# Performance Evaluation of Matrix-Matrix Multiplication using Parallel Programming Models on CPU Platforms

**Alireza Akoushideh** (akushide@tvu.ac.ir)
Technical and Vocational University (TVU)
**Asadollah Shahbahrami**
University of Guilan

**Research Article**

**Keywords:**

**Posted Date:** October 11th, 2022

**DOI:** https://doi.org/10.21203/rs.3.rs-2135830/v1

## 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. Matrix-matrix multiplication is an important kernel used in many scientific and multimedia applications. This kernel is a compute-intensive program too. This paper aims to efficiently implement this kernel using different parallel programming models such as SIMD, OpenMP, and OpenCL. Our experiment results for different matrix sizes show that speedups up to 6.5x, 3.2x, 16.7x, and 32x are achieved for SIMD, OpenMP, hybrid OpenMP-SIMD, and OpenCL implementations over-optimized serial implementation, respectively.

## 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].

OpenMP is a high-level programming API that enables Thread Level Parallelism (TLP) for sequential programs [2]. It enables parallelization of the sequential algorithm without restructuring, where the programmer defines the amount of parallelism granularity by a total number of threads, where a master thread schedules the given task among slave threads. OpenCL was introduced by Apple and jointly developed by AMD/ATI, Intel, and NVIDIA for both CPU and GPU platforms[3, 4]. The OpenCL platform has one host and one or more devices, the host is typically the main CPU and devices could be either CPUs or GPUs [5]. On the CPU side, it has an auto-vectorization methodology for given issues. Many researchers have worked on improving MMM performance in both CPU and GPU platforms using SIMD, OpenMP, and OpenCL while achieving different results[6-14]. However, there is not a clear evaluation and comparison between these parallel programming models, especially for hybrid models.

Matrix-Matrix Multiplication (MMM) is one of the main kernels in multimedia and scientific applications [12-14]. There are different ways to implement this kernel such as basic multiplication, matrix transposition, loop interchange, and blocking. However, these implementations for large matrix multiplication are computationally intensive [6-10]. For example, the multiplication of two matrix sizes of 6400×6400 with the improved algorithm is about 180 seconds on an Intel Core i7 system.

The goal of this paper is to implement the MMM on different CPU platforms using different parallel programming models such as SIMD, OpenMP, Hybrid OpenMP-SIMD, and OpenCL. In addition, we try to answer this question in this paper, what kind of implementation in CPU platforms can achieve more speedup for MMM? And what kind of parallel programming models yields more speedup compared to others? Our experimental result shows that speedups up to 6.5x, 3.2x, 16.7x, and 32x are achieved using SIMD, OpenMP, OpenMP-SIMD, and OpenCL for different matrix sizes, respectively.

This paper is organized as follows, in section II background is described. Section IIIillustrates related works on matrix-matrix multiplication. In section IV proposed approach is presented. Section V illustrates our selected benchmarks, test-bed machine, and experimental result in detail. Finally, conclusions are presented in Section VI.

## 2 Background Information

## 2.1 Matrix Multiplication

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 Fig 1. In a basic implementation, the cost of fetching matrix B column has a bad great impact on overall performance. While for the transposed method the only extra cost is transposing Matrix B. However, in loop interchange and blocking the data reuses gains much better performance compared to basic and transposed methods.

Hence, the loop interchange method has been chosen as our basic CPU implementation. In basic MMM where the hole element of one row of A are multiplied into one whole column of the matrix B, the loading element of B columns is a bottleneck, on the other hand, MMM is a regular application, that in basic MMM when a row is multiplied into a column of the second matrix, so we can do it individually by loop interchange. C implementation code for basic MMM and loop interchange are demonstrated in Fig 2.

In loop interchange, every single element of matrix A will be selected once and multiplied to the hole element of the same row of matrix B and the result will be stored in matrix C elements, loop interchange algorithm decreases cache miss and increases data reuse. Each product of this multiplication will be added to one element of matrix C to create a C element in a very small step-by-step order. See Fig 3 for more detail. This code runs up to 2 times faster than transposed MMM and also 8 times better than basic MMM in our selected matrix ranges.

## 2.2 Parallel Programming Models

There are many parallel programming techniques such as instruction-level parallelism (ILP), data-level parallelism (DLP) using SIMD, thread-level parallelism (TLP) using OpenMP, as well as OpenCL in terms of auto-vectorization. These programming models can be implemented in single and multi-core CPUs which will be described in the proceeding [1, 4, 6-8, 11, 15].

### 2.2.1 SIMD

Single Instruction Multiple Data (SIMD), a type of parallel computing architecture that is classified under Flynn's taxonomy, enables the processing of multiple data with a single instruction which is supported by recent CPUs manufactured by Intel and AMD.

The first SIMD instruction set for x86 architecture was called MMX, developed by Intel in 1996 that is also known as the Multimedia Multiple Math extension. By MMX instructions, Intel introduced eight registers (MM0 to MM7) each register is 64 bits wide but only is used for integer operations. Thereafter, Intel introduced SSE (Streaming SIMD Extensions). The SSE has eight 128-bit wide register files (XMM0–XMM7) for x86 systems and sixteen 128-bit registers (XMM0-XMM15) for x64 systems that concentrate more on single-precision floating-point operations rather than integer operations [1].

SSE instructions implement the integer vector operations in a wider range of bits compared to MMX, besides supporting double-precision floating-point in 64-bit wide. Other extensions were introduced with some new instructions and changes. As AVX registers are twofold bigger than SSE registers, AVX provides the ability to use 8 single-precision floating-point numbers or 4 double-precision floating-point numbers in comparison to SSE instructions that can process only half of them. An example is illustrated in Fig 4.

### 2.2.2 OpenMP

Open multiprocessing (OpenMP) is an application programming interface for shared-memory multiprocessor systems that are designed for CPU-based applications [30]. OpenMP method is implementing a multithreading strategy, whereby a master thread forks a determined number of slave threads while the OpenMP runtime divides tasks between them, as shown in Fig 5. OpenMP uses *pragma* to fork additional threads to expedite the work in parallel. The original thread will be signified as a master thread with thread ID 0.

OpenMP determines a global view of programming, thereby the algorithm remains general like the original sequential one and will be parallelized with compiler directives at runtime. OpenMP provides a portable and scalable model for programmers on shared memory parallelization by exploiting Single Process Multiple Data (SPMD) and Multiple Process Multiple Data (MPMD) [16]. In addition, SIMD vectorization is also supported since OpenMP 4.0 where programmers could easily use “ *pragma omp simd* ” to vectorize their code [2]. Due to the goal of this paper which in part is to compare SSE and AVX in the OpenMP-SIMD version of MMM, the code will be utterly written in *intrinsic function,* and “ *pragma omp simd* ” will not be used.

### 2.2.3 OpenCL

Several important corporations, including Intel and NVIDIA, developed a standardized programming model called Open Compute Language (OpenCL) [3]. OpenCL is very similar to CUDA in the case of parallelism management and data delivery in parallel processing. In comparison to CUDA, OpenCL relies more on APIs, OpenCL can run correctly without modification on all processors that support it [5].

Many libraries use OpenCL for acceleration, such as MAGMA, clAMDBLAS, BOLT C++ Template library, and JACKET which accelerates MATLAB on GPUs, also some Java bindings are available for OpenCL. Hence the OpenCL is a cross-platform parallel computing API that defines a C-like language for writing programs, this framework contains a c99-based language for writing kernels and an API for controlling OpenCL supported devices. The OpenCL key factor is its portability across many multi-processors like CPUs, GPUs, and FPGAs. The aim of this API for programmers is to put aside whole hardware specification concerns by offering a hardware abstraction [17].

OpenCL provides a unique programming environment for programmers to write efficient, portable code for high-performance computing in CPUs and GPUs. The OpenCL specification describes its programming architecture details, and a set of APIs to perform specific tasks, which are all required by an application developer. This specification is provided by the Khronos OpenCL consortium. OpenCL employs a data-parallel execution model relying on its fine granularity based on target platforms. Moreover, it has two special parts, one called *kernel* which executes in parallel, and another called *host* which is standard C/C++ form. Kernels lunches by *work-items*, the particular number of work-item creates *a work-group*. The OpenCL mapping strategy makes it suitable for implementing BLAS (Basic Linear Algebra Subprograms), Level 1 BLAS perform scalar, vector, and vector-vector operations, Level 2 BLAS perform matrix-vector operations, and Level 3 BLAS perform matrix-matrix operations as shown in (1) where $\alpha$ and $\beta$ are random scalars [9, 18].

$$C = \alpha AB + \beta C \qquad (\mathbf{1})$$

## 3 Related Works

MMM speedup has been the major goal of many studies[6–11] and is still ongoing today. BLAS [9, 19] is Basic Linear Algebra Subprograms (BLAS) that provides a standard blocking method for matrix multiplication. It has been widely used in many libraries, like ATLAS, and NVIDIA cuBLAS [6]. Related works with clear results are listed in Table 1.

Table 1
Related works on CPU Platform, Core-thread stands for some physical cores and total supported thread respectively, Max (N) stands for Maximum Matrix dimension used in the related article.

| Ref | CPU | Core-tread | Max (N) | Programming model | Applied Algorithms |
|---|---|---|---|---|---|
| [6] | Xeon E3-1241 | 4–8 | 4800 | SIMD | Tiling, cache optimization |
|  | Core i7-2600K | 4–8 | 4800 | SIMD | Tiling, cache optimization |
|  | ARMv7-a | – | 920 | – | cache optimization |
|  | Xilinx Virtex-5 FPGA | – | 920 | – | cache optimization |
| [7] | Core i7-5600U | 2–4 | 1120 | C++ SIMD | Basic, transposed |
| [8] | Pentium core 2 E6550 | 2–2 | 4800 | SIMD | Tiling and cache management |
|  | core i7 2600k | 4–8 | 4800 | SIMD | Tiling and cache management |
| [9] | Core i7 3960X | 6–12 | 8192 | OpenCL | BLAS |
|  | Xeon Phi 5110P | 60–240 | 8192 |  |  |
| [18] | Intel Core i7 3960X | 6–12 | 6144 | OpenCL | Basic, transposed |
|  | AMD Bulldozer FX-8150 | 8–8 | 6144 |  |  |
| [15] | Intel Pentium D 3.40GHz | 2–2 | 2100 | MPI MPI + OpenMP | Blocking |
| [20] | Intel Xeon quad-core | 4–8 | 4096 | OpenMP | Loop parallelization |
| [21] | AMD Athlon II X2 270 | 2–2 | 834 | OpenMP | Loop parallelization |
| [11] | Intel Xeon Phi 7120P | 61–244 | 60000 | OpenMP MPI + OpenMP | Basic |

SIMD is widely used to improve MMM performance [6–8, 10], there are three types of writing SIMD code, *inline assembly*, *intrinsic function*, and *vector class*. Although in inline assembly code the programmer has full control over registers and their behaviour, it’s not always the best way, especially for new compilers. Therefore, in the SIMD version of MMM the *intrinsic function* is used which is written as the standard form of C. SSE could be an eligible way to implement MMM, it has been implemented for square matrix sizes in a wide range of hardware [6–8], while giving their method to minimize DDR access as well as finding suitable parameters like tiling size based on cache sizes, we also implement a SIMD version of MMM in for Intel CPU while combining it with OpenMP which has a great effort on some selected platforms. Moreover, AVX has been implemented for small matrix sizes, while comparing Intel and Visual Studio compiler [7], also two *inline assembly* and *intrinsic function* AVX code for matrices up to 1120 sizes, the result revealed that *the intrinsic function* works faster.

OpenMP programming language [11, 15, 20] was studied for MMM. In [20] authors explore MPMD for the cache-aware algorithm of MMM and gain better performance compared to the single-core version of their code, but we will show that in some cases SIMD is better than OpenMP and the best case is to use them together.

Hybrid parallel programming model [11, 15] significantly improves overall performance for MMM, [15] presented MMM implemented in a multicore-cluster using MPI and hybrid MPI + OpenMP, while the Blocking technique is used, where each node receives a block of A and multiply it into a block of B, A blocks stay at the node till the end while B blocks are in cycle turning.

CPUs are different in cache level, the number of physical cores, the total number of supported threads, frequency, and memory bandwidth. But there is not much work on comparing different parallel programming languages in the CPU platform. Here we compare the most common MMM algorithm in SIMD, OpenMP, OpenCL, and hybrid OpenMP-SIMD.

There are also several written libraries such as Intel MKL [22] and GotoBLAS [19]. MKL and loop kernels are written in assembly code, while our method is in C and assembly code is always more efficient the same is true about GotoBLAS assembly code. In terms of MKL, the developers have access to all the processor architecture details, for example, victim cache, and hardware prefetchers. The MKL library is the fastest library on Intel processors only. So it will not be fair to compare the MKL or GotoBLAS to our results. On the other hand, our approach was to compare SSE, AVX, OpenMP, OpenMP-SIMD, and OpenCL as parallel programming languages in the case of matrix-matrix multiplication algorithms, and achieving the best possible performance for selected platforms was not the goal of this paper.

## 4 Parallel Implementation Of Matrix Multiplication

## 4.1 SIMD implementation of MMM

SIMD programming allows parallel processing in a single CPU. There are three methods to implement SIMD programming: inline assembly, intrinsic function, and vector class [1]. SIMD version of MMM was implemented using the *intrinsic function*. Inline assembly gives us the most liberty to gain the best possible performance by managing each register one by one, but it is so complicated and it requires cautious handling of data transfer between CPUs and memories [1]. SSE and AVX instructions provide parallel execution of data in the MMM kernel.

Due to SSE and AVX capability, we will do multiplication with a step size of four and eight floating-point operations using the intrinsic function, and store them into another register allocated before. The loop interchange algorithm did not work well for SSE and AVX, thus transposed matrix multiplication was used, and transpose time was added to kernel time. In kernel code, two registers (XMM0, XMM1 for SSE and YMM0, YMM1 for AVX) will load matrix A and B elements, and the summation of multiplication is stored in another register allocated before (XMM2 or YMM2). While the kernel is running single instruction to multiple data the different results are stored in an array size of 4 or 8, finally, the summation of array elements creates each element of matrix C. The AVX code is demonstrated in Fig. 6.

## 4.2 Multi-Core CPUs

By programming in SIMD fashion, it’s still a CPU single-core operation, but nowadays CPUs have at least 2-4-8 physical cores, and with multi-threading, by order 4-8-16 and more threads are available.

### 4.2.1 OpenMP

Open Multi-Processing (OpenMP) is a shared memory architecture API that provides TLP for multi-core CPUs using C, C + + or FORTRAN. OpenMP supports both task and data level parallelism [2]. The OpenMP version of MMM is illustrated in Fig. 7.

In OpenMP a loop can be parallelized by including the OpenMP library and using *pragma* with specific parameters for loops, also the granularity can be controlled by adjusting loop chunks with a scheduling type, such as *static* or *dynamic*. The rest of the parallelization is on the OpenMP compiler, which generates parallel code and the runtime system, and manages parallel threads and recourses. For the OpenMP code version, the loop interchange model was used and the outer loop was parallelized.

### 4.2.2 OpenCL

In OpenCL, the host code coordinates and queues the data transfer and kernel execution commands. The device code executes the kernel code in an array of work items called work-groups.

In this paper, we used the Intel SGEMM library [23] which uses BLAS and measures its effects on CPUs to compare it with hybrid OpenMP-SIMD kernel code. The code contains basic, transpose, and Blocking techniques, but discovering suitable work-item and work-group dimensions plays a major role in gaining the best performance, we will use the method that is mentioned in (2). To fully utilize automatic vectorization in selected platforms we set work-item dimensions as 1 and 16 where 16 is equal to the total number of floating-point operations in a SIMD manner. Also, the work-group dimensions should be at least bigger than 8 because of our selected platforms which all support AVX.

$$\boxed{\text{Work-group } (W/2, W/128), \text{ Work-item } (16,1) \qquad (2)}$$

### 4.2.3 Hybrid OpenMP-SIMD

A combination of OpenMP and SIMD instruction has been used, in this version, each thread running on the CPU has full advantage of SIMD functions but the number of registers is limited. OpenMP uses Fork and joins method where a master thread will automatically spread works beyond slave threads.

Moreover, SIMD vectorization is supported since OpenMP 4.0 is released, so by combining OpenMP and SIMD the kernel code will be broken into 4–8 tasks which have several threads provided by the runtime system and can increase register allocation to up to 8–16 for loading matrix elements (each can load 4 up to 8 elements) and 4–8 register for storing multiplication in it. As it is clear there are up to 16 registers available in both SSE (for 64-bit systems) and AVX, so a drop-down in performance is predictable, moreover loading 8 elements will need much more bandwidth and it could be another bottleneck.

## 5 Performance Evaluation

## 5.1 environment and hardware system

We evaluated the performance of the MMM Implemented by SSE, AVX, OpenMP, and OpenMP-SIMD programmed using intrinsic functions and compared the best cases with an OpenCL version of MMM. Tested CPU platforms, all contain 256-bit wide vector units which gave the ability to perform up to 16 single-precision floating-point operations. Testbeds are specified in Table 2, for the rest of our implementation results, square matrices have been selected and loaded with random floating-point numbers. On CPU platforms, all code versions such as single-core, SIMD, OpenMP, and hybrid OpenMP-SIMD kernel time are measured by Visual Studio 2013(MSVC++) Profiler aside from the CPU *clock ()* function. Withal, Intel SDK 5.3 is used for OpenCL application analysis. The minimum runtime has been measured for each selected system.

Table 2
Platform's specification. There are four systems named F, R, T, and A.

| | F | | R | | T | | A | |
|---|---|---|---|---|---|---|---|---|
| CPU | Intel® Core™ i3-2370M | | Intel® Core™ i3-5005U | | Intel® Core™ i7-2630QM | | Intel® Core™ i7-6700HQ | |
| Architecture | Sandy Bridge | | Broadwell | | Sandy Bridge | | Skylake | |
| Number of physical cores | 2 | | 2 | | 4 | | 4 | |
| OS | Windows 8 64bit | | Windows 8.1 64bit | | Windows 7 64bit | | Windows 10 64bit | |
| Core Clock | 2400 MHz | | 2000 MHz | | 2000–2900 MHz | | 2600–3500 MHz | |
| L1 | 128 KB | | 128 KB | | 256 KB | | 256 KB | |
| L2 | 512 KB | | 512 KB | | 1024 KB | | 1024 KB | |
| L3 | 3072 KB | | 3072 KB | | 6144 KB | | 6144 KB | |
| Memory Bandwidth | 21.3 GB/s | | 21.3 GB/s | | 25.6 GB/s | | 34.1 GB/s | |
| Instruction Set Extensions | SSE | AVX | SSE | AVX 2.0 | SSE | AVX | SSE | AVX 2.0 |
| SIMD vector widths | 128-bit | 256-bit | 128-bit | 256-bit | 128-bit | 256-bit | 128-bit | 256-bit |

## 5.2 Experimental Result for Sequential Implementation of MMM

The loop interchange algorithm has been implemented in four Testbeds and the results are shown in Table 3. For all speedup comparisons, the result of each parallel implementation of MMM is compared to the sequential loop interchange algorithm result. The execution times for sequential MMM are presented in Table 3. For the rest of this paper, all parallel results of the F, R, T, and A systems will be compared to the result illustrated in the above Table.

Table 3
Execution time in seconds for sequential loop interchange algorithm on four different CPU platforms.

| Matrix size | F | R | T | A |
|---|---|---|---|---|
| 256×256 | 0.017 | 0.015 | 0.015 | **0.013** |
| 512×512 | 0.132 | 0.125 | 0.119 | **0.093** |
| 1024×1024 | 1.332 | 1.131 | 0.979 | **0.750** |
| 2048×2048 | 9.405 | 9.006 | 8.235 | **6.031** |
| 4096×4096 | 76.473 | 72.507 | 66.058 | **48.05** |
| 6400×6400 | 278.436 | 270.857 | 246.786 | **180.2** |

## 5.3 Experimental Results for SIMD and OpenMP Implementation

SSE and AVX have better performance for small matrix sizes compared to OpenMP. As it has been illustrated in Fig. 8, the range of SSE and AVX speedup over loop interchange algorithm is about 2 to 3.2 and 2.6 to 6.5 respectively. Due to the algorithm selected for SSE and AVX, which does not employ proper cache blocking by increasing matrix sizes the performance of both SSE and AVX decreases. Although AVX can process 8 elements of the matrix at the same time and has twice the speedup for small matrices compared to SSE, the achieved speedup for matrices larger than 2048 showed that there is not much difference between SSE and AVX and bandwidth limitations directly affect the SIMD.

In A-system AVX keeps its speed up in most cases and has fewer drops down compared to other Testbeds because of 34.1GBs bandwidths. On the other hand, OpenMP has the most stable results, although it did not represent interesting performance for small matrices, still, it has 3.2 speedups for a matrix size of 6400, we should mention that in the best-tested CPU the total number of threads was 8 which restricts OpenMP functionalities. In addition, SIMD relies more on memory bandwidth and is more suitable for small matrices while Open MP's multi-threading strategy showed a stable speedup overall. But the performance of MMM with OpenMP increases in all platforms by increasing matrix sizes. This represents that the volume of computations per thread increases respectively, while AVX still waiting for the exact 8 data elements to be computed.

### 5.3.1 SSE and AVX comparison.

As it is shown in Table 4, assembly codes generated for SSE and AVX are quite equal in the number of operations. The further analysis illustrated that latency is the key point in SSE and AVX compression, as we use an algorithm that does not cover memory bottleneck in MMM, the memory operation latency plays a major role in larger matrices.

Table 4
The number of assembly operations for SSE and AVX version of MMM, here “NUM. of” stands for the number of operations. Additionally, X and Y stand for 128 and 256-bit registers, respectively, while m128 and m256 refer to memory operations in 128 and 256-bit. Also, memory operations have two types which are illustrated as (register, memory) or (memory, register).

| | Memory Operations | | | Mathematical Operations | | |
|---|---|---|---|---|---|---|
| | Type | Num. of | Latency | Addition | Multiplication | Latency |
| AVX | Y,m256 | 14 | 3 | 9 | 12 | 6 |
| | m256,Y | 13 | 3 | | | |
| SSE | X,m128 | 14 | 2 | 5 | 8 | 6 |
| | m128, X | 14 | 3 | | | |

Although AVX has twice register size and calculation it also has a slower memory operation compared to SSE. These slower memory operations along with more arithmetic operations in each iteration cause the performance dropdown in larger matrices.

## 5.4 Hybrid OpenMP-SIMD and OpenCL Result

In multi-core CPUs, as shown in Fig. 9, we represented our hybrid methods, OpenMP-SSE and OpenMP-AVX as well as compared them to Intel OpenCL SGEMM [23]. Also, it has been said that for OpenCL a hybrid CPU-GPU has better results [24] but it was not the topic of this paper, so for OpenCL results the integrated GPUs were disabled and execution times are only for selected CPUs. On the other hand, SIMD vectorization has been supported since OpenMP 4.0 and we could simply use “*#pragma omp simd*” to vectorize our MMM code, but the goal of this paper was to compare two SIMD technologies which typically should have very different result and AVX code should be twice faster SSE while the experimental result revealed it is not.

The result exhibited that OpenCL can gain the best performance but as the matrix size increases the performance of all three versions decreases concurrently. In this experiment, T-system could not handle OpenCL code correctly due to an unknown error while it had the same Intel OpenCL driver installed, so T results are not mentioned for OpenCL. However, OpenCL has a more inclined drop-down in charts compared to OpenMP-AVX. Also in some cases, the performance of OpenMP-AVX increases, while OpenCL performance is falling by a factor of -1.7x, and it gives the idea that in matrices larger than 6400 the differences between OpenCL and OpenMP-AVX will not be considerable.

Although achieving the best possible performance was not beyond the purpose of this paper, the best performance achieved for the MMM is illustrated in Fig. 10. These results determined that programming languages with a fine granularity model are more suitable for the MMM and larger registers do not have that much impact on applications that are significantly dependent on memory operations.

As the number of floating-point operations in matrix multiplication is ( $2n^3\text{FLOPS}$) [7]. The performance measuring selected for Fig. 10 is GFlops which stands for the total number of Floating-point operations executed per second in the selected platform. The performance comparison showed that OpenCL has better performance, but two major issues should be taken into account. First, the Intel OpenCL SGEMM uses a blocking algorithm that could hide memory bandwidth limits. Second, the selected CPUs did not support more than 8 threads, so OpenMP results for better CPUs could be better.

## 5.5 performance comparison

Since the CPU used in the R-system is very close to the one used in [7], comprehensive results are illustrated in Fig. 11. Both systems use Intel Broadwell family CPUs, except that in R-system frequency is 2GHz compared to 3.2GHz in [7]. Both systems have two cores and 4 threads as well as the same instruction set supported by processors. The result showed that hybrid OpenMP-AVX has much better performance in all selected matrices.

Best achieved performance for [7] was 8.16GFlops, while our hybrid OpenMP-AVX code version achieved 25.48 GFlops in the best case. The comparison revealed that using SIMD only to boost scientific applications could not be a significant choice.

## 6 Conclusions

The performance of different parallel programming models such as SIMD, OpenMP, and OpenCL for an efficient implementation of matrix-matrix multiplication on different platforms has been evaluated. The result demonstrated that a larger register and an increased number of floating-point operations for recent SIMD instruction sets have better performance for smaller matrices than large matrices. In addition, our experimental results show that the combination of SIMD and OpenMP has much higher performance compared to SIMD and OpenMP implementations.

## Declarations

**Funding** The author(s) received no financial support for the research, authorship, and publication of this article.

**Conflict of Interest** The authors declared no potential conflicts of interest concerning the research, authorship, and publication of this article.

**Data Sharing** Data sharing is not applicable to this article as no datasets were generated or analysed during the current study.

**Consent for publication:** The Authors have agreed to publish this manuscript.

**Authors' contributions:**  All authors contributed to the study's conception and design.

**Code Availability** Not applicable.

**Ethical Approval** Not applicable.

**Consent to participate:** Not applicable

## References

1. H. Jeong, S. Kim, W. Lee, and S.-H. Myung, "Performance of SSE and AVX Instruction Sets," *arXiv,* 2012.
2. "OpenMP 5.1 Specification." The OpenMP ARB (Architecture Review Boards). https://www.openmp.org/specifications/ (accessed.
3. C. Bertoni *et al.*, "Performance Portability Evaluation of OpenCL Benchmarks across Intel and NVIDIA Platforms," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 18-22 May 2020, pp. 330-339, doi: 10.1109/IPDPSW50202.2020.00067.
4. S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler, "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption," presented at the Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing, Washington, DC, USA, 2017. [Online]. Available: https://doi.org/10.1145/3110355.3110356.
5. R. Banger, B. Bhattacharyya, and K. Bhattacharyya, *OpenCL Programming by Example*. Packt Publishing, 2013.
6. 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, 2016/03/01 2016, doi: 10.1007/s11227-015-1613-7.
7. 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.
8. 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, 2014/06/01 2014, doi: 10.1007/s11227-014-1098-9.
9. 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
10. 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, 2010/11/20/ 2010, doi: https://doi.org/10.1016/j.seppur.2010.08.023.
11. 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)*, 1-4 July 2015 2015, pp. 1-2, doi: 10.1109/CEM.2015.7237429.

12. 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.
13. 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: https://doi.org/10.1145/2663761.2664192.
14. 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.
15. L. He, W. Shen, Y. Li, A. Shi, and D. Zhao, "MPI+OpenMP Implementation and Results Analysis of Matrix Multiplication Based on Rowwise and Columnwise Block-Striped Decomposition of the Matrices," in *Third International Joint Conference on Computational Science and Optimization*, 28-31 May 2010, vol. 2, pp. 304-307, doi: 10.1109/CSO.2010.123.
16. J. Fang, C. Huang, T. Tang, and Z. Wang, "Parallel programming models for heterogeneous many-cores: a comprehensive survey," *CCF Transactions on High Performance Computing,* vol. 2, no. 4, pp. 382-400, 2020, doi: 10.1007/s42514-020-00039-4.
17. K. Uchiyama, M. Hariyama, and H. M. Waidyasooriya, *Design of FPGA-Based Computing Systems with OpenCL*. Springer, 2017.
18. K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Performance Tuning of Matrix Multiplication in OpenCL on Different GPUs and CPUs," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 10-16 Nov. 2012 2012, pp. 396-405, doi: 10.1109/SC.Companion.2012.59.
19. 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.
20. J. Li, S. Ranka, and S. Sahni, in *GPU matrix multiplication*: Chapman-Hall/CRC Press, 2013, ch. Multicore Computing: Algorithms, Architectures, and Applications.
21. L. Cleverson, D. Ledur, C. Zeve, and D. Anjos, *Comparative Analysis of OpenACC, OpenMP and CUDA using Sequential and Parallel Algorithms*. 2013.
22. "Intel® oneAPI Math Kernel Library." Microsoft. https://www.intel.com (accessed.
23. "General Matrix Multiply - Intel® SDK for OpenCL™ Applications." Intel. https://software.intel.com (accessed.
24. S. Mittal and J. S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Comput. Surv.,* vol. 47, no. 4, p. Article 69, 2015, doi: 10.1145/2788396.

## Figures

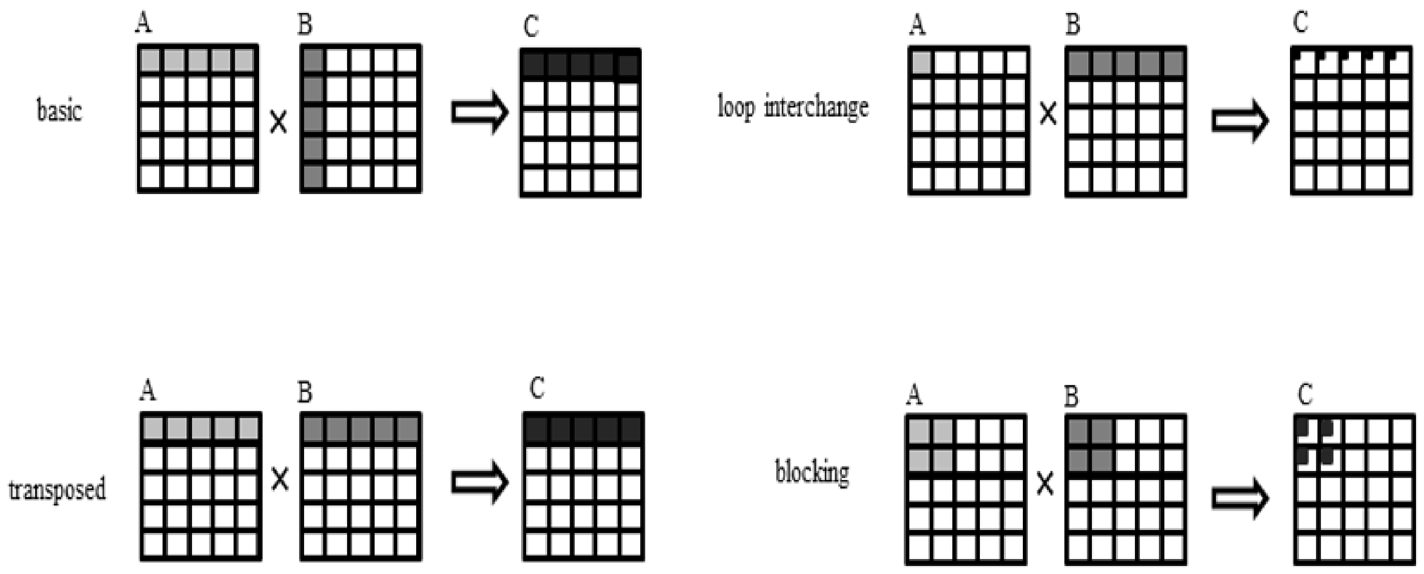
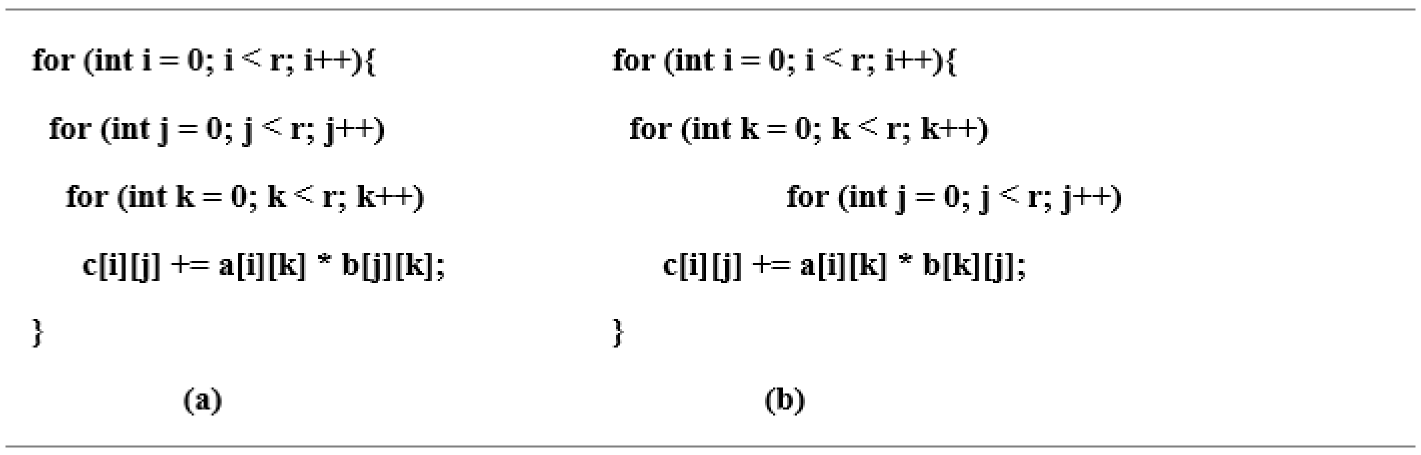


**Figure 1**

Mapping matrix-matrix multiplication for four different techniques such as basic, transposed, loop interchange, and Blocking.



```
for (int i = 0; i < r; i++){
  for (int j = 0; j < r; j++)
    for (int k = 0; k < r; k++)
      c[i][j] += a[i][k] * b[j][k];
}
            (a)
```

```
for (int i = 0; i < r; i++){
  for (int k = 0; k < r; k++)
            for (int j = 0; j < r; j++)
      c[i][j] += a[i][k] * b[k][j];
}
            (b)
```

**Figure 2**

Here “ *i* ” stand for row index and “ *j* ” stand for column index, and “ *k* ” is used as a counter index. (a) Demonstrate basic MMM where matrix B elements are selected in column and (b) exhibit the loop interchange.

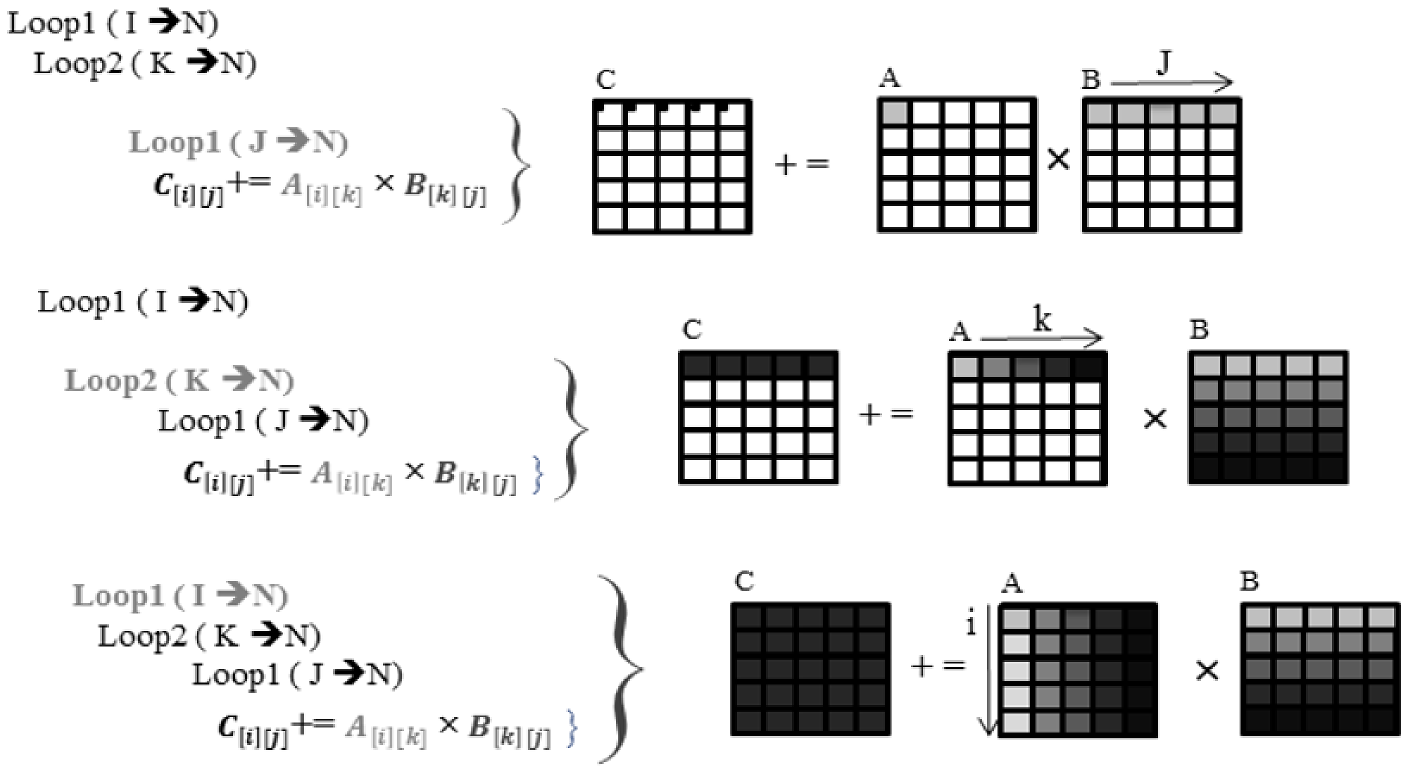

Loop1 ( I →N)
Loop2 ( K →N)
Loop1 ( J →N)
$C_{[i][j]} += A_{[i][k]} \times B_{[k][j]}$
C
A
B
J
+ =
×
Loop1 ( I →N)
Loop2 ( K →N)
Loop1 ( J →N)
$C_{[i][j]} += A_{[i][k]} \times B_{[k][j]}$
k
Loop1 ( I →N)
Loop2 ( K →N)
Loop1 ( J →N)
$C_{[i][j]} += A_{[i][k]} \times B_{[k][j]}$
i

Figure 3

In loop interchange mapping, each element of matrix A will be used only once and all available multiplications will be performed while the results are added to elements of matrix C. In each row of matrix C, elements are calculated little by little and the results remain in cache till one row of matrix C completely be calculated, finally first loop iteration for "*i*" selects the second row of matrix A and starts computing the same row of Matrix C, this will continue as long as all elements of matrix C are computed.

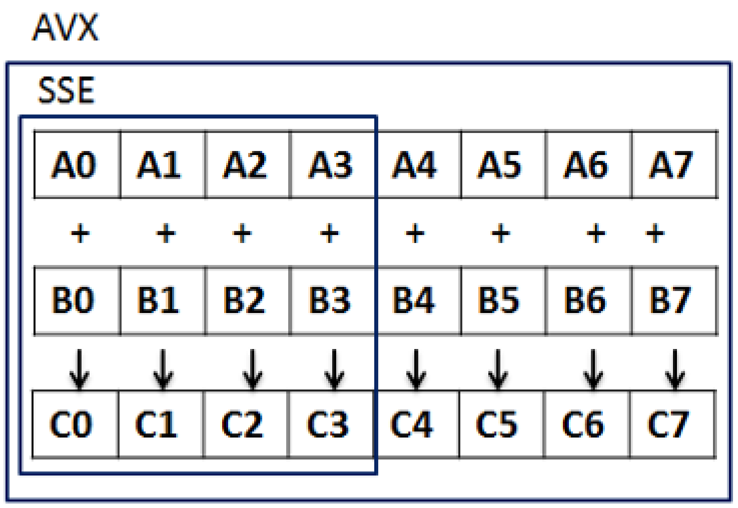


Figure 4

The total number of floating-point operations in AVX and SSE for vector addition, where A and B elements are single floating-point numbers.

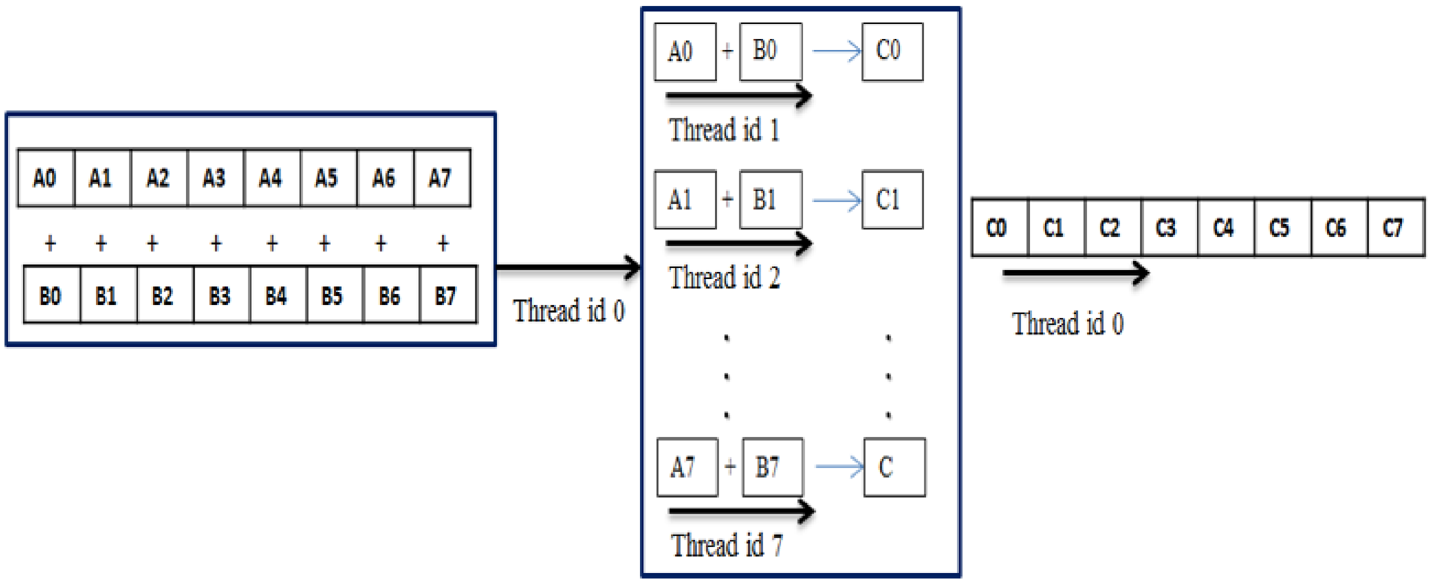
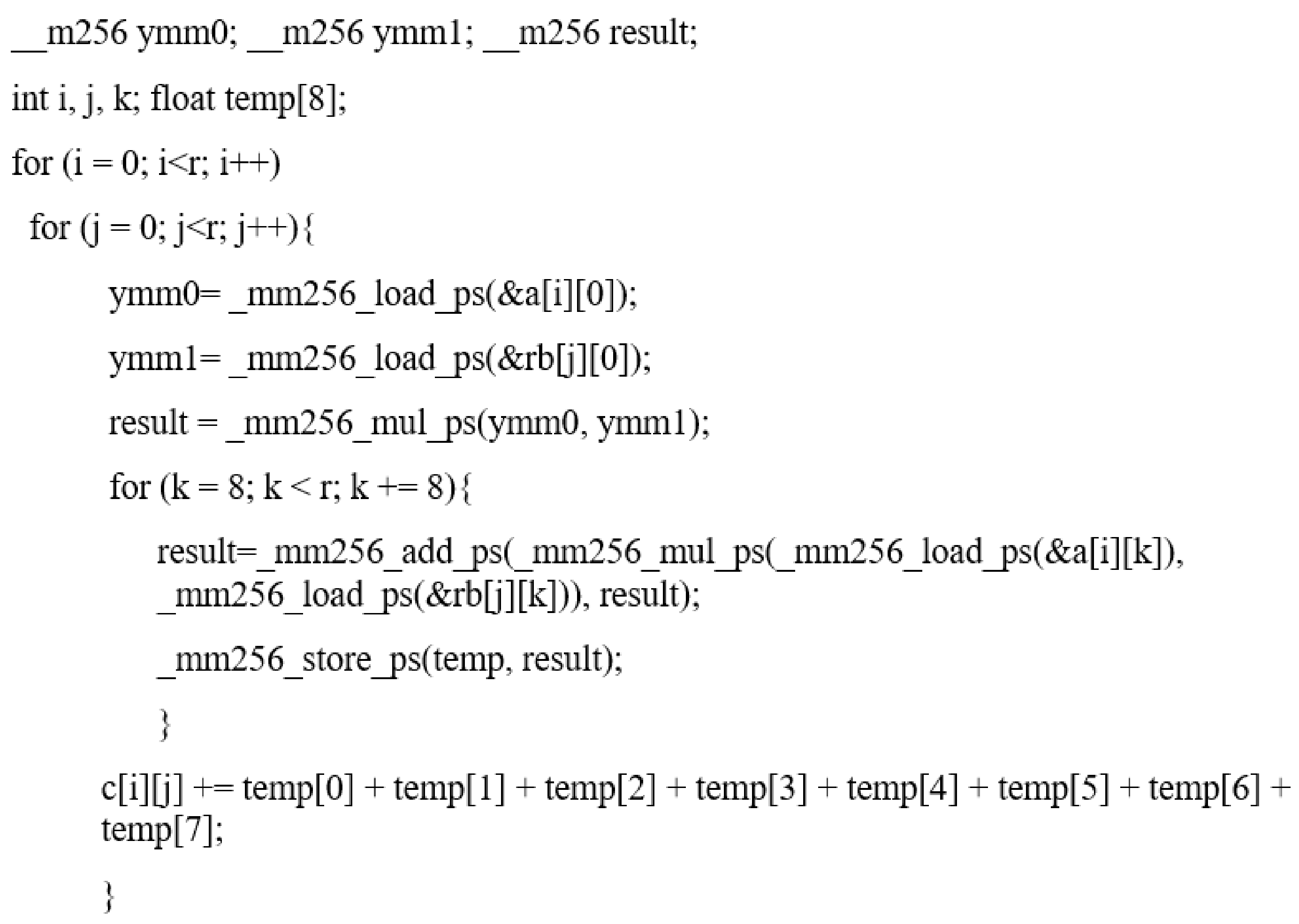


**Figure 5**

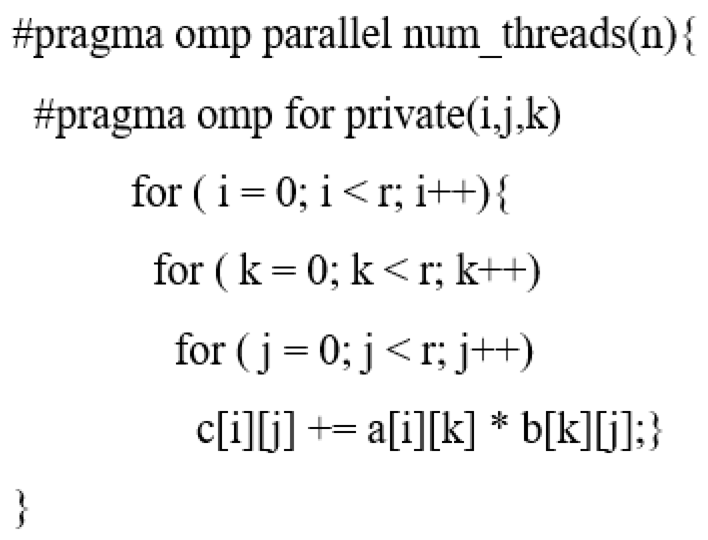
Fork and join method for two vector addition.



```
__m256 ymm0; __m256 ymm1; __m256 result;
int i, j, k; float temp[8];
for (i = 0; i<r; i++)
  for (j = 0; j<r; j++){
          ymm0= _mm256_load_ps(&a[i][0]);
          ymm1= _mm256_load_ps(&rb[j][0]);
          result = _mm256_mul_ps(ymm0, ymm1);
          for (k = 8; k < r; k += 8){
               result=_mm256_add_ps(_mm256_mul_ps(_mm256_load_ps(&a[i][k]),
               _mm256_load_ps(&rb[j][k])), result);
               _mm256_store_ps(temp, result);
               }
         c[i][j] += temp[0] + temp[1] + temp[2] + temp[3] + temp[4] + temp[5] + temp[6] +
         temp[7];
         }
```

**Figure 6**

AVX intrinsic function code for matrix-matrix multiplication.



```
#pragma omp parallel num_threads(n){
  #pragma omp for private(i,j,k)
         for ( i = 0; i < r; i++){
           for ( k = 0; k < r; k++)
             for ( j = 0; j < r; j++)
               c[i][j] += a[i][k] * b[k][j];}
}
```

**Figure 7**

OpenMP code for Matrix-Matrix Multiplication.

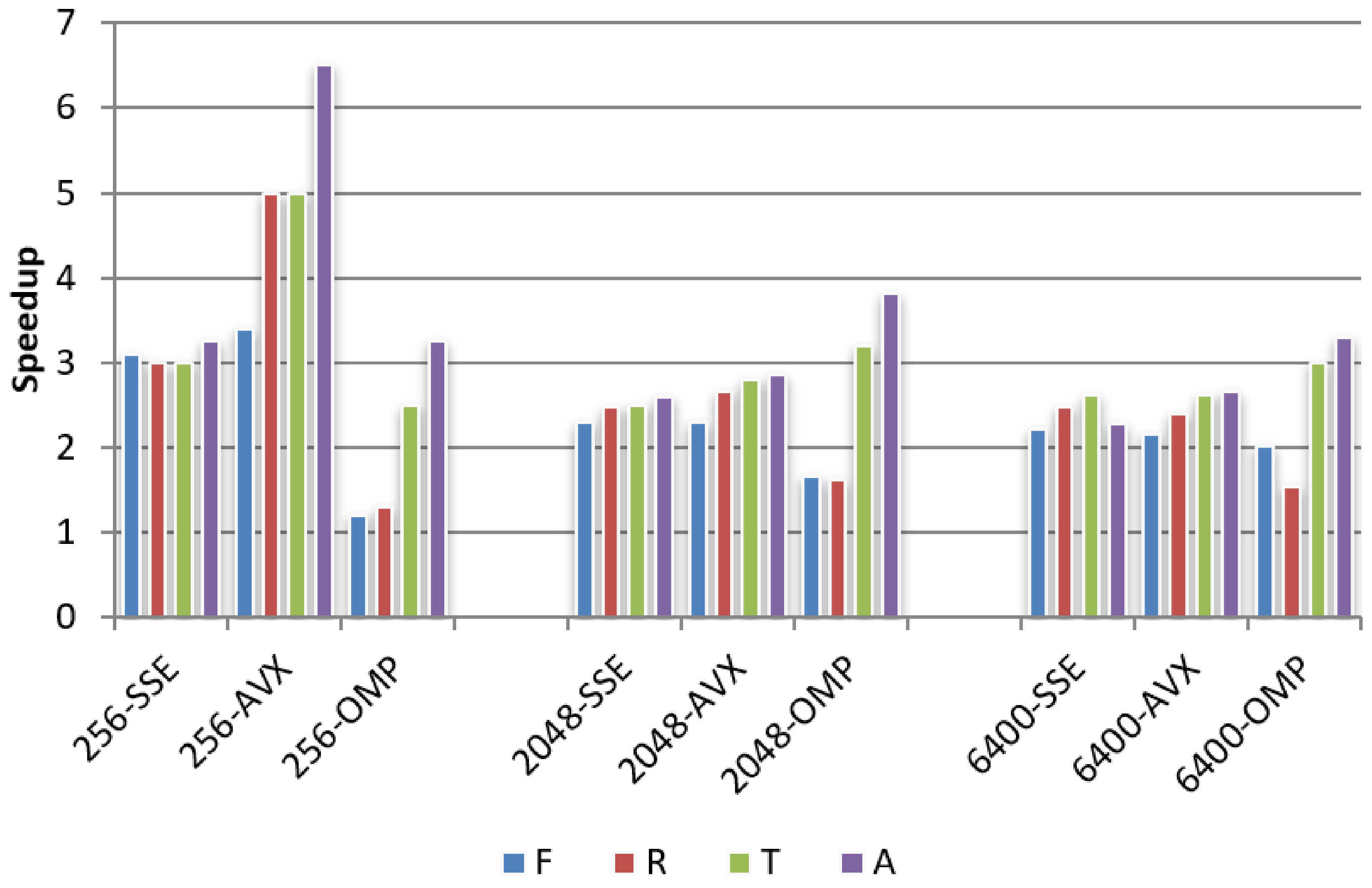


**Figure 8**

Speedup of SSE, AVX, and OpenMP over the best sequential implementation, note that each system speedup is compared to its ownbest sequential result and less speedup does not mean higher execution time in the particular system. For instance, the AVX execution time for a matrix size of 6400 in R and A systems is 109 and 68 seconds.

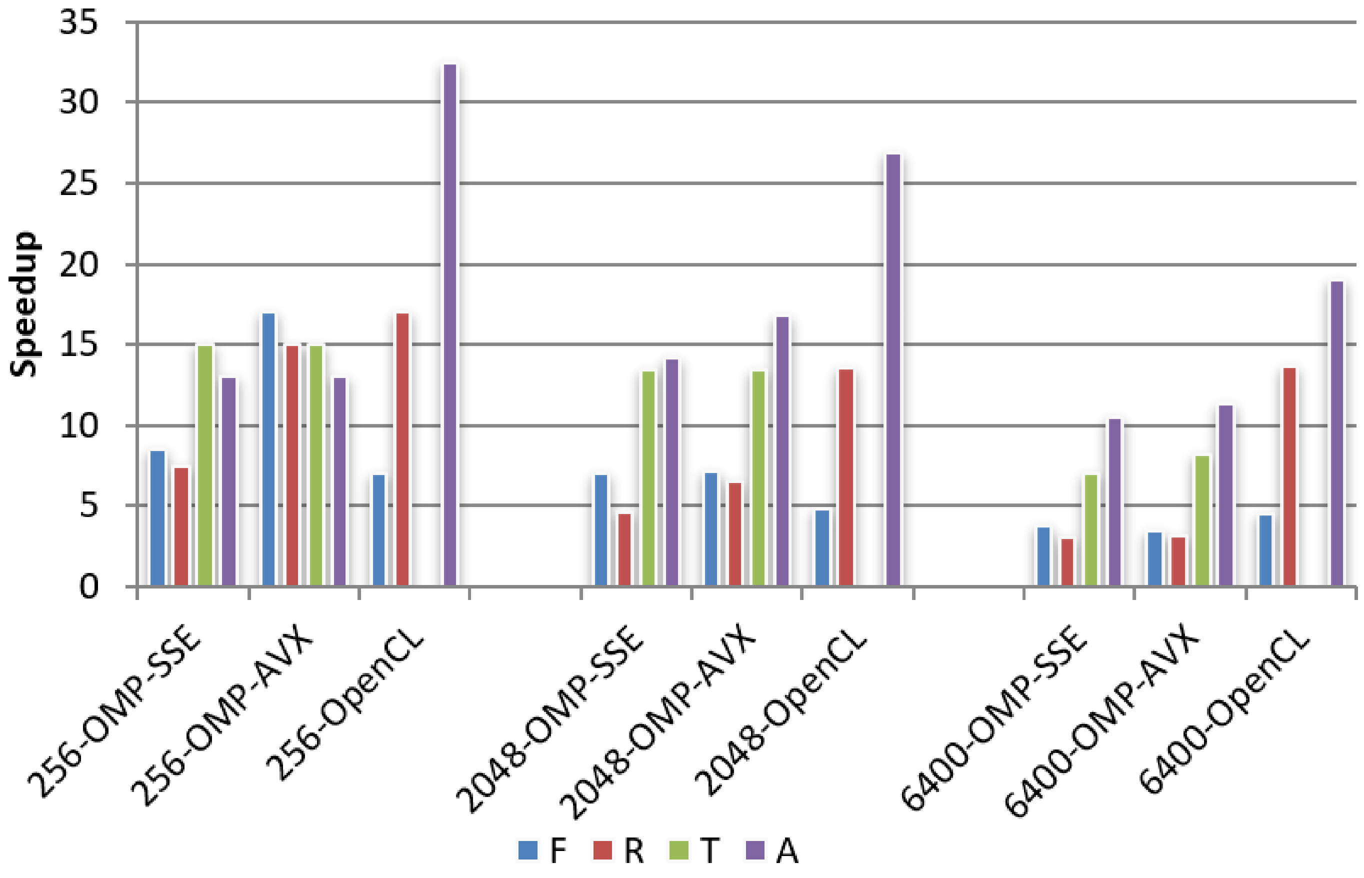


**Figure 9**

Performance comparison of hybrid OpenMP-SIMD and OpenCL for four selected platforms, the hybrid, and OpenCL speedup compare to the best sequential result of the same system. Although the best speedup in hybrid OpenMP-AVX is in F, the lowest achieved time is for A, and that's because the sequential time for F was larger than A.

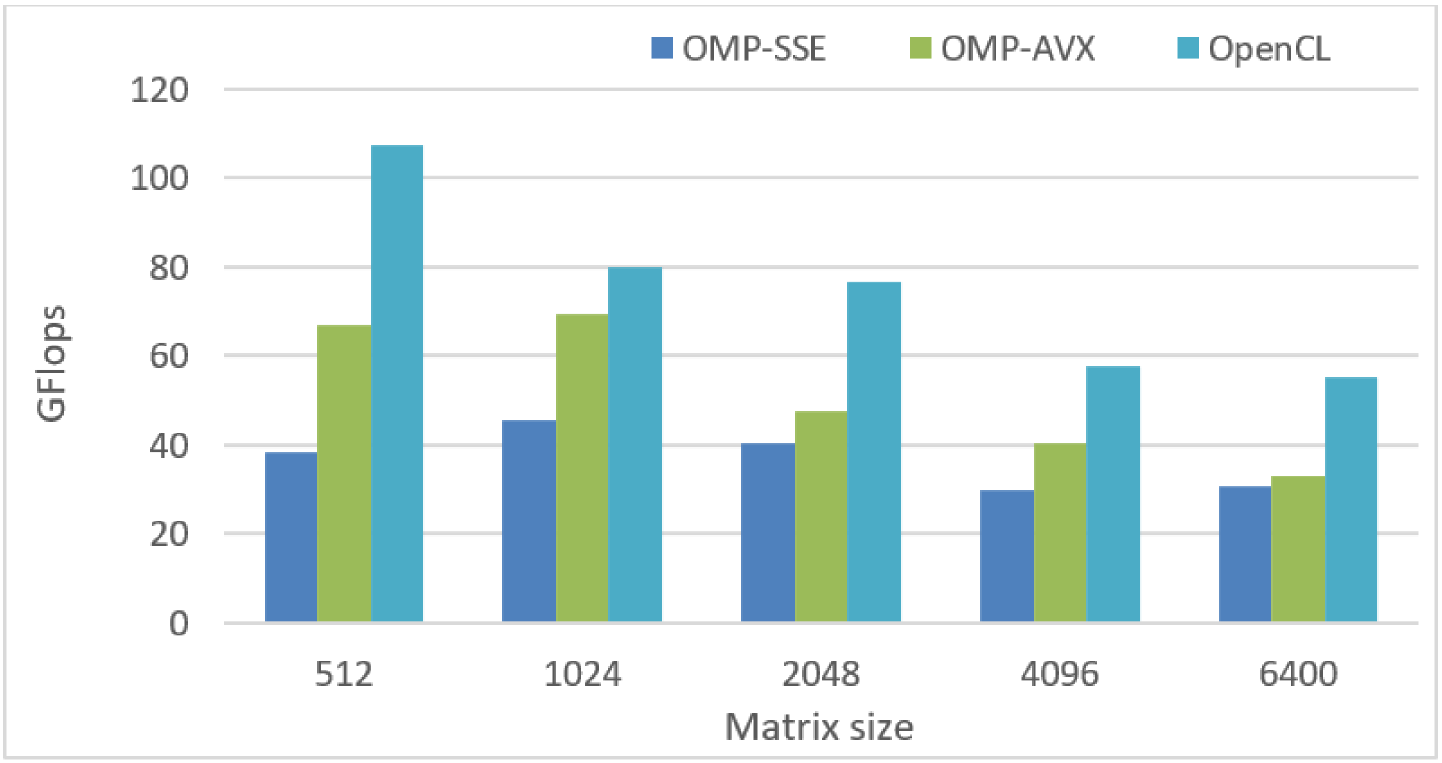


**Figure 10**

Performance comparison of hybrid OpenMP-SIMD and OpenCL in system A.

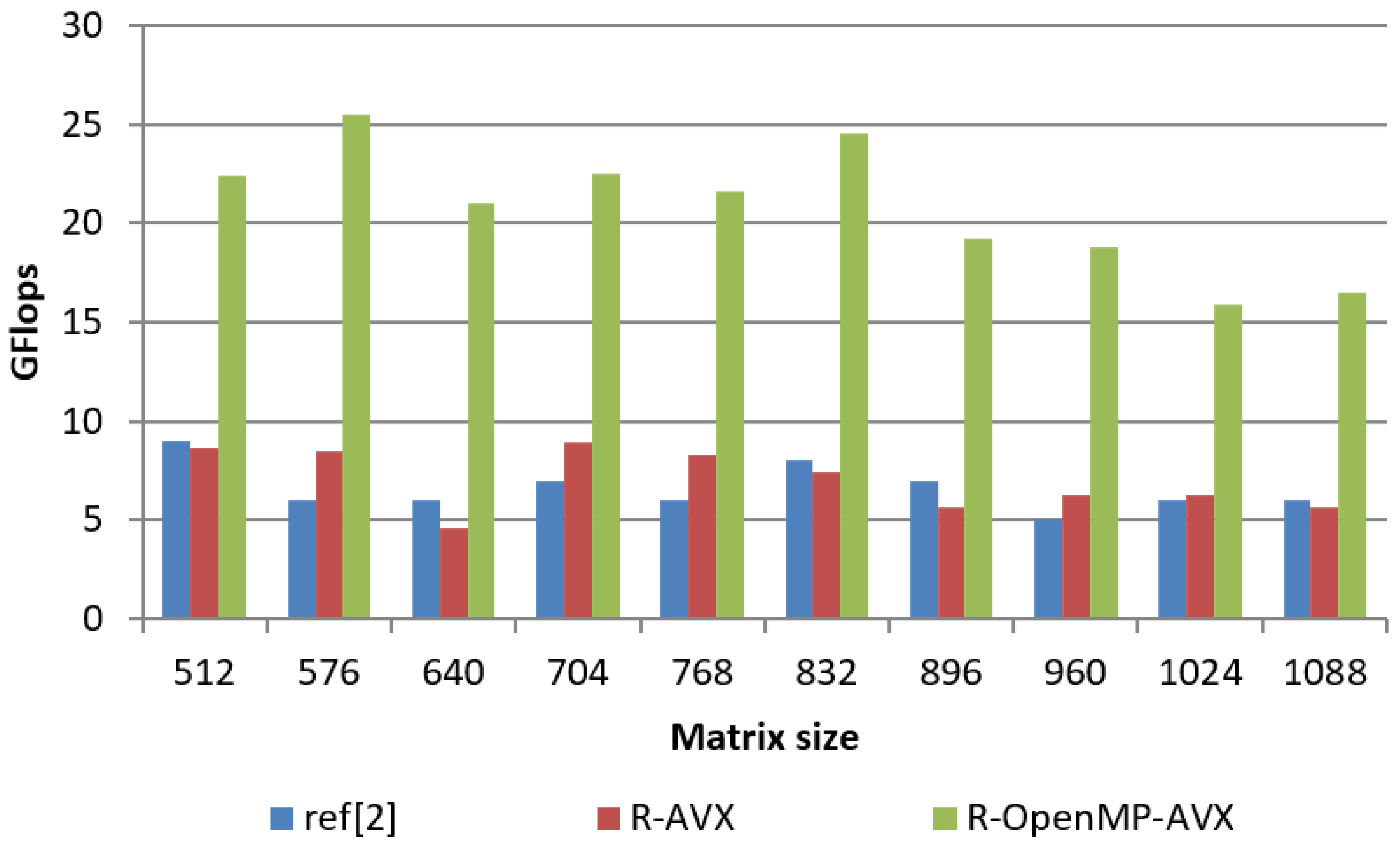


**Figure 11**

A fair comparison to the most recent work on AVX intrinsic function code, GFlops for [7] are approximately selected and compared to the R-system result because of similarity in both system CPUs.