}_5)$, $(\mathbf{t}_3, \mathbf{t}_7)$. The common subexpressions are listed as follows:

- $(\mathbf{t}_0, \mathbf{t}_6)$: $cv_0 + bv_6$ occurs three times.
- $(\mathbf{t}_1, \mathbf{t}_5)$: $bv_1 + av_5$ occurs three times.
- $(\mathbf{t}_2, \mathbf{t}_4)$: $av_2 + bv_4$ occurs three times.
- $(\mathbf{t}_3, \mathbf{t}_7)$: $av_3 + bv_7$ occurs twice and $bv_3 + cv_7$ occurs twice.

The gain becomes

$$gain^{1,2} = (3-1) + (3-1) + (3-1) + [(2-1) + (2-1)] = 8 \tag{4.9}$$

according to the new pair order. The exchange attempt of $\mathbf{t}_4$ and $\mathbf{t}_5$ is accepted as $gain^{1,2}$ is greater than $gain^{1,0}$.

Two improvement attempts are shown in the first iteration. The user decides the number of attempts. The following attempts are performed on the last accepted pair order in the improvement phase. Therefore, the number of addition eliminations either stays the same or increases through the attempts. Once an iteration is halted, the last pair order is marked as the result of the current iteration. The common subexpressions of the selected pairs are used in forming $\mathbf{y}_p^1$, the partial result vector of the first iteration, as

$$\mathbf{y}_p^1 = \mathbf{T}_p^1 \mathbf{s}^1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} cv_0 + bv_6 \\ bv_1 + av_5 \\ av_2 + bv_4 \\ av_3 + bv_7 \\ bv_3 + cv_7 \end{bmatrix} . \tag{4.10}$$

T_p^1 and s^1 represent the row indices of the common subexpressions found in the first iteration and the values of those common subexpressions, respectively. Let $\mathbf{T}_r^1$ represent the remainder matrix after the first iteration. Then, the remaining operations that will take place on $\mathbf{T}_r^1\mathbf{v}$ can be written as

$$\mathbf{y}_r^1 = \mathbf{T}_r^1 \mathbf{v} = \begin{bmatrix} av_0 \\ 0v_0 \\ 0v_0 \\ 0v_0 \\ bv_0 \\ 0v_0 \\ 0v_0 \\ av_0 \end{bmatrix} + \begin{bmatrix} 0v_1 \\ av_1 \\ 0v_1 \\ 0v_1 \\ cv_1 \\ 0v_1 \\ cv_1 \\ 0v_1 \end{bmatrix} + \begin{bmatrix} 0v_2 \\ 0v_2 \\ 0v_2 \\ bv_2 \\ av_2 \\ cv_2 \\ 0v_2 \\ 0v_2 \end{bmatrix} + \begin{bmatrix} cv_3 \\ 0v_3 \\ 0v_3 \\ 0v_3 \\ 0v_3 \\ cv_3 \\ 0v_3 \\ 0v_3 \end{bmatrix} + \begin{bmatrix} av_4 \\ 0v_4 \\ cv_4 \\ 0v_4 \\ 0v_4 \\ av_4 \\ 0v_4 \\ 0v_4 \end{bmatrix} + \begin{bmatrix} 0v_5 \\ cv_5 \\ 0v_5 \\ bv_5 \\ 0v_5 \\ cv_5 \\ 0v_5 \\ 0v_5 \end{bmatrix} + \begin{bmatrix} cv_6 \\ 0v_6 \\ 0v_6 \\ 0v_6 \\ av_6 \\ 0v_6 \\ av_6 \\ 0v_6 \end{bmatrix} + \begin{bmatrix} 0v_7 \\ 0v_7 \\ av_7 \\ 0v_7 \\ 0v_7 \\ 0v_7 \\ 0v_7 \\ cv_7 \end{bmatrix} . \tag{4.11}$$

The pairs are reshuffled for the initial phase of the second iteration. The common subexpressions of the pair order $(\mathbf{t}_0, \mathbf{t}_2)$, $(\mathbf{t}_1, \mathbf{t}_7)$, $(\mathbf{t}_3, \mathbf{t}_4)$, $(\mathbf{t}_5, \mathbf{t}_6)$ on $\mathbf{T}_1$ are listed as follows:

- $(\mathbf{t}_0, \mathbf{t}_2)$: This pair does not include a common subexpression.
- $(\mathbf{t}_1, \mathbf{t}_7)$: This pair does not include a common subexpression.
- $(\mathbf{t}_3, \mathbf{t}_4)$: $cv_3 + av_4$ occurs twice.
- $(\mathbf{t}_5, \mathbf{t}_6)$: This pair does not include a common subexpression.

The gain becomes

$$gain^{2,0} = 0 + 0 + (2 - 1) + 0 = 1 \tag{4.12}$$

according to the selected pair order.

$\mathbf{t}_6$ and $\mathbf{t}_7$ are randomly picked in the first attempt of the improvement phase. They are temporarily exchanged to form the pair order $(\mathbf{t}_0, \mathbf{t}_2)$, $(\mathbf{t}_1, \mathbf{t}_6)$, $(\mathbf{t}_3, \mathbf{t}_4)$, $(\mathbf{t}_5, \mathbf{t}_7)$. The common subexpressions of the pair order is listed as follows:

- $(\mathbf{t}_0, \mathbf{t}_2)$: This pair does not include a common subexpression.
- $(\mathbf{t}_1, \mathbf{t}_6)$: $cv_1 + av_6$ occurs twice.
- $(\mathbf{t}_3, \mathbf{t}_4)$: $cv_3 + av_4$ occurs twice.
- $(\mathbf{t}_5, \mathbf{t}_7)$: This pair does not include a common subexpression.

The gain becomes

$$gain^{2,1} = 0 + (2 - 1) + (2 - 1) + 0 = 2 \tag{4.13}$$

according to the selected pair order. The exchange attempt of $\mathbf{t}_6$ and $\mathbf{t}_7$ is accepted as $gain^{2,1}$ is greater than $gain^{2,0}$. The pair order $(\mathbf{t}_0, \mathbf{t}_2)$, $(\mathbf{t}_1, \mathbf{t}_6)$, $(\mathbf{t}_3, \mathbf{t}_4)$, $(\mathbf{t}_5, \mathbf{t}_7)$ is marked as the current result.

The next improvement attempt is temporarily exchanging $\mathbf{t}_2$ and $\mathbf{t}_5$ on the current pair order $(\mathbf{t}_0, \mathbf{t}_2)$, $(\mathbf{t}_1, \mathbf{t}_6)$, $(\mathbf{t}_3, \mathbf{t}_4)$, $(\mathbf{t}_5, \mathbf{t}_7)$. The common subexpressions of the pair order $(\mathbf{t}_0, \mathbf{t}_5)$, $(\mathbf{t}_1, \mathbf{t}_6)$, $(\mathbf{t}_3, \mathbf{t}_4)$, $(\mathbf{t}_2, \mathbf{t}_7)$ are listed as follows:

- $(\mathbf{t}_0, \mathbf{t}_5)$: This pair does not include a common subexpression.
- $(\mathbf{t}_1, \mathbf{t}_6)$: $cv_1 + av_6$ occurs twice.
- $(\mathbf{t}_3, \mathbf{t}_4)$: $cv_3 + av_4$ occurs twice.
- $(\mathbf{t}_2, \mathbf{t}_7)$: This pair does not include a common subexpression.

The gain becomes

$$gain^{2,2} = 0 + (2-1) + (2-1) + 0 = 2 \tag{4.14}$$

according to the selected pair order. The improvement attempt is rejected as the new elimination number 2 equals the last marked gain, i.e. $gain^{2,1}$.

The second iteration is limited to two improvement attempts. The pair order $(\mathbf{t}_0, \mathbf{t}_2)$, $(\mathbf{t}_1, \mathbf{t}_6)$, $(\mathbf{t}_3, \mathbf{t}_4)$, $(\mathbf{t}_5, \mathbf{t}_7)$ is marked as the result of the second iteration. $\mathbf{y}_p^2$ is built as

$$\mathbf{y}_p^2 = \mathbf{T}_p^2 \mathbf{s}^2 = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} cv_1 + av_6 \\ cv_3 + av_4 \end{bmatrix}. \tag{4.15}$$

The common subexpressions are removed from $\mathbf{T}_r^1$ to produce $\mathbf{T}_r^2$. Then, $\mathbf{T}_r^2\mathbf{v}$ can be written as

$$\mathbf{y}_r^2 = \mathbf{T}_r^2 \mathbf{v} = \begin{bmatrix} av_0 \\ 0v_0 \\ 0v_0 \\ 0v_0 \\ bv_0 \\ 0v_0 \\ 0v_0 \\ av_0 \end{bmatrix} + \begin{bmatrix} 0v_1 \\ av_1 \\ 0v_1 \\ 0v_1 \\ 0v_1 \\ 0v_1 \\ 0v_1 \\ 0v_1 \end{bmatrix} + \begin{bmatrix} 0v_2 \\ 0v_2 \\ 0v_2 \\ bv_2 \\ av_2 \\ cv_2 \\ 0v_2 \\ 0v_2 \end{bmatrix} + \begin{bmatrix} 0v_3 \\ 0v_3 \\ 0v_3 \\ 0v_3 \\ 0v_3 \\ 0v_3 \\ 0v_3 \\ 0v_3 \end{bmatrix} + \begin{bmatrix} 0v_4 \\ 0v_4 \\ cv_4 \\ 0v_4 \\ 0v_4 \\ 0v_4 \\ 0v_4 \\ 0v_4 \end{bmatrix} + \begin{bmatrix} 0v_5 \\ cv_5 \\ 0v_5 \\ bv_5 \\ 0v_5 \\ cv_5 \\ 0v_5 \\ 0v_5 \end{bmatrix} + \begin{bmatrix} cv_6 \\ 0v_6 \\ 0v_6 \\ 0v_6 \\ 0v_6 \\ 0v_6 \\ 0v_6 \\ 0v_6 \end{bmatrix} + \begin{bmatrix} 0v_7 \\ 0v_7 \\ av_7 \\ 0v_7 \\ 0v_7 \\ 0v_7 \\ 0v_7 \\ cv_7 \end{bmatrix}. \tag{4.16}$$

The search of the common subexpressions ends as the number of iterations is kept as two for the example. The result of **Tv** is written as

$$\mathbf{y} = \mathbf{Tv} = (\sum_{i=1}^{It} \mathbf{y}_p^i) + \mathbf{y}_r^{It}, \tag{4.17}$$

where It is the number of iterations. So, the results of all iterations, $\mathbf{y}_p^i$, and the second remainder matrix which is the last remainder matrix, $\mathbf{y}_r^{lt}$, are proceeded to the compression step explained in the next section.

The time complexity of the algorithm is analyzed for a given $N \times M$ matrix, iteration number and attempt number for each iteration. In the worst case, the column pair does not include any zero elements and any duplicated element pairs. Then, the first element pair is compared with $N-1$ element pairs, where N is the number of rows. After it is removed from the list, the second picked element pair is compared with $N-2$ element pair. The sequence continues until the last element pair is removed. The summation of the sequence indicates the number of element pair comparisons while calculating the gain in the worst case. It is shown as

$$\sum_{i=1}^{N-1}(N-i) = \frac{N(N-1)}{2} = \frac{N^2-N}{2}. \tag{4.18}$$

The iteration number and attempt numbers indicate how many pairs are processed. The initial phase of an iteration considers $\frac{M}{2}$ pairs where M is the column number. The user defines the number of iterations and the number of attempts for each iteration. Note that each iteration may perform a different number of attempts. However, the distribution of the attempts over iterations is not considered in the worst case. Only the total number of attempts takes place in the time complexity notation, as each attempt is assumed to consume the same time duration. Let At be the total number of attempts in all iterations. Overall pair checking number is shown as

$$(It \times \frac{M}{2}) + At. \tag{4.19}$$

The multiplication of both equations produces the time complexity of the proposed algorithm for the worst case. It is built as $\mathcal{O}(N^2(ItM + At))$ when the constants are

removed. The user may estimate the run-time of the method by tuning the iteration and attempt numbers. Note that the assumptions of the worst case may not hold in an average run. The attempts in the later iterations may take less time than those in earlier iterations as the density of the matrix may reduce through the iterations.

4.2. Matrix Compression

The constant matrix is compressed into one-dimensional arrays according to the CSE results. Three units are introduced. The Intermediate Values Unit (IVU) receives the input vector and performs all multiplication operations, shown in Equation (4.4), to prepare the *multiplication result array* (MRA). Pair Copy Unit (PCU) receives it and produces the first part of the result vector. This part contains the summation of the results of all CSE iterations shown as $\sum_{i=1}^{It} \mathbf{y}_p^i$. Element Copy Unit (ECU) receives the first part of the result vector and MRA. It accumulates the remaining elements, shown as $\mathbf{y}_r^{It}$, on the received part of the result vector. This operation produces the result vector.

IVU requires two one-dimensional arrays named *unique elements array* (UEA) and *unique elements separator array* (UESA). The values of $\mathbf{T}$ are column-wise sorted, and duplicated elements are removed to build the UEA. The end position of each column is recorded into the UESA. IVU processes those arrays to build the MRA. Three arrays are filled according to $\mathbf{T}$ in Equation (4.3) and listed as follows:

- UEA: | $a\ b\ c\ a\ b\ c\ a\ b\ c\ a\ b\ c\ a\ b\ c\ a\ b\ c\ a\ b\ c\ a\ b\ c$ |.
- UESA: | 3 6 9 12 15 18 21 24 |.
- MRA: | $av_0\ bv_0\ cv_0\ av_1\ bv_1\ cv_1\ av_2\ bv_2\ cv_2\ av_3\ bv_3\ cv_3\ av_4\ bv_4\ cv_4\ av_5\ bv_5\ cv_5\ av_6\ bv_6\ cv_6\ av_7\ bv_7\ cv_7$ |.

The MRA size equals the UEA size, and the UESA size equals the number of columns. Pseudocode of IVU is shown in Figure 4.1.

Input: Vector[]

Output: Multiplication Result[]

1: $multiplication\_result[] = \{0\}$
2: $unique\_elements[] = \{...\}$ {Predefined}
3: $unique\_elements\_separator[] = \{...\}$ {Predefined}
4: $start = 0$
5: $end = 0$
6: **for** $j = 0$ **to** $j < Column$ **do**
7: $start = end$
8: $end = unique\_elements\_separator[j]$
9: **for** $k = start$ **to** $k < end$ **do**
10: $value = unique\_elements[k]$
11: $multiplication\_result[k] = value * vector[j]$
12: **end for**
13: **end for**
14: **return** $multiplication\_result[]$

Figure 4.1. Pseudocode of Intermediate Values Unit.

PCU calculates the results of the common subexpressions and accumulates them on $\mathbf{y}^i$. The pairs acquired from $\mathbf{T}_p^i$, $\mathbf{s}^i$ and filled into two one-dimensional arrays named *copy pairs array* (CPA) and *copy pairs separator array* (CPSA). The common subexpressions are grouped and saved into the CPA. A pair group includes one element from the $\mathbf{s}^i$. An element contains two values from the MRA. Indices of those values are written into a pair group as the first two elements. Then, the row indices of the corresponding column in $\mathbf{T}_p^i$ are appended to the pair group. The pair groups are listed in the CPA. The order of the pairs are not important. The end position of each pair group is recorded in the CPSA. Two arrays are filled according to the results of the iterations in Equations (4.10), (4.15) and listed as follows:

- CPA: | 2 19 2 3 5 4 15 0 2 7 6 13 1 6 7 9 22 3 6 10 23 1 4 5 18 4 6 11 12 0 5 |.
- CPSA: | 5 10 15 19 23 27 31 |.

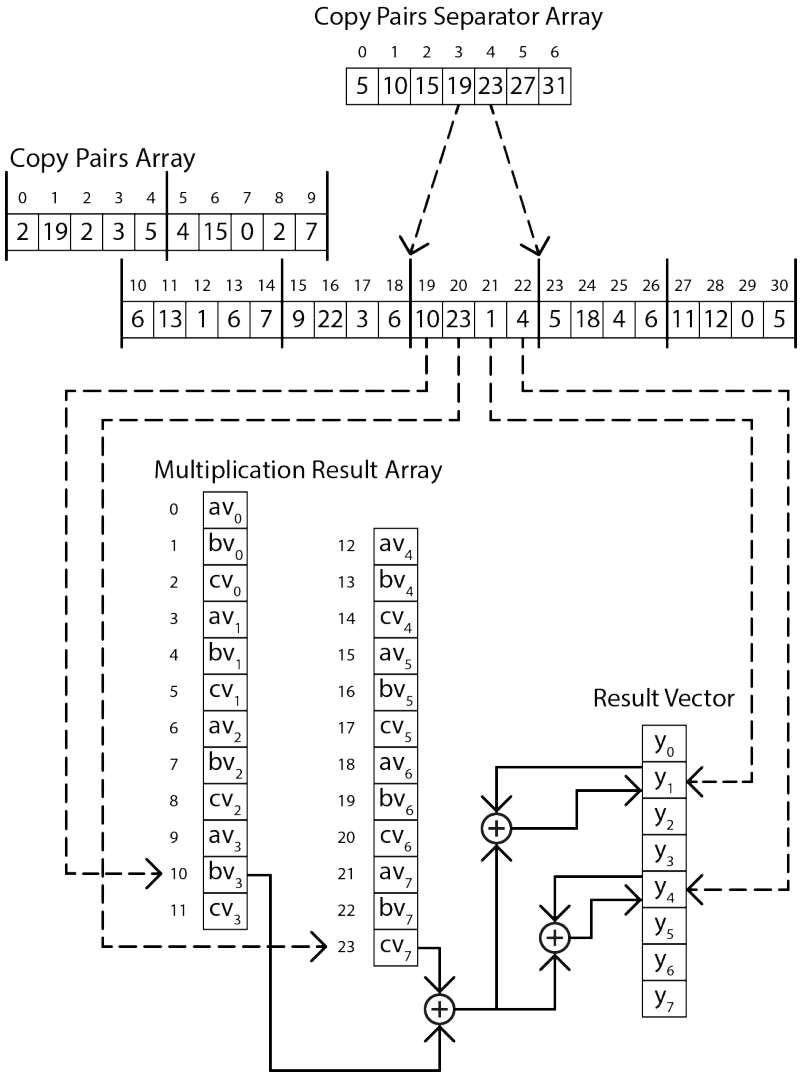


Figure 4.2. An Example Process in PCU.

The CPSA size equals the number of pair groups. It is not related to the row number or column number. Processing the CPSA elements 19 and 23 are illustrated in Figure 4.2 to show the processing procedure of the pair $bv_3 + cv_7$. The CPSA values 19 and 23 show the CPA indices from 19 to 22. The first two elements, 10 and 23, show the MRA elements to be summed. The last two values, 1 and 4, show that the addition is accumulated on y_1 and y_4. Pseudocode of PCU is shown in Figure 4.3.

Input: Multiplication Result[]

Output: Result Vector[]

1: $result\_vector[] = \{0\}$
2: $copy\_pairs[] = \{...\}$ {Predefined}
3: $copy\_pairs\_separator[] = \{...\}$ {Predefined}
4: $pairs\_number = \{...\}$ {Predefined}
5: $start = 0$
6: $end = 0$
7: **for** $i = 0$ **to** $i < pairs\_number$ **do**
8: $start = end$
9: $end = copy\_pairs\_separator[i]$
10: $index1 = copy\_pairs[start]$
11: $index2 = copy\_pairs[start + 1]$
12: $value1 = multiplication\_result[index1]$
13: $value2 = multiplication\_result[index2]$
14: $sum = value1 + value2$
15: **for** $k = start + 2$ **to** $k < end$ **do**
16: $index = copy\_pairs[k]$
17: $result\_vector[index]$ += sum
18: **end for**
19: **end for**
20: **return** $result\_vector[]$

Figure 4.3. Pseudocode of Pair Copy Unit.

PCU produces $\mathbf{y}_p = \sum_{i=1}^{It} \mathbf{y}_p^i$ and sends it to ECU. ECU accumulates the remaining elements, $\mathbf{y}_r^{It}$ on $\mathbf{y}_p$ to produce $\mathbf{y}$ by processing two arrays named *copy elements array* (CEA) and *copy elements separator array* (CESA). The remaining elements are row-wise ordered. Each element corresponds to a value from the MRA. Corresponding indices are recorded into the CEA. The end position of each row is recorded into the CESA. Both arrays are filled according to the remaining elements shown in Equation (4.16) and listed as follows:

- Copy elements array: | 0 20 3 17 14 21 7 16 1 6 8 17 0 23 |.
- Copy elements separator array: | 2 4 6 8 10 12 12 14 |.

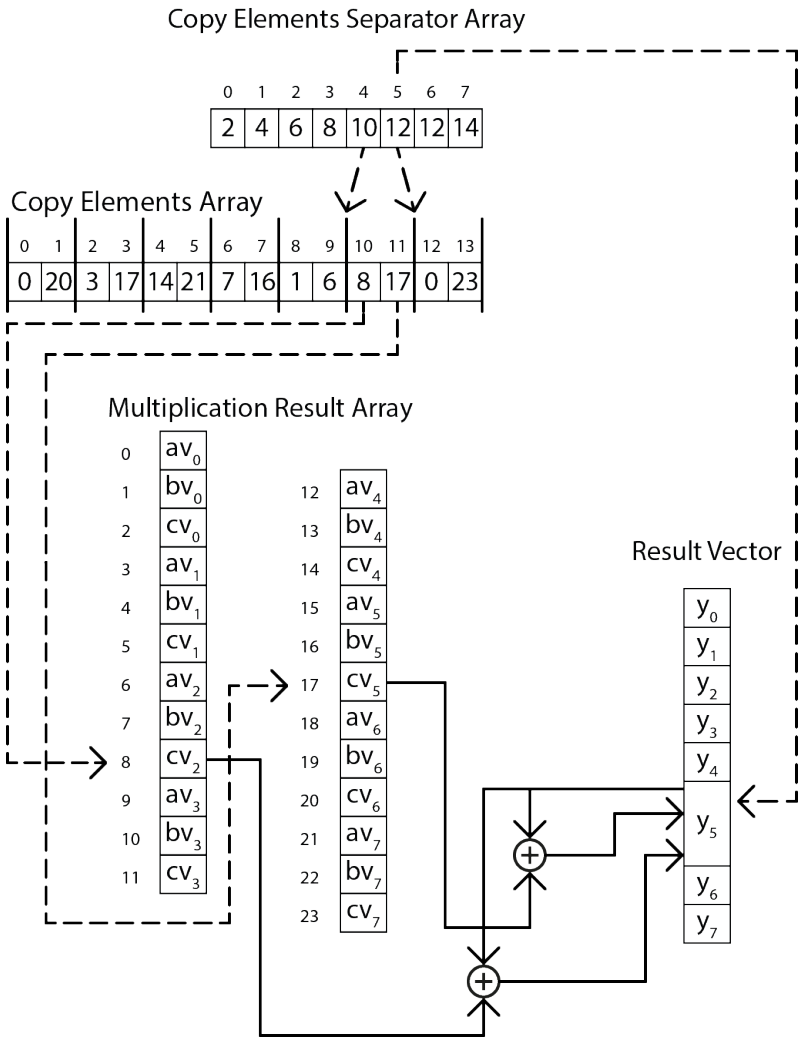


Figure 4.4. An Example Process in ECU.

Note that the sixth row does not contain any remaining elements. The CESA contains 12 twice for this reason. The CESA size equals the number of rows. Processing the 5^{th} row of $\mathbf{y}_r^2$ is illustrated in Figure 4.4. The fourth and fifth indices of the CESA indicate the start position inclusively and end position exclusively of the 5^{th} row of $\mathbf{y}_r^2$ in the CEA, respectively. This region contains the values 8 and 17. The values in the 8^{th} and 17^{th} indices of the MRA are acquired. They are accumulated on the 5^{th} index of the CESA one by one. Pseudocode of ECU is shown in Figure 4.5.

Input: Multiplication Result[], Result Vector[]

Output: Result Vector[]

```
1: copy_elements[] = {...} {Predefined}
2: copy_elements_separator[] = {...} {Predefined}
3: start = 0
4: end = 0
5: for i = 0 to i < Row do
6:    start = end
7:    end = copy_elements_separator[i]
8:    for k = start to k < end do
9:       index = copy_elements[k]
10:      value = multiplication_result[index]
11:      result_vector[i] += value
12:   end for
13: end for
14: return result_vector[]
```

Figure 4.5. Pseudocode of Element Copy Unit.

The output of ECU equals **y**. As a result, the matrix multiplication is performed with six constant arrays and one temporary MRA. Array sizes depend on the matrix elements and CSE solution.

5. EXPERIMENTS

Sample matrices are constructed for given $N \times M$, UV and NZR for the experiments. The non-zero elements are equally distributed to each row. The non-zeros inside a row are shuffled with a uniform distribution. Three groups of experiments are performed. Firstly, three random distributions are tested for the improvement phase of the proposed method. The number of additions is measured on the proposed method as the idea is specific to the search algorithm of the proposed method. Secondly, the state-of-the-art methods, the proposed method and the baseline are compared in terms of the number of additions and latency. The baseline contains a nested for-loop for two-dimensional matrix multiplication. Thirdly, a set of experiments are conducted to investigate the performance of the proposed method on various matrix parameters such as $N \times M$, UV and NZR against the baseline. The number of additions, latency and storage size are measured. The proposed method is indicated with "P". The methods from [9] and [10] are indicated with "[9]" and "[10]", respectively. The baseline is identified with "F".

The state-of-the-art methods produce an adder tree as a two-dimensional matrix. The adder tree for the state-of-the-art methods and the constant matrix for the baseline are processed with the pseudocode provided in Figure 5.1. The number of multiplications and additions are calculated according to the pseudocode for the state-of-the-art methods and the baseline. Zeros are ignored, and each non-zero is counted as an addition and multiplication.

The number of multiplications is not measured in the experiments as the state-of-the-art methods and baseline do not include a mechanism to reduce the number of multiplications. The number of multiplications is equal to the NNZ for the state-of-the-art methods and baseline. It equals $\sum_{j=1}^{M} UV_j \times M$. Note that the number of multiplications is equal to UV $\times$ M in all sample matrices as all columns contain all of the unique values.

Input: Vector[]

Output: Result Vector[]

```
1: result_vector[] = {0}
2: matrix[][] = {...} {Predefined}
3: for i = 0 to i < row_number do
4:    for j = 0 to j < column_number do
5:       if matrix[i][j] != 0 then
6:          result_vector[i] += matrix[i][j] * vector[j]
7:       end if
8:    end for
9: end for
10: return result_vector[]
```

Figure 5.1. Pseudocode of Matrix Multiplication.

The state-of-the-art methods produce a matrix according to their notation. The NNZ of the produced matrix is equal to the number of additions for the state-of-the-art methods. It is equal to the NNZ for the baseline. It is equal to |CEA| + |CPA| − |CPSA| for the proposed method. |CEA| indicates the NNZ of the remaining elements matrix, $\mathbf{y}_r^{It}$. |CPA| − |CPSA| indicates the total number of additions of all pair groups.

The storage size is not measured for the state-of-the-art methods because their notation increases the matrix size, and they do not provide a compression format. The storage size is equal to N × M for the baseline. However, it can be measured according to the CSR format for the baseline. So, zeros are not counted for the storage size in the baseline. It is calculated as 2 × N × M × NZR + N for the CSR format. The storage size equals |UEA| + |UESA| + |CPA| + |CPSA| + |CEA| + |CESA| for the proposed method.

The gem5 simulation tool is used to simulate the matrix multiplication on a CPU [8]. The same simulation configuration is used for all experiments. TimingSimpleCPU is used to calculate the latency of the matrix multiplication. It runs x86 instructions. L1 instruction cache and L1 data cache sizes are selected as 256 kB for both. L2 cache

size is selected as 1 MB. A single channel DDR3 1600 MHz RAM is used. CPU clock is set to 1 GHz. So, the number of CPU cycles mentions the latency of a method. The miss rates of three caches are provided in the results to show that if a memory shortage occurs in any method. Note that the simulator accumulates the CPU cycles for each system call to calculate the total number of CPU cycles passed during the program. However, some of the calls are missing in the implementation. Therefore, the results may differ in a real hardware implementation.

5.1. Random Distribution Options for the Improvement Phase

The columns are randomly picked in the improvement phase. The distribution of the random number generator may affect the improvement phase results. An idea is introduced to test this thought. The pairs are sorted by their gains at the start of each improvement attempt. A biased random distribution is used to break and tie low-gained pairs more frequently than high-gained pairs. Three cumulative distribution functions (CDF) are selected. The first one is Rayleigh CDF

$$F(x) = 1 - e^{\frac{-x^2}{2\sigma^2}} \text{ where } x \in [0.0, 10.0] \tag{5.1}$$

includes the parameter σ. The second one is Exponential CDF

$$F(x) = 1 - e^{-\lambda x} \text{ where } x \in [0.0, 1.0] \tag{5.2}$$

includes the parameter λ. The third one is Continuous Uniform CDF

$$F(x) = \begin{cases} 0, & \text{if } x < a \\ \frac{x-a}{b-a}, & \text{if } x \in [a, b] \text{ where } x \in [0.0, 1.0] \\ 1, & \text{if } x > b \end{cases} \tag{5.3}$$

includes the parameters a and b. The parameter a is set to zero in all experiments. Three CDFs are illustrated in Figure 5.2 to show that all of them serve high probability to the pairs with low gain.

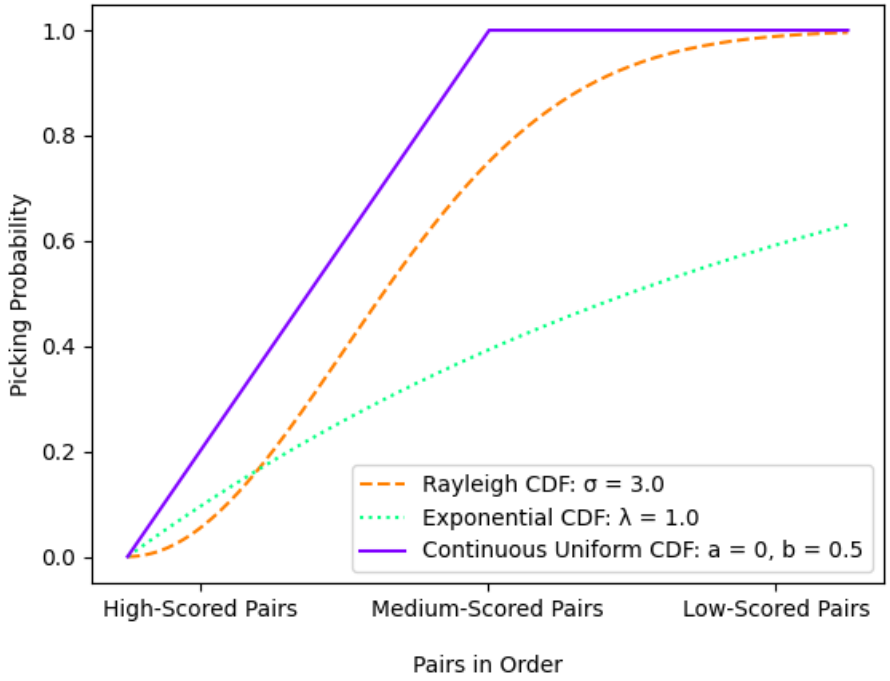
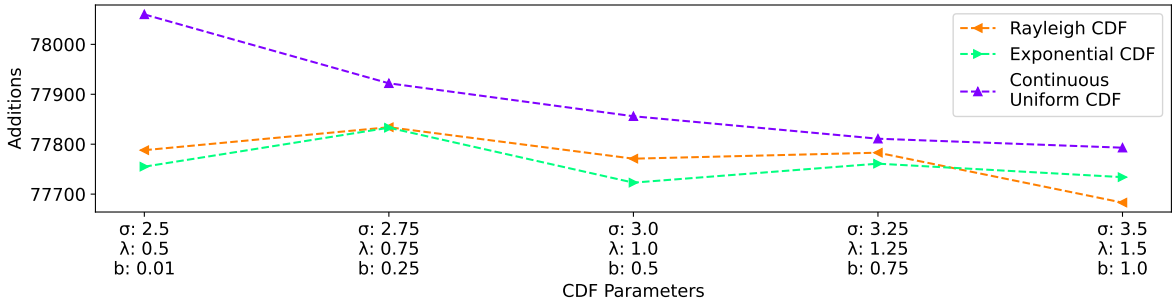
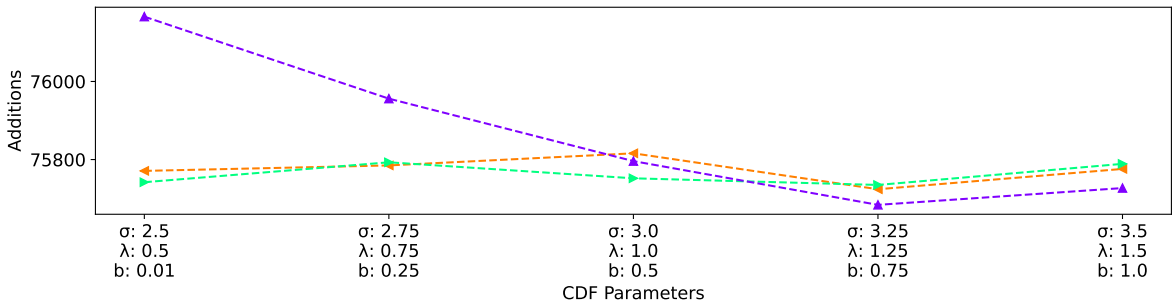


Figure 5.2. An example illustration of three CDFs.

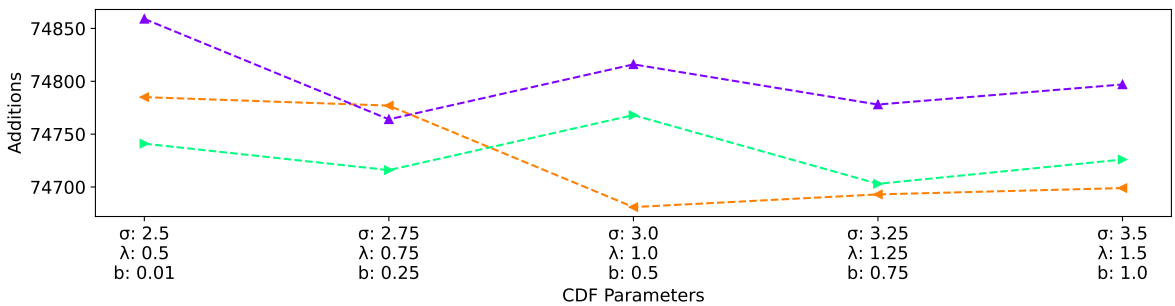
Three CDFs are tested on the same constant matrix for consistency. Its parameters are set as $N \times M = 500 \times 500$, UV $= 2$ and NZR $= 0.5$. The iteration and improvement attempt numbers, It and At, are selected as $10-100$, $10-1000$, $100-100$ and $100-1000$, respectively. The number of additions is calculated in Figure 5.3. The parameter values of three CDFs σ, λ and b are shown on the $X-$axis. The Continuous Uniform CDF with $b = 0.01$ is selected to simulate the uniform distribution. It produces the lowest result in all cases. For this reason, using a biased distribution function achieves better results than the uniform distribution. Note that the results change in every run if different seeds are fed to the random number generator. The results show that each option indicates a different CDF for the lowest addition number. The results may change in the next run. Besides, another CDF with a different parameter value may produce the lowest addition number.



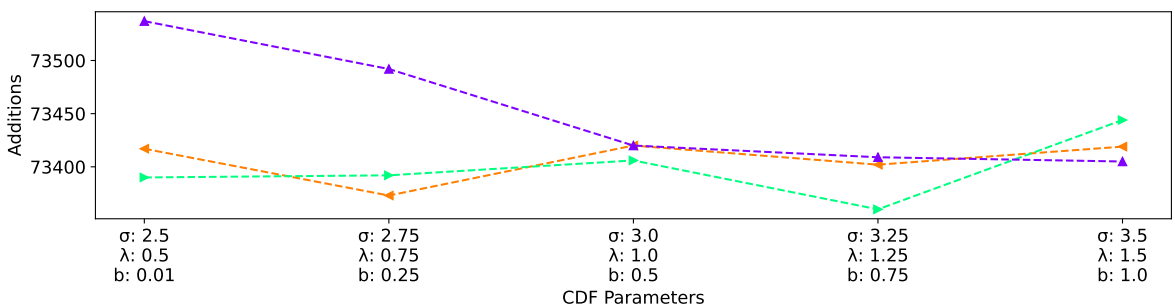
(a)



(b)



(c)



(d)

Figure 5.3. The comparison of three CDFs when UV = 2, NZR = 0.1 and $N \times M = 500 \times 500$ for a) $It = 10$, $At = 100$ b) $It = 10$, $At = 1000$ c) $It = 100$, $At = 100$ d) $It = 100$, $At = 1000$.

5.2. Comparison with the State-of-the-art Methods

The proposed method is compared with [9], [10] and the baseline. Nine matrices are prepared for the experiments. UV is set as 2 for all of them. NZR is selected as 0.25, 0.5 and 0.75. The matrix size is determined as 24×24, 50×50, 100×100. The sizes are picked small to keep the run-time of the compared methods less than one hour. The iteration (It) and attempt (At) are selected as 100 for all matrices in the proposed method. The random distribution is selected as the Rayleigh CDF with $\sigma = 3.25$. The results of the proposed method are prepared in less than ten seconds for each run.

The number of additions for all methods is illustrated in Figure 5.4. The results show that increasing NZR on the same size increases the length of the common subexpressions. Therefore the proposed method can fall behind the compared methods for a higher NZR in some experiments. On the other hand, the gap between the proposed method and state-of-the-art methods reduces as the NZR reduces.

The state-of-the-art methods require registers to save the intermediate results of the common subexpressions. The proposed method does not require registers as it considers only size-of-two common subexpressions. The baseline does not require any register as it does not include a CSE algorithm. The number of the register writes for storing intermediate results in Figure 5.5. The results show that the state-of-the-art methods require a compression mechanism. Appended rows contains two non-zero elements in [9]. For example, when two non-zeros in the appended rows are counted as two additions, the total number of additions becomes 2093 for a given 100×100 matrix with NZR = 0.25. The adder tree contains 435 appended rows. When two non-zeros in the appended row are counted as one addition, the total number of additions becomes $2093 - 435 = 1658$. So, [9] can outperform the proposed method with a compression format as the proposed method performs the matrix multiplication with 1923 additions for the same matrix. Also, matrix sizes are not listed in this section as the state-of-the-art methods do not provide a compression technique.

The number of CPU cycles is shown in Figure 5.6. The CPU cycles show that the state-of-the-art methods require a compression method for processing one-dimensional arrays to perform matrix multiplication. Although their notation reduces the addition number more than the proposed method in some experiments, their process times fall behind the proposed method and baseline. Their notation does not utilize the addition reduction in the CPU. Those methods are built for customizable hardware such as ASIC or FPGA. Their CPU implementation requires a revision to reduce the latency. The miss rates of the L1 instruction cache, L1 data cache and L2 cache are shown in Figure 5.7, Figure 5.8 and Figure 5.9, respectively. The results do not imply a significant memory shortage for any method.

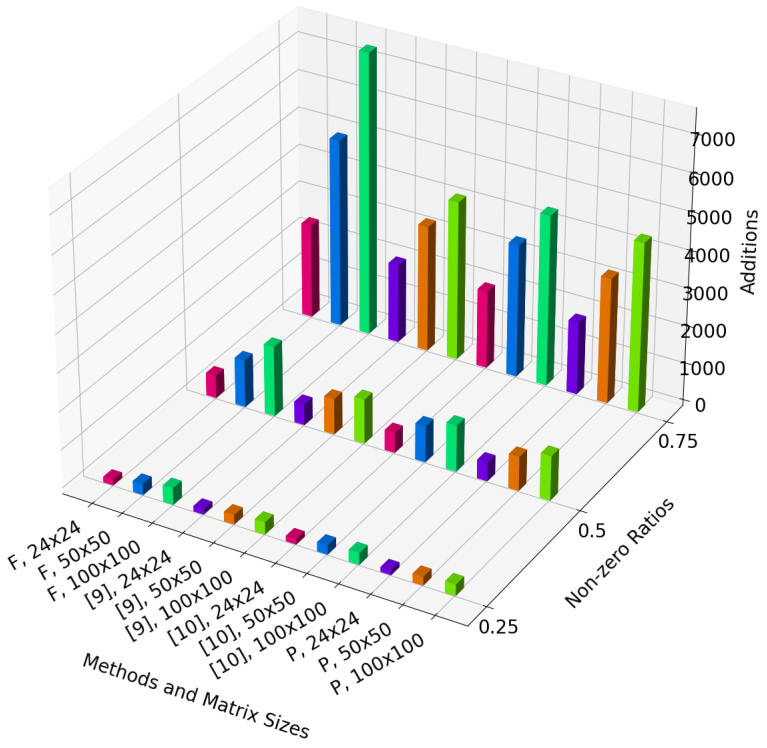


Figure 5.4. The number of additions when UV = 2. F: Baseline, P: Proposed.

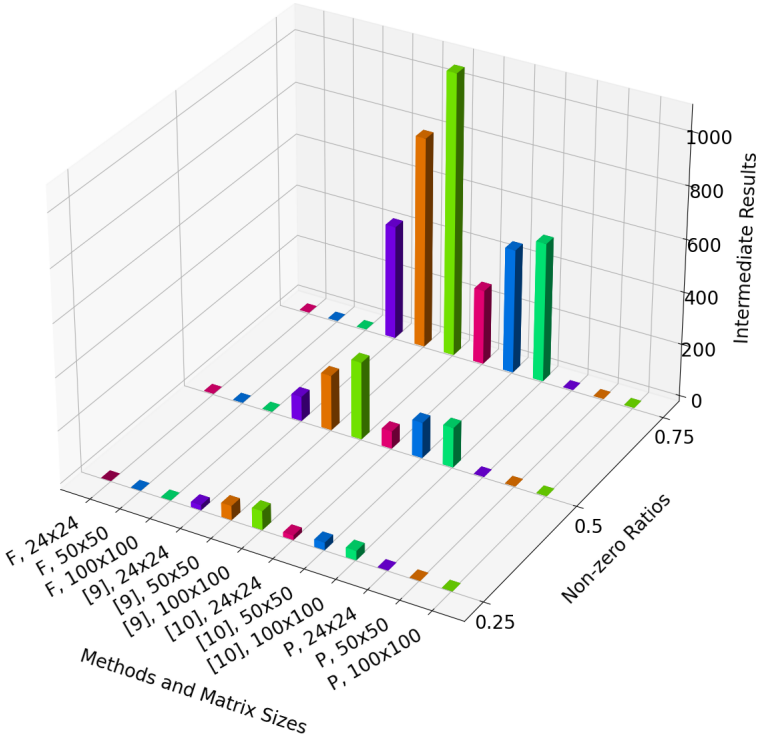


Figure 5.5. The number of intermediate results when UV = 2. F: Baseline, P: Proposed.

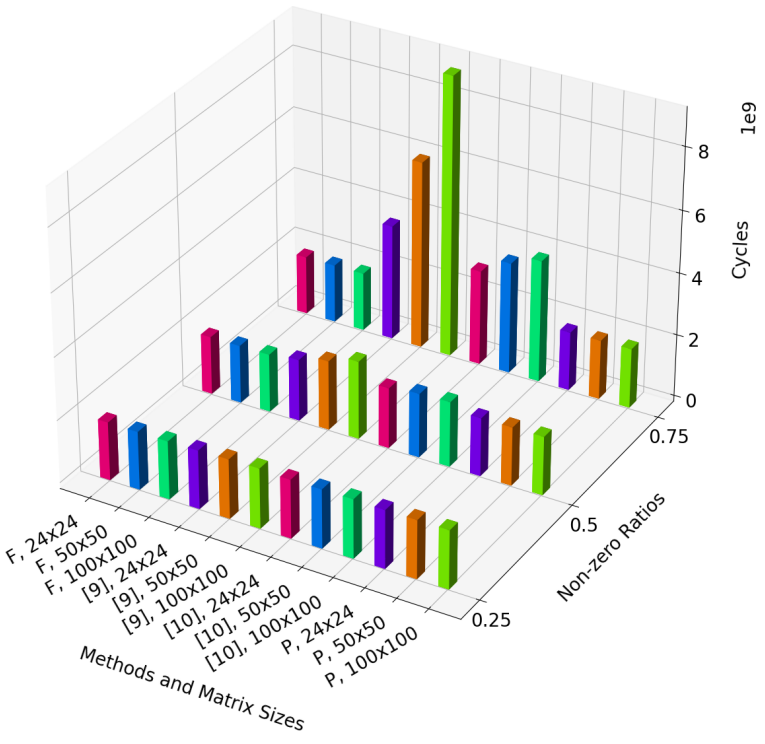


Figure 5.6. The number of cycles when UV = 2. F: Baseline, P: Proposed.

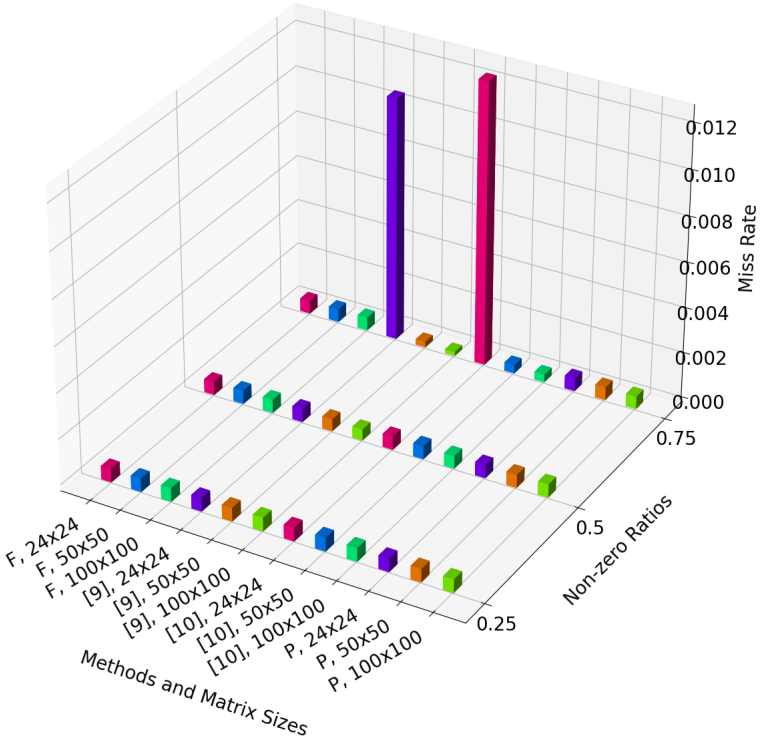


Figure 5.7. The L1 instruction cache miss rate when UV = 2. F: Baseline, P: Proposed.

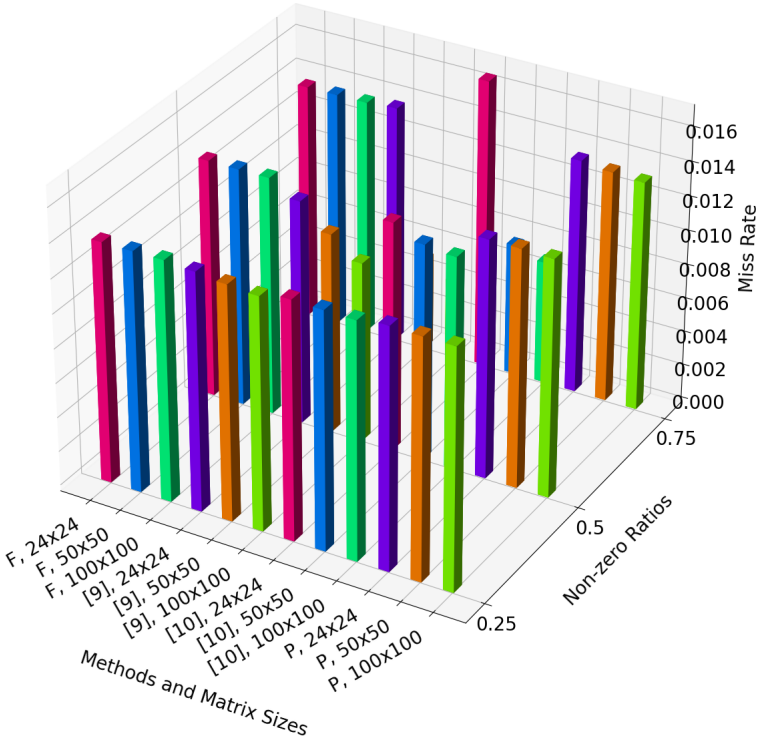


Figure 5.8. The L1 data cache miss rate when UV = 2. F: Baseline, P: Proposed.

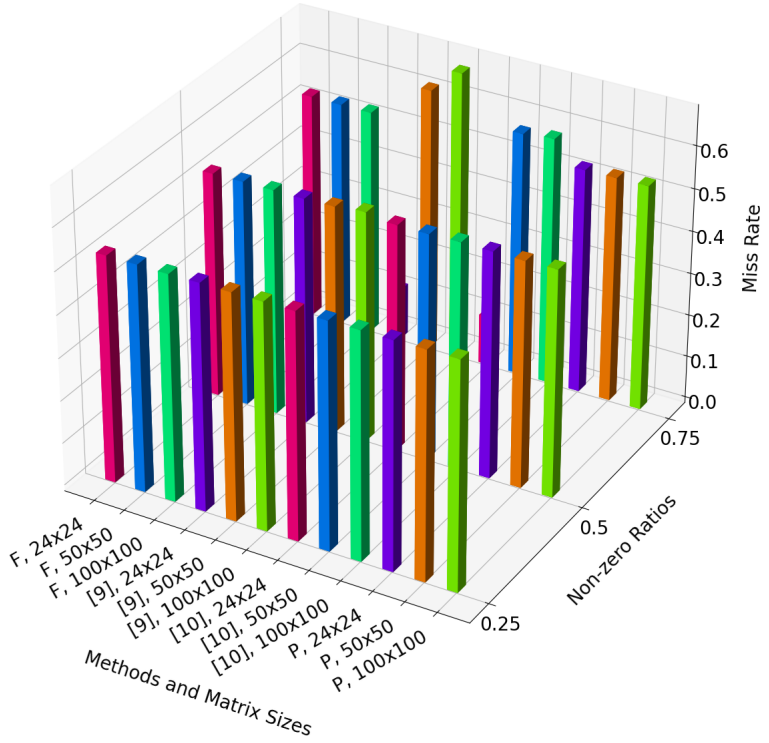


Figure 5.9. The L2 cache miss rate when UV = 2. F: Baseline, P: Proposed.

5.3. The Effects of the Matrix Properties

The effects of the matrix properties, UV, NZR and $N \times M$, are investigated on the proposed method and the baseline. The iteration (It) and attempt (At) numbers are selected as 100. The random distribution is selected as the Rayleigh CDF with $\sigma = 3.25$. The proposed method results are prepared in less than 70 seconds for each run. The state-of-the-art methods are not included in this section as they do not scale for 1000×1000 matrices. Their run-time reaches nine hours for a 200×200 matrix.

5.3.1. Unique Values

1000×1000 matrices with UV = $\{2, 4, 6, 8\}$ are constructed. NZR = 0.1 and NZR = 0.25 are used to check whether the results depend on a specific NZR. The number of additions is shown in Figure 5.10. The storage size is shown in Figure 5.11. The number of CPU cycles is shown in Figure 5.12.

The results emphasize the importance of quantization. The quantization operation increases the similarity of the coefficients. When the similarity increases, the number of additions, storage size and latency reduce in the proposed method. The results also indicate the consistency of the proposed method. One of the aims of the proposed method is to reduce the computation cost. The computation cost includes the number of additions, storage size and latency. When the similarity increases, three indicators reduce at the same time.

On the other hand, the baseline is not affected by UV change in terms of the addition number and storage size. However, the simulation results show negligible differences. For example, the cycle number increases less than 0.1% when UV increases. The miss rates of the L1 instruction cache, L1 data cache and L2 cache are shown in Figure 5.13, Figure 5.14 and Figure 5.15, respectively. The results do not imply a significant memory shortage for the proposed method and baseline.

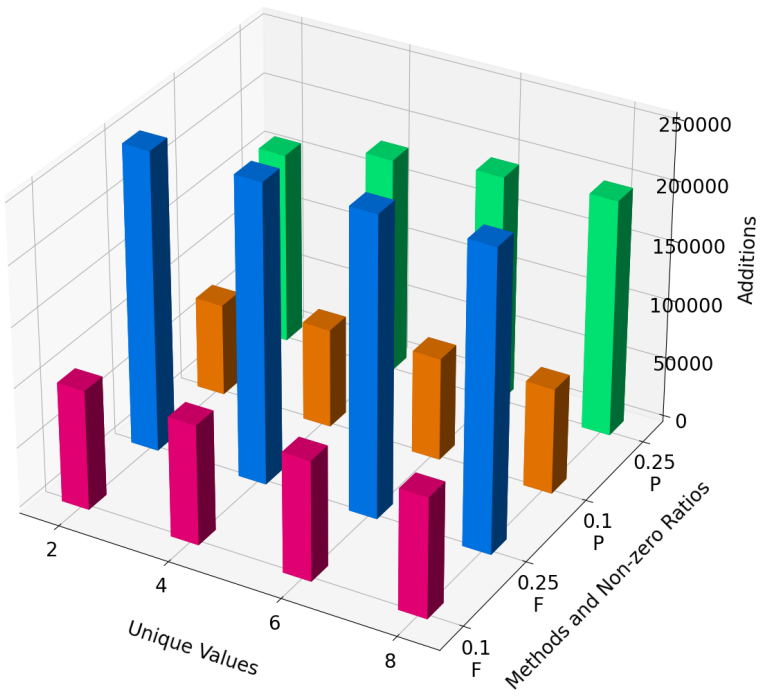


Figure 5.10. The number of additions when $N \times M = 1000 \times 1000$ for different UV. F: Baseline, P: Proposed.

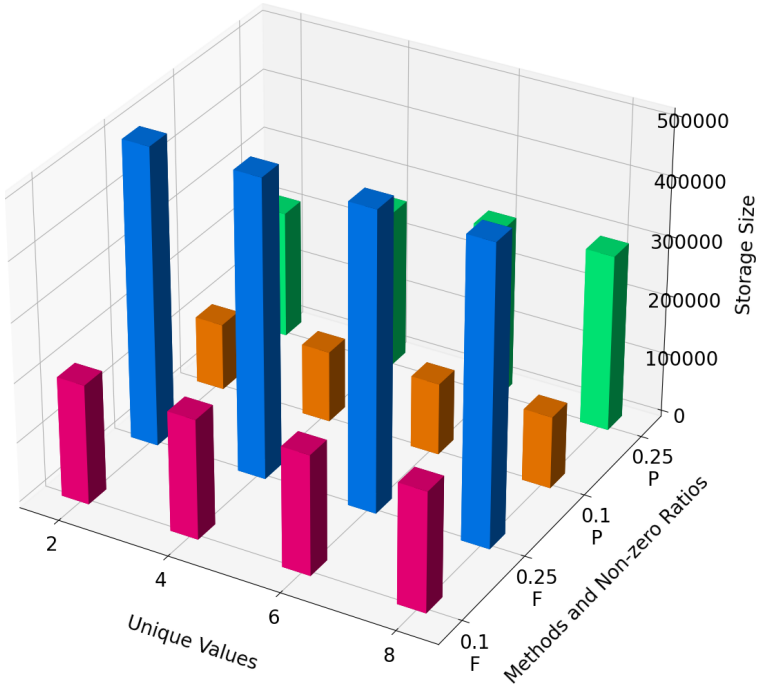


Figure 5.11. The storage size when $N \times M = 1000 \times 1000$ for different UV. F: Baseline, P: Proposed.

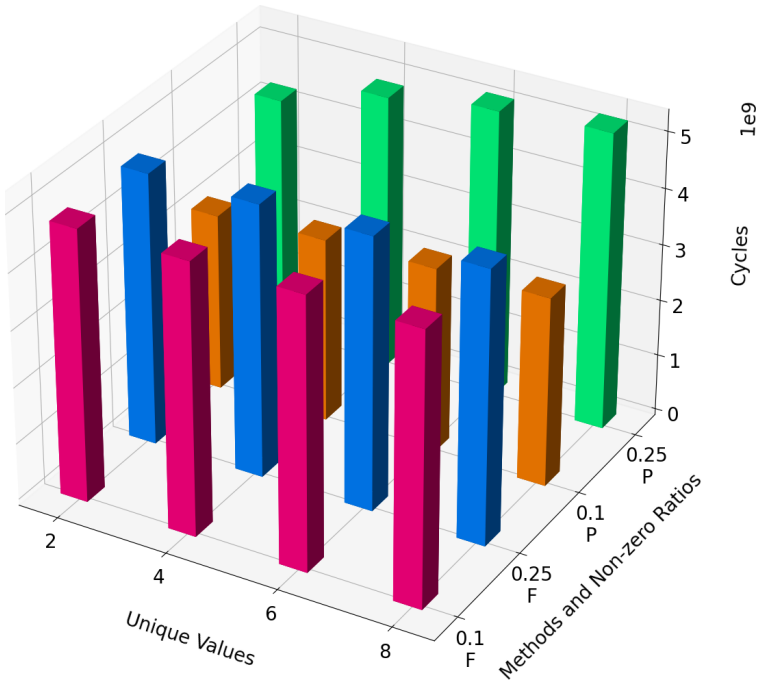


Figure 5.12. The number of cycles when $N \times M = 1000 \times 1000$ for different UV. F: Baseline, P: Proposed.

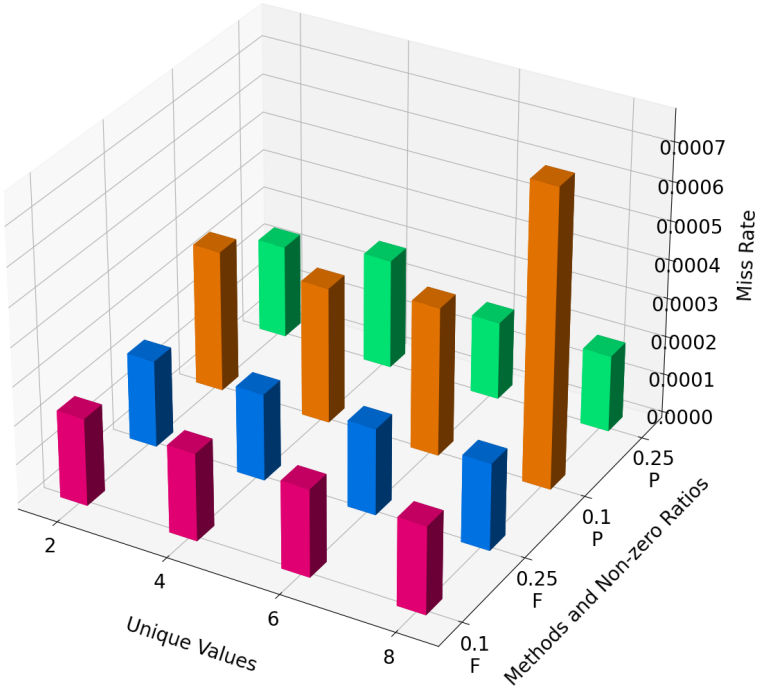


Figure 5.13. The L1 instruction cache miss rate when $N \times M = 1000 \times 1000$ for different UV. F: Baseline, P: Proposed.

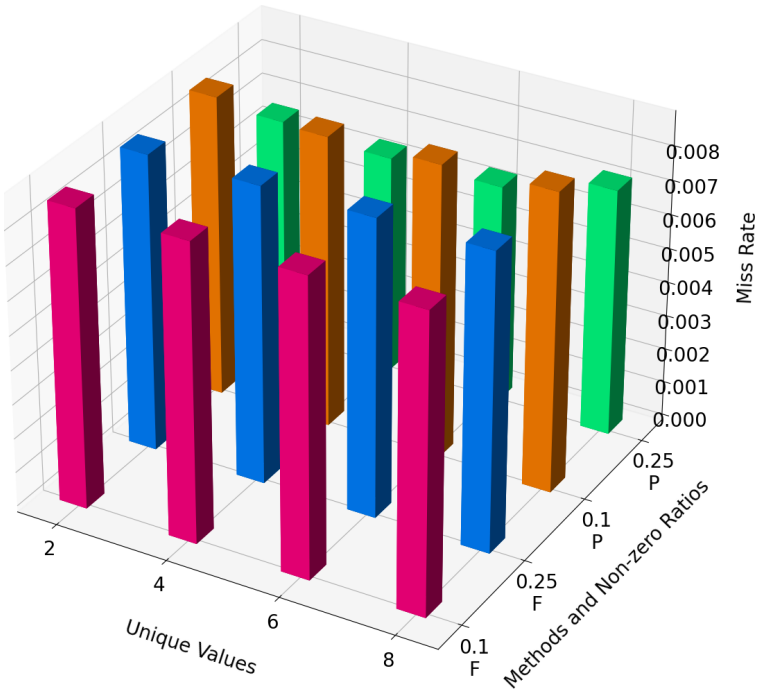


Figure 5.14. The L1 data cache miss rate when $N \times M = 1000 \times 1000$ for different UV. F: Baseline, P: Proposed.

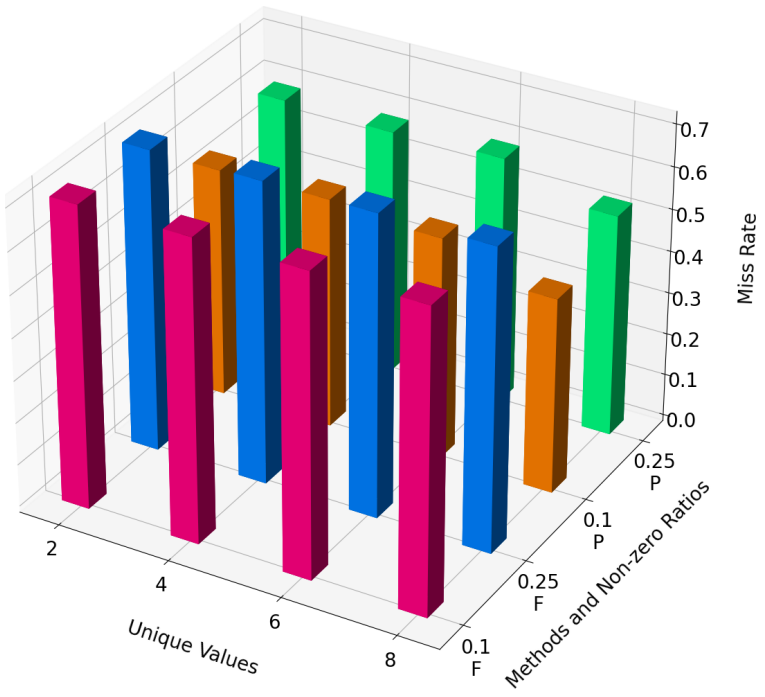


Figure 5.15. The L2 cache miss rate when $N \times M = 1000 \times 1000$ for different UV. F: Baseline, P: Proposed.

5.3.2. Non-zero Ratio

1000×1000 matrices with NZR $= \{0.1, 0.25, 0.5, 0.75\}$ are constructed. UV $= 2$ and UV $= 4$ are used to check whether the results depend on a UV. The number of additions is shown in Figure 5.16. The storage size is shown in Figure 5.17. The number of CPU cycles is shown in Figure 5.18.

The results show that the efficiency of the proposed method increases when the NZR increases. For example, a given NZR $= 0.5$ matrix contains twice the amount of non-zero elements compared to a given NZR $= 0.25$ matrix. But, the number of additions increases by 1.78 for the given UV $= 2$ matrix and 1.73 for the given UV $= 4$ matrix. In addition, a given NZR $= 0.75$ matrix contains 1.5 times of non-zero elements compared to the given NZR $= 0.5$ matrix. The addition number ratio becomes 1.42 for the given UV $= 2$ matrix and 1.37 for the given UV $= 4$ matrix. The storage size increases with a lower slope than the slope of the NNZ. The storage size for a given UV $= 2$ and NZR $= 0.1$ matrix is 201000 elements for the CSR format and 112195

elements for the proposed method. The ratio of the CSR format over the proposed method in terms of storage size is approximately 1.8. This ratio is approximately 2.3, 2.9 and 3.2 for given matrices UV = 2 and NZR = $\{0.25, 0.5, 0.75\}$, respectively.

The cycle numbers show that the latency of the baseline is approximately 4.8 seconds for NZR = $\{0.1, 0.25, 0.5, 0.75\}$ with less than 1% change between each run. The latency of the proposed method is approximately 3.2 seconds for NZR = 0.1, and it increases to 7.5 for NZR = 0.75. The curves of the baseline and proposed method intersect nearly at NZR = 0.25. So, the proposed method outperforms the base for the matrices with NZR < 0.25, and the baseline outperforms the proposed method for the matrices with NZR > 0.25. The miss rates of the L1 instruction cache, L1 data cache and L2 cache are shown in Figure 5.19, Figure 5.20 and Figure 5.21, respectively. The results do not imply that the cache sizes affect the process time for the proposed method and baseline. As a result, the pruning ratio needs to be decided by considering the changes in the latency and storage size.

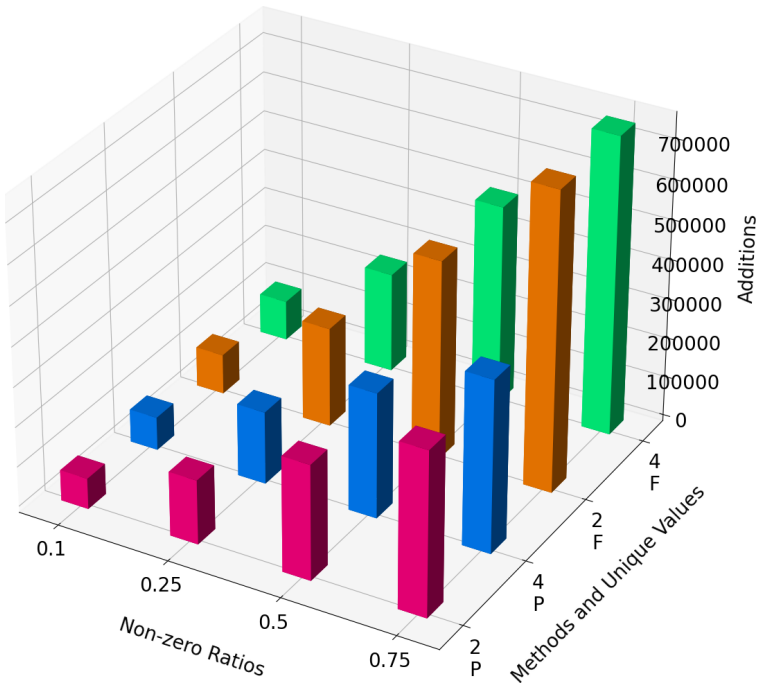


Figure 5.16. The number of additions when $N \times M = 1000 \times 1000$ for different NZR. F: Baseline, P: Proposed.

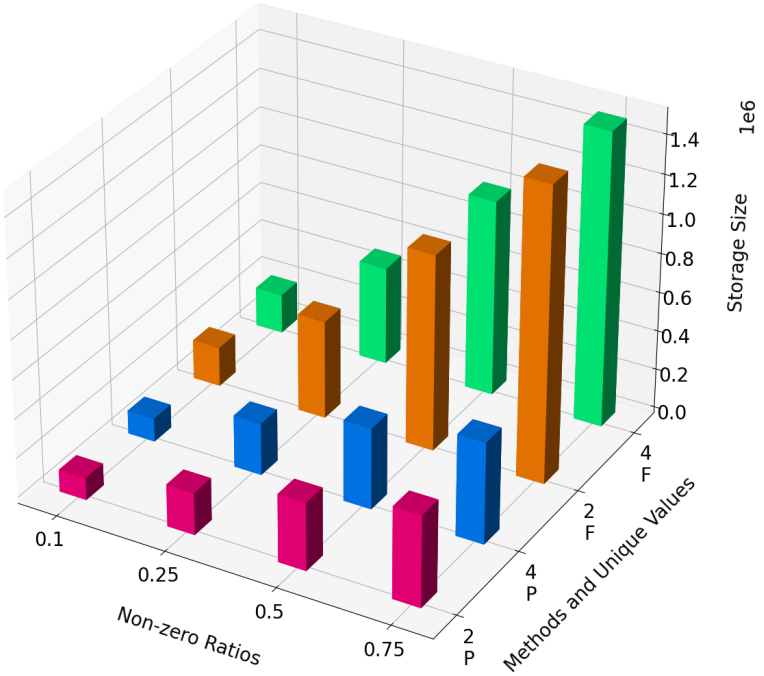


Figure 5.17. The storage size when $N \times M = 1000 \times 1000$ for different NZR. F: Baseline, P: Proposed.

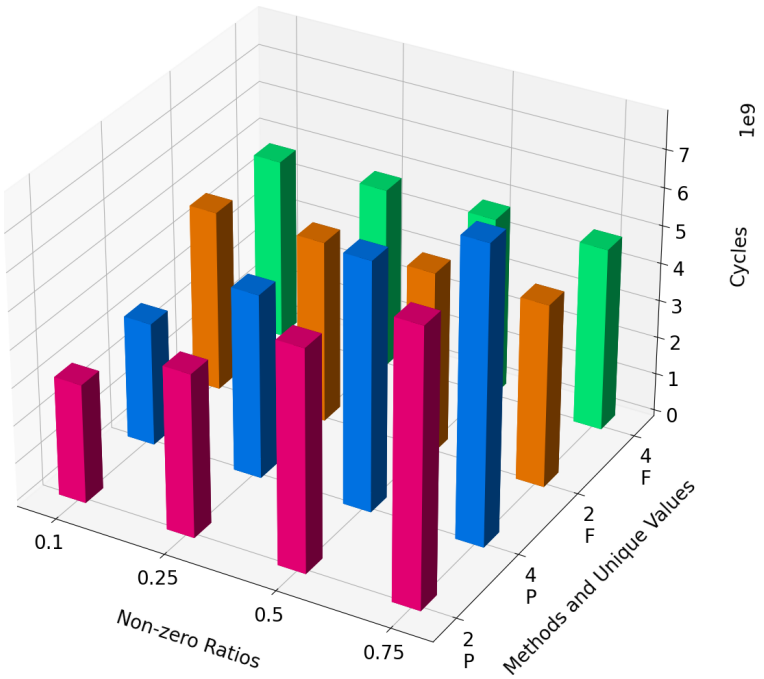


Figure 5.18. The number of cycles when $N \times M = 1000 \times 1000$ for different NZR. F: Baseline, P: Proposed.

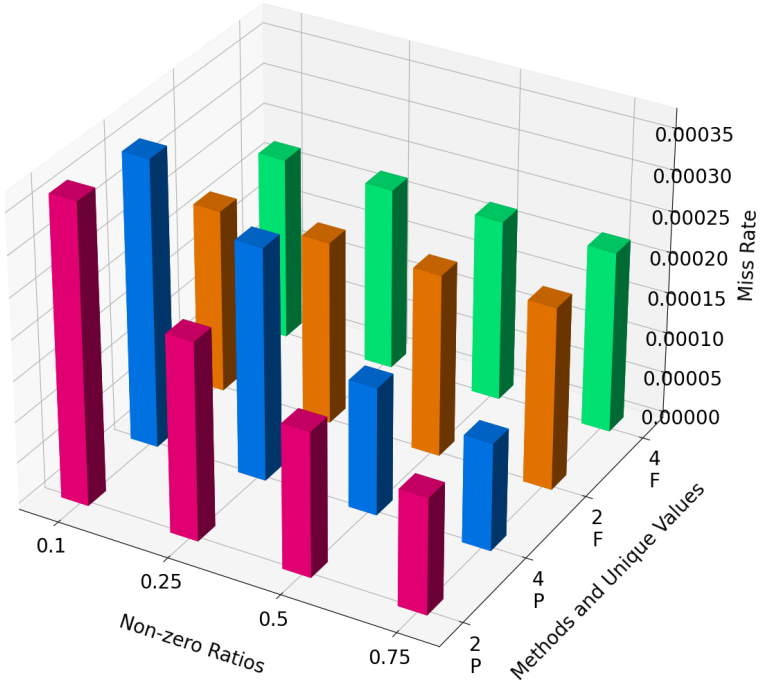


Figure 5.19. The L1 instruction cache miss rate when $N \times M = 1000 \times 1000$ for different NZR. F: Baseline, P: Proposed.

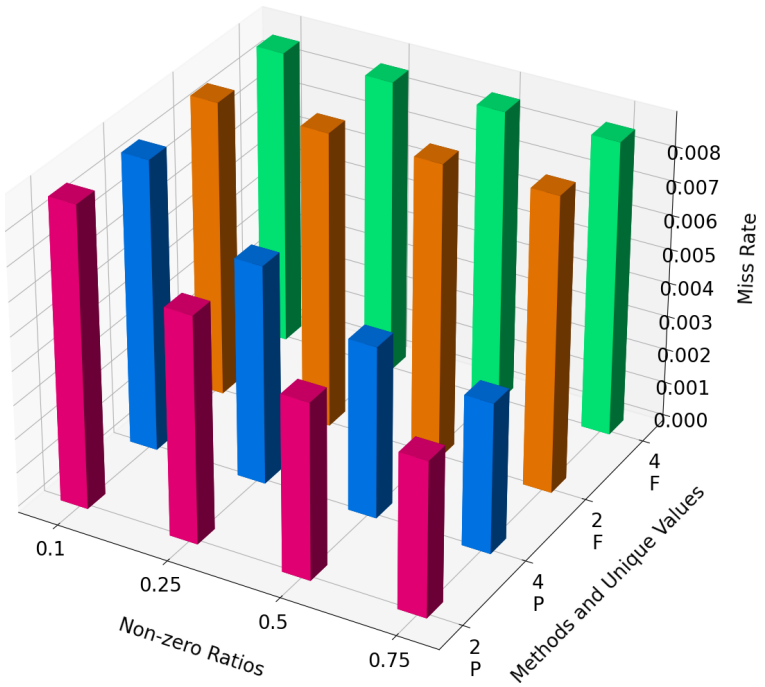


Figure 5.20. The L1 data cache miss rate when $N \times M = 1000 \times 1000$ for different NZR. F: Baseline, P: Proposed.

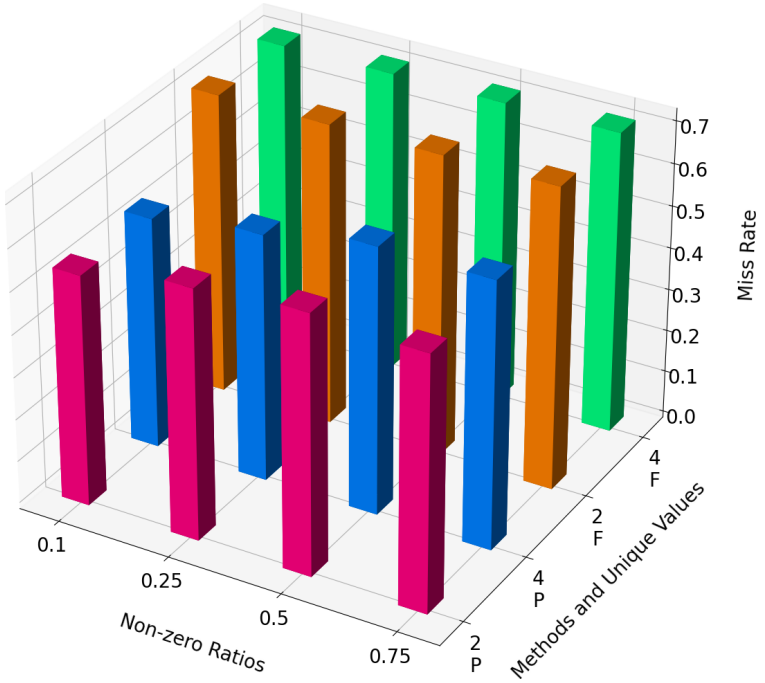


Figure 5.21. The L2 cache miss rate when $N \times M = 1000 \times 1000$ for different NZR. F: Baseline, P: Proposed.

5.3.3. Matrix Size

Several matrices of size $N \times M = \{250 \times 250, 250 \times 500, 250 \times 1000, 500 \times 250, 500 \times 500, 500 \times 1000, 1000 \times 250, 1000 \times 500, 1000 \times 1000\}$ are constructed with UV $= 2$ and NZR $= 0.1$. The additions number is shown in Figure 5.22. The storage size is shown in Figure 5.23. The number of CPU cycles is shown in Figure 5.24.

When the given 1000×250 matrix is compared with the given 250×1000 matrix, the proposed method produces 9% fewer additions, 10.8% less storage size and 1.9% less latency for the 1000×250 matrix. When the row size is doubled, the number of additions and storage size are multiplied by approximately 1.9. Besides, when the column size is doubled, the number of additions and storage size are multiplied by approximately 2.1. Doubling the column number increases the latency more than doubling the row number. The results imply that increasing the row number causes less overhead than increasing the column number in the proposed method. This property needs to be considered while restructuring the deep learning model.

Besides, doubling the row number or column number double the number of additions in the baseline. In addition, doubling the row number or column number nearly doubles the storage size in the CSR and CSC formats. Note that if the row number is higher than the column number for a given matrix, the CSC format can be used. The latency is very near for the given 250×1000 and 1000×250 matrices in the baseline. The miss rates of the L1 instruction cache, L1 data cache and L2 cache are shown in Figure 5.25, Figure 5.26 and Figure 5.27, respectively. The results show that the miss rates do not change significantly between the matrices for the proposed method and baseline.

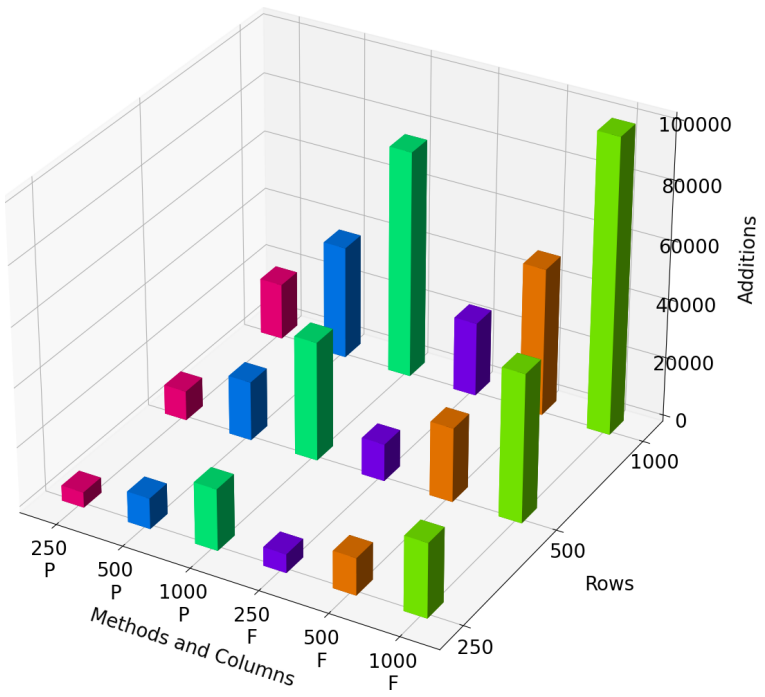


Figure 5.22. The number of additions when UV = 2 and NZR = 0.1 for different matrix sizes. F: Baseline, P: Proposed.

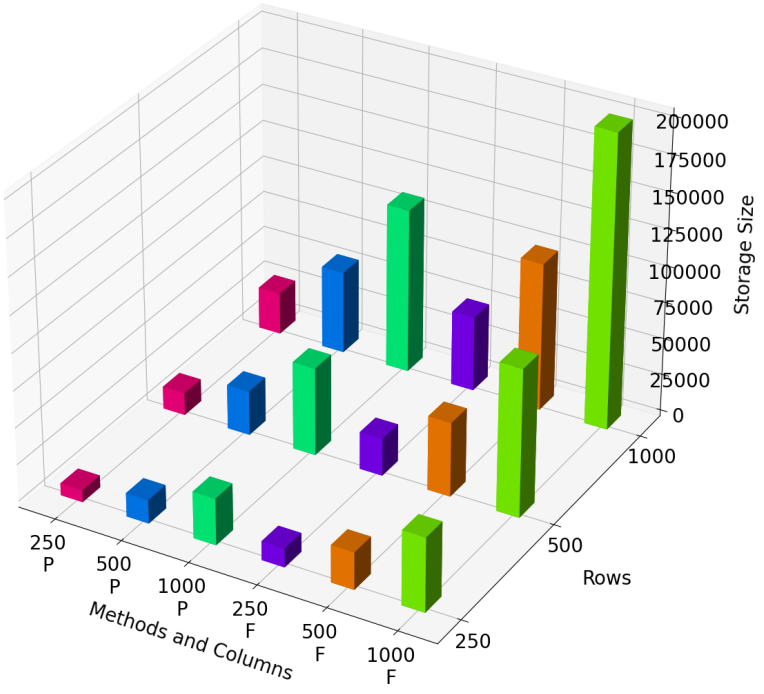


Figure 5.23. The storage size when UV = 2 and NZR = 0.1 for different matrix sizes. F: Baseline, P: Proposed.

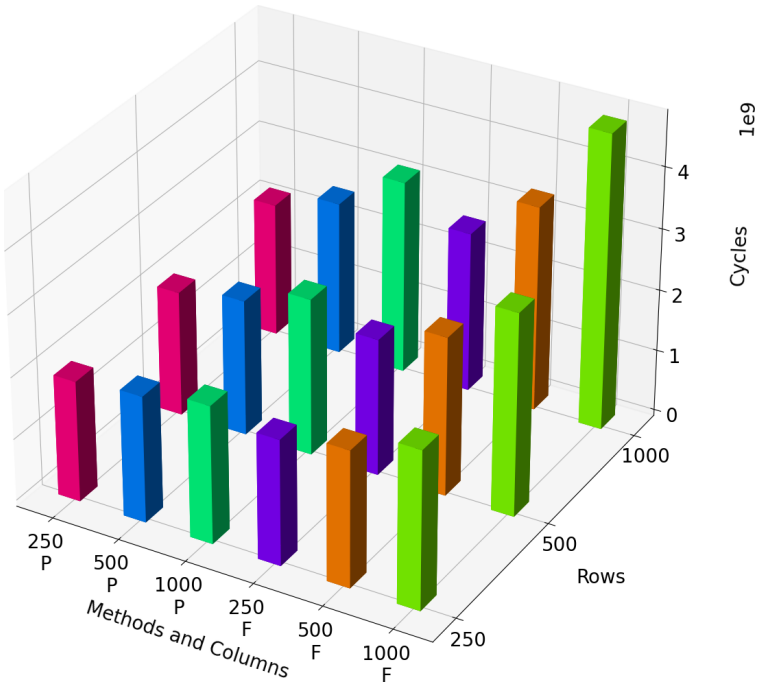


Figure 5.24. The number of cycles when UV = 2 and NZR = 0.1 for different matrix sizes. F: Baseline, P: Proposed.

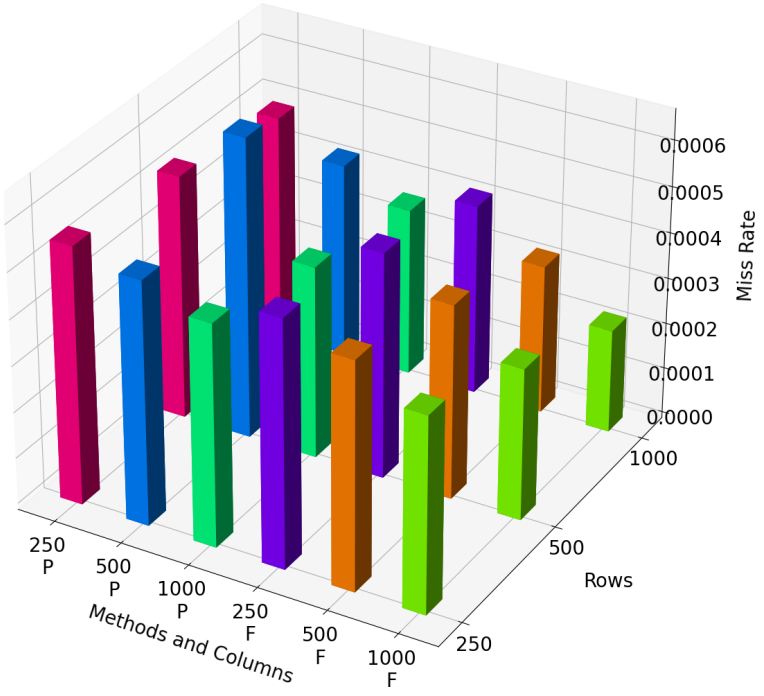


Figure 5.25. The L1 instruction cache miss rate when UV = 2 and NZR = 0.1 for different matrix sizes. F: Baseline, P: Proposed.

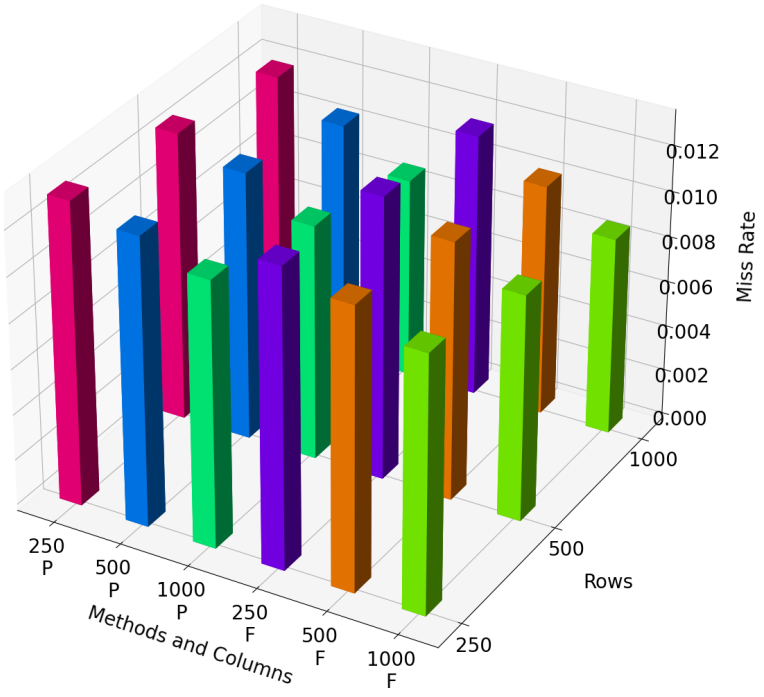


Figure 5.26. The L1 data cache miss rate when UV = 2 and NZR = 0.1 for different matrix sizes. F: Baseline, P: Proposed.

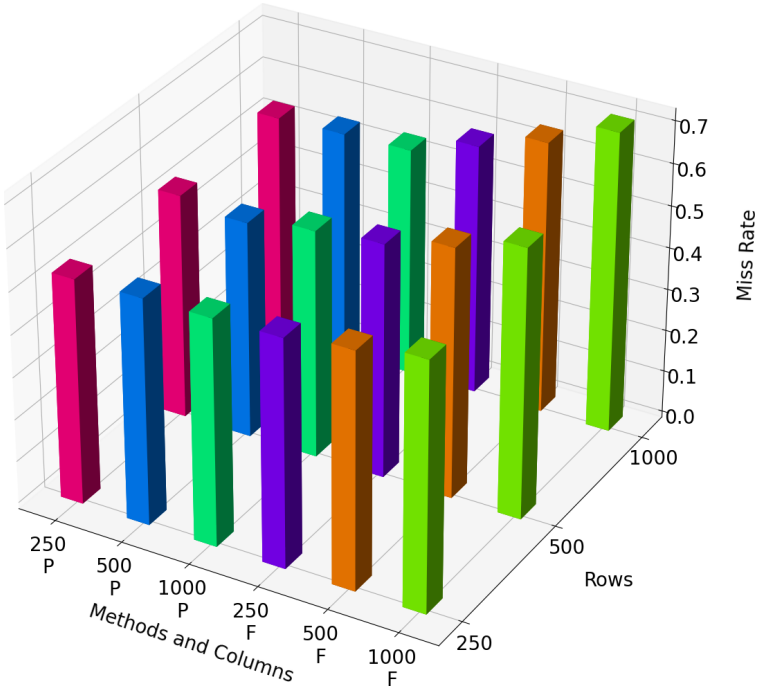


Figure 5.27. The L2 cache miss rate when UV = 2 and NZR = 0.1 for different matrix sizes. F: Baseline, P: Proposed.

6. CONCLUSION

In this thesis, a random search-based CSE method with a compression format is built. Its run-time can be defined by the iteration and attempt parameters. The run-time of producing the adder tree stays under ten seconds for 100×100 matrices and under hundred seconds for 1000×1000 matrices in the experiments. Although it shows a weaker performance against the compared methods, it can produce an adder tree for large matrices in a short time. The compared methods are supposed to be used for small layers of a deep learning model if a one-dimensional matrix multiplication format is provided. In contrast, the proposed method is recommended for large layers of a deep learning model.

The proposed method can also be used to compress the matrices. If the matrix contains a portion of similar elements, the proposed method compresses the matrix more than CSR and CSC formats. Decompression can be performed without data loss. Therefore, the proposed method can be used as a compression technique. Both compressed and decompressed versions of the matrices can be used in matrix multiplication.

The proposed method is simulated on gem5 to investigate its deployment on a single-core embedded device. A small cache setup and DDR3 memory are picked in the simulation to imitate an embedded device. The proposed method shows better performance than the baseline for sparse matrices up to approximately 25% NZR. So, the sparse models can be deployed on low-end devices, and they can be supported with the proposed method.

In the next step of this thesis, data arrangement on the compression matrices needs to be analyzed for the latency decrease. The placement of the common subexpressions in the compression matrices may be reconsidered to reduce the memory accesses. In addition, parallel processing on a multi-core device or FPGA can be studied.

The compression format and processing the compressed arrays need to be manipulated to fit an FPGA well.

In conclusion, deploying a deep learning application needs to be considered as a whole. Pruning, quantization and CSE algorithms are applied to reduce the computation cost. The efficiency of the improvement attempts depends on the target hardware. Pruning, quantization and CSE methods need to be manipulated according to the target hardware to increase utilization. A CSE algorithm and its compression format must be specialized for CPU, GPU or FPGA.

REFERENCES

1. LeCun, Y., "1.1 Deep Learning Hardware: Past, Present, and Future", *IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 12–19, San Francisco, USA, 2019.

2. Zhang, C., P. Patras and H. Haddadi, "Deep Learning in Mobile and Wireless Networking: A Survey", *IEEE Communications Surveys & Tutorials*, Vol. 21, pp. 2224–2287, 2019.

3. Shawahna, A., S. M. Sait and A. El-Maleh, "FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review", *IEEE Access*, Vol. 7, pp. 7823–7859, 2019.

4. Zaman, K. S., M. B. I. Reaz, S. H. M. Ali, A. A. A. Bakar and M. E. H. Chowdhury, "Custom Hardware Architectures for Deep Learning on Portable Devices: A Review", *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2021.

5. Han, S., H. Mao and W. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding", ArXiv:1510.00149 [cs], 2016.

6. Pasko, R., P. Schaumont, V. Derudder, S. Vernalde and D. Durackova, "A New Algorithm for Elimination of Common Subexpressions", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 18, No. 1, pp. 58–68, 1999.

7. Goharian, N., A. Jain and Q. Sun, "Comparative Analysis of Sparse Matrix Algorithms for Information Retrieval", *Computer*, Vol. 2, pp. 0–4, 2003.

8. Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hest-

ness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, "The Gem5 Simulator", *SIGARCH Computer Architecture News*, Vol. 39, No. 2, pp. 1–7, 2011.

9. Hsiao, S.-F., M.-C. Chen and C.-S. Tu, "Memory-free Low-cost Designs of Advanced Encryption Standard Using Common Subexpression Elimination for Subfunctions in Transformations", *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 53, No. 3, pp. 615–626, 2006.

10. Wu, N., X. Zhang, Y. Ye and L. Lan, "Improving Common Subexpression Elimination Algorithm with A New Gate-Level Delay Computing Method", *Lecture Notes in Engineering and Computer Science*, Vol. 2, pp. 677–682, 2013.

11. Alzubaidi, L., J. Zhang, A. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. Fadhel, M. Al-Amidie and L. Farhan, "Review of Deep Learning: Concepts, CNN Architectures, Challenges, Applications, Future Directions", *Journal of Big Data*, Vol. 8, p. 53, 2021.

12. Justus, D., J. Brennan, S. Bonner and A. S. McGough, "Predicting the Computational Cost of Deep Learning Models", *IEEE International Conference on Big Data (Big Data)*, pp. 3873–3882, Seattle, WA, USA, 2018.

13. Liu, L., Y. Wu, W. Wei, W. Cao, S. Sahin and Q. Zhang, "Benchmarking Deep Learning Frameworks: Design Considerations, Metrics and Beyond", *IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1258–1269, Vienna, Austria, 2018.

14. Gupta, S., W. Zhang and F. Wang, "Model Accuracy and Runtime Tradeoff in Distributed Deep Learning: A Systematic Study", *IEEE 16th International Conference on Data Mining (ICDM)*, pp. 171–180, Barcelona, Spain, 2016.

15. Goel, A., C. Tung, Y.-H. Lu and G. K. Thiruvathukal, "A Survey of Methods

for Low-Power Deep Learning and Computer Vision", *IEEE 6th World Forum on Internet of Things (WF-IoT)*, pp. 1–6, New Orleans, LA, USA, 2020.

16. Deng, L., G. Li, S. Han, L. Shi and Y. Xie, "Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey", *Proceedings of the IEEE*, Vol. 108, No. 4, pp. 485–532, 2020.

17. Wang, Y., W. ". Guo and X. Yue, "Tensor Decomposition to Compress Convolutional Layers in Deep Learning", *IISE Transactions*, pp. 1–60, 2021.

18. Phan, A.-H., K. Sobolev, K. Sozykin, D. Ermilov, J. Gusak, P. Tichavský, V. Glukhov, I. Oseledets and A. Cichocki, "Stable Low-Rank Tensor Decomposition for Compression of Convolutional Neural Network", A. Vedaldi, H. Bischof, T. Brox and J.-M. Frahm (Editors), *Computer Vision – ECCV*, pp. 522–539, Glasgow, Scotland, 2020.

19. Kim, E., C. Ahn and S. Oh, "NestedNet: Learning Nested Sparse Structures in Deep Neural Networks", *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8669–8678, Salt Palace Convention Center, Utah, United States, 2018.

20. Liu, Z., M. Sun, T. Zhou, G. Huang and T. Darrell, "Rethinking the Value of Network Pruning", ArXiv:1810.05270 [cs], 2019.

21. Xu, S., A. Huang, L. Chen and B. Zhang, "Convolutional Neural Network Pruning: A Survey", *39th Chinese Control Conference (CCC)*, pp. 7458–7463, Shenyang, China, 2020.

22. Ma, X., S. Lin, S. Ye, Z. He, L. Zhang, G. Yuan, S. H. Tan, Z. Li, D. Fan, X. Qian, X. Lin, K. Ma and Y. Wang, "Non-Structured DNN Weight Pruning–Is It Beneficial in Any Platform?", *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2021.

23. Cao, S., C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu and L. Zhang, "Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity", *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 63–72, New York, NY, USA, 2019.

24. Yang, J., X. Shen, J. Xing, X. Tian, H. Li, B. Deng, J. Huang and X.-s. Hua, "Quantization Networks", *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Long Beach, California, 2019.

25. Liang, T., J. Glossner, L. Wang, S. Shi and X. Zhang, "Pruning and Quantization for Deep Neural Network Acceleration: A Survey", *Neurocomputing*, Vol. 461, No. C, pp. 370–403, 2021.

26. Hou, L. and J. T.-Y. Kwok, "Loss-aware Weight Quantization of Deep Networks", ArXiv:1802.08635 [cs], 2018.

27. Chen, X., Y. Zhao, Y. Wang, P. Xu, H. You, C. Li, Y. Fu, Y. Lin and Z. Wang, "SmartDeal: Remodeling Deep Network Weights for Efficient Inference and Training", *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–15, 2022.

28. You, H., X. Chen, Y. Zhang, C. Li, S. Li, Z. Liu, Z. Wang and Y. Lin, "ShiftAddNet: A Hardware-Inspired Deep Network", H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan and H. Lin (Editors), *Advances in Neural Information Processing Systems*, Vol. 33, pp. 2771–2783, 2020.

29. Langr, D. 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.

30. Willcock, J. and A. Lumsdaine, "Accelerating Sparse Matrix Computations via Data Compression", *Proceedings of the 20th Annual International Conference on*

Supercomputing, pp. 307–316, New York, NY, USA, 2006.

31. Hosseinabady, M. and J. L. Núñez-Yáñez, "Sparse Matrix-Dense Matrix Multiplication on Heterogeneous CPU+FPGA Embedded System", *Proceedings of the 11th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures / 9th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms*, Bologna, Italy, 2020.

32. Hosseinabady, M. and J. Nunez-Yanez, "A Streaming Dataflow Engine for Sparse Matrix-Vector Multiplication Using High-Level Synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 39, pp. 1272–1285, 2020.

33. Greathouse, J. L. and M. Daga, "Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format", *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 769–780, New Orleans, LA, USA, 2014.

34. Wang, Y., J. Wang, W. Zhang, Y. Zhan, S. Guo, Q. Zheng and X. Wang, "A Survey on Deploying Mobile Deep Learning Applications: A Systemic and Technical Perspective", *Digital Communications and Networks*, Vol. 8, pp. 1–17, 2021.

35. Chen, J. and X. Ran, "Deep Learning With Edge Computing: A Review", *Proceedings of the IEEE*, Vol. 107, No. 8, pp. 1655–1674, 2019.

36. Ma, X., F. Guo, W. Niu, X. Lin, J. Tang, K. Ma, B. Ren and Y. Wang, "PCONV: The Missing but Desirable Sparsity in DNN Weight Pruning for Real-time Execution on Mobile Devices", ArXiv:1909.05073 [cs], 2019.

37. Chen, Y., B. Zheng, Z. Zhang, Q. Wang, C. Shen and Q. Zhang, "Deep Learning on Mobile and Embedded Devices: State-of-the-Art, Challenges and Future Directions", *ACM Computing Surveys*, Vol. 53, No. 84, pp. 1–37, 2021.

38. Goumas, G., K. Kourtis, N. Anastopoulos, V. Karakasis and N. Koziris, "Understanding the Performance of Sparse Matrix-Vector Multiplication", *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 283–292, Toulouse, France, 2008.

39. Nurvitadhi, E., D. Sheffield, J. Sim, A. Mishra, G. Venkatesh and D. Marr, "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC", *International Conference on Field-Programmable Technology (FPT)*, pp. 77–84, Xi'an, China, 2016.

40. Tridgell, S., M. Kumm, M. Hardieck, D. Boland, D. J. M. Moss, P. Zipf and P. H. W. Leong, "Unrolling Ternary Neural Networks", *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, Vol. 12, pp. 1–23, 2019.

41. Kumm, M., M. Hardieck and P. Zipf, "Optimization of Constant Matrix Multiplication with Low Power and High Throughput", *IEEE Transactions on Computers*, Vol. 66, No. 12, pp. 2072–2080, 2017.

APPENDIX A: USED GEM5 OPTIONS

The arguments to run the gem5 executable for a matrix multiplication executable are listed as follows:

- "../../../gem5/build/X86/gem5.opt",
- "../../../gem5/configs/example/se.py",
- "–cpu-type=TimingSimpleCPU",
- "–cpu-clock=1GHz",
- "–l1d_size=256kB",
- "–l1i_size=256kB",
- "–caches",
- "–l2cache",
- "–l2_size=1MB",
- "–cmd=./matrix_executable".