ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20191104183930

Original scientific paper

MULTIPLICATION OF MEDIUM-DENSITY MATRICES USING TENSORFLOW ON MULTICORE CPUs

Siraphob THEERACHEEP, Jaruloj CHONGSTITVATANA

Abstract: Matrix multiplication is an essential part of many applications, such as linear algebra, image processing and machine learning. One platform used in such applications is TensorFlow, which is a machine learning library whose structure is based on dataflow programming paradigm. In this work, a method for multiplication of medium-density matrices on multicore CPUs using TensorFlow platform is proposed. This method, called *tbt_matmul*, utilizes TensorFlow built-in methods *tf.matmul* and *tf.sparse_matmul*. By partitioning each input matrix into four smaller sub-matrices, called tiles, and applying an appropriate multiplication method to each pair depending on their density, the proposed method outperforms the built-in methods for matrices of medium density and matrices of significantly uneven distribution of non-zeros.

Keywords: Sparse matrix; Matrix multiplication; TensorFlow

1 INTRODUCTION

Matrix multiplication is a basis for computation in many areas, such as linear algebra, machine learning and image processing. Various implementations of matrix multiplication are studied in different environments. For dense matrices, many efficient matrix multiplication algorithms such as Strassen algorithm [1] and Coppersmith-Winograd algorithm [2] are proposed. In these algorithms all matrices are treated as dense matrices. Another approach to improve the efficiency of matrix multiplication is based on the sparsity of matrices. In [3], a hardware accelerator for neural networks, utilizing irregularity in sparse neural networks and the sparsity of weight matrices, is developed. In [4], a sparse tensor representation, F-COO, is proposed, and, based on F-COO, sparse tensors operations and sparse matrix multiplication on GPU are optimized.

TensorFlow [5] also provides the matrix multiplications for both dense matrices and sparse matrices. Like many other lower-level functions, these matrix multiplication methods are built on various kernel-implementations, depending on the platform. For example, methods in Eigen [6] are built for dense and sparse matrix multiplication on CPUs and methods in cuBLAS [7] are built for dense matrix multiplication on GPUs.

Medium-density matrices cannot benefit from matrix multiplication designed for sparse matrices. However, a medium-density matrix can be divided into sub-matrices, some of which are dense, and some are sparse. In this paper, we propose an approach to reduce the computation time for medium-density matrix multiplication. This approach divides a medium-density matrix into four equal-size sub-matrices and chooses an appropriate matrix multiplication method for each pair of sub-matrices, based on the density of the sub-matrices.

Using this approach, we implement a matrix multiplication function for medium-density matrices on TensorFlow. An experiment is performed to compare the proposed method to TensorFlows's matrix multiplication for dense matrices and that for sparse matrices. It is found that the proposed method is faster than both matrix multiplication methods in TensorFlow for medium-density matrices.

The proposed approach can be applied for matrix multiplication on any platform, and any efficient implementation of matrix multiplication can be used for sub-matrices. This is an advantage of this approach because it can benefit from any performance improvement in the low-level libraries.

The remaining sections of this paper are organized as follows. Section 2 presents related works on matrix multiplication for tensors, dense matrices and sparse matrices on various environments to motivate the need for efficient multiplication for medium density matrices. Section 3 describes the proposed matrix multiplication algorithm for medium-density matrices. The performance evaluation of the proposed method is discussed in Section 4. Section 5 presents conclusion and future works.

2 BACKGROUND AND MOTIVATION

Since matrix multiplication is often used in many applications, various implementations of matrix multiplication are studied in different environments. A hardware accelerator for neural network [3] which also includes matrix multiplication, was proposed. The accelerator utilized the sparsity and irregularity of weight matrices of neural network models to improve the computation efficiency. Many efficient matrix multiplication algorithms are proposed for different architectures. For example, in [8], a table and heap-based algorithm for sparse matrix multiplication is optimized for multicore and Intel KNL processors. This implementation outperforms Intel MKL sparse matrix multiplication for large matrices. In [9], software libraries for linear algebra computation, including matrix multiplication, are developed for GPUs and CPUs. Nvidia cuBLAS library [7] is a linear algebra library for Nvidia's GPU. Intel MKL [10] is a mathematical library optimized for Intel's processor. OpenBLAS [11] is an open-source linear algebra library. Higher-level libraries, such as Numpy, scikit-learn, Theano, PyTorch and TensorFlow, are built on these linear algebra and mathematical libraries in order to utilize performance optimization of low-level libraries.

Many of these implementations are designed for dense matrices. On multicore CPUs, dense matrix multiplication can be made faster by maximizing the parallelism while optimizing the overhead. When the operations are performed on sparse matrices, many operations are wasted on multiplying zeroes and no performance improvement can be gained from the sparsity of matrices. The performance gain for sparse matrix multiplication depends on the representation of sparse matrices. F-COO [4] is proposed as a representation of sparse tensors which is used for tensor operations on GPU. A sparse tensor-times- dense matrix multiplication (SpTTM) [12] is implemented for both CPU and GPU platforms.

TensorFlow, developed by Google, is an open-source machine learning library that contains various methods that help in the development and training of machine learning models. It also includes low level functions such as matrix multiplication. On GPUs, TensorFlow provides the matrix multiplication only for dense matrices. But, on CPUs, it provides the matrix multiplication for both dense matrices and sparse matrices. Since TensorFlow provides kernel implementations for many platforms, these multiplication functions are optimized for different platforms. It is suggested that, using TensorFlow on multicore CPUs, the multiplication method for sparse matrices could be used, instead of the dense matrix multiplication method, when the density of the input matrices is lower than 70% [13].

However, a matrix can be divided into some dense sub-matrices and some sparse sub-matrices, and the multiplication of a pair of matrices can be decomposed into the multiplication of pairs of sub-matrices. The idea of multiplying sub-matrices is called tiled matrix multiplication, which is used to perform parallel matrix multiplication by partitioning a pair of matrices into multiple 'tiles' and distribute them to different computing units to perform concurrently. This method allows multiple parts of the result matrix to be computed simultaneously [14], [15], [16]. Based on tiled matrix multiplication, an efficient multiplication method can be chosen for each pair of tiles, instead of using one multiplication method for the whole matrix.

3 PROPOSED MULTIPLICATION FOR MEDIUM-DENSITY MATRICES

We propose an approach for matrix multiplication which is optimized for matrices with medium density. Each of the two input matrices is divided into four sub-matrices. An $m \times n$ matrix is divided into four $m/2 \times n/2$ sub-matrices. Pairs of sub-matrices are multiplied, using an appropriate matrix multiplication method. In order to optimize the computation time, an appropriate multiplication method is chosen for each pair of sub-matrices based on their density. The proposed method, referred to as two-by-two matrix multiplication function *tbt_matmul*, is described below.

Function: *tbt_matmul*(*A, B, C*)

Let $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ be matrices of size $(m \times n)$, $(n \times p)$ and $(m \times p)$, respectively.

Algorithm:

1. Divide $\boldsymbol{A}$ into $\boldsymbol{A}_{11}$, $\boldsymbol{A}_{12}$, $\boldsymbol{A}_{21}$, and A_{22} of size $m/2 \times n/2$.
2. Divide $\boldsymbol{B}$ into $\boldsymbol{B}_{11}$, $\boldsymbol{B}_{12}$, $\boldsymbol{B}_{21}$, and $\boldsymbol{B}_{22}$ of size $n/2 \times p/2$.
3. Find the density of $\boldsymbol{A}_{11}$, $\boldsymbol{A}_{12}$, $\boldsymbol{A}_{21}$, and $\boldsymbol{A}_{22}$ as $\boldsymbol{D}_{A11}$, $\boldsymbol{D}_{A12}$, $\boldsymbol{D}_{A21}$ and $\boldsymbol{D}_{A22}$, respectively.
4. Find the density of $\boldsymbol{B}_{11}$, $\boldsymbol{B}_{12}$, $\boldsymbol{B}_{21}$, and $\boldsymbol{B}_{22}$ as $\boldsymbol{D}_{B11}$, $\boldsymbol{D}_{B12}$, $\boldsymbol{D}_{B21}$ and $\boldsymbol{D}_{B22}$, respectively.
5. Choose the matrix multiplication method for each pair of sub-matrices.
   $\boldsymbol{A}_{11}\boldsymbol{B}_{11}matmulfn = getMatmulFn(\boldsymbol{D}_{A11}, \boldsymbol{D}_{B11})$
   $\boldsymbol{A}_{12}\boldsymbol{B}_{21}matmulfn = getMatmulFn(\boldsymbol{D}_{A12}, \boldsymbol{D}_{B21})$
   …
   $\boldsymbol{A}_{22}\boldsymbol{B}_{22}matmulfn = getMatmulFn(\boldsymbol{D}_{A22}, \boldsymbol{D}_{B22})$
6. Use the chosen multiplication method to multiply each pair of sub-matrices.
   $\boldsymbol{C}_{A11B11} = \boldsymbol{A}_{11}\boldsymbol{B}_{11}matmulfn(\boldsymbol{A}_{11}, \boldsymbol{B}_{11})$
   $\boldsymbol{C}_{A12B21} = A_{12}B_{21}matmulfn(\boldsymbol{A}_{12}, \boldsymbol{B}_{21})$
   …
   $\boldsymbol{C}_{A22B22} = \boldsymbol{A}_{22}\boldsymbol{B}_{22}matmulfn(\boldsymbol{A}_{22}, \boldsymbol{B}_{22})$
7. Find sub-matrices of the result matrix $\boldsymbol{C}$ from matrices obtained in Step 6.
   $\boldsymbol{C}_{11} = \boldsymbol{C}_{A11B11} + \boldsymbol{C}_{A12B21}$
   $\boldsymbol{C}_{12} = \boldsymbol{C}_{A11B12} + \boldsymbol{C}_{A12B22}$
   $\boldsymbol{C}_{21} = \boldsymbol{C}_{A21B11} + \boldsymbol{C}_{A22B21}$
   $\boldsymbol{C}_{22} = \boldsymbol{C}_{A21B12} + \boldsymbol{C}_{A22B22}$
8. Combine sub-matrices $\boldsymbol{C}_{11}$, $\boldsymbol{C}_{12}$, $\boldsymbol{C}_{21}$ and $\boldsymbol{C}_{22}$ into $\boldsymbol{C}$.

This algorithm is implemented on Python 3.6.5, TensorFlow API r1.11 using TensorFlow's matrix multiplications, *tf.matmul* and *tf.sparse_matmul*. The method *tf.matmul* is used when both operands are dense matrices, while the method *tf.sparse_matmul* is used when at least one of the operands is a sparse matrix. For the method *tf.sparse_matmul*(*A, B, A_is_sparse, B_is_sparse*), the parameters ***A*** and ***B*** are the input matrices, and the parameters *A_is_sparse* and *B_is_sparse* are boolean values specifying if the input matrices are sparse.

Table 1 Matrix Multiplication Method Chosen for Sub-matrices

Density of *B*	**Density of *A***				
	0-20	20-30	30-40	40-50	50-100
0-10	*tf.sparse_matmul*(*A, B, A_is_sparse=True, B_is_sparse=True*)	*tf.sparse_matmul(A, B, A_is_sparse=False, B_is_sparse=True)*	*tf.sparse_matmul(A, B, A_is_sparse=False, B_is_sparse=True)*	*tf.sparse_matmul(A, B, A_is_sparse=False, B_is_sparse=True)*	*tf.sparse_matmul(A, B, A_is_sparse=False, B_is_sparse=True)*
10-20	*tf.sparse_matmul*(*A, B, A_is_sparse=True, B_is_sparse=True*)	*tf.sparse_matmul*(*A, B, A_is_sparse=True, B_is_sparse=True*)	*tf.sparse_matmul(A, B, A_is_sparse=False, B_is_sparse=True)*	*tf.sparse_matmul(A, B, A_is_sparse=False, B_is_sparse=True)*	*tf.sparse_matmul(A, B, A_is_sparse=False, B_is_sparse=True)*
20-30	*tf.sparse_matmul*(*A, B, A_is_sparse=True, B_is_sparse=True*)	*tf.sparse_matmul*(*A, B, A_is_sparse=True, B_is_sparse=True*)	*tf.sparse_matmul*(*A, B, A_is_sparse=True, B_is_sparse=True*)	*tf.sparse_matmul(A, B, A_is_sparse=False, B_is_sparse=True)*	*tf.sparse_matmul(A, B, A_is_sparse=False, B_is_sparse=True)*
30-40	*tf.sparse_matmul*(*A, B, A_is_sparse=True, B_is_sparse=True*)	*tf.sparse_matmul*(*A, B, A_is_sparse=True, B_is_sparse=True*)	*tf.sparse_matmul*(*A, B, A_is_sparse=True, B_is_sparse=True*)	*tf.sparse_matmul*(*A, B, A_is_sparse=True, B_is_sparse=True*)	*tf.sparse_matmul(A, B, A_is_sparse=False, B_is_sparse=True)*
40-100	*tf.sparse_matmul(A, B, A_is_sparse=True, B_is_sparse=False)*	*tf.sparse_matmul(A, B, A_is_sparse=True, B_is_sparse=False)*	*tf.sparse_matmul(A, B, A_is_sparse=True, B_is_sparse=False)*	*tf.sparse_matmul(A, B, A_is_sparse=True, B_is_sparse=False)*	*tf.matmul*(*A, B*)

The criteria for choosing between *tf.matmul* and *tf.sparse_matmul* is based on the multiplication time for varying density of input matrices on the platform used to evaluate the performance. For the multiplication of sub-matrices, if a matrix is sparse enough that *tf.sparse_matmul* is faster than *tf.matmul*, *tf.sparse_matmul* is chosen.

Otherwise, *tf.matmul* is used. Using different multiplication methods for each multiplication of sub-matrices can reduce the computation time if the performance difference of the *tf.matmul* and *tf.sparse_matmul* outweighs the overhead cost of partitioning and merging matrices. The criteria for choosing the multiplication method for sub-matrices shown in Tab. 1 is used in the function *getMatmulFn* in this paper.

4 PERFORMANCE EVALUATION

tbt_matmul is our implementation of the proposed matrix multiplication method on Python 3.6.5 with TensorFlow API r1.11. In this implementation, *tf.matmul* is used for dense matrices and *tf.sparse_matmul* is used for sparse matrices. *tbt_matmul* is compared with the multiplication using *tf.matmul* and *tf.sparse_matmul* for the whole matrices. The experiments were performed on a Laptop with Intel Core i7-4720HQ quad-core CPU, 16GB DDR3 RAM. The operating system used in this experiment was Linux Ubuntu 18.04.2 with Python 3.6.5 and TensorFlow API r1.11. In the experiments, the default configuration for thread pool was used, and the number of threads is set to maximum [17]. Thus, TensorFlow used eight threads in this experiment.

We performed the experiment on matrices with various overall density, ranging from 0.3 to 0.8. Furthermore, the performance of the proposed method is evaluated on matrices with different combination of higher-density and lower-density sub-matrices. These matrices are randomly generated so that two of their sub-matrices have higher density of non-zeros and two have lower density of non-zeros as shown in Fig 1. The non-zero distribution of sub-matrices is varied, as shown in Tab. 2. Twenty pairs of matrices are generated for each non-zero matrices, and the average computation time is used for the comparison.

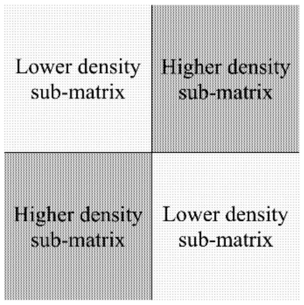

Figure 1 The density distribution pattern of input matrices

Table 2 Non-zero Distribution of Sub-matrices

Overall Density	(**Higher Density**, *Lower Density*)			
0.3	(**0.5**, 0.1)	(**0.4**, 0.2)		
0.4	(**0.7**, 0.1)	(**0.6**, 0.2)	(**0.5**, 0.3)	
0.5	(**0.9**, 0.1)	(**0.8**, 0.2)	(**0.7**, 0.3)	(**0.6**, 0.4)
0.6	(**1.0**, 0.2)	(**0.9**, 0.3)	(**0.8**, 0.4)	(**0.7**, 0.5)
0.7	(**1.0**, 0.4)	(**0.9**, 0.5)	(**0.8**, 0.6)	
0.8	(**1.0**, 0.6)	(**0.9**, 0.7)		

In order to evaluate the proposed method for different sizes of input, the experiments were performed on large matrices, i.e. 12,000×12,000 matrices, and small matrices, i.e. 3,000×3,000 matrices.

The graphs in Figs. 2-5 show the execution time for all three methods for matrices with different densities and sizes. The execution time for all three methods is grouped together for the comparison. The x-axis shows the density and non-zero distribution of the input matrices. The non-zero distribution of matrices is labelled in the graphs as (***d***, *s*), where ***d*** is the density of the higher-density sub-matrices and *s* is the density of the lower-density sub-matrices. The overall density is the average of ***d*** and *s*. For example, (**0.5**, 0.1) indicates that the density of the higher-density sub-matrices is 0.5, the density of lower-density sub-matrices is 0.1, and overall density is 0.3.

In section 4.1-4.3, the performance of the three methods are compared on large matrices of the size 12,000×12,000. Section 4.1 shows the performance on medium-density matrices, section 4.2 shows the performance on low-density matrices, and section 4.3 shows the performance on high-density matrices. Furthermore, section 4.4 compares the performance of the three methods on small, medium-density matrices of the size 3,000×3,000.

4.1 On Medium-density Matrices

Fig. 2 shows the execution time of the three multiplication methods on medium-density matrices of 50% and 60% density. The proposed method performs better than *tf_sparse_matmul* and *tf_matmul* for almost all non-zero distribution. When the density difference between the higher-density sub-matrices and the lower-density sub-matrices is small, i.e. for the non-zero distribution of (**0.6**, 0.4) and (**0.7**, 0.5), the proposed method is not the fastest. However, it is faster than *tf.sparse_matmul* , but slower than *tf_matmul* for the non-zero distribution of (**0.6**, 0.4).

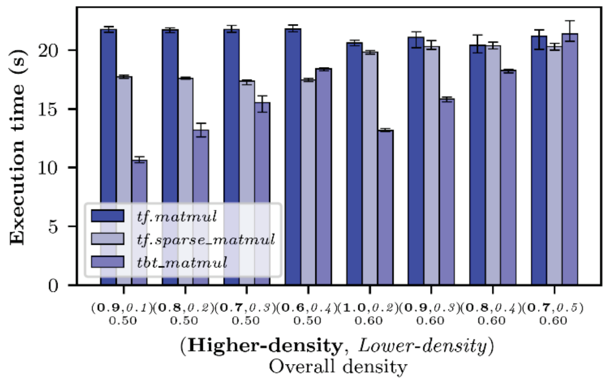

Figure 2 Execution time of *tf.matmul*, *tf.sparse_matmul* and *tbt_matmul* for 12000×12000 matrices with 50-60% density.

Furthermore, all three methods took almost the same amount of time for the non-zero distribution of (**0.7**, 0.3).

4.2 On Low-density Matrices

The execution time of the three multiplication methods on low-density matrices of 30% and 40% density is shown in Fig. 3. As suggested in [13], the method *tf.matmul* is not suitable for sparse matrices, and takes longest time of the three. The proposed method performs better than *tf.sparse_matmul* when the density of the higher-density and lower-density sub-matrices are significantly different, i.e. the non-zero distribution of (**0.5**, 0.1), (**0.7**, 0.1) and (**0.6**, 0.2). But when the density difference between the higher-density and the lower-density sub-matrices are small, our method is minutely slower than *tf.sparse_matmul*.

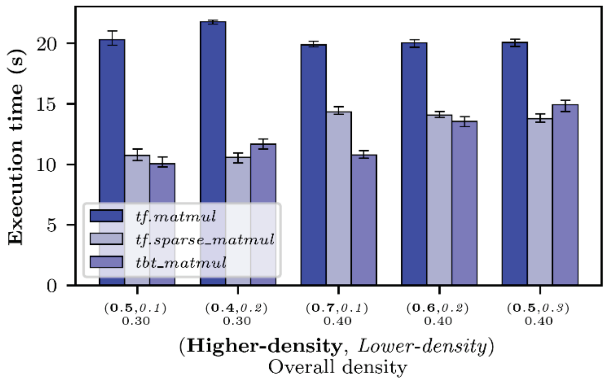

Figure 3 Execution time of *tf.matmul*, *tf.sparse_matmul* and *tbt_matmul* for 12000×12000 matrices with 30-40% density.

4.3 On High-density Matrices

Fig. 4 shows the execution time of the three multiplication methods on high-density matrices of 70% and 80% density. As expected, the method *tf.sparse_matmul* is not good for dense matrices and takes longest time of the three. For matrices with 70% density, the proposed method performs better than, or equally well as *tf.matmul* when the density of the higher-density and lower-density sub-matrices are significantly different, i.e. (**1.0**, 0.4) and (**0.9**, 0.5).

For matrices with 80% density, our proposed method is slower than *tf.matmul*, regardless of the non-zero distribution.

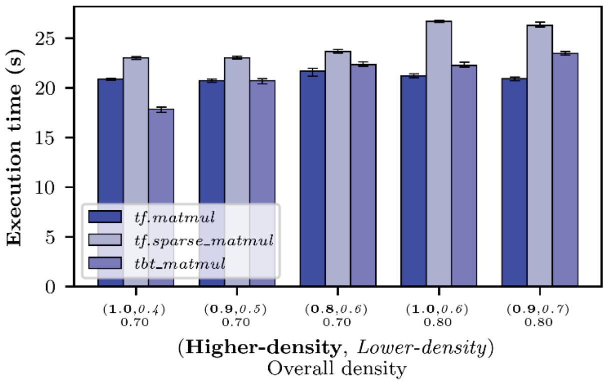

Figure 4 Execution time of *tf.matmul*, *tf.sparse_matmul* and *tbt_matmul* for 12000×12000 matrices with 70-80% density.

4.4 On Small Matrices with Medium-density

For input matrices of size 3,000×3,000, *tf.matmul* performs best on matrices with density at least 70% but was outperformed by tf.sparse_matmul on matrices with density under 70%. On the other hand, the method *tbt_matmul* is the slowest in all datasets most likely due to overhead of matrix partitioning and merging outweighing the performance gained. The execution time on small medium-density matrices is shown in Fig. 5

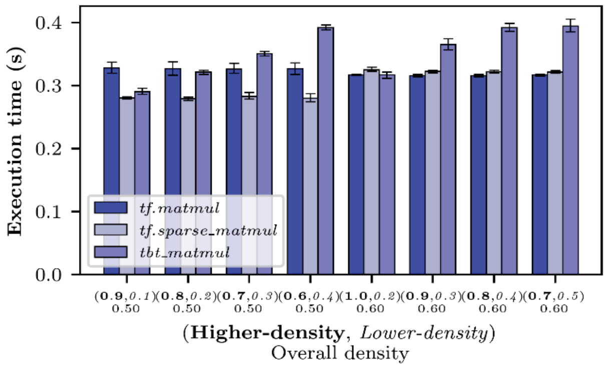

Figure 5 Execution time of *tf.matmul*, *tf.sparse_matmul* and *tbt_matmul* for 3000×3000 matrices with 50-60% density.

Based on the experiment, the density difference between the higher-density and lower-density sub-matrices is the important factor for the performance of our proposed method. When density difference between the higher-density and lower-density sub-matrices is large, the method *tbt_matmul* outperforms the other two methods for the input matrices with density of 70% or lower. However, for medium-density matrices, the proposed method is faster or comparable to the other two for all non-zero distribution. For sparse matrices, the proposed method is comparable to the *tf.sparse_matmul*. For dense matrices, the proposed method is comparable to *tf.matmul*. Furthermore, our proposed method is not suitable for small matrices.

5 DISCUSSION

The computation time of *tf.matmul* does not depend on the density of input matrices because it treats all matrices regardless of their density. It only performs better than the other two methods when the density of input matrices are very high.

The method *tf.sparse_matmul* takes into account the density of the input matrices and therefore performs better than tf.matmul at density lower than 70%. However, its performance is the same for different distribution of non-zeros in the matrices.

On the other hand, the performance of the proposed method *tbt_matmul* depends on both the matrix density and the density difference between the higher-density and lower-density sub-matrices of the input matrices. For medium density matrices, *tbt_matmul* is the fastest, especially when the density difference between the higher-density and lower-density sub-matrices is large. The performance gain can be attributed to the selection of the appropriate multiplication methods in step 5 of the algorithm. For example, for the input dataset (**0.9**, 0.1), 6 out of 8 multiplications of sub-matrices use *tf.sparse_matmul* and two multiplication of sub-matrices use *tf.matmul*. Using dense matrix multiplication for these two pairs of sub-matrices can reduce the computation time, compared to using sparse matrix multiplication for the whole matrix. On the other hand, when the difference between the higher-density and lower-density sub-matrices is small, such as (**1.0**, 0.6) or (**0.7**, 0.5), the multiplication method chosen for all pairs of sub-matrices pair is *tf.matmul* in many cases, because the density of each submatrix is relatively high. As a result, we do not get the performance gain because it is the same as using that multiplication method for the whole matrix, with additional cost of splitting and merging.

In this work, a matrix is divided into 4 sub-matrices. Further dividing a matrix into smaller sub-matrices, such as 16, does not lead to better performance, most likely due to the size of the dataflow graphs required for the computation.

6 CONCLUSION

In this work, we have proposed a multiplication method, *tbt_matmul*, for medium-density matrices on TensorFlow. The proposed method divides each input matrix into four sub-matrices and multiplies them using one of the TensorFlow built-in matrix multiplication methods, *tf.matmul* or *tf.sparse_matmul*, depending on the density of each sub-matrices. The method outperforms *tf.matmul* when the input matrices have overall density of 70% or lower and outperforms *tf.sparse_matmul* when the input matrices have overall density of 40% or higher. This method performs especially well when the densities of sub-matrices are significantly different.

The concept of the proposed method, multiplying different sub-matrices with the appropriate multiplication methods, can be used in any platform with efficient multiplication methods for dense matrices and for sparse matrices. It can benefit from the matrix multiplication methods, which are optimized on a specific platform.

For the future work, this concept can also be applied for matrix multiplication on GPUs. However, no sparse matrix multiplication is provided for GPUs on TensorFlow and it should be implemented in order to further improve matrix multiplication on GPU.

Remark

The article was orally presented at the 23rd International Computer Science and Engineering Conference (ICSEC2019).

7 REFERENCES

[1] Volker, S. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, 13, 354-356. https://doi.org/10.1007/BF02165411

[2] Don, C. & Shmuel, W. (1990). Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 251-280. https://doi.org/10.1016/S0747-7171(08)80013-2

[3] Shijin Z., Zidong D., Lei Z., Huiying L., Shaoli L., Ling L., Qi G., Tianshi, C., & Yunji, C. (2016). Cambricon-X: An accelerator for sparse neural networks. *49th Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO*), Taipei, 1-12.

[4] Bangtian, L., Chengyao, W, Anand, D. S., & Maryam, M. D. (2017). A unified optimization approach for sparse tensor operations on GPUs. *2017 IEEE International Conference on Cluster Computing* (*CLUSTER*), Honolulu, HI, 47-57.

[5] Abadi, M., Barham, P., Chen, J. et al. (2016). TensorFlow: a system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI '16*).

[6] Benoît, J. & Gaël, G. Eigen library for linear algebra. http://eigen.tuxfamily.org/

[7] NVIDIA Corporation. (2008). Santa Clara, California, Cublas library, https://developer.nvidia.com/cublas

[8] Yusuke, N., Satoshi, M., Ariful, A., & Aydın, B. (2018). High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. *ICPP '18*.

[9] Gudula, R. & Michael, S. (2010). Fast recursive matrix multiplication for multi-core architectures. *ICCS 2010*, 67-76. https://doi.org/10.1016/j.procs.2010.04.009

[10] Intel Corporation, Intel® Math Kernel Library, https://software.intel.com/en-us/mkl

[11] Zhang, X., Martin, K., Werner, S., Wang, Q., Zaheer, C., Chen, S., & Luo, W. *OpenBlas*. https://www.openblas.net

[12] Jiajia, L., Yuchen, M., Chenggang, Y., & Richard, V. (2016). Optimizing sparse tensor times matrix on multi-core and many-Core architectures. *6th Workshop on Irregular Applications: Architecture and Algorithms* (*IA3*), Salt Lake City, UT, 26-33.

[13] TensorFlow Python API Documentation, https://www.tensorflow.org/api_docs/python.

[14] Chien, S. W. D., Markidis, S., Olshevsky, V., Bulatov, Y., Laure, E., & Vetter, J. S. (2019). TensorFlow doing HPC: an evaluation of TensorFlow performance in HPC applications. *The Ninth International Workshop on Accelerators and Hybrid Exascale Systems* (*AsHES'19*), Rio de Janeiro, Copacabana, Brazil. https://doi.org/10.1109/IPDPSW.2019.00092

[15] Minwoo, K., Yong, J. J., & Won, W. R. (2011). Parallel transpose of matrix multiplication based on the tiling algorithms. *IEEE 54th International Midwest Symposium on Circuits and Systems* (*MWSCAS*), Seoul, 1-3.

[16] Haitao, W., Guang, R. G., & Elkin, G. (2015). Energy efficient multi-level tiling for dense matrix multiplication on many-core architecture. *Sixth International Green and Sustainable Computing Conference* (*IGSC)*, Las Vegas, NV, 1-6. https://doi.org/10.1109/IGCC.2015.7393735

[17] TensorFlow Performance Optimization Guide, https://www.tensorflow.org/guide/performance/overview

Authors' contacts:

Siraphob THEERACHEEP
Department of Mathematics and Computer Science,
Faculty of Science, Chulalongkorn University,
Bangkok 10330, Thailand
siraphob47@gmail.com

Jaruloj CHONGSTITVATANA, Assistant Professor
Department of Mathematics and Computer Science,
Faculty of Science, Chulalongkorn University,
Bangkok 10330, Thailand
jaruloj@gmail.com