

ADAPTIVE MATRIX MULTIPLICATION FOR VARIOUS DEGREE OF SPARSITY USING
TENSORFLOW



A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Computer Science and Information Technology
Department of Mathematics and Computer Science
FACULTY OF SCIENCE
Chulalongkorn University
Academic Year 2019
Copyright of Chulalongkorn University

การคูณเมทริกซ์ที่ปรับได้สำหรับความว่างระดับต่างๆ โดยใช้เทนเซอร์-โฟลว



วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ ภาควิชาคณิตศาสตร์และวิทยาการ

คอมพิวเตอร์

คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2562

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

Thesis Title ADAPTIVE MATRIX MULTIPLICATION FOR VARIOUS
DEGREE OF SPARSITY USING TENSORFLOW
By Mr. Siraphob Theeracheep
Field of Study Computer Science and Information Technology
Thesis Advisor Associate Professor JARULOJ CHONGSTITVATANA, Ph.D.

Accepted by the FACULTY OF SCIENCE, Chulalongkorn University in Partial
Fulfillment of the Requirement for the Master of Science

..... Dean of the FACULTY OF SCIENCE
(Professor POLKIT SANGVANICH, Ph.D.)

THESIS COMMITTEE

..... Chairman
(Assistant Professor DITTAYA WANVARIE, Ph.D.)

..... Thesis Advisor
(Associate Professor JARULOJ CHONGSTITVATANA, Ph.D.)

..... External Examiner
(Associate Professor Worasait Suwannik, Ph.D.)

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

สิริภพ ธีระชีพ : การคูณเมทริกซ์ที่ปรับได้สำหรับความว่างระดับต่างๆ โดยใช้เทนเซอร์-
โฟลว. (ADAPTIVE MATRIX MULTIPLICATION FOR VARIOUS DEGREE OF
SPARSITY USING TENSORFLOW) อ.ที่ปรึกษาหลัก : รศ. ดร.จาร์โลจน์ จงสถิตย์วัฒนา

การคูณเมทริกซ์เป็นการดำเนินการทางคณิตศาสตร์ที่นำไปประยุกต์ใช้แก้ไขปัญหาหลายประเภทและมีการนำเสนอขั้นตอนวิธีการคูณเมทริกซ์สำหรับหลายแพลตฟอร์ม เทนเซอร์โฟลวเป็นแพลตฟอร์มสำหรับการเรียนรู้ด้วยเครื่องที่ประกอบด้วยชุดคำสั่งทางคณิตศาสตร์หลายคำสั่งรวมถึงคำสั่งการคูณเมทริกซ์ เทนเซอร์โฟลวมีคำสั่งพื้นฐานสำหรับคูณเมทริกซ์สองคำสั่งคือ *tf.matmul* และ *tf.sparse_matmul* ซึ่งแนะนำให้ใช้กับเมทริกซ์ที่มีเลขศูนย์น้อยและเมทริกซ์ที่มีเลขศูนย์มาก ตามลำดับ วิทยานิพนธ์ฉบับนี้นำวิธีปรับปรุงประสิทธิภาพของการคูณเมทริกซ์บนแพลตฟอร์มเทนเซอร์โฟลว วิธีการคูณเมทริกซ์ที่นำเสนอจะแบ่งแต่ละเมทริกซ์เป็นสี่เมทริกซ์ย่อย จากนั้นจึงเลือกระหว่าง *tf.matmul* และ *tf.sparse_matmul* ที่เป็นคำสั่งพื้นฐานสำหรับคูณเมทริกซ์ เพื่อคูณคู่ของเมทริกซ์ย่อยแต่ละคู่ ตามความหนาแน่นของเมทริกซ์ย่อย เราพบว่าคำสั่งการคูณเมทริกซ์ที่นำเสนอนี้สามารถคูณเมทริกซ์ได้เร็วกว่า *tf.matmul* และ *tf.sparse_matmul* ในกรณีที่เมทริกซ์มีการกระจายตัวของค่าที่ไม่ใช่ศูนย์ไม่สม่ำเสมอ สำหรับกรณีอื่นๆคำสั่งการคูณเมทริกซ์ดังกล่าวสามารถคูณเมทริกซ์ได้ช้ากว่าคำสั่งที่พื้นฐานที่เร็วที่สุดระหว่าง *tf.matmul* หรือ *tf.sparse_matmul* เล็กน้อย อย่างไรก็ตาม คำสั่งการคูณเมทริกซ์ดังกล่าวใช้ได้เฉพาะบน CPU เนื่องจาก *tf.sparse_matmul* รองรับการทำงานเฉพาะบน CPU ไม่รองรับการทำงานบน GPU

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

สาขาวิชา วิทยาการคอมพิวเตอร์และ เทคโนโลยีสารสนเทศ
ปีการศึกษา 2562
ลายมือชื่อนิสิต
ลายมือชื่อ อ.ที่ปรึกษาหลัก

5972633323 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

KEYWORD: TensorFlow, Matrix multiplication, Dataflow programming

Siraphob Theeracheep : ADAPTIVE MATRIX MULTIPLICATION FOR VARIOUS DEGREE OF SPARSITY USING TENSORFLOW. Advisor: Assoc. Prof. JARULOJ CHONGSTITVATANA, Ph.D.

Matrix multiplication is a fundamental operation used in many problems, and many matrix multiplication algorithms are proposed for many computing environments. TensorFlow is a machine learning platform with many mathematic library functions including matrix multiplication. TensorFlow provides two methods, *tf.matmul* and *tf.sparse_matmul*, for matrix multiplication. It is suggested that *tf.matmul* should be used for dense matrices, and *tf.sparse_matmul* should be used for sparse matrices. In this work, an approach is proposed to improve the efficiency of matrix multiplication in TensorFlow. The proposed approach divides each matrix into four submatrices, and chooses either *tf.matmul* or *tf.sparse_matmul* for the multiplication of each pair of submatrices, based on the density of the submatrices. We found that it is faster than both *tf.matmul* and *tf.sparse_matmul* for input matrices that have uneven distribution of non-zero values. For other inputs, it is almost as fast as the faster one between *tf.matmul* and *tf.sparse_matmul*. However, this approach can only be used for CPUs because *tf.sparse_matmul* is supported only on CPUs but not GPUs.

Field of Study: Computer Science and
Information Technology

Student's Signature

Academic Year: 2019

Advisor's Signature

ACKNOWLEDGEMENTS

This thesis will not come to fruition without the help and support of all the faculty and staffs at the Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University.

First, I would like to thank Ms. Nonglak Pootachareon, for all the help and support regarding the paperwork and student's affairs.

Next, I would like to thank my family and friends who throughout the creation of this thesis were always there for me when I'm in the darkest of places.

Finally, and most importantly, I would like to thank my thesis advisor, Associate Professor Jaruloj Chongstitvatana Ph.D. for always being kind and patient with me.

Siraphob Theeracheep

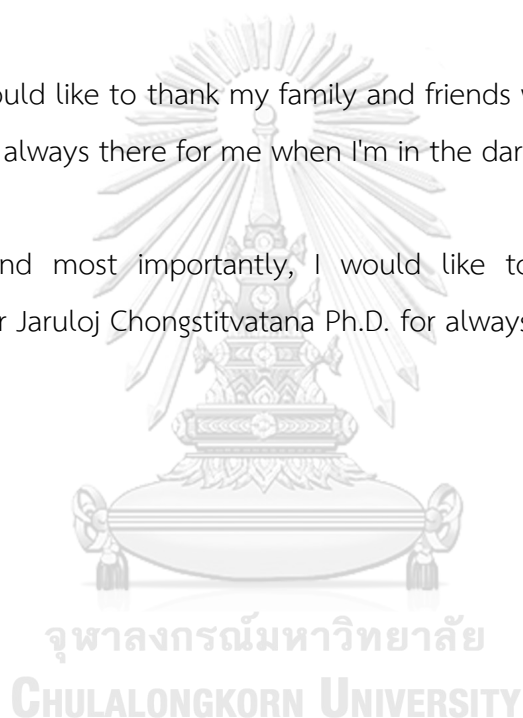


TABLE OF CONTENTS

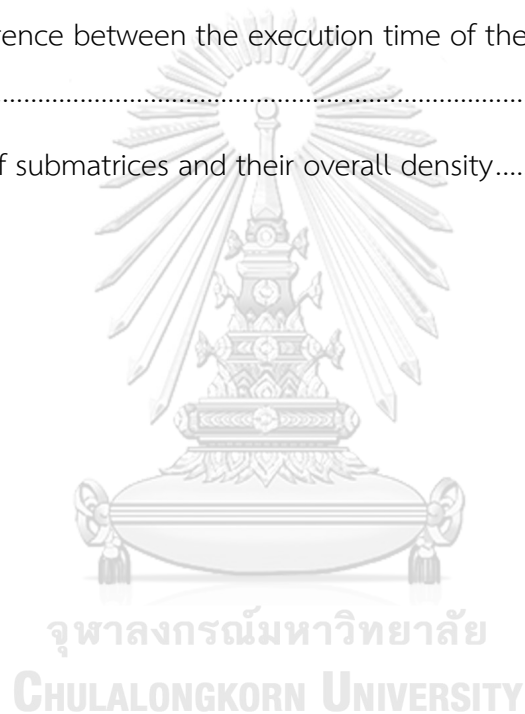
	Page
.....	iii
ABSTRACT (THAI).....	iii
.....	iv
ABSTRACT (ENGLISH).....	iv
ACKNOWLEDGEMENTS.....	v
TABLE OF CONTENTS.....	vi
LIST OF TABLES.....	viii
LIST OF FIGURES.....	ix
Chapter 1 Introduction.....	11
Chapter 2 Related Works.....	14
2.1 Dense Matrix Multiplication.....	14
2.2 Sparse Matrix Representation.....	15
2.3 Example of Sparse Matrix Multiplication Implementations.....	18
2.4 TensorFlow Programming.....	19
Chapter 3 Proposed Method.....	22
3.1 Block-adaptive Matrix Multiplication.....	22
Chapter 4 Experiments.....	27
4.1 Finding the Criteria for Choosing between Dense Matrix Multiplication and Sparse Matrix Multiplication Methods.....	27
4.2 Performance Evaluation of <i>block_adapt_matmul</i> on CPU.....	31
Chapter 5 Results and Discussion.....	33

5.1 Performance of <i>block_adapt_matmul</i> with Varying Overall Density of Input Matrices.....	33
5.2 Performance of <i>block_adapt_matmul</i> with Varying Density Differences of Input Matrices.....	38
Chapter 6 Conclusion	42
REFERENCES	44
VITA.....	47



LIST OF TABLES

	Page
Table 1 Criteria for selecting multiplication methods in block_adapt_matmul.....	28
Table 2 The fastest matrix multiplication method for each combination of input matrices densities measured from experiments.....	29
Table 3 The criteria for deciding whether matrix B is sparse or dense	30
Table 4 The difference between the execution time of the two fastest multiplication methods	31
Table 5 Density of submatrices and their overall density.....	32



LIST OF FIGURES

	Page
Figure 1 An arbitrary matrix converted to 2x2 block matrix.....	15
Figure 2 Example of sparse matrix in DOK format	16
Figure 3 Example of sparse matrix stored in LIL format	16
Figure 4 Example of sparse matrix stored in COO format	17
Figure 5 Example of sparse matrix stored in CSR format and CSC format.....	17
Figure 6 Example of matrix multiplications of block matrices	23
Figure 7 Pseudocode of block_adapt_matmul.....	25
Figure 8 The diagram of the algorithm for block_adapt_matmul.....	26
Figure 9 Checkered pattern of higher density and lower density input matrix	32
Figure 10 Execution times of block_adapt_matmul, tf.sparse_matmul and tf.matmul on matrices of sizes 3000x3000, 6000x6000, 9000x9000, and 1200x12000 with varying overall density.....	34
Figure 11 Speedup of block_adapt_matmul compared to tf.matmul for matrices of overall density of 30%, 40%, 50%, 60%, 70%, and 80%, with varying sizes.	36
Figure 12 Speedup of block_adapt_matmul compared to tf.sparse_matmul for matrices of overall density of 30%, 40%, 50%, 60%, 70%, and 80%, with varying sizes	37
Figure 13 Execution times of block_adapt_matmul, tf.sparse_matmul and tf.matmul on matrices of size 3000x3000, 6000x6000, 9000x9000, and 12000x12000 with varying density difference.....	39
Figure 14 Speedup of block_adapt_matmul compared to tf.matmul for matrices of density differences of 20, 40, 60 and 80 with varying sizes	40

Figure 15 Speedup of block_adapt_matmul compared to tf.sparse_matmul for matrices of density differences of 20, 40, 60 and 80 with varying sizes 41



Chapter 1

Introduction

Massive amount of data is constantly being processed by billions of computers all over the world. They are processed on computers on a large network of datacenter, desktop computers, laptop computers and even smartphones. For these computers handle large amount of data, efficient algorithms must be provided. An important algorithm frequently used in wide range of applications and scales is matrix multiplication. A matrix is a data representation, organizing data into two-dimensional array, which consists of rows and columns. Each row or column can contain any number of elements. For each element in a matrix, its position in the matrix can be specified using two integers, a row index, and a column index. Matrices are used in many applications, such as solving a system of equations in linear algebra [1]. These systems of equations are also used to model many problems in neural networks, image processing, machine learning, and many others. Solving these problems rely on matrix computation, especially matrix multiplication. For example, in neural networks, a training session requires many iterations of matrix multiplication [2]. Since the matrix multiplication is a computationally expensive operation whose time complexity is in $O(n^3)$ for naïve implementation [3], there are many approaches to make matrix multiplication operation more efficient. An approach for optimizing matrix multiplication is to exploit the property of input matrices. For example, if input matrices contain a large number of zeros, many steps in matrix multiplication will be wasted on multiplying zeros [4]. A matrix multiplication method can be implemented to reduce numbers of these wasted computation steps. Matrices with this property is called sparse matrix and matrix multiplication optimized for these matrices is called sparse matrix multiplication. One approach to optimize sparse matrix multiplication is to use a representation for sparse matrix which stores only non-zero elements of a matrix. Then, algorithms for matrix operations must be optimized for such representations. Some sparse representations are optimized for the creation of matrices, while others are optimized for fast access and operations [5]. These representations are also less efficient when used with matrices that are

less sparse. Some efficient matrix operations on sparse matrices are done on low-level computing kernel that directly communicates with hardware, which depend on specific hardware and may not be applicable on others. There are many implementations of matrix multiplication with varying degree of optimizations which target different fields of application. Many implementations of matrix multiplication can be found in various libraries of high-level programming languages. An example of such libraries is TensorFlow [6].

TensorFlow is a dataflow-paradigm-based machine learning library that can be used on many types of processing units, such as CPU, GPU and TPU. It allows users to design and construct computation graphs of computing tasks using a set of pre-built nodes representing basic operations through a coding interface written in high-level languages such as Python and Java. After the computation graph is constructed, TensorFlow evaluates the dependency of computing nodes in the graph and automatically allocate processing units for these nodes in order to make parallelization of the tasks possible. TensorFlow hides the complexity of parallel programming in low-level languages while still allows customizability through various pre-built operation nodes. TensorFlow was developed as an open-source software not only available for use with supercomputers or datacenters, but also for smartphones and personal computers.

TensorFlow provides two different implementations of matrix multiplication methods, *tf.matmul*, which is designed for general purpose matrix multiplication and *tf.sparse_matmul*, which is optimized for sparse matrix multiplication. The method *tf.matmul* is supported on both CPUs and GPUs. However, the method *tf.sparse_matmul* is supported only on CPUs because a GPU kernel for sparse matrix multiplication is not readily available in TensorFlow. Some matrices are of medium density, e.g. matrices with half zeros and half non-zeros, and sparse matrix multiplication is not well optimized for such matrices. However, if the method *tf.matmul* is used, many multiplications of zeros are still performed.

In this work, we aim to increase the performance of multiplication of medium-density matrices, using only pre-built operations on TensorFlow without incorporating any customized low-level computing kernels. We propose an approach

for matrix multiplication and implement a computation graph of matrix multiplication from basic operations in TensorFlow, including its two built in matrix multiplication methods. The proposed approach divides each input matrices into four submatrices and perform multiplication on each pair of submatrices using the appropriate matrix multiplication method based on their density. This approach can only be used on CPUs because the method *tf.sparse_matmul* is not supported on GPUs.

Experiments are performed to evaluate the proposed method. It is found that the method, *block_adapt_matmul*, outperforms *tf.matmul* and *tf.sparse_matmul* when the distribution of non-zero elements in the input matrices are uneven. The more uneven the non-zero distribution in the input matrices are, the faster the performance of the proposed. Although the method also requires a preliminary experiment to determine the criteria for choosing multiplication methods for different density.

In conclusion, *block_adapt_matmul* should be used when the input matrices have uneven non-zero distribution and moderate overall density.

Chapter 2

Related Works

Matrix multiplication is used in many problems such as pattern recognition, neural networks, and machine learning. In pattern recognition, an image is represented by a matrix of values of each pixel in the image, and operations, such as edge detection, are done by multiplying a matrix with the matrix of the image [7]. In neural networks, the multiplication of the input and the weight matrix is performed numerous times throughout the training process [2]. Many machine learning algorithms, such as bert [8], word2vec [9] and GloVe [10], are based on neural networks and, thus, rely heavily on matrix multiplication. As a result, reducing the time for matrix multiplication can reduce the computation time for many algorithms.

Many approaches are proposed to improve the efficiency of matrix multiplication. Some matrix multiplication algorithms are designed for dense matrixes, and they are described in section 2.1. Section 2.2 describes the representation of sparse matrices. Then, some implementations of matrix multiplication for sparse matrices are explained in section 2.3. Some of these implementations are designed for single-core CPUs, some are designed for multi-core CPU, and some are of GPUs. Finally, section 2.4 describes TensorFlow, a machine learning library which will be used as the experimental platform in this work.

2.1 Dense Matrix Multiplication

Early study proposes algorithms such as Strassen's algorithm [11] and Coppersmith-Winograd algorithm [12] which follow the same principle of recursively reducing the total number of multiplications required in a matrix multiplication. In Strassen's algorithm, the multiplication of matrices is done by doing seven multiplications of submatrices which are created by either adding or subtracting a pair of submatrices of the input matrices. Compare to the basic divide-and-conquer matrix multiplication which uses eight multiplications of submatrices, it reduces the time complexity of matrix multiplication from $O(n^3)$ to $O(n^{2.807})$. After Strassen's algorithm, many matrix multiplication algorithms using similar optimization principle

which have better time complexity are proposed [12-14]. One of these algorithms is the Coppersmith-Winograd algorithm, whose time complexity is $O(n^{2.375})$. However, all these matrix multiplication algorithms have an exceptionally large computational overhead cost. This makes these algorithms faster, in practice, than conventional matrix multiplication only for extremely large matrices. Therefore, these algorithms are not used in practice while Strassen's algorithm can be used.

In order to utilize parallelization in hardware, Steven W. D. Chien et al [15] uses tiling algorithm to divide matrices into smaller submatrices, as shown in figure 1. Matrix multiplications are then performed on these submatrices in parallel on GPU. In addition to parallelism, tiling algorithm also enable multiplication of matrices too large to fit in the memory of a GPU.

$$\begin{aligned}
 A_{m \times n} &= \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \\
 A_{11} &= \begin{bmatrix} a_{11} & \cdots & a_{1(\frac{1}{2}n)} \\ \vdots & \ddots & \vdots \\ a_{(\frac{1}{2}m)1} & \cdots & a_{(\frac{1}{2}m)(\frac{1}{2}n)} \end{bmatrix}, A_{12} = \begin{bmatrix} a_{1(\frac{1}{2}n+1)} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{(\frac{1}{2}m)(\frac{1}{2}n+1)} & \cdots & a_{(\frac{1}{2}m)n} \end{bmatrix} \\
 A_{21} &= \begin{bmatrix} a_{(\frac{1}{2}m+1)1} & \cdots & a_{(\frac{1}{2}m+1)(\frac{1}{2}n)} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{m(\frac{1}{2}n)} \end{bmatrix}, A_{22} = \begin{bmatrix} a_{(\frac{1}{2}m+1)(\frac{1}{2}n+1)} & \cdots & a_{(\frac{1}{2}m+1)n} \\ \vdots & \ddots & \vdots \\ a_{m(\frac{1}{2}n+1)} & \cdots & a_{mn} \end{bmatrix}
 \end{aligned}$$

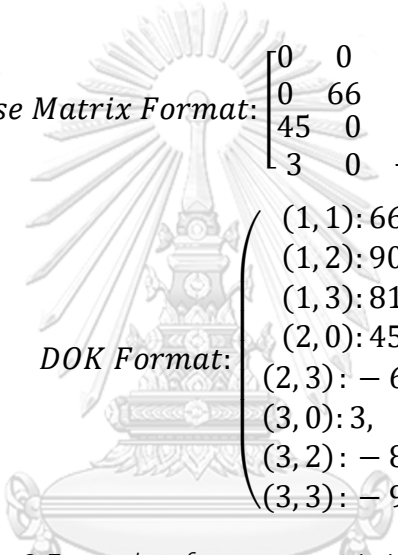
Figure 1 An arbitrary matrix converted to 2x2 block matrix

These algorithms are designed for dense matrices. However, in many real-world problems, data are represented by matrices may contain many zero-valued elements, called sparse matrices. Based on the sparsity of matrices, many algorithms are designed for sparse matrix multiplication.

2.2 Sparse Matrix Representation

One approach to optimize sparse matrix multiplication is based on the representation of sparse matrices, which stores a collection of the non-zero values in the matrix, together with their positions. This can reduce the memory required to store sparse matrices, and also make it easy to avoid the useless multiplication of

zeros. Many sparse matrix storage formats have been proposed [5]. For sparse matrices, dictionary of keys (DOK), List of lists (LIL) or Coordinate list (COO) are examples of the representations that can be used to efficiently construct sparse matrix or convert matrix to sparse representation from dense format. Figure 2, 3 and 4 show examples of matrices in DOK, LIL and COO sparse matrix representation, respectively. For DOK format, a matrix is stored in a dictionary, and the key in the dictionary is the row and the column of a non-zero element and the value in the dictionary is the non-zero element.



$$\begin{array}{l}
 \text{Dense Matrix Format:} \\
 \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 66 & 90 & 81 \\ 45 & 0 & 0 & -62 \\ 3 & 0 & -82 & -97 \end{bmatrix} \\
 \\
 \text{DOK Format:} \\
 \left(\begin{array}{l} (1,1): 66, \\ (1,2): 90, \\ (1,3): 81, \\ (2,0): 45, \\ (2,3): -62, \\ (3,0): 3, \\ (3,2): -82, \\ (3,3): -97, \end{array} \right)
 \end{array}$$

Figure 2 Example of sparse matrix in DOK format

For LIL format, each row in a matrix is stored as a list of pairs of column index and its value, and a matrix is a list of all rows.

$$\begin{array}{l}
 \text{Dense Matrix Format:} \\
 \begin{bmatrix} 0 & 0 & 0 & 12 \\ -90 & 13 & 0 & -5 \\ 0 & 48 & 0 & -116 \\ -77 & 0 & 82 & 0 \end{bmatrix} \\
 \\
 \text{LIL Format:} \\
 \left[\begin{array}{l} [(3, 12)], \\ [(0, -90), (1, 13), (3, -5)], \\ [(1, 48), (3, -116)], \\ [(0, -77), (2, 82)] \end{array} \right]
 \end{array}$$

Figure 3 Example of sparse matrix stored in LIL format

For COO format, a matrix is stored as a list of three-element tuples, where each element is row index, column index and value, respectively.

$$\begin{array}{l}
 \text{Dense Matrix Format:} \begin{bmatrix} 0 & 0 & 63 & 0 \\ 0 & 76 & 0 & -105 \\ 0 & 121 & 38 & 10 \\ 92 & -38 & 0 & 0 \end{bmatrix} \\
 \\
 \text{COO Format:} \begin{pmatrix} (0, 2, 63) \\ (1, 1, 76) \\ (1, 3, -105) \\ (2, 1, 121) \\ (2, 2, 38) \\ (2, 3, 10) \\ (3, 0, 92) \\ (3, 1, -38) \end{pmatrix}
 \end{array}$$

Figure 4 Example of sparse matrix stored in COO format

When performing matrix operations, including matrix multiplication, some other sparse matrix representations are more efficient [5], such as compressed sparse row (CSR) or compressed sparse column (CSC). Examples of matrix in these formats are shown in figure 5. These two formats are similar, with the only difference being the stored values are ordered by rows first in the former representation and by column first in the latter. The CSR format is efficient for accessing values of the matrix by rows, while the CSC format is efficient for accessing values of the matrix by columns.

$$\begin{array}{l}
 \text{Dense Matrix Format:} \begin{bmatrix} -11 & -3 & 0 & 0 \\ 0 & 0 & 24 & 0 \\ 95 & -123 & 87 & 0 \\ 0 & -94 & 17 & 0 \end{bmatrix} \\
 \\
 \text{CSR Format:} \begin{array}{l}
 \text{Value} = [-11 \quad -3 \quad 24 \quad 95 \quad -123 \quad 87 \quad -94 \quad 17] \\
 \text{Column} = [0 \quad 1 \quad 2 \quad 0 \quad 1 \quad 2 \quad 1 \quad 2] \\
 \text{Row} = [0 \quad 2 \quad 3 \quad 6 \quad 8]
 \end{array} \\
 \\
 \text{CSC Format:} \begin{array}{l}
 \text{Value} = [-11 \quad -3 \quad 24 \quad 95 \quad -123 \quad 87 \quad -94 \quad 17] \\
 \text{Row} = [0 \quad 2 \quad 0 \quad 2 \quad 3 \quad 1 \quad 2 \quad 3] \\
 \text{Column} = [0 \quad 2 \quad 5 \quad 8 \quad 8]
 \end{array}
 \end{array}$$

Figure 5 Example of sparse matrix stored in CSR format and CSC format

Tao Wu et al. [16], proposed a block-based sparse matrix format combining COO and CSR format called BCSR&BCOO. In BCSR&BCOO format, a sparse matrix is divided into smaller submatrices or tiles. Each tile is then stored in either COO or CSR

format depending on the density of the tile. If the density of a tile is above a threshold, it is stored in CSR format; otherwise, it is stored in COO format. The BCSR&BCOO was designed for Multicore CPUs.

Monika Shah et al. [17], proposed an extension of COO format called ALIGNED_COO, in which the storage format is not only optimized by exploiting the sparsity of the matrices, but also their non-zero distribution. ALIGNED_COO format is specifically designed to optimize the performing of sparse matrix-vector multiplication on GPU where the sparse matrix has skewed non-zero distribution. By realigning and padding the sparse matrix into multiple segments of the same size, the ALIGNED_COO format helps increasing the degree of concurrency while performing the computation in GPU.

The optimization of sparse matrix multiplication, based on a representation of sparse matrix, is effective when the matrix is sufficiently sparse. The computation time is higher when the number of non-zero elements in the matrix is higher. Thus, it does not perform well for matrices with medium density.

2.3 Example of Sparse Matrix Multiplication Implementations

Another common approach to optimize sparse matrix multiplication is to develop a computing kernel specific for sparse matrix multiplication for certain types of input matrices. A computing kernel is a program written in low-level code in order to optimize the computation on a specific hardware. Some works propose computing kernels optimized for specific type of sparse matrix multiplication, such as SpMV [18], where the multiplier matrix is sparse and the multiplicand is a single-row matrix or a vector, while other works propose computing kernels for sparse matrix multiplication in general.

Tao Wu et al. [16], proposed a sparse matrix-matrix multiplication computing kernel BSpMM for use specifically with their proposed sparse matrix representation on a CPU with L1 and L2 cache, BCSR&BCOO, which was described in section 2.2. The sparse matrix multiplier in BCSR&BCOO format and dense matrix multiplicand are loaded into L2 and L1 cache of a CPU core, respectively. Then the non-zero values

in the multiplier matrix is multiplied with all values in the dense matrix at the same rows as the column index of each of the non-zero value.

Carl Yang et al. [19], proposed a GPU computing kernel for sparse matrix-sparse vector multiplication, called SpMSpV, which is an improvement to SpMV, which is a computing kernel for sparse matrix-vector multiplication. They developed this kernel initially for breadth-first-search algorithm, which is a problem that can be interpreted in terms of sparse matrix-vector multiplication [20] and then, based on the algorithm for generalized sparse matrix-vector multiplication, the sparse matrix-sparse vector multiplication kernel was developed.

Jeongmyung Lee et al. [21], developed a sparse matrix multiplication algorithm on GPU called *block reorganizer*. The block reorganizer divides input matrices into smaller submatrices and calculates the computational cost required for each submatrix. The submatrices that contain greater number of non-zero values are further divided into smaller blocks, while submatrices with smaller number of non-zero values are grouped together into a larger block. By doing so, the block reorganizer adjusts the computation load in each block to the similar level which leads to better thread utilization and better performance.

The downside of optimization of matrix multiplication by developing a computing kernel is that it is usually specific to the device the kernel is developed for. Moreover, writing low-level computing kernel codes usually requires a lot of technical knowledge and is mostly done on high-performance computing devices, which make it less accessible to non-technical users. Nowadays there are platforms that helps alleviate these downsides by wrapping operations like matrix multiplication and other matrix manipulation techniques in higher level of abstraction that is easier to understand and customize. One such platform is TensorFlow, which is the platform that will be used in our experiments.

2.4 TensorFlow Programming

TensorFlow is a machine learning library made by Google Brain [6]. It utilizes dataflow programming paradigm to manage, distribute and parallelize computations represented as dataflow graphs. A dataflow graph in TensorFlow consists of nodes

and edges. A node in TensorFlow graph represents a primitive, simple operator that take zero or more tensor as input from edges and output zero or more tensor. An operation in a node can be executed when all its input on edges are available. Since the communication between nodes is explicitly defined by edges, the computation dependencies between operations can be identified and parallel computation can be done efficiently.

Writing a TensorFlow program can be divided into two parts. The first part is the creation of dataflow graph. In this step, the program creates computational nodes, which may include constant value tensors, variable tensors, placeholder tensors or operation nodes. The constant value tensors are defined with specific values at the creation of the dataflow graph, and then cannot be changed. The variable tensors are initialized with values at the start of the dataflow execution, and then can be further changed again throughout the computation. The placeholder tensors take input values from outside of the dataflow graph at the start of the computation. The operation nodes are nodes that take other tensors or operation nodes as input and return output tensors or nodes depending on the operation performed. The second part of the program executes the dataflow graph and return output values. The dataflow graph is created only once and can be executed as many times as needed. Any node of the dataflow graph can be selected as the end of the computation, meaning that the dataflow graph can be partially executed up to any node.

At the user-level, TensorFlow is available particularly in Python, C++ and Java, although it has also been ported to other programming languages. While writing a TensorFlow program in these high-level languages, such as when defining an operation node, the high-level method for defining a node communicates through API written in C with a low-level kernel of that operations called “ops”. TensorFlow utilizes many high-performance computing libraries, such as Eigen [22] and BLAS [23] in CPU implementation and cuBLAS [24] in GPU implementation.

TensorFlow provides two matrix multiplication methods. One is for general purpose multiplication for arbitrary input matrices. This method, called *tf.matmul*, performs multiplication of input matrices as if both inputs are dense matrix. The

other one is called *tf.sparse_matmul*, which is specifically designed to efficiently perform multiplication on sparse matrices. This method has additional arguments for specifying if the first input matrix and/or the second input matrix is dense or sparse. The TensorFlow's *tf.sparse_matmul* is suitable for efficient matrix multiplication when the density of input matrices is below 70% as suggested in TensorFlow documentation [25]. Both multiplication method can be executed using CPU, using Eigen kernel implementation, while only the first method can be executed with GPU, using cuBLAS kernel implementation.

Many of the multiplication methods described here utilize block matrix multiplication or tiling algorithm, which divides input matrices into smaller submatrices before further applying optimization techniques. In this work, we propose a matrix multiplication method implemented on TensorFlow platform. The proposed method also utilizes block matrix multiplication to increase the performance for certain types of input matrices.

Chapter 3

Proposed Method

Most of the works on matrix multiplication focus on either dense matrices or sparse matrices. In this work, we focus on matrix multiplication on TensorFlow, and propose a matrix multiplication method. This method is based on the observation that, regardless of the density of a matrix, some of its submatrices are possibly sparse and some are possibly dense. If a faster multiplication method is chosen for each pair of submatrices based on their density, the performance of matrix multiplication can be improved. To create submatrices, a matrix can be divided by row or by column.

The proposed method is called block-adaptive matrix multiplication or *block_adapt_matmul*. Each input matrix is divided into four submatrices by both rows and columns. Then, one of two matrix multiplication methods in TensorFlow, *tf.matmul* and *tf.sparse_matmul*, is chosen for each pair of submatrices, based on their density. For dense matrices with some sparse submatrices, this method should be faster than the dense matrix multiplication. For sparse matrices with some dense submatrices, this method should be faster than the sparse matrix multiplication. However, this method is applicable for computation on CPU, but not on GPU, because TensorFlow does not provide the sparse matrix multiplication method on GPU.

3.1 Block-adaptive Matrix Multiplication

The product of two matrices is composed of submatrices that are the products of submatrices of the multiplier and the multiplicand. For example, let A be a matrix of size $m \times n$, B be a matrix of size $n \times p$ and C be the product of A and B , which is of the size $m \times p$. If the matrices A , B and C each are composed of 2×2 block matrices, all block matrices of C , which are C_{11} , C_{12} , C_{21} and C_{22} , can be created from block matrices of A and B as shown in figure. 6.

$$A_{(m \times n)} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, B_{(n \times p)} = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$A \times B = C_{(m \times p)} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11} \times B_{11} + A_{12} \times B_{21} \\ C_{12} &= A_{11} \times B_{12} + A_{12} \times B_{22} \\ C_{21} &= A_{21} \times B_{11} + A_{22} \times B_{21} \\ C_{22} &= A_{21} \times B_{12} + A_{22} \times B_{22} \end{aligned}$$

Where

Figure 6 Example of matrix multiplications of block matrices

The method partitions the multiplier and the multiplicand into four submatrices each by dividing both rows and columns. Then submatrices from the multiplier and the multiplicand are paired for matrix multiplications. Total of 8 pairs of submatrices multiplication are prepared. The pairing is equivalent to the multiplication of two 2x2 matrix. Two pairs of multiplication must be performed for each submatrix in the product. The upper left submatrix of the product (C_{11}) is the addition of the products from the multiplication between the upper two submatrices of the multiplier (A_{11}, A_{12}) and the two left submatrices of the multiplicand (B_{11}, B_{21}). The upper right submatrix of the product is the addition of the products from the multiplication between the same upper two submatrices in the multiplier and the two right submatrices of the multiplicand (B_{12}, B_{22}). The lower left submatrix of the product is the addition of the products from the multiplication between the lower two submatrices of the multiplier (A_{21}, A_{22}) and the two left submatrices of the multiplicand. The lower right submatrix of the product is the addition of the products from the multiplication between the same lower two submatrices in the multiplier and the two right submatrices of the multiplicand. After the multiplications are performed, each product submatrices are concatenated, resulting in the final product (C).

Based on the observation that each of the eight submatrices may be dense or may be sparse, different multiplication method can be selected for each pair of submatrices multiplication for the most efficient performance. The multiplication

methods available in TensorFlow are *tf.matmul* and *tf.sparse_matmul*. The first method, *tf.matmul* is optimized for dense matrix multiplication. The method does not consider the zero values inside the input matrices. The second method, *tf.sparse_matmul* is optimized for multiplication of sparse matrices. The *tf.sparse_matmul* has additional parameter to specify if either the multiplier and multiplicand matrices are sparse, resulting in three possible configuration of *tf.sparse_matmul*, for when the multiplier matrix is sparse, when the multiplicand matrix is sparse, or when both input matrices are sparse. For multiplication of dense matrices, *tf.matmul* is selected as the multiplication method. For pairs of submatrices that include at least one sparse matrix, one of the three configurations of *tf.sparse_matmul* is selected as the multiplication method. The criteria to determine if a submatrix is dense or sparse must be determined before the *block_adapt_matmul* method can be implemented. The criteria are used in the *selectMatmulMethod* function which take density of two matrices as input (DA and DB) and return either the *tf.matmul* method or one of the three configurations of *tf.sparse_matmul* as an output (*ABmatmulmethod*), described in chapter 4.1. After multiplication of each pair of submatrices are performed using their determined multiplication method, the product from each pair is combined into the final result matrix. The pseudocode of *block_adapt_matmul* can be shown in figure 7 below.

Function: *block_adapt_matmul* (A, B, C) where A, B and C are matrices of size $m \times n, n \times p$ and $m \times p$ respectively:

1. Divide A into $A_{11}, A_{12}, A_{21}, A_{22}$ of size $(\frac{m}{2} \times \frac{n}{2})$
2. Divide B into $B_{11}, B_{12}, B_{21}, B_{22}$ of size $(\frac{n}{2} \times \frac{p}{2})$
3. Calculate density of $A_{11}, A_{12}, A_{21}, A_{22}$ as $D_{A11}, D_{A12}, D_{A21}, D_{A22}$, respectively
4. Calculate density of $B_{11}, B_{12}, B_{21}, B_{22}$ as $D_{B11}, D_{B12}, D_{B21}, D_{B22}$, respectively
5. Determine the multiplication method for each pair of submatrices.

$$A_{11}B_{11}matmulmethod = selectMatmulMethod(D_{A11}, D_{B11})$$

$$A_{11}B_{12}matmulmethod = selectMatmulMethod(D_{A11}, D_{B12})$$

$$A_{12}B_{21}matmulmethod = selectMatmulMethod(D_{A12}, D_{B21})$$

$$A_{12}B_{22}matmulmethod = selectMatmulMethod(D_{A12}, D_{B22})$$

$$A_{21}B_{11}matmulmethod = selectMatmulMethod(D_{A21}, D_{B11})$$

$$A_{21}B_{12}matmulmethod = selectMatmulMethod(D_{A21}, D_{B12})$$

$$A_{22}B_{21}matmulmethod = selectMatmulMethod(D_{A22}, D_{B21})$$

$$A_{22}B_{22}matmulmethod = selectMatmulMethod(D_{A22}, D_{B22})$$

6. Perform matrix multiplication on each pair of submatrices.

$$C_{11} = A_{11}B_{11}matmulmethod(A_{11}, B_{11}) + A_{12}B_{21}matmulmethod(A_{12}, B_{21})$$

$$C_{12} = A_{11}B_{12}matmulmethod(A_{11}, B_{12}) + A_{12}B_{22}matmulmethod(A_{12}, B_{22})$$

$$C_{21} = A_{21}B_{11}matmulmethod(A_{21}, B_{11}) + A_{22}B_{21}matmulmethod(A_{22}, B_{21})$$

$$C_{22} = A_{21}B_{12}matmulmethod(A_{21}, B_{12}) + A_{22}B_{22}matmulmethod(A_{22}, B_{22})$$

7. Combine $C_{11}, C_{12}, C_{21}, C_{22}$ into C .

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

8. Return C

Figure 7 Pseudocode of *block_adapt_matmul*

The *block_adapt_matmul* is implemented in python using TensorFlow library and can be described in as a computation graph shown in figure 8. In figure 8, A, B and C are the two input matrices and the result matrix, respectively. The subscripted A s and B s are submatrices derived from their corresponding input matrix. The subscripted D s are the density of the submatrices. The input matrices are partitioned

using the *tf.split* method. To find the density of each submatrix, the method *tf.count_nonzero* is used to find the number of non-zero elements in the input matrix and the function *tf.size* is used to find the size or the total number of elements in the input matrix. The function *selectMatmulMethod* uses the density of each submatrix to determine the appropriate method for the multiplication of a pair of matrices according to predetermined criteria. All submatrices of the product are combined using *tf.add* and *tf.concat*. The method *tf.add* is used to add the products of submatrix multiplications, and the output of these operations are assembled into the final matrix multiplication product *tf.concat*.

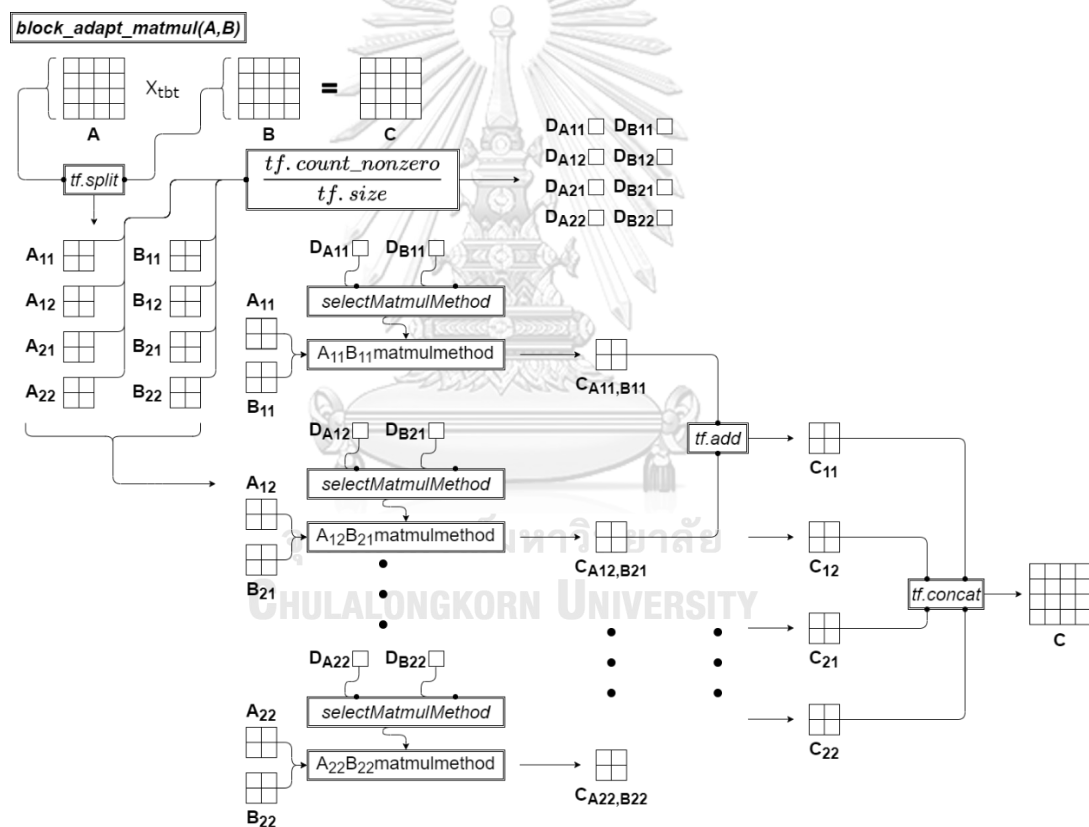


Figure 8 The diagram of the algorithm for `block_adapt_matmul`

Chapter 4

Experiments

All experiments were performed on a laptop with Intel Core i7-4720HQ quad-core CPU with 16GB DDR3 RAM and Nvidia Geforce GTX 870M GPU with 3GB VRAM. The operating system used was Linux Ubuntu 18.04.2. The proposed matrix multiplication methods are implemented in Python 3.6.5 and TensorFlow r1.11. The computation time used in all performance evaluation is the average of the execution time of the multiplication of 20 pairs of matrices generated randomly.

In this chapter, we will describe the experiments performed in this work. We compare the performance of our proposed matrix multiplication method and the TensorFlow built-in multiplication methods, *tf.matmul* and *tf.sparse_matmul* on input matrices with different non-zero values distribution patterns. A preliminary experiment is performed to determine the criteria to select the appropriate matrix multiplication method and it is described in section 4.1. In section 4.2, performance of *block_adapt_matmul* is evaluated against TensorFlow's provided methods, *tf.matmul* and *tf.sparse_matmul* on CPU.

4.1 Finding the Criteria for Choosing between Dense Matrix Multiplication and Sparse Matrix Multiplication Methods

Because the performance of the proposed matrix multiplication method, *block_adapt_matmul*, depends on the matrix multiplication method chosen for each pair of the submatrices, it is necessary to determine the criteria for choosing one of TensorFlow's multiplication methods for submatrices. TensorFlow offers a multiplication method for dense matrices which is *tf.matmul*, and a multiplication method for sparse matrices, which is *tf.sparse_matmul*. For *tf.sparse_matmul* there are three variations of the method which are used according to the argument of the method. This argument is used to indicate which of the input matrices is sparse. The argument *a_is_sparse* indicates that the first input matrix is sparse, the argument *b_is_sparse* indicates that the second input matrices is sparse, and the parameter

ab_is_sparse indicates that both of the input matrices are sparse. An optimized sparse matrix multiplication method is used according to these parameters.

In order to optimize the performance of the proposed multiplication method, *block_adapt_matmul*, it is needed to find the cut-off criteria to determine when one of these four configurations of TensorFlow's matrix multiplication is best for each pair of input matrices. The criteria for selecting multiplication methods used in *selectMatmulMethod* in the algorithm of *block_adapt_matmul* is shown in table 1. For multiplication of arbitrary matrices, A and B, each column in table 1 represents the density of the multiplier matrix or matrix A, while each row in table 1 represents the density of the multiplicand matrix or matrix B. The different colors in the table represent the multiplication method selected based on the densities of the input matrices A and B. The cell is shown in pink when *tf.matmul* is chosen, the cell is shown in blue when *tf.matmul(a_is_sparse)* is chosen, the cell is shown in yellow when *tf.matmul(b_is_sparse)* is chosen, and the cell is shown in green when *tf.matmul(ab_is_sparse)* is chosen. This criteria is derived from the experiment to find the fastest multiplication method for matrices of various densities.

Table 1 Criteria for selecting multiplication methods in *block_adapt_matmul*

DA (%) \ DB (%)											
		10	20	30	40	50	60	70	80	90	100
	10	Green	Green	Yellow	Yellow	Yellow	Yellow	Yellow	Yellow	Yellow	Yellow
	20	Green	Green	Yellow	Yellow	Yellow	Yellow	Yellow	Yellow	Yellow	Yellow
	30	Green	Green	Green	Yellow	Yellow	Yellow	Yellow	Yellow	Yellow	Yellow
	40	Green	Green	Green	Green	Yellow	Yellow	Yellow	Yellow	Yellow	Yellow
	50	Blue	Blue	Blue	Blue	Blue	Blue	Pink	Pink	Pink	Pink
	60	Blue	Blue	Blue	Blue	Blue	Blue	Pink	Pink	Pink	Pink
	70	Blue	Blue	Blue	Blue	Blue	Blue	Pink	Pink	Pink	Pink
	80	Blue	Blue	Blue	Blue	Blue	Blue	Pink	Pink	Pink	Pink
	90	Blue	Blue	Blue	Blue	Blue	Blue	Pink	Pink	Pink	Pink
	100	Blue	Blue	Blue	Blue	Blue	Blue	Pink	Pink	Pink	Pink

tf.sparse_matmul(ab_is_sparse)

tf.sparse_matmul(a_is_sparse)

tf.sparse_matmul(b_is_sparse)

tf.matmul

In the initial steps of determining the criteria, an experiment is performed to compare the computation time of these four matrix multiplication methods. All four multiplication methods were executed on input matrices with different density, varying from 10%, 20% up to 100%, and different sizes, including 3000x3000, 6000x6000 and 9000x9000. Then, for each combination of input matrix densities, the average execution time of each method is calculated from multiplication of 20 pairs of matrices from each matrix size. The fastest multiplication method for each combination of matrices density according to the experiment is shown in table 2.

Table 2 The fastest matrix multiplication method for each combination of input matrices densities measured from experiments.

		D _A (%)									
		10	20	30	40	50	60	70	80	90	100
D _B (%)	10										
	20										
	30										
	40										
	50										
	60										
	70										
	80										
	90										
	100										

tf.sparse_matmul(ab_is_sparse)

tf.sparse_matmul(a_is_sparse)

tf.sparse_matmul(b_is_sparse)

tf.matmul

From table 2, the multiplication method *tf.matmul* and *tf.sparse_matmul(b_is_sparse)* are the fastest multiplication methods in ranges of input matrices densities that are easily identifiable. Therefore, we use the result from the experiment directly as the criteria for selecting these two methods as shown in table 3. That is, the first matrix, A, is considered dense when its density exceeds 60%, and the second matrix, B, considered sparse when its density does not exceed 40%.

Table 3 The criteria for deciding whether matrix B is sparse or dense

D _A (%) \ D _B (%)		D _B (%)									
		10	20	30	40	50	60	70	80	90	100
D _A (%)	10										
	20										
	30										
	40										
	50										
	60										
	70										
	80										
	90										
	100										

↑

↓

tf.sparse_matmul(b_is_sparse)

B is considered as a sparse matrix

tf.matmul

B is considered as a dense matrix

However, based on this criteria, the first matrix, A, is considered dense when its density exceeds 60%, and the second matrix, B, is considered sparse when its density does not exceeds 40%. If these cut-off points are used to determine if the two matrices are dense or sparse, there are some cases that we will not choose the fastest method. For example, when the density of matrix B is 30%, in which B should be considered sparse, and the density of matrix A is 20%, in which A is considered sparse, we should use *tf.sparse_matmul(ab_is_sparse)*. But, from the experiment, *tf.sparse_matmul(a_is_sparse)* is faster than *tf.sparse_matmul(ab_is_sparse)*, as shown in table 2. However, when we compare the average execution time of *tf.sparse_matmul(a_is_sparse)* and *tf.sparse_matmul(ab_is_sparse)* in these cases, which is shown in table 4, it is found that the difference is very small, i.e. it does not exceed 5%.

Table 4 The difference between the execution time of the two fastest multiplication methods

		D _A (%)									
		10	20	30	40	50	60	70	80	90	100
D _B (%)	10	.35%	.31%								
	20										
	30		.64%		2.25%						
	40	1.30%	.32%		2.11%	1.04%					
	50	.90%	.25%	.20%							
	60		.19%								
	70	.44%	.89%	.72%							
	80	1.89%		2.49%		.73%	.09%				
	90	.83%	2.38%	.31%							
	100	.73%	2.27%	.25%	.15%		<.01%				

tf.sparse_matmul(ab_is_sparse)

tf.sparse_matmul(a_is_sparse)

tf.sparse_matmul(b_is_sparse)

tf.matmul

4.2 Performance Evaluation of *block_adapt_matmul* on CPU

The proposed *block_adapt_matmul* is evaluated against TensorFlow's *tf.matmul* and *tf.sparse_matmul* on CPU. All three matrix multiplication methods, *block_adapt_matmul*, TensorFlow's *tf.matmul* and TensorFlow's *tf.sparse_matmul* were performed on input matrices of size 3000x3000, 6000x6000, 9000x9000, and 12000x12000. To study the performance of the proposed method on matrices with uneven distribution of non-zero elements, the experiments are performed on input matrices with varying degree of unevenness. Uneven-density matrices are generated randomly by creating submatrices with some low-density submatrices and some high-density submatrices, as shown in figure 9. The density of matrices used in the experiments have different degree of unevenness as shown in Table 5. The evaluation is performed by measuring the average execution time of matrix multiplication of each input matrices setting. Furthermore, for each density combination of the higher-density and the lower-density regions, the performance is also measured in order to evaluate the performance of the proposed method for matrices with different degree of unevenness of the matrix density.

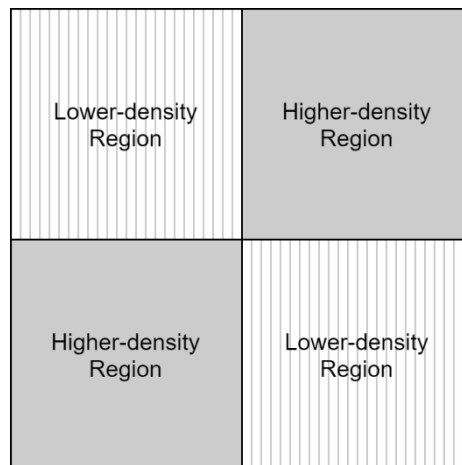


Figure 9 Checkered pattern of higher density and lower density input matrix

Table 5 Density of submatrices and their overall density

Overall Density	(Higher Density, Lower Density, Density Difference)			
30%	(50%,10%,40)	(40%,20%,20)		
40%	(70%,10%,60)	(60%,20%,40)	(50%,30%,20)	
50%	(90%,10%,80)	(80%,20%,60)	(70%,30%,40)	(60%,40%,20)
60%	(100%,20%,80)	(90%,30%,60)	(80%,40%,40)	(70%,50%,20)
70%	(100%,40%,60)	(90%,50%,40)	(80%,60%,20)	
80%	(100%,60%,40)	(90%,70%,20)		

Chapter 5

Results and Discussion

In this chapter, we will describe the results of the experiments described in Chapter 4 to evaluate the proposed method. The performance of *block_adapt_matmul* is compared with the performance of the TensorFlow built-in methods *tf.matmul* and *tf.sparse_matmul*.

The performance of *block_adapt_matmul* is evaluated with respect to two different independent variables. The first variable is the overall density of the input matrices, and the performance of *block_adapt_matmul* for different matrix densities is described in section 5.1. The second variable indicates how uneven the non-zero values is distributed in the input matrices. For this variable, we use the difference between the density of the higher-density and the lower-density regions in the checkered-pattern matrices, called *density difference*. For example, if the density of the higher-density region is 70% and the density of the lower-density region is 30%, the density difference is 40. If a matrix has high density difference, the density of the dense part of the matrix is much higher than the density of the sparse part, which means that the non-zero values distribution in the input matrices is more uneven. The performance of *block_adapt_matmul* for different degree of density difference is described in section 5.2. Additionally, we also examine how the performance of the proposed method varies with respect to the size of the input matrices.

5.1 Performance of *block_adapt_matmul* with Varying Overall Density of Input Matrices

The performance of *block_adapt_matmul*, *tf.matmul*, and *tf.sparse_matmul* is measured as the execution time in seconds. Figure 10 shows the execution time of all three multiplication methods for input matrices with overall density of 30%, 40%, 50%, 60%, 70%, and 80%. For one overall density, we consider various values of density difference, i.e. we use all density for the high-density and the low-density regions as shown in table 5, and the execution time is the average time for all cases. In figure 10 the, x-axis is the overall density of input matrices, and the y-axis is the

execution time. Each line represents the execution time of a multiplication method for one matrix size. The execution time of *block_adapt_matmul*, *tf.sparse_matmul*, and *tf.matmul* are shown in red, blue and green lines, respectively. Different marks on the line show the execution time for different matrix sizes. The execution time for 3000x3000 matrices, 6000x6000 matrices, 9000x9000 matrices, and 12000x12000 matrices are shown with \dagger , \blacksquare , \ast and \times , respectively.

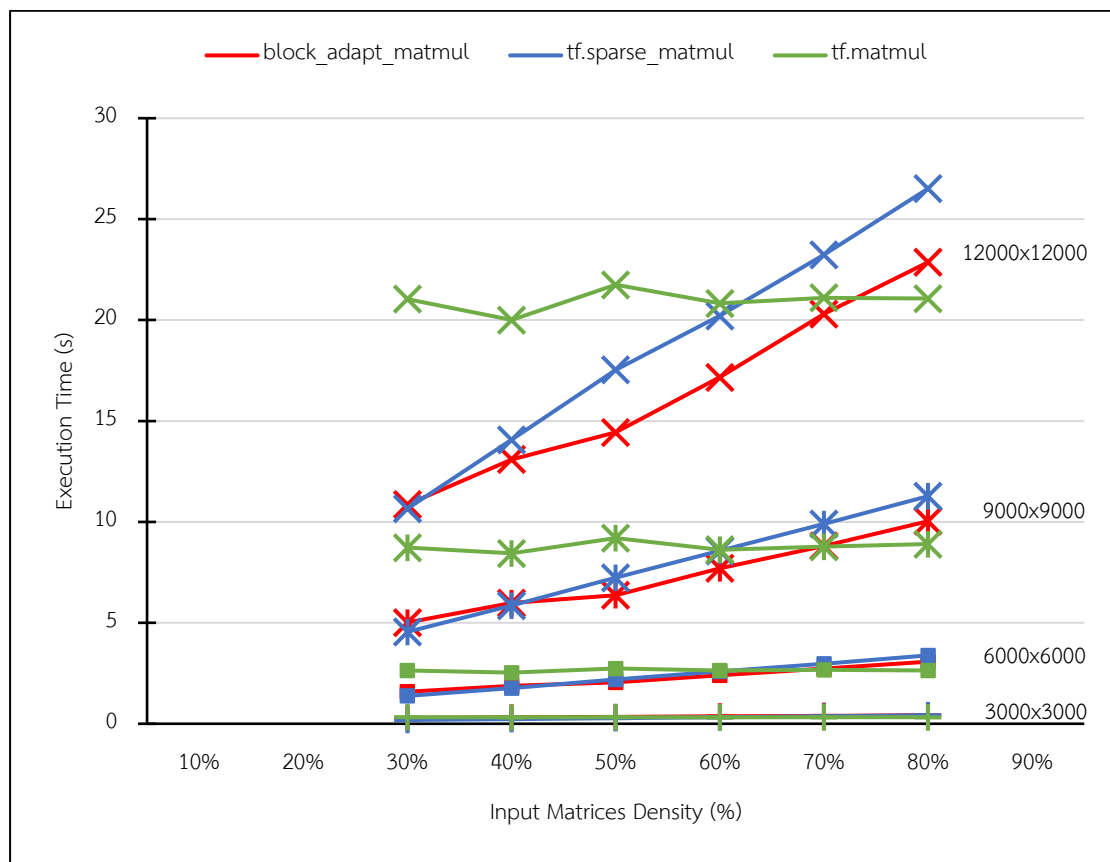


Figure 10 Execution times of *block_adapt_matmul*, *tf.sparse_matmul* and *tf.matmul* on matrices of sizes 3000x3000, 6000x6000, 9000x9000, and 1200x12000 with varying overall density.

As shown in figure 10, both *tf.sparse_matmul* and *block_adapt_matmul* are faster than *tf.matmul* for matrices with lower overall density. On the other hand, the execution time of *tf.matmul* almost does not vary with the overall density of the matrices. When the overall density is low (e.g. 30%), *block_adapt_matmul* is faster

than *tf.matmul*, but slightly slower than *tf.sparse_matmul*. When the overall density is high (e.g. 80%), *block_adapt_matmul* is faster than *tf.sparse_matmul*, but slightly slower than *tf.matmul*. However, for medium-density matrices (e.g. 60%), *block_adapt_matmul* is faster than both *tf.matmul* and *tf.sparse_matmul*. Specifically, for 12000x12000 matrices, *block_adapt_matmul* is faster than both *tf.matmul* and *tf.sparse_matmul* when the overall density is between 40% and 70%.

However, when the size of the input matrices is smaller, the performance difference between *block_adapt_matmul* and the two TensorFlow's built-in methods are also smaller. The performance difference between the *block_adapt_matmul* and the two built-in methods are measured as the speedup of the proposed method compared to each existing method. The speedup is calculated from the ratio of average execution time of an existing method to the average execution time of the proposed method. Figure 11 shows the speedup of *block_adapt_matmul* compared to *tf.matmul* and figure 12 shows the speedup of *block_adapt_matmul* compared to *tf.sparse_matmul*. For both figures, the x-axis is the sizes of the input matrices, and the y-axis is the speedup of *block_adapt_matmul* compared to each built-in method. The speedup for matrices of different densities are shown in different colors. The blue, orange, grey, yellow, red, and green lines are the speedup for input matrices with density of 30%, 40%, 50%, 60%, 70%, and 80%, respectively. When the speedup is 1.0, shown by the dashed thick line in figure 11 and 12, the performance of the proposed method and the reference method are equally good. When the speedup is higher than 1.0, i.e. above the dashed thick line, the proposed method is faster than the reference method. On the other hand, when the speedup is lower than 1.0, i.e. below the dashed thick line, the proposed method is slower than the reference method.

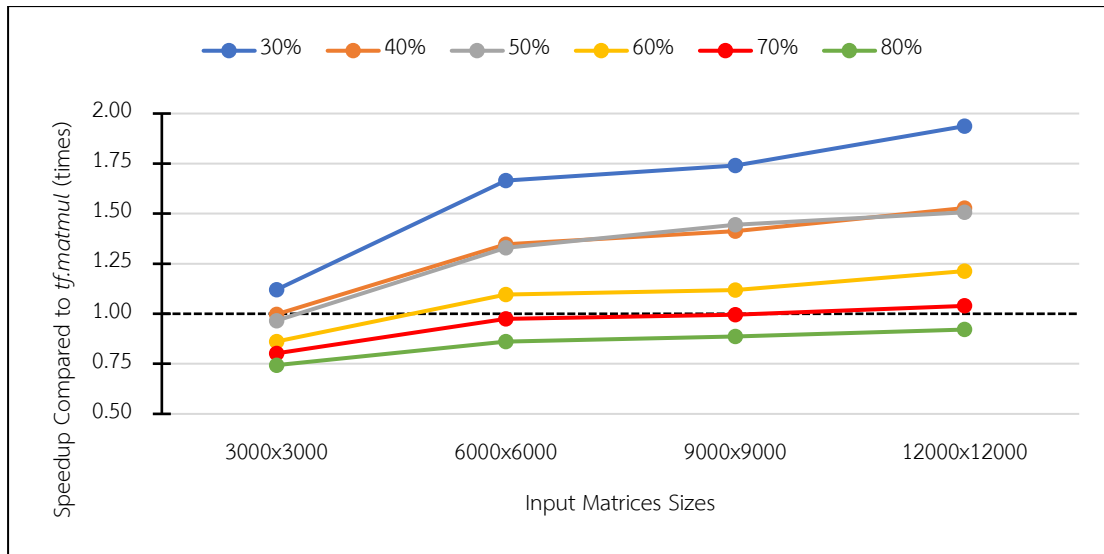


Figure 11 Speedup of *block_adapt_matmul* compared to *tf.matmul* for matrices of overall density of 30%, 40%, 50%, 60%, 70%, and 80%, with varying sizes.

From both figure 11 and figure 12, when compared to both *tf.matmul* and *tf.sparse_matmul*, the speedup of *block_adapt_matmul* increases as the matrix size increases. Compared to *tf.matmul* as shown in figure 11, the performance of *block_adapt_matmul* is significantly faster than *tf.matmul* for larger matrices with density of lower than 60%. Otherwise, the *block_adapt_matmul* is slower than *tf.matmul*. However, the performance of *block_adapt_matmul* drastically decreases for smaller matrices such as 3000x3000 matrices because the overhead of dividing and merging submatrices outweighs the gain of using appropriate multiplication methods.

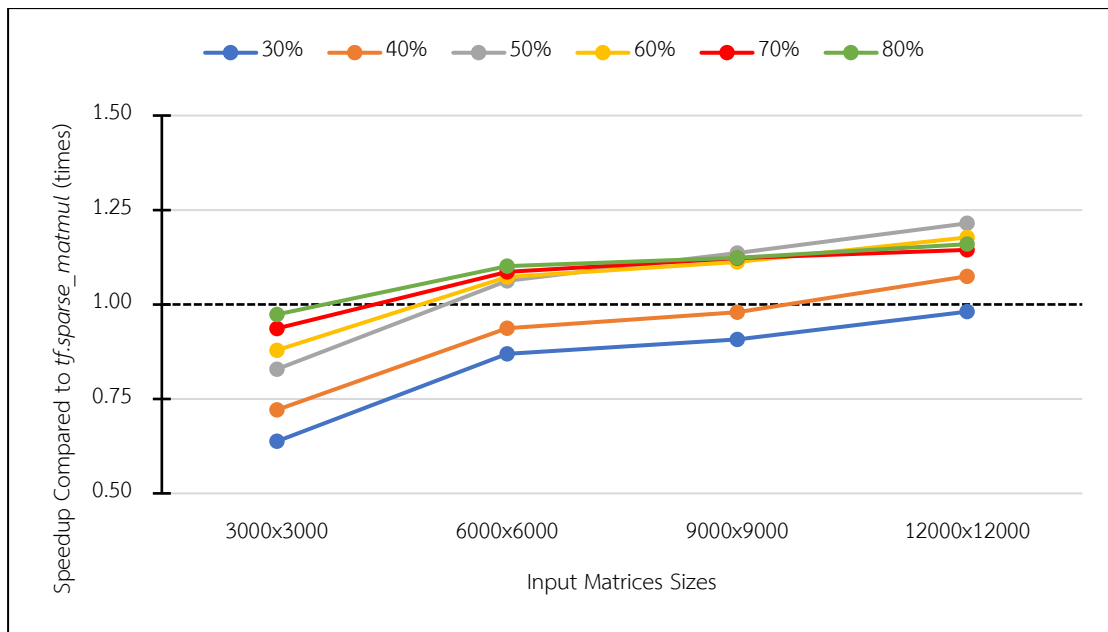


Figure 12 Speedup of `block_adapt_matmul` compared to `tf.sparse_matmul` for matrices of overall density of 30%, 40%, 50%, 60%, 70%, and 80%, with varying sizes

Compared to `tf.sparse_matmul` as shown in figure 12, `block_adapt_matmul` is faster than `tf.sparse_matmul` for 12000x12000 matrices when the density is as low as 40%. For small matrices such as 3000x3000 matrices, `block_adapt_matmul` is slower than `tf.sparse_matmul` because the overhead is more than the gain of reducing the multiplication. For medium-size matrices such as 6000x6000 matrices, `block_adapt_matmul` is faster than `tf.sparse_matmul` when the density of matrix is higher than 40%.

As we consider matrices with average distribution of nonzero values, the proposed method is not faster than `tf.matmul` and `tf.sparse_matmul` in some cases. We further examine the performance of the proposed method when the density difference is varied.

5.2 Performance of *block_adapt_matmul* with Varying Density Differences of Input Matrices

In this section, the execution time of *block_adapt_matmul*, *tf.sparse_matmul*, and *tf.matmul* is measured for matrices with varying density differences, i.e. the difference between the density of the higher-density region and the lower-density region of the input matrices. Figure 13 shows the execution time of each multiplication method on input matrices of size 3000x3000, 6000x6000, 9000x9000, and 12000x12000, with density differences of 20, 40, 60, and 80. The y-axis is the execution time in seconds, and the x-axis is the density differences. The execution time of *block_adapt_matmul*, *tf.sparse_matmul*, and *tf.matmul* are shown in red, blue and green lines, respectively. Different marks on the line show the execution time for different matrix sizes. The execution time for 3000x3000 matrices, 6000x6000 matrices, 9000x9000 matrices, and 12000x12000 matrices are shown with **+**, **■**, ***** and **×**, respectively.

As shown in figure13, for all matrix sizes, the average execution time of the two TensorFlow's built-in methods *tf.matmul* and *tf.sparse_matmul* do not vary much with the density difference, and *tf.sparse_matmul* is slightly faster than *tf.matmul*. On the other hand, the proposed method *block_adapt_matmul* is faster when the density difference is higher. Furthermore, the performance of *block_adapt_matmul* is comparable to *tf.sparse_matmul* when the density difference is low.

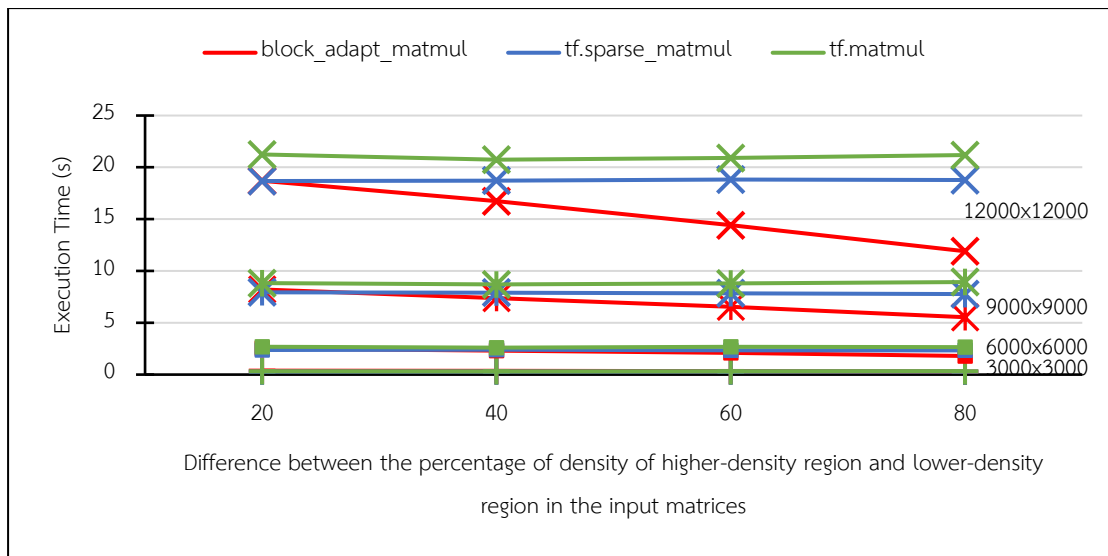


Figure 13 Execution times of *block_adapt_matmul*, *tf.sparse_matmul* and *tf.matmul* on matrices of size 3000x3000, 6000x6000, 9000x9000, and 12000x12000 with varying density difference

Figure 14 and figure 15 show the speedup of *block_adapt_matmul* compared to *tf.matmul* or *tf.sparse_matmul*, and the y-axis is the speedup and the x-axis is the matrix size. For a density difference d , the average execution time is measured from the multiplication of matrices with the density difference d with all the overall densities. For example, for the density difference of 40% we use matrices with overall density of 40%, 50%, 60%, 70% and 80%. The speedups when the density difference is 20, 40, 60 and 80 are shown in blue line, orange line, grey line, and yellow line, respectively. Both figure 14 and figure 15 show that the speedup increases when the density difference increases. The *block_adapt_matmul* is faster when the density difference is at least 40%. However, when the matrix is as small as 3000x3000, the proposed method is only faster than *tf.matmul* when the density difference is as high as 80%.

From all experiments, we can conclude that the performance of our proposed method increases as the density difference of the input matrix increases. When the density difference of the input matrix is large, the proposed method outperforms both TensorFlow's built-in methods, except for small matrices. As a

result, our method should be used when the distribution of nonzero values in the matrix is uneven.

When we measure the average execution time for matrices with varying density difference, our proposed method is faster than both TensorFlow's built-in methods for medium-density matrices. However, when the distribution of nonzero values in the matrix is even, *tf.matmul* is still faster for very-dense matrices and *tf.sparse_matmul* is still faster for very-sparse matrices.

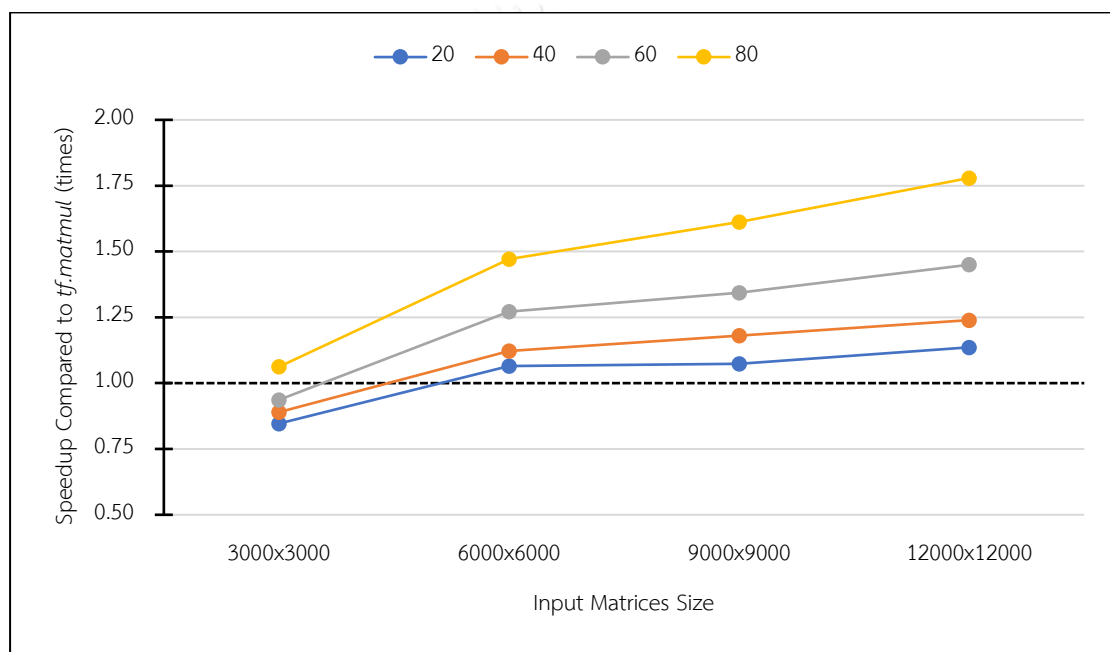


Figure 14 Speedup of *block_adapt_matmul* compared to *tf.matmul* for matrices of density differences of 20, 40, 60 and 80 with varying sizes

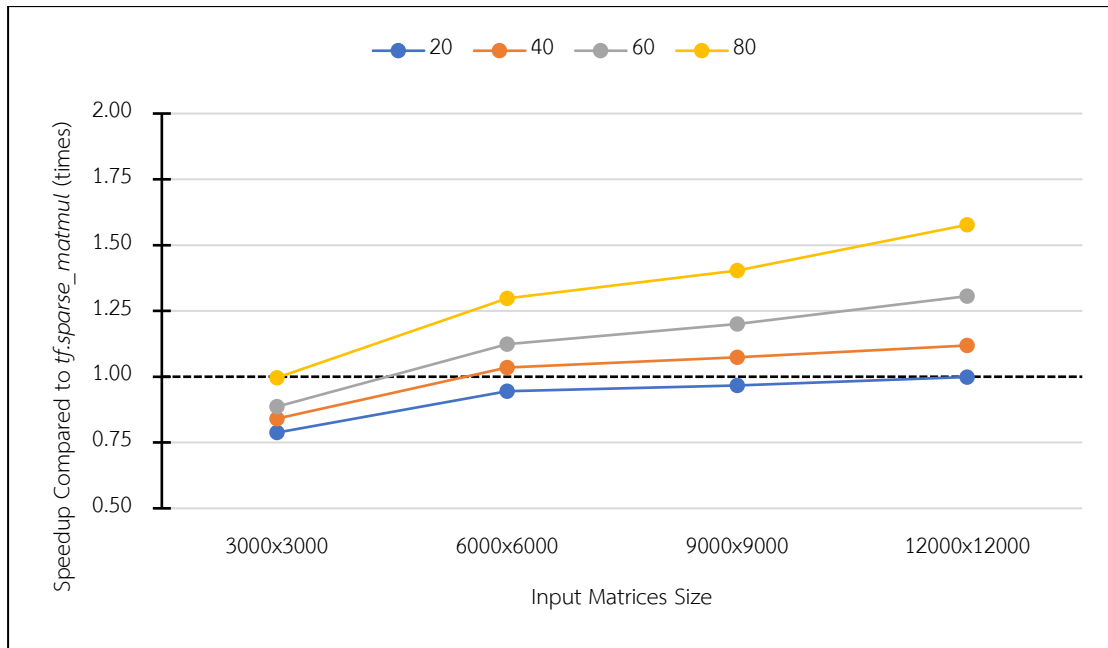


Figure 15 Speedup of `block_adapt_matmul` compared to `tf.sparse_matmul` for matrices of density differences of 20, 40, 60 and 80 with varying sizes

Chapter 6

Conclusion

In conclusion, we proposed an implementation of matrix multiplication, which is called block-adaptive matrix multiplication or *block_adapt_matmul*, on TensorFlow platform. The proposed multiplication method is optimized for matrices which have large degree of unevenness of non-zero values distribution.

The method *block_adapt_matmul* partitions each input matrix into four smaller submatrices or “blocks”, and then multiplies each pair of blocks using the appropriate multiplication method, chosen between *tf.matmul* or *tf.sparse_matmul* depending on the density of each pair of submatrices. This method is compared to TensorFlow’s matrix multiplication methods, i.e. *tf.matmul* and *tf.sparse_matmul*, on CPUs. For input matrices larger than 6000x6000, the proposed method outperforms *tf.matmul* when the overall density of input matrices is not more than 70% and outperforms *tf.sparse_matmul* when the overall density of input matrices is at least 50%. The performance of the proposed method increases proportionately with the density differences of the input matrices. Thus, the proposed method *block_adapt_matmul* is suitable for medium density matrices with uneven non-zero values distribution.

In conclusion, our proposed method *block_adapt_matmul* can perform well when the distribution of zeros in the input matrices is uneven. It performs better than both TensorFlow’s matrix multiplication methods for large medium-density matrices. However, when it is slower than one of the two TensorFlow’s matrix multiplication methods, the difference is small. The performance of this method depends on the difference between the performance of the two TensorFlow’s matrix multiplication methods, which reflects on the criteria for choosing the multiplication method for submatrices. An advantage of our proposed method is that it can be implemented on any platform that supports TensorFlow. Additionally, the idea that appropriate multiplication methods being applied to different submatrices could be extended to any platform similar to TensorFlow that provides many implementations of matrix multiplication methods.

However, the disadvantage of our proposed method is that it can only be used on CPUs but not on GPUs because it depends of the built-in method *tf.sparse_matmul*, which is not supported on GPUs. As a preliminary step for future works, on GPUs, we have also explored using similar concept of separating input matrices into multiple submatrices and process each submatrix differently according to their properties. We divide input matrices into submatrices by either rows or columns based on their density. Then we applied *tf.matmul*, which is supported on GPUs directly to the dense submatrices, while each sparse submatrix is packed into a dense matrix before being applied the same method. We found that the performance of multiplying matrices directly using *tf.matmul* on GPUs is significantly more efficient compared to our pilot method most likely because *tf.matmul* is more suitable with GPUs SIMD parallelism.



REFERENCES

1. Strang, S., *Linear Algebra and Its Applications*. 2018: Cengage Learning.
2. Nielsen, M.A., *Neural Networks and Deep Learning*. 2015: Determination Press.
3. Cormen, T.H., *Introduction to Algorithms, 3rd Edition*. 2009: MIT Press.
4. Russell, S.J., et al., *Artificial Intelligence: A Modern Approach*. 2010: Prentice Hall.
5. Jones, E., et al., *SciPy: Open source scientific tools for Python*. 2001.
6. Abadi, M., et al., *TensorFlow: a system for large-scale machine learning*, in *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*. 2016, USENIX Association: Savannah, GA, USA. p. 265–283.
7. Davis, P.J., *Circulant Matrices*. 1979: Wiley.
8. Devlin, J., et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. in *NAACL-HLT*. 2019.
9. Mikolov, T., et al., *Efficient Estimation of Word Representations in Vector Space*. CoRR, 2013. **abs/1301.3781**.
10. Pennington, J., R. Socher, and C. Manning. *GloVe: Global Vectors for Word Representation*. in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014. Doha, Qatar: Association for Computational Linguistics.
11. Strassen, V., *Gaussian elimination is not optimal*. *Numer. Math.*, 1969. **13**(4): p. 354–356.
12. Coppersmith, D. and S. Winograd, *Matrix multiplication via arithmetic progressions*, in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 1987, Association for Computing Machinery: New York, New York, USA. p. 1–6.
13. Stothers, A.J. *On the complexity of matrix multiplication*. 2010.
14. Davie, A.M. and A.J. Stothers, *Improved bound for complexity of matrix multiplication*. *Proceedings of The Royal Society A: Mathematical, Physical and Engineering Sciences*, 2013. **143**: p. 351-369.

15. Chien, S.W.D., et al., *TensorFlow Doing HPC*, in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2019. p. 509-518.
16. Wu, T., et al., *An efficient sparse-dense matrix multiplication on a multicore system*. 2017 IEEE 17th International Conference on Communication Technology (ICCT), 2017: p. 1880-1883.
17. Shah, M. and V. Patel, *An Efficient Sparse Matrix Multiplication for Skewed Matrix on GPU*, in *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. 2012. p. 1301-1306.
18. Bell, N. and M. Garland. *Implementing sparse matrix-vector multiplication on throughput-oriented processors*. in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009.
19. Yang, C., Y. Wang, and J.D. Owens, *Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU*, in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 2015. p. 841-847.
20. Shah, V., J. Gilbert, and S. Reinhardt, *Some Graph Algorithms in an Array-Based Language*. 2011. p. 29-44.
21. Lee, J., et al., *Optimization of GPU-based Sparse Matrix Multiplication for Large Sparse Networks*, in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 2020. p. 925-936.
22. Guennebaud, G., B. Jacob, and others, *Eigen v3*. 2010.
23. Wang, E., et al., *Intel Math Kernel Library*. 2014. p. 167-188.
24. Chetlur, S., et al., *cuDNN: Efficient Primitives for Deep Learning*. ArXiv, 2014. **abs/1410.0759**.
25. *TensorFlow Python API Documentation*. Available from: https://www.tensorflow.org/api_docs/python/.



จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

VITA

NAME Siraphob Theeracheep

DATE OF BIRTH 03 Feb 1994

PLACE OF BIRTH Bangkok, Thailand

INSTITUTIONS ATTENDED Mr. Siraphob Theeracheep graduated in 2016 from the Department of Biochemistry, Faculty of Science, Chulalongkorn University with a degree in Bachelor of Science (B.Sc.) (1st Class Honours)

HOME ADDRESS 660/925 Ideo Q Chula-Samyan, Rama 4 road, Si-praya, Bangrak, Bangkok, 10500

PUBLICATION "Multiplication of medium-density matrices using TensorFlow on multicore CPUs", Tehnički glasnik, vol.13, no. 4, pp. 286-290, 2019.