Indonesian Journal of Electrical Engineering and Computer Science
Vol. 28, No. 2, November 2022, pp. 1012~1019
ISSN: 2502-4752, DOI: 10.11591/ijeecs.v28.i2.pp1012-1019

Matrix-matrix multiplication on graphics processing unit platform using tiling technique

Rahman Ghasempour Balagafshe[1], Alireza Akoushideh[2], Asadollah Shahbahrami[1]
[1]Department of Computer Engineering, University of Guilan, Rasht, Iran
[2]Department of Electrical Engineering, College of Shahid-Chamran, Technical and Vocational University (TVU), Rasht, Iran

Article Info

Article history:

Received Dec 21, 2021
Revised Jul 25, 2022
Accepted Aug 26, 2022

Keywords:

Dense
Matrix-matrix multiplication
CUDA
Shared memory
Tiling

ABSTRACT

Today's hardware platforms have parallel processing capabilities and many parallel programming models have been developed. It is necessary to research an efficient implementation of compute-intensive applications using available platforms. Dense matrix-matrix multiplication is an important kernel that is used in many applications, while it is computationally intensive, especially for large matrix sizes. To improve the performance of this kernel, we implement it on the graphics processing unit (GPU) platform using the tiling technique with different tile sizes. Our experimental results show the tiling approach improves speed by 56.89% (2.32× faster) against straightforward (STF). And tile size of 32 has the highest speed compared to other tile sizes of 8 and 16.

Corresponding Author:

Alireza Akoushideh
Department of Electrical Engineering, College of Shahid-Chamran, Technical and Vocational University Guilan Branch, Rasht, Iran
Email: akushide@tvu.ac.ir

1. INTRODUCTION

Processor vendors have been developing single instruction multiple data (SIMD) extinctions to improve the performance of multimedia applications. SIMD was developed as a data level parallelism (DLP) solution in parallelism. For example, Intel has introduced MMX, SSE, and AVX/AVX2 since 1996. In the beginning, SIMD registers were 64-bit and each new extinction doubled it to 128, 256-bit. So, in SIMD technology AVX and AVX2 could provide at least 256-bit registers. On the other hand, inline-assembly, intrinsic functions, and auto-vectorization also known as vector class has been applied for SIMD programming [1], [2].

Dense matrix-matrix multiplication (DMMM) is an important kernel that is used in many applications such as weka data mining and network analysis [3]-[5]. This kernel is computationally intensive, for example for a matrix size of 8000 with floating-point numbers, it approximately takes 789 seconds as can be seen in Figure 1. Many researchers have worked on the high-performance implementation of MMM [4], [6], [7]. Some cases have been performed on CPU platforms [8], [9], while other implementations have been performed on graphics processing unit (GPU) platforms. There are some software optimization techniques in both CPU and GPU implementation such as instruction-level parallelism (ILP), data-level parallelism (DLP), and thread level parallelism (TLP) [8]. In addition, the tiling technique has already been used, while due to hardware limitations tiling sizes up to 16 have been applied [10]. To evaluate the effect of different tile sizes on the performance, we have applied tile sizes of 8, 16, and 32 on the GPU Kepler architecture. Our experimental results show that the tile size of 32 has the highest speedup compare to the other tile sizes. The paper is organized as follows. We discuss related works in section 2. In section 3 the DMMM is represented using different techniques such as

tiling and shared memory. Section 4 illustrates our methodology, the selected benchmarks, and the test-bed machine in detail. Finally, conclusions are presented in section 5.

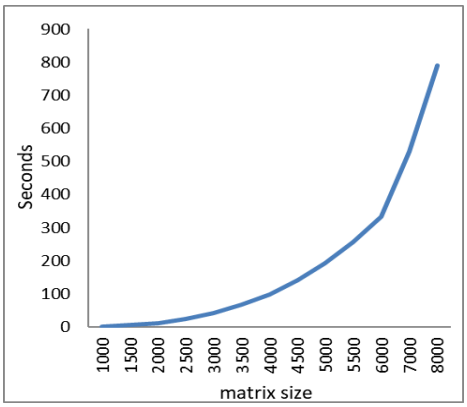

Figure 1. Execution time of dense matrix-matrix multiplication KERNEL on CPU platform, Intel Core i3, 2.0 GHz

2. RELATED WORK

Matrix-matrix multiplication or in brief MMM is a binary operation that multiplies an entire row of matrix A into an entire column of matrix B to produce each element of matrix C. There are different ways to implement this kernel such as basic mapping, matrix transposition, loop interchange and blocking, see Figure 2. In a basic implementation, shown in Figure 2(a), the cost of fetching matrix B column has a bad great impact on overall performance. While for the transposed method that we can see in Figure 2(b), the only extra cost is transposing Matrix B. However, in loop interchange and blocking the data reuses gain much better performance compared to basic and transposed methods which are shown in Figure 2(c) and Figure 2(d) respectively. MMM speedup has been the major goal of many studies [8], [11]-[15] and is still ongoing today. BLAS [13], [16] is a basic linear algebra subprogram (BLAS) that provides a standard blocking method for matrix multiplication. It has been widely used in many libraries, like ATLAS, and NVIDIA cuBLAS [8].

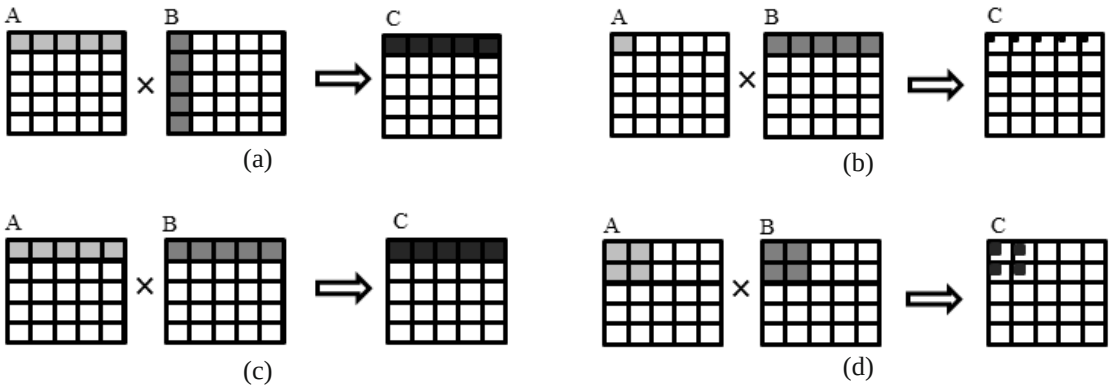

Figure 2. Mapping matrix-matrix multiplication for (a) basic, (b) loop interchange, (c) transposed, and (d) blocking techniques

Zachariadis *et al.* [17] proposed a GPU-based framework for sparse general MMM (spGEMM) computation. Their methodology groups elements into tiles and uses the fast MM of tensor core units (TCUs) to multiply the tiles. Srivastava *et al.* [18] introduce C^2SR, as a new architecture that allows different parallel processing engines (PEs) to access the data in a vectorized and streaming manner leading to high utilization of the available memory bandwidth. Their SpGEMM accelerator (MatRaptor), efficiently implements the row-wise product approach and fully exploits the C^2SR format to achieve better performance. Park *et al.* [19] proposed an approach to improve and adjust the parallel double-precision general matrix-matrix multiplication routine for new intel cores such as Knights landing and xeon scalable processors. The authors

proposed that performance improvements are achieved in the case of smaller matrix multiplications on the SKL clusters. Masliah *et al.* [20] proposed a method to compute many small general dense matrix-matrix multiplication and its performance portability for a wide range of computer architectures. The dense matrix-matrix multiplication (DMMM) has been implemented on different platforms using different techniques, some of these are depicted in Table 1. The ILP and TLP and a combination of these techniques have been used [10], [21]. The SIMD technique, OpenCL, and sub-blocks have been applied for DMMM implementation [8], [9], [22]-[24].

Table 1. Different platforms and techniques have been used for dense matrix-matrix multiplication kernels, the T refers to Tiling, and T&U refers to tiling and unroll

		[8]	[21]	[22]	[10]	[4]	[9]
GPU Platform		GeForce GTX 580	GeForce 8800 GTX & Tesla C870	GeForce GTX 280	GeForce 8800 GTX	GeForce 8800GTX & 8800GTS & 8600GTS & Quadro FX5600	Tesla C1060
CUDA Cores		512	128 & 128	240	128	128 & 96 & 32 &128	240
Single precision Theatrical peak performance (GFlops)		1581.06	345.6 & 345.6	622.08	345.6	345.6 & 230.04 & 92.8 & 345.6	624
Memory bandwidth (GB/s)		192.4	86.4 & 76.8	141.7	86.4	86.4 & 63.36 & 32 & 76.8	102.4
Method	Tiling	Yes	Yes	---	Yes	Yes	yes
	Tile and unroll	---	Yes	---	Yes		Yes
	Rectangular Blocking	---	---	yes	---	Yes	Yes
	Square Blocking	Yes	Yes	---	Yes	---	Yes
Maximum matrix size		8192	4096	8192	4096	11264	16384
Maximum GFlops		not mentioned	43	393	46(*T*) - 91(*T&U*)	206	383

3. PROPOSED APPROACH

Considering we have two square matrices size of N×N (called matrix A and matrix B) the multiplication result is going to be stored in the same size matrix *C*. Also, each element of matrix *C* would be an inner product of an entire row *A* and an entire column *B*. As shown in Figure 3, in the straightforward matrix-matrix multiplication form, the code consists of three nested loops iterating over the 3 dimensions *i*, *j*, and *k*.

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        C[i] [j] =0;
            for (k=0; k<N; k++)
            C[i] [j] =C[i] [j] +A[i] [k] × B[j]
```

Figure 3. DMMM basic code

3.1. Step 1

Straight-forward mapping DMMM on CUDA needs each thread to produce one element of matrix *C* by producing an entire row of matrix *A* into an entire column of matrix B. We will transmit the number of grid and block needed to process, and also manually change block dimensions size (number of threads in a block) and grid dimensions size (number of blocks in a grid) [5] to assign one element to each matrix *C* element. For example, for a matrix size of 2048×2048 if we chose a block size of 64 threads (setting block dimension to 8×8, we will need 65536 blocks for a total coverage of matrix *C* elements.

3.2. Step 2

Tiling is a method for improving DMMM, also it has been used by many researchers as mentioned in Table 1, in tiling we divide big matrices into sub-matrices, considering their sizes are $m \times m$ and multiply an entire row of A sub-matrices as a with an entire column of B sub-matrices as b to produce an entire sub-matrix of C as mentioned in (1), see Figure 4 for more details.

$$c(0,0) = a(0,0) \times b(0,0) + a(0,1) \times b(1,0) + \ldots\ldots\ldots + a\left(0, \frac{N}{m} - 1\right) \times b\left(\frac{N}{m} - 1,\ 0\right) + a\left(0, \frac{N}{m}\right) \times b\left(\frac{N}{m}, 0\right) \quad (1)$$

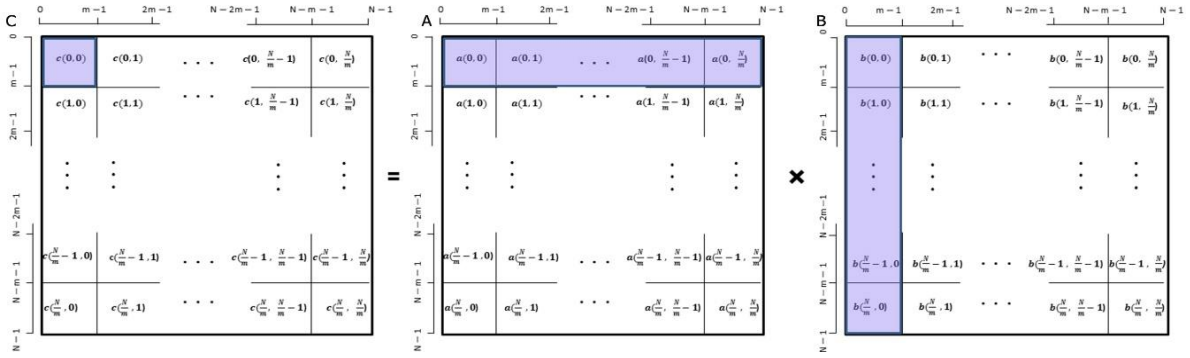

Figure 4. DMMM using tile methodology

Each C sub-matrix will be the sum of the product of all a and b sub-matrices, the number of middle sum (sum of sub-matrices) is directly connected with m (tile dimension) as it gets bigger middle sum decreases by a factor of $((N/m) - 1) \times (N/m)$. Tiling in DMMM creates 2 more inner loops which compute a product of a sub-matrix of A and a sub-matrix of B and replace the old result of matrix C with a new one after updating. Avoiding divergence and avoiding Bank Conflicts by coalescing memory access is another positive aspect of tiling [25]. Conceptually in straight forward DMMM threads can execute concurrently or independently, and in no particular order, to avoid this mess we will use *__syncthreads()* to make sure any thread would complete its job before starting another one. In the tiled kernel, we will use shared memory to increase reusability and reduce the traffic to global memory. Every thread in a tile will load m elements of each row of matrix A and m elements of each column of matrix B into the shared memory. These elements are used in the calculation of the dot product, each element will be used m times (for a tile size of $m \times m$) therefore the number of global memory access will be divided per m see Figure 5.

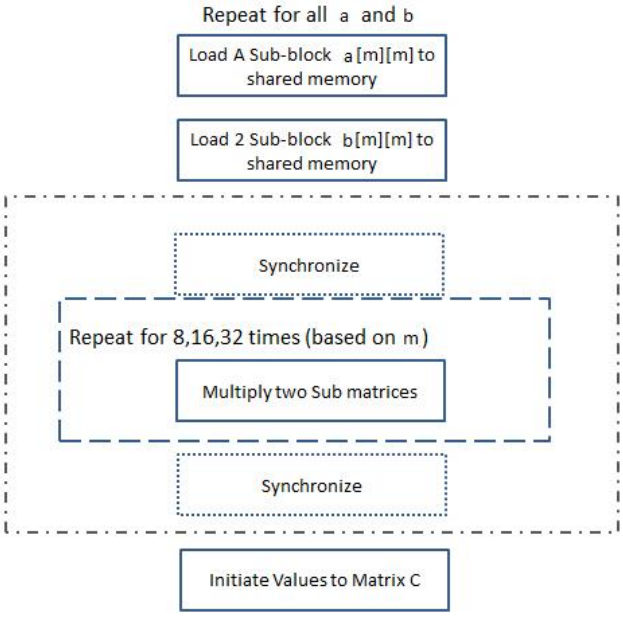

Figure 5. Tiling and shared memory kernel

Moreover, in CUDA each multiprocessor runs in a SIMD manner. In each clock cycle, all stream processors of a multiprocessor execute the same instruction while operating on different data. Besides this, threads in a block are grouped by Warps and the executions of warps are implemented by SIMD hardware [5].

4. RESULTS AND DISCUSSION

To compare the performance of different tile size effects, we will fill our two matrices with random floating-point numbers from files we have created before. This is done on the CPU side of our application. After that, two matrices will be passed to devise memory to run the kernel function and measure it in detail. We will use speed up and floating-point operation per seconds (FLOPS) as well as Achieved IOPS, Pipeline utilization, Branch statistics and memory statistics.

4.1. Selected benchmark

We selected Nvidia Nsight visual studio edition [26]. For CUDA version 3.5. It provides a variety type of benchmarking for CUDA kernels.

4.2. Experimental test-bed

We accelerate the computation of matrix multiplication with one platform, consisting of two parts host (CPU) and device (GPU). You can see the detail in Table 2. As mentioned before, we are going to discuss tile size impacts on DMMM and also the percentage of load/store, number of integer operation, and global memory request (GMR) in kernel launches for different tile sizes. In Figure 6 kernel speedup over sequential DMMM size of 4096×4096 has been shown, for tile size of 16×16 speed up increment compared to tile size of 8×8 is admissible but for tile size of 32×32 speed up is not what is accepted for it. Due to Nvidia Nsight, there are some differences in the number of integer additions, global memory request, achieved occupancy, and some other factors.

Table 2. GPU specifications

		GeForce 920M (Kepler (cc 3.5)
Load/Store unit		32
Arithmetic	FP32 unit	192
	SFU unit	32
GPU Memory Size		1024MB
Bus Width		64 bit
Bandwidth		14.4 Gb/s
GPU Clock		954 MHz
Memory Clock		900MHz
Streaming Multiprocessors (SMs)		2
Max Thread per SM		2048
Max active Block per SM		16
Max active Warp per SM		64

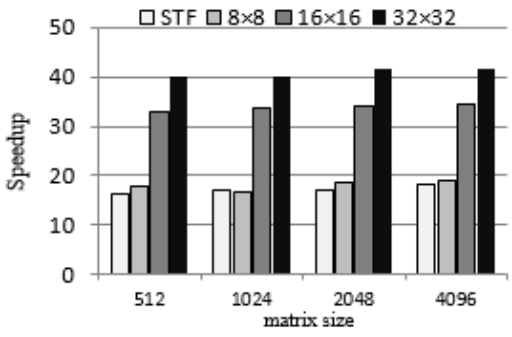

Figure 6. Speedup of the straightforward (STF) and tiling over serial

As a result, in Table 3 you can see the influence factors on the final result of DMMM for matrix size of 4096×4096 in different tile sizes. In a straightforward (STF) kernel we have 3 integer multiplications, 3 additions, one floating-point addition, and one multiplication, they are the arithmetic operations needed for

each multiplication of one element of matrix *A* and one element of matrix *B*, plus storing the result in matrix *C*. These operations need one load and one store for each *C* element separately. Respectively increasing tile number will decrease total Integer ADD operation, the "sum of all executed integer additions (IADD)" needed in kernel launch see Table 3, the total number of ADD operations for tile size of 8×8 in our code is 5.3 times, in 16×16, 10.6 times and in 32×32 is 21.2 times lower than STF. In conclusion, by using a larger tile the ADD instructions would decrease dramatically, while MADD which stands for the sum of all executed integer multiply-add (IMAD) instructions and SHIFTs meaning the sum of all executed shift instructions, covering shift-right (SHR), shift-left (SHL), and funnel-shift (SHF) operations are fixed. Despite this our 32×32 tile size didn't achieve the expected GFLOPS, The reason for this low efficiency is the arithmetic and load/store increments than other tile sizes [26].

Table 3. Pipeline utilization for DMMM size of "4096×4096"

	STF	8×8	16×16	32×32
Load/Store	100%	100%	69.36%	73.57%
Arithmetic	87.72%	31.85%	10.39%	11.64%
Control flow	22.88%	23.13%	4.36%	2.74%
Int. operation ADD	137,455,730,688	25,803,358,208	12,918,456,320	6,476,005,376
Shared memory request	NO	3,221,225,000	2,952,792,000	2,818,572,000
GMR	4,295,492,000	537,395,200	268,959,700	137,742,000
Achieved Active warps per SM	1.72	2.32	2.48	0.89
Theatrical Active warps per SM	64	32	64	64
Achieved Occupancy	2.69%	3.62%	3.88%	1.39%
Reg per thread	14	25	25	25
Static shared memory per block	NO	512 bytes	2048 byte	8192 bytes
Kernel time	5.9524sec	5.6916sec	3.1018sec	2.5658sec

Figure 7 depicts floating-point operation per second for DMMM with the size of "4096×4096". Maybe the greatest effect of tiling using shared memory methodology is its reduction in global memory access, as mentioned in the sections above a tile size of $m \times m$ can respectively decrease global memory access by a factor of m. But due to tile size limitations, the best output will have 32 times less global memory access request (GMR). According to the results obtained in Table 3, tile sizes of 8×8, 16×16, and 32×32 in sequence reduce GMR by 7.9, 15.9, and 31.1 times compared to the STF version. GMR is a bottleneck [5] for CUDA applications and reducing it directly improves total performance. Overall loop tiling is performed to maximize reusability and minimize the number of loads and stores. In 32×32 increases slightly which causes the decrease in performance.

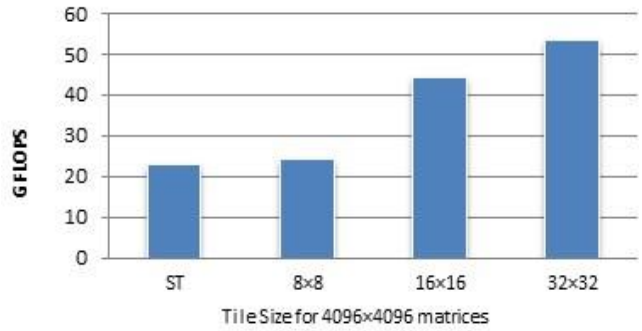

Figure 7. Floating-point operation per second for DMMM size of "4096×4096"

5. CONCLUSION

The dense matrix-matrix multiplication is the main kernel in many applications and there are different kinds of implementations on the CPU and GPU platforms. We have used the tiling technique to improve the performance of the kernel on the GPU platform and evaluate the effect of different tile sizes on performance. Our experimental results show the tiling technique improves the performance and the tile size of 32 has the highest speedup compared to other tile sizes.

REFERENCES

[1] H. Jeong, S. Kim, W. Lee, and S.-H. Myung, "Performance of SSE and AVX Instruction Sets," *arXiv*, 2012.

[2] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe, "VeGen: a vectorizer generator for SIMD and beyond," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 902-914.

[3] D. Heryadi and S. Hampton, "Characterizing performance improvement of GPUs," presented at the Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning), Chicago, IL, USA, 2019. [Online]. Available: doi: 10.1145/3332186.3332237.

[4] Y.-Y. Jo, S.-W. Kim, and D.-H. Bae, "GPU-based matrix multiplication methods for social networks analysis," presented at the Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems, Towson, Maryland, 2014. [Online]. Available: doi: 10.1145/2663761.2664192.

[5] D. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors, A Hands-on Approach*, 3rd ed. 2016.

[6] Y. Ouerhani, M. Jridi, and A. AlFalou, "Fast face recognition approach using a graphical processing unit "GPU"," in *IEEE International Conference on Imaging Systems and Techniques*, 1-2 July 2010 2010, pp. 80-84, doi: 10.1109/IST.2010.5548545.

[7] L. Bustio-Martínez, R. Cumplido, M. Letras, R. Hernández-León, C. Feregrino-Uribe, and J. Hernández-Palancar, "FPGA/GPU-based acceleration for frequent itemsets mining: a comprehensive review," *ACM Comput. Surv.*, vol. 54, no. 9, p. Article 179, 2021, doi: 10.1145/3472289.

[8] V. Kelefouras, A. Kritikakou, I. Mporas, and V. Kolonias, "A high-performance matrix–matrix multiplication methodology for CPU and GPU architectures," *The Journal of Supercomputing*, vol. 72, no. 3, pp. 804-844, 2016, doi: 10.1007/s11227-015-1613-7.

[9] J. Li, S. Ranka, and S. Sahni, in *GPU matrix multiplication*: Chapman-Hall/CRC Press, 2013, ch. Multicore Computing: Algorithms, Architectures, and Applications.

[10] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," presented at the Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, Salt Lake City, UT, USA, 2008. [Online]. Available: doi: 10.1145/1345206.1345220.

[11] S. A. Hassan, A. M. Hemeida, and M. M. M. Mahmoud, "Performance evaluation of matrix-matrix multiplications using intel's advanced vector extensions (AVX)," *Microprocess. Microsystems*, vol. 47, pp. 369-374, 2016.

[12] V. Kelefouras, A. Kritikakou, and C. Goutis, "A matrix–matrix multiplication methodology for single/multi-core architectures using SIMD," *The Journal of Supercomputing*, vol. 68, no. 3, pp. 1418-1440, 2014, doi: 10.1007/s11227-014-1098-9.

[13] T. Gautier and J. V. F. Lima, "Evaluation of two topology-aware heuristics on level-3 BLAS library for multi-GPU platforms," in *PAW-ATM 2021 - 4th Annual Parallel Applications Workshop, Alternatives To MPI+X*, Saint Louis, United States, 2021-11-19 2021, https://hal.inria.fr/hal-03363275/document, https://hal.inria.fr/hal-03363275/file/xkblas_pawatm106s1-file1.pdf, pp. 1-11. [Online]. Available: https://hal.inria.fr/hal-03363275. [Online]. Available: https://hal.inria.fr/hal-03363275

[14] M. A. Aroon, A. F. Ismail, T. Matsuura, and M. M. Montazer-Rahmati, "Performance studies of mixed matrix membranes for gas separation: A review," *Separation and Purification Technology*, vol. 75, no. 3, pp. 229-242, 2010, doi: 10.1016/j.seppur.2010.08.023.

[15] M. Salim, A. O. Akkirman, M. Hidayetoglu, and L. Gurel, "Comparative benchmarking: matrix multiplication on a multicore coprocessor and a GPU," in *2015 Computational Electromagnetics International Workshop (CEM)*, 2015, pp. 1-2, doi: 10.1109/CEM.2015.7237429.

[16] K. Goto and R. A. V. D. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, p. Article 12, 2008, doi: 10.1145/1356052.1356053.

[17] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, "Accelerating sparse matrix–matrix multiplication with GPU tensor cores," *Computers & Electrical Engineering*, vol. 88, p. 106848, 2020, doi: 10.1016/j.compeleceng.2020.106848.

[18] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: a sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 766-780, 2020, doi: 10.1109/MICRO50266.2020.00068.

[19] Y. Park, R. Kim, T. M. T. Nguyen, and J. Choi, "Improving blocked matrix-matrix multiplication routine by utilizing AVX-512 instructions on intel knights landing and xeon scalable processors," *Cluster Computing*, 2021, doi: 10.1007/s10586-021-03274-8.

[20] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra, "Algorithms and optimization techniques for high-performance matrix-matrix multiplications of very small matrices," *Parallel Computing*, vol. 81, pp. 1-21, 2019, doi: 10.1016/j.parco.2018.10.003.

[21] J. M. Cecilia, J. M. García, and M. Ujaldón, "The GPU on the matrix-matrix multiply: performance study and contributions," in *PARCO*, 2009.

[22] X. Cui, Y. Chen, and H. Mei, "Improving performance of matrix multiplication and FFT on GPU," in *2009 15th International Conference on Parallel and Distributed Systems*, 2009, pp. 42-48, doi: 10.1109/ICPADS.2009.8.

[23] H. H. Holm, A. R. Brodtkorb, and M. L. Sætra, "Performance and energy efficiency of CUDA and OpenCL for GPU computing using python," *Advances in Parallel Computing*, vol. 36, p. 12, 2020, doi: 10.3233/APC200089.

[24] S. Singh and M. D. Graef, "GPU-accelerated matrix exponentiation for 5-D STEM-DCI simulations," *Microscopy and Microanalysis*, vol. 24, no. S1, pp. 222-223, 2018, doi: 10.1017/S1431927618001605.

[25] P. Anastasiadis, N. Papadopoulou, G. Goumas, and N. Koziris, "CoCoPeLia: communication-computation overlap prediction for efficient linear algebra on GPUs," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 28-30 March 2021 2021, pp. 36-47, doi: 10.1109/ISPASS51385.2021.00015.

[26] Nsight, N. V. I. D. I. A., and Visual Studio Edition "Nsight visual studio edition user guide," NVIDIA. NVIDIA Nsight Visual Studio Edition 2019.3 User Guide (accessed Aug. 25, 2019)

BIOGRAPHIES OF AUTHORS

Rahman Ghasempour Balagafshe received a B.Sc. degree in computer engineering (software engineering) from Payame Noor University, Rasht Branch, in 2014. He got his M.Sc. degree in Computer Engineering, Software Engineering, from Guilan University. His Master's thesis was "evaluating deep learning on TSR" including training a model on a self-collected image database of local Traffic signs. His current job title is Technical Support Engineer. His research interests include parallel programming HPC, web technology, machine vision and deep learning. He can be contacted at email: rahman.ghasempour@gmail.com.

Alireza Akoushideh received the B.Sc. and M.Sc. degree in Electrical engineering from the University of Guilan and Amirkabir University of Technology (Tehran Polytechnic) in 1997 and 2000, respectively. From 2001 until now, he is a faculty member of Technical and Vocational University, Shahid-Chamran community college, Rasht, Iran. He got his PhD degree from Shahid-Beheshti University, Tehran, Iran in 2016. As a visiting researcher, he worked with the SCS group at Twente University, the Netherlands from January to September 2015. He has taught courses in FPGA, microprocessor and microcontrollers, computer architecture, and digital circuits. His research interests include machine vision, texture analysis, Intelligence Transformation System (ITS), and FPGA implementation. He can be contacted at email: akushide@tvu.ac.ir.

Prof. Asadollah Shahbahrami received the B.Sc. and M.Sc. degrees in computer engineering (hardware and machine intelligence) from Iran University of science and technology and Shiraz University in 1993 and 1996, respectively. He got his PhD degree from the Delft University of Technology, The Netherlands in 2008. He has been working at the University of Guilan since August 1996. He is a professor position in the Department of Computer Engineering at the University of Guilan. Dr Shahbahrami research interests include advanced computer architecture, image and video processing, multimedia instructions set design, reconfigurable computing, parallel processing, and SIMD programming. He can be contacted at email: shahbahrami@guilan.ac.ir.