Hindawi
Mobile Information Systems
Volume 2022, Article ID 9650652, 9 pages
https://doi.org/10.1155/2022/9650652

*Research Article*

# Improved Matrix Multiplication by Changing Loop Order

**Abdullah Almurayh**

*Imam Abdulrahman Bin Faisal University, P. O. Box 1982, Dammam, Saudi Arabia*

Correspondence should be addressed to Abdullah Almurayh; asalmurayh@iau.edu.sa

Received 26 August 2022; Revised 7 September 2022; Accepted 21 September 2022; Published 11 October 2022

Academic Editor: Santosh Tirunagari



Matrix multiplication has been implemented in various programming languages, and improved performance has been reported in many articles under various settings. Matrix multiplication is of paramount interest to machine learning, a lightweight matrix-based key management protocol for IoT networks, animation, and so on. There has always been a need and an interest for improved performance in terms of algorithm implementation. In this work, the authors compared the run times of matrix multiplication in popular languages such as C++, Java, and Python. This analysis showed that Python's implementation was poor while Java was relatively slower compared to the C++ implementation. All the aforementioned languages use a row-major scheme, and hence, there are many cache misses encountered when implemented through simple looping. In contrast, the authors show that by changing the loop order, more performance gains are possible. Moreover, we evaluated the performance of matrix multiplication by comparing the execution time under various loop settings. The authors observed tremendous performance gains due to better spatial locality. In addition, the authors also implemented a parallel version of the same algorithm using OpenMP with eight logical cores and achieved a speed-up of seven times compared to the serial implementation.

## 1. Introduction

With the emergence of smart devices in many application areas, such as healthcare, location-based services, and self-driving vehicles heavily depends on the efficient processing of data produced by these devices. Processing such data are very challenging, especially when data is produced in zettabytes. On the other hand, the performance of an application is no more linearly linked to the processor clock frequency as it was a norm until 2004, where doubling the system speed would roughly result in a 50% performance gain. However, in 2004, the chip manufacturers encountered a "Power Wall" and faced fundamental constraints in power delivery and heat dissipation. "Power Wall" refers to the difficulty of scaling the performance of computing chips and systems due to fundamental constraints imposed by physics. This limitation on increasing processor frequency introduced the era of multicore computing, where multiple cores run an application in parallel for better performance. Multicore systems and even low-end small devices such as tablets and smartphones, which have two or more cores, currently dominate the computing domain. On the other hand, writing algorithms to fully exploit these multicore systems remains a challenging task.

With multicore systems, personal computers today offer the computation power of supercomputers. Such speed provides many prospects for performing computationally extensive operations. To exploit this facility, several serial applications present the opportunity to write parallel versions, especially in applications where it is of principal interest to exploit the potential parallelism in the application. This trend of parallel processing theoretically shows linear performance gains with an increased number of processors at an abstract level and excellent performance when run as a parallel system. However, not all code can run in parallel and, hence, serial parts seriously limit the system performance. Unlike serial counterparts where performance is measured through complexity analysis, the performance of parallel systems is evaluated through Amdahl's or Gustafson's laws [1, 2].

Traditionally, until 2002, the majority of computer systems had one processor, and accordingly, sequential

algorithms were developed. Today, the latest computers, which consist of multiple processing elements (either multiple CPU cores or GPU), are more powerful due to multiple processing units. Understandable serial codes are slow as they need to process the instructions one by one, in contrast to a parallel execution where multiple instructions are processed simultaneously. The performance of many serial programs can be improved by exploiting the parallel architectures, which can be in terms of loop reorder, pipelining, and speculation, and so on. It is worth noting that not every program can be converted in parallel, and there are cases where the parallel version exhibits poor performance compared to the serial version. With the latest computing architecture and parallel languages, tremendous improvements have been reported for machine learning, AI, a lightweight matrix-based key management protocol for IoT networks, graphics, computational photography, and computer vision by exploiting parallelization [3–7].

Benefits such as power efficiency, resource pooling, cost reduction, system availability, and improved computational power can be obtained with cloud computing infrastructure. All such benefits attract computer scientists, systems engineers, the research community, and high-performance computing (HPC) customers to the cloud domain. On the other hand, HPC programs often use a large number of processors to lower the run times of tasks. An issue with such processors is synchronization, in addition to communication overheads. It has been reported in [3] that shifting an HPC application to the cloud environment can negatively impact the aforementioned difficulties and even introduce additional issues of virtualization, multitenancy, network latency, and so forth.

In this paper, we limit our analysis to a multicore system by observing the program behavior and changing loop orderings [4–7]. We apply a parallel construct for matrix multiplication to gain better performance. In this work, we study the following:

The impact of loop reordering on performance

Parallel implementation under OpenMP framework to illustrate the speed-up obtained through parallelism

The remaining part of this paper is divided into four sections as. In the background Section, we discuss the groundwork, followed by an explanation of the row and column major order of data in the memory in the impact of row-major programming constructs Section. We discuss the behavior of matrix multiplication for a square matrix by changing the loop ordering. In the Discussion Section, we provide the parallel version of the code and discuss the experimental results. The paper then offers a conclusion as the last section.

## 2. Background

Until the beginning of the 21st century, advances in technology would simply be considered an increase in the clock speed. Naturally, the software would effectively "speed-up" automatically over time because of running faster processors. The clock speed of microprocessors increased exponentially through the 1990s and beyond, but after 2004, it reached a limit due to physics, and the clock speed is now limited by power consumption/heat dissipation. With little improvement in clock speed, such performance gain convenience is no longer an option for software engineers.

As a solution to lower power consumption, the dynamic voltage scaling concept was introduced in CMOS technologies. The literature shows that the relationship between frequency and voltage in modern processors [8] can be written as: $E = P * T$, where "E" denotes the power consumption, "P" represents the average power, and "T" is the time taken for this average power.

Today, advances in technology mean increased parallelism and not enhanced clock speed. Thus, exploiting such parallelism is one of the outstanding challenges of modern computer science.

The parallel programming and parallelization of the tasks are done for the main purpose of allowing tasks to be executed at the same time by utilizing multiple computer resources and multiple cores on the same CPU. This process is very critical, especially for large-scale projects where speed is needed. Parallel programming is making its way into various domains, ranging from drug discovery to data analytics to the animation industry. All these applications are computation intensive and traditional sequential code becomes inefficient. However, just increasing the number of processors does not always guarantee performance gains and depends on the nature of the problem to be parallelized. Parallel implementation is prone to overheads such as: task start-up, time, synchronization, SATA communications, and software overhead imposed by parallel languages, libraries, operating system, and so on. When a parallel code is developed, these factors need to be considered carefully.

In the parallel programming literature, a massive parallel system means that the hardware of a given parallel system is comprised of many processing elements, and currently, the largest parallel computers include processing components in the range of hundreds of thousands to millions. Similarly, embarrassing parallel applications refer to a set of applications where independent tasks can run simultaneously and there exists very little to no need for coordination between the tasks. Another term that is used in the parallel programming domain is "scalability," which points to a parallel system's ability to demonstrate an adequate increase in parallel speed-up. Such a situation is understandable with the addition of more resources. Factors that contribute to scalability include algorithms, overheads, hardware, the characteristics of a program, and so forth. In parallel programming, efficiency is defined as the amount of work needed to be done, while "performance" points to how fast an algorithm can finish a particular work. A faster implementation is not necessarily an efficient one, where efficiency points to the full exploitation of available hardware resources.

In a program running on a parallel system, it is possible that some instructions need to be accomplished in sequence. This sequential execution has a limiting factor on program speed-up such that even adding more processors may not make the program run any faster. For instance, if a program

takes 20 minutes to finish using a serial code with one thread, and when the 5 minute portion of the code cannot be made parallel, the remaining 15 minutes of processing can be written as parallel code. In such situations, irrespective of how many threads are devoted to the parallelized execution of this program, the minimum execution time cannot be less than 15 minutes. For such evaluation of a program in the parallel computing domain, Amdahl's law [1] is used and can be represented as follows:

$$S = 1/((1 - p) + p/speed), \quad (1)$$

where S shows the theoretical speed-up of the execution, speed represents the speed-up of the part of the task that benefits from improved system resources, and P is the proportion of the execution time that the part benefiting from the improved resources originally occupied.

One of the most important criteria in parallel computing is to actually measure how much faster a parallel algorithm runs with respect to the best sequential one. This measure is known as "speed-up." In other words, speed-up is the gain in speed made by a parallel execution compared to a sequential execution. Any program that results in higher speed is not necessarily efficient. The efficiency of a program is described as using $p$ processors, or how effectively all system hardware elements are being utilized. If the efficiency is 1.0, then it is the maximum theoretical efficiency and shows the optimal usage of computational resources available for execution. If speed-up is not greater than linear, the efficiency will be less than or equal to 1.0, and this is normally the situation in practical cases.

In computer science, matrix multiplication is of great interest to many application areas, and a lot of work has been done in this regard in the literature [3, 9–23]. It has been shown that the number of processes does not necessarily result in performance gain. Recently, the authors in [3] measured the speed-up and efficiency of a matrix multiplication benchmark running on Amazon EC2. Their experiment shows why the performance of HPC applications on the cloud is not predictable due to the shared resources and multitenant environment of the cloud [3]. Various improvements in matrix multiplication have been discussed in the literature [16, 24, 25], and further gains are possible using GPUs [26–29]. Recently, for more secure communication between these IoT devices, the authors in [30] extended the work by proposing a lightweight matrix-based key management protocol for IoT networks.

OpenMP is an open specification for multiprocessing and offers a standard API for defining multithreaded, shared-memory programs [31]. The OpenMP high-level API consists of 80% preprocessor (compiler) directives, 19% library calls, and around 1% of environment variables. This framework presents the fork-join model of parallel execution. OpenMP is an API that is portable, supports threading, and can work with shared-memory programming specifications with "light" syntax. It is to be noted that the exact behavior depends on the OpenMP implementation and the number of threads. OpenMP is an advanced API and works for both C and C++; and it requires compiler support. Since a program can have serial and parallel sections, OpenMP allows a programmer to separate a program into serial regions and parallel regions, hide stack management, and provide synchronization constructs. As a potential drawback, OpenMP cannot detect dependencies in the code nor guarantee speed-up. In addition, it cannot provide freedom from data races, and it is the responsibility of the programmer to avoid such cases.

## 3. Impact of Row-Major Programming Constructs

In the computer science domain, two methods exist for storing multidimensional arrays, such as matrices, in linear storage and in random access memory. They are called column-major and row-major orders. These methods are different in the way in which elements are stored contiguously in the memory. In column-major order, elements are arranged consecutively along the column, while elements are arranged consecutively along the row under row-major sequence. Python, C, C++, Objective-C, and Java implement the column-major order when storing elements in memory, while FORTRAN, MATLAB, Julia, and Pascal use the column-major order.

In row-major order, elements are placed in memory as shown in Table 1, where the entire row is placed at one location in the memory, ignoring the cache line size such that there are multiple rows of the matrix. Table 2 represents the representation of the elements of Matrix C in cache, where a minimal block of data is transferred between the memory and the cache in a better algorithm, and the entire block of memory is placed in the cache line i.e., the word length and block length are of the same size in this paper. We then study the effect of spatial locality on program performance.

We name the first Matrix A, the second B; the product of Matrixes A and B is stored in Matrix C, as shown in Figure 1. It can be seen that under the *j, k, i* loop ordering, the program takes the longest due to poor placements of elements in memory. In Figure 2, we can see B has excellent spatial locality, but the code is dominated by two other Matrices A and C, where the placement is poor and, hence, the program shows worse case behavior when implanted in C, C++, Python, or Java.

As an alternative, we implemented the program as follows, written in C++ in Figure 3. We can now visualize the memory layout in Figure 4, where Matrix B offers poor, Matrix A offers good, while Matrix C presents the best spatial locality for the elements. In Figure 4, it can be seen that for Matrix C, the elements are placed $n$ elements apart, and that is why there is a miss, which takes more time for the system to load the value. For Matrix B, it is relatively closer, as only a desired number of steps are taken, but for Matrix C, only one location is updated, and hence, it offers an excellent spatial locality. This program improves the performance by a factor of three when the matrix is of dimension 4096 and implemented in C++.

We then rewrote the code and adjusted the loop order (see Figure 5). This arrangement did not affect the correctness of the program. The aim of this arrangement was to

Table 1: Matrix representation and placement in memory with row-major order.

| | | | |
|---|---|---|---|
| Row 1 | | | |
| Row 2 | | | |
| Row 3 | | | |
| ... | | | |
| Row n | | | |
| Memory: | Row 1 | Row 2 | Row 3 |

Table 2: Representation of elements of Matrix C in cache.

| Matrix C | | | | Cache line |
|---|---|---|---|---|
| C [0][0] | C [0][1] | C [0][2] | C [0][n] | Cache line-0 |
| C [1][0] | C [1][1] | C [1][2] | C [1][n] | Cache line-1 |
| C [2][0] | C [2][1] | C [2][2] | C [2][n] | Cache line-2 |
| ... | ... | ... | ... | ... |
| C [n][0] | C [n][1] | C [n][2] | C [n][n] | Cache line-n |

```
For (j=0; j<n; j++){
        For (k=0;k<n; k++){
                For (i=0; i<n; i++){
                C[i][j]=A[i][k]*B[k][j];}}}
```

Figure 1: Matrix multiplication with orders *j*, *k*, and *i*.

obtain a better spatial locality, as the program takes much less time compared to the counterpart implements. The corresponding spatial locality for Figure 5 is shown in Figure 6. It can be seen that since Matrixes C and B offer good spatial locality, Matrix A has excellent spatial locality, and hence, this arrangement surpasses its other counterparts in performance. Theoretically, the code shown in Figure 5 is approximately 14 times faster than the one implemented in Figure 2 due to better spatial locality. This arrangement favors row-major languages, while it will hurt the performance of column-major languages due to the inappropriate memory layout of array elements.

## 4. Discussion

This experiment was conducted on Windows 10 and the system information is shown in Table 3. The complexity of the program is $2n^3$, where $n$ represents the loop iterations. Our analysis shows that Python's implementation is poor in all three languages. While Java is relatively slow compared to the C++ implementation. This study is limited to matric multiplication and test programs written in C++. Java and Python to perform matrix-matrix multiplication as follows:

(i) Dimensions of each matrix are n×n and the elements/values of the matrix are of type double.

(ii) Populate each matrix with randomly generated values, etc.

(iii) Add the necessary code to measure the time taken by the matrix-matrix multiplication.

(iv) Observe the behavior of loop reordering.

(v) Evaluate the performance gains with parallel implementation in OpenMP.

We first ran sequential versions of the program while changing the matrix size $n$ from 500 to 2500 in steps of 500. We recorded the time to perform matrix-matrix multiplication for each execution. It is worth noting that interpreters can easily support high-level programming features [5]. This is due to the flexibility with which the interpreter reads, interprets, and performs each program statement and updates the machine state. However, the features of dynamic code alteration come at the cost of performance loss. This is why Python is the slowest in this class. While Python is interpreted, Java overcomes this drawback with a just-in-time compiler feature. Since C++ is compiled, it presents the best performance. In Table 4, we show our results where the runtime for C++, Java, and Python is extracted for a matrix starting from size 500 to 2500.

The just-in-time compiler of Java can recover some of the performance lost by interpretation. In Java, when the code is executed for the first time, it is interpreted first. Interestingly, the system keeps information about how often various pieces of code are executed. Whenever a piece of code executes frequently, the code is compiled to machine code in real time, and the next executions of that code use the more efficient compiled version [3].

We further studied the behavior of C++ by changing the loop order. In matric multiplication, there exist combinations where changing the order of the loops will not affect the correctness of the program. In the following experiment, we showed such combinations and provided the performance of the code (see Table 5). In all the aforementioned three programming languages, matrices were placed in row-major order. It is worth mentioning that the matrix size was the same for all experiments under C++, Java, and Python, while the time varied significantly due to the arrangements of the matrix elements in the memory. As discussed earlier, the poor arrangement of elements results in more cache misses and thus the program takes more time, while an excellent arrangement guarantees improved performance. The parallel version of the code is shown in Figure 7, where pragma "omp parallel for" is used. When this code runs on real hardware, the number of threads, which depends on the hardware and operating system, becomes of interest for performance. In Figure 7, we did not specify the number of threads, but the code can run on any system where more threads will definitely enhance the execution of the code significantly.

The results show the improvement of code in C++ over Java and Python. In our last experiment, we extracted results for a parallel code written in C++. We used the pragma "# pragma omp parallel for" in the parallel implementation of the code. The parallel matrix multiplication program with the support of OpenMP, shows a noticeable gain in speed-up for varying matrix sizes. Through our analysis, we conclude that the row-major policy is better for the matrix multiplication using the loop ordering *i, k, j*. It is worth noting that although *i, j, k* order is the natural one and easy to

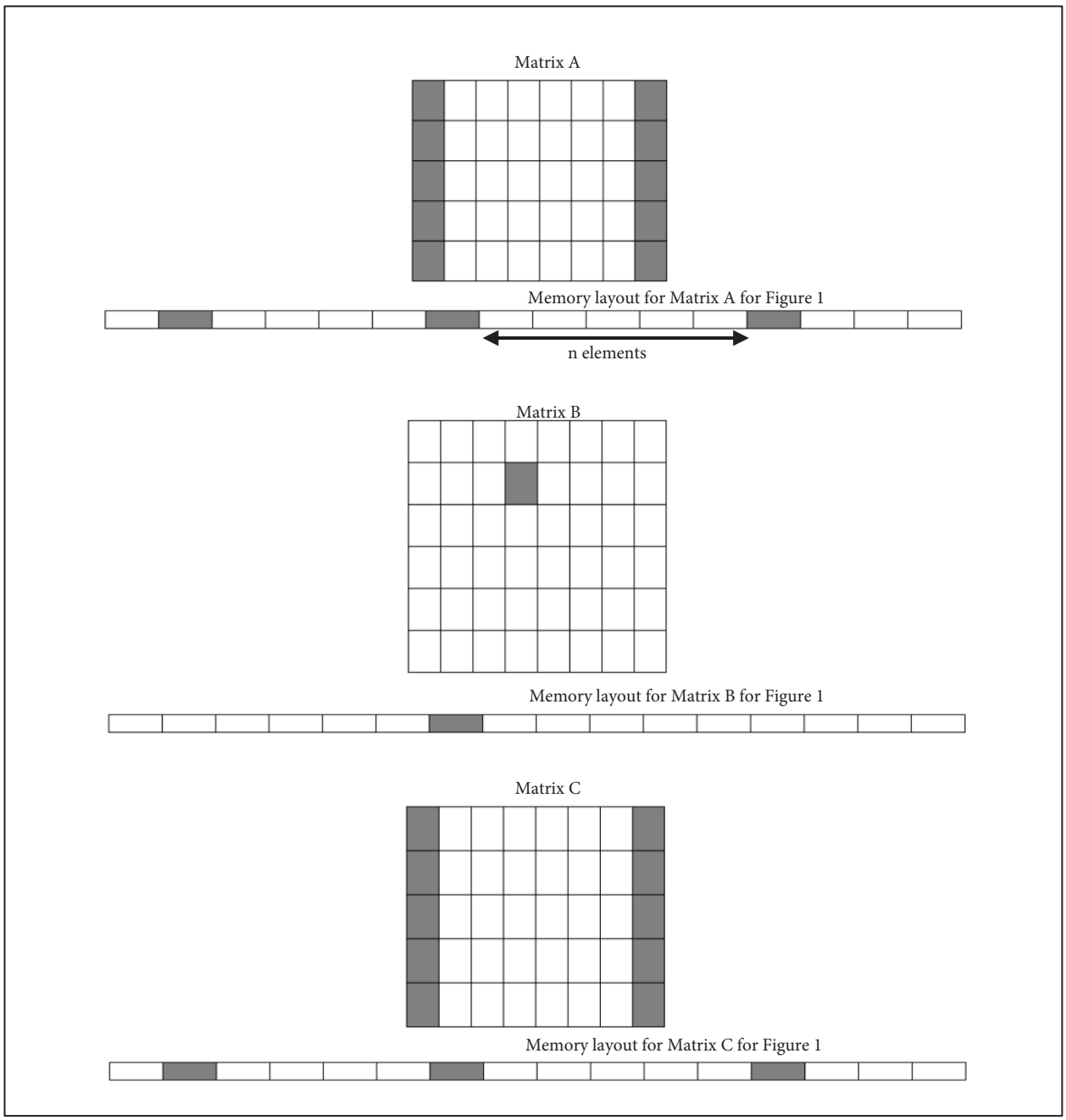


Figure 2: Spatial locality of Matrixes A, B, and C++ using *j*, *k*, and *i* ordering.

```
For (i=0; i<n; i++){
        For (j=0;j<n; j++){
                For (k=0; k<n; k++){
                        C[i][j]=A[i][k]*B[k][j];}}}
```

Figure 3: Matrix multiplication with orders *i*, *j*, and *k*.

understand, it results in poor memory layout, and therefore, the performance is poor. The order *i, k, j* places the data in such a way that the cache hit is increased, and thus, more gains are observed. Considering the *i, k, j* order as the most suitable looping order, we recommend this strategy for improved performance, irrespective of serial and parallel versions. We show our results for both serial and parallel implementation in Table 6. Hence, we implemented the best serial code, which was *i, k, j* ordering, and hence, the results were superior.

After implementing the code using the OpenMP framework, it is observed that the execution time of the parallel version is approximately seven times better than the execution time of the serial one, and thus, an improvement of around seven times has been obtained. It is worth noting

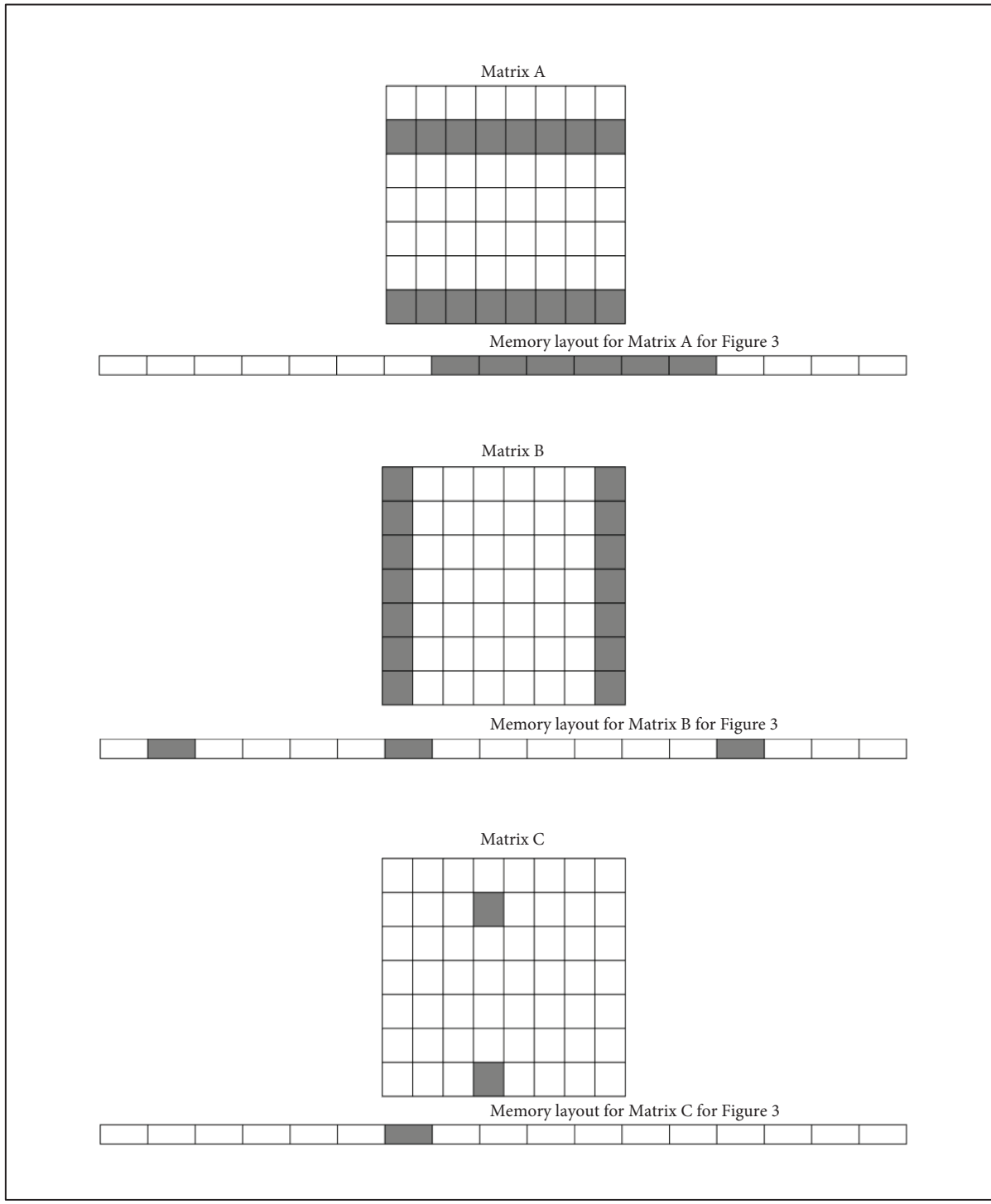


Figure 4: Spatial locality of Matrixes A, B, and C using *i*, *j*, and *k* ordering.

```
For (i=0; i<n; i++){
        For (k=0;k<n; k++){
                For (j=0; j<n; j++){
                        C[i][j]=A[i][k]*B[k][j];}}}
```

Figure 5: Matrix multiplication with orders *i*, *k*, and *j*.

that the efficiency is achieved mainly due to the algorithm, while the system performance depends on the data structure. It can be noted that the performance is not linear in our experimentation, and this is understandable. In theory, the speed should be eight times faster as we have eight cores on the system, but since there are communication, synchronization, etc., overheads involved, and speed-up is not linear,

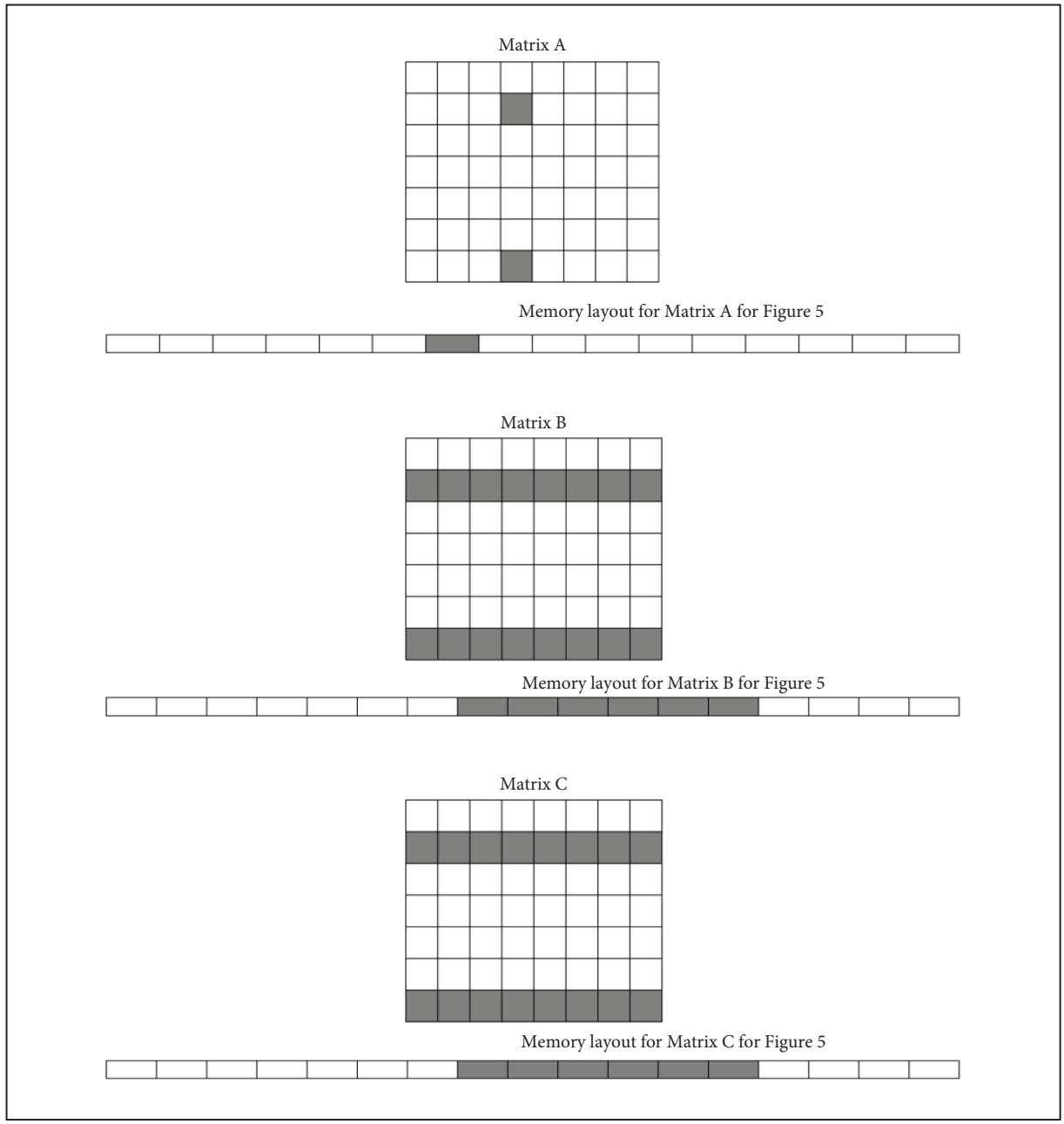


Figure 6: Spatial locality of Matrixes A, B, and C using *i*, *k*, and *j* ordering.

Table 3: Experimental setup.

| Item | Specification |
|---|---|
| Processor | Intel ® core (TM) i7-8650U |
| Installed memory (RAM) | 16 GB |
| System type | 64- Bit operating system |
| Base speed | 2.11 GHz |
| Cores | 8 |
| Logical processors | 16 |
| Virtualization | Enabled |
| Cache (L1, L2, and L3) | 256 KB, 1 MB, and 8 MB |

Table 4: Results extracted by changing row and columns using serial code.

| N | Loop order (i is the outermost loop) | Time taken (seconds) | | |
|---|---|---|---|---|
| | | C++ (serial code) | Java (serial code) | Python (serial code) |
| 500 | *i, j, k* | 0.312 | 0.324 | 2.345 |
| 1000 | *i, j, k* | 2.334 | 3.932 | 5.456 |
| 1500 | *i, j, k* | 10.881 | 16.245 | 23.453 |
| 2000 | *i, j, k* | 37.382 | 43.88 | 122.304 |
| 2500 | *i, j, k* | 97.293 | 124.593 | 566.452 |

Table 5: Results of serial code by changing loop order.

| N | Loop order | Time (in seconds) | | |
|---|---|---|---|---|
| | | *C++* (serial code) | Java (serial code) | Python (serial code) |
| 1000 | *j, k, i* | 5.436 | 11.492 | 27.342 |
| 1000 | *i, j, k* | 2.334 | 3.932 | 15.456 |
| 1000 | *i, k, j* | 0.802 | 1.803 | 5.453 |
| 1500 | *j, k, i* | 7.532 | 18.432 | 43.345 |
| 1500 | *i, j, k* | 10.881 | 16.245 | 23.453 |
| 1500 | *i, k, j* | 1.342 | 5.352 | 8.345 |

```
# pragma omp parallel for private (tid)

For (i=0; i<n; i++){
        tid = omp_get_thread_num();
        For (k=0;k<n; k++){
                For (j=0; j<n; j++){
                        C[i][j]=A[i][k]*B[k][j];}}}
```

Figure 7: Parallel implementation of matrix multiplication in OpenMP.

Table 6: Parallel code matrix-matrix multiplication in C/C++.

| N | C++ (serial code) [i, k, j] | C/C++ (parallel code) [i, k, j] |
|---|---|---|
| 500 | 0.4378 | 0.093 |
| 1000 | 2.815 | 0.456 |
| 1500 | 20.814 | 2.845 |
| 2000 | 40.748 | 8.102 |
| 2500 | 115.456 | 16.324 |

a seven-time improvement is still achieved with parallel implementation using *i, k,* and *j* orders.

## 5. Conclusions

We implemented matrix multiplication for row-major order languages and observed that C++ is more efficient in terms of runtime. The performance gap in programming languages, such as C++, Java, and Python, is due to the use of interpreters and compilers. Since C++ becomes compiled to machine language, it is the fastest. As a workout solution, the just-in-time feature of Java makes it faster than Python for matrix multiplication. We observed that the layout of elements in the memory has a large impact on the computational cost of a program. The parallel implementation of C++ is obtained in a speed-up of a factor of 7.0. As a future work, it will be interesting to extend this work to implement applications such as chess, binary decision diagrams, and logistic regression, and so on.

## Data Availability

Data will be made available from the author upon request.

## Conflicts of Interest

The author declares that there are no conflicts of interest.

## Acknowledgments

The resources were provided by the Imam Abdulrahman Bin Faisal University.

## References

[1] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532-533, 1988.

[2] G. M. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, Article ID 13967, Atlantic City New Jersey, April 1967.

[3] S. Jamaliannasrabadi, "High performance computing as a service in the cloud using software-defined networking," Doctoral dissertation, Bowling Green State University, Ohino, OH, USA, 2015.

[4] C. E. Leiserson, "The Cilk++ concurrency platform," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, 2010.

[5] M. D. McCool, A. D. Robison, and J. Reinders, *2.5 Performance Theory*, pp. 61-62, Elsevier, Amsterdam, Netherlands, 2012.

[6] H. Rajaei and S. Jamalian, "HPC as a service in education," in *Proceedings of the ASEE Annual Conference & and Exposition*, 2016.

[7] A. Fernández, C. Fernández, J. Á. Miguel-Dávila, and M. . Á. Conde, "Integrating supercomputing clusters into education: a case study in biotechnology," *The Journal of Supercomputing*, vol. 77, no. 3, pp. 2302–2325, 2021.

[8] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low power CMOS digital design," *IEEE Transactions on Electronics*, vol. 75, no. 4, pp. 371–382, 1992.

[9] K. Zhong, R. Guo, S. Kumar, B. Yan, D. Simcha, and I. Dhillon, "Fast classification with binary prototypes," in *Proceedings of the Artificial Intelligence and Statistics*, pp. 1255–1263, April 2017.

[10] H. Zou and L. Xue, "A selective overview of sparse principal component analysis," *Proceedings of the IEEE*, vol. 106, no. 8, pp. 1311–1320, 2018.

[11] F. Andre, A. M. Kermarrec, and N. Le Scouarnec, "Quicker adc: unlocking the hidden potential of product quantization with simd," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 5, pp. 1666–1677, 2021.

[12] J. Wang, W. Liu, S. Kumar, and S. F. Chang, "Learning to hash for indexing big data—a survey," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 34–57, 2016.

[13] W. Wang, C. Chen, W. Chen, P. Rai, and L. Carin, "Deep metric learning with data summarization," in *Proceedings of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 777–794, Springer, Riva del Garda, Italy, September 2016.

[14] C. J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, and M. Dukhan, "Machine learning at Facebook: understanding inference at the edge," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 331–344, IEEE, Washington, D.C., USA, February 2019.

[15] Q. Ye, L. Luo, and Z. Zhang, "Frequent direction algorithms for approximate matrix multiplication with applications in CCA," *Computational Complexity*, vol. 1, 2016.

[16] Q. Yu, M. Maddah-Ali, and S. Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[17] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, Honolulu, HI, USA, July 2017.

[18] Z. Huang, "Near optimal frequent directions for sketching dense and sparse matrices," *Journal of Machine Learning Research*, vol. 20, no. 1, p. 23, 2018.

[19] P. L. Bartlett and S. Mendelson, "Rademacher and Gaussian complexities: risk bounds and structural results," *Journal of Machine Learning Research*, vol. 3, pp. 463–482, 2002.

[20] D. W. Blalock and J. V. Guttag, "Bolt: accelerated data mining with fast vector compression," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 727–735, Halifax NS Canada, August 2017.

[21] D. W. Blalock, J. J. G. Ortiz, J. Frankle, and J. V. Guttag, "What is the state of neural network pruning?" *Proceedings of Machine Learning and Systems*, vol. 2, pp. 129–146, 2020.

[22] J. Camacho, A. K. Smilde, E. Saccenti, and J. A. Westerhuis, "All sparse PCA models are wrong, but some are useful. Part I: computation of scores, residuals and explained variance," *Chemometrics and Intelligent Laboratory Systems*, vol. 196, Article ID 103907, 2019.

[23] OpenMP, *The OpenMP API Specification for Parallel Programming*, OpenMP, 2021, https://www.openmp.org/.

[24] R. Yuster, "Efficient algorithms on sets of permutations, dominance, and real-weighted APSP," in *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, no. 950–957, pp. 950–957, New York, NY, USA, January 2009.

[25] Nvidia Corp, "CUDA CUFFT library," 2022, https://developer.nvidia.com/content/cuda-toolkit-11-june-2007.

[26] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–11, IEEE, Austin, TX, USA, November 2008.

[27] Nvidia Corp, *Dense Linear Algebra. SC08 NVIDIA Corp. CUDA Compute Unified Device Architecture*, NVIDIA Corp, California, CL, USA, 2008.

[28] Nvidia Corp, "Programming guide," 2022, http://oz.nthu.edu.tw/%7Ed947207/NVIDIA/NVIDIA_CUDA_Programming_Guide_2.0.pdf.

[29] Nvidia Corp, *CUDA Compute Unified Device Architecture, Programming Guide*, NVIDIA Corp, California, CL, USA, 2008.

[30] M. Nafi, S. Bouzefrane, and M. Omar, "Matrix-based key management scheme for IoT networks," *Ad Hoc Networks*, vol. 97, Article ID 102003, 2020.

[31] A. J. Stothers, "On the complexity of matrix multiplication," PhD thesis, University of Edinburgh, United Kingdom, 2010.