Journal of Engineering and Applied Research (JEAR)

National University of Skill

Winter & Spring 2026, Vol. 3, No. 1, p. 39- 68

Journal Homepage: https://jear.tvu.ac.ir

:10.48301/jear.2024.421627.1005

Efficient Implementation of Matrix-Matrix Multiplication using SIMD and OpenMP Models on CPU Platforms

Alireza Akoushideh[1*], Asadollah Shahbahrami[2]

[1]Department of Electrical and Computer Engineering, Technical and Vocational University, Tehran, Iran.

[2]Department of Computer Engineering, Guilan University, Rasht, Iran.

ARTICLE INFO

Article Type:
Original Research

Received: 2023.10.21
Revised: 2024.10.20
Accepted: 2024.11.12

Keyword:
Matrix-Matrix Multiplication
Parallel programming, SIMD
OpenMP
OpenCL

***Corresponding Author:**
Alireza Akoushideh
Email: *akushide@tvu.ac.ir*

ABSTRACT

Modern CPUs and GPUs come with multiple processing units, enabling the parallel execution of tasks, which enhances overall system performance. Various parallel programming models have been developed to take full advantage of this parallelism. For computation-heavy applications, such as matrix-matrix multiplication—a key operation in linear algebra frequently applied in scientific simulations and multimedia processing—achieving efficient implementation is essential to fully utilize hardware resources. This paper explores optimized implementations of matrix-matrix multiplication using several parallel programming techniques: SIMD, OpenMP, a Hybrid OpenMP-SIMD model, and OpenCL. Our experimental results highlight the effectiveness of these approaches, with observed speedups of up to 6.5x using SIMD, 3.2x with OpenMP, 16.7x via hybrid OpenMP-SIMD, and a notable 32x speedup with OpenCL, compared to a highly optimized serial baseline. These results demonstrate significant performance improvements when leveraging parallelism for matrix-matrix multiplication.

E-ISSN: 3060-6357

1)Introduction

Matrix-Matrix Multiplication (MMM) plays a pivotal role in linear algebra and numerical computing. It involves multiplying two matrices to generate a third matrix, where the dimensions of the output matrix are defined by the input matrices' sizes. MMM is a fundamental operation in numerous fields, such as computer graphics [1-3], scientific computing [4; 5], and machine learning [6; 7], and is frequently used in algorithms that manipulate large datasets or solve systems of linear equations. Given its computational intensity, developing optimized algorithms and implementations for matrix-matrix multiplication is essential for improving the performance of numerical computations. Common approaches to implementing this operation include basic multiplication, matrix transposition, loop reordering, blocking techniques, and deep learning methodologies [6]. However, large-scale matrix multiplication remains a highly computationally demanding task [8-12]. For instance, multiplying two 6400×6400 matrices using an optimized algorithm can still take approximately 180 seconds on an Intel Core i7 processor. The pursuit of faster MMM algorithms continues to be a major area of research [8-14]. One such effort is the Basic Linear Algebra Subprograms (BLAS) [11; 15], which standardizes a blocking method for matrix multiplication and is widely utilized in libraries like ATLAS and NVIDIA cuBLAS [8].

To improve multimedia application performance, processor manufacturers have introduced Single Instruction Multiple Data (SIMD) extensions. SIMD was designed to exploit Data Level Parallelism (DLP) in parallel computing. For instance, Intel has launched several SIMD extensions, such as MMX, SSE, and AVX/AVX2, since 1996. Over time, these SIMD registers have evolved from 64-bit to 128-bit, and most recently, 256-bit with AVX and AVX2. Additionally, SIMD programming utilizes techniques such as Inline Assembly, Intrinsic Functions, and Auto-vectorization [16].

Leveraging parallel computing techniques like SIMD, OpenMP, and OpenCL for matrix-matrix multiplication offers significant benefits, including enhanced performance, more efficient use of multicore processors, and the ability to exploit data-level parallelism [17; 18]. This can lead to faster execution of compute-intensive tasks, better scalability, real-time performance improvements, and broader applications in areas such as scientific simulations, machine learning, and multimedia processing [19]. This study focuses on identifying the most efficient parallel programming approach for implementing MMM on CPU platforms, with a specific interest in the performance impact of different models. The paper examines the use of SIMD, OpenMP, Hybrid OpenMP-SIMD, and OpenCL in the implementation of MMM on various CPUs. The main research question seeks to determine which approach yields the greatest speedup for MMM and which parallel programming model provides the most significant performance gains. The study compares widely-used MMM algorithms across SIMD, OpenMP, OpenCL, and hybrid OpenMP-SIMD implementations. Additionally, it references optimized libraries like Intel MKL [20] and GotoBLAS [15], which achieve superior performance due to their low-level assembly code and knowledge of processor architecture. However, this research focuses on comparing parallel programming models for MMM algorithms rather than optimizing for specific hardware platforms.

Our research explores the effectiveness of different parallel programming paradigms, including SIMD, OpenMP, Hybrid OpenMP-SIMD, and OpenCL, in improving MMM performance on CPU architectures [21-23]. By taking advantage of the parallel execution capabilities of modern multicore processors, these programming models offer promising pathways for improving computational efficiency in matrix-matrix multiplication. This investigation seeks to determine which programming model offers the best approach for accelerating MMM operations across various matrix sizes on CPU platforms. Through detailed experiments and analysis, we aim to assess the performance advantages of each parallel model and provide insights into their optimization for numerical computation tasks on CPUs. Furthermore, this research highlights the importance of carefully evaluating and comparing these parallel programming approaches to determine their suitability for MMM implementations. By identifying the strengths and weaknesses of each model, we hope to contribute to the ongoing discourse on optimizing numerical computations in computational mathematics and related fields.

In summary, this paper addresses the goal of improving MMM performance by exploring and comparing various parallel programming models for CPU platforms. By emphasizing the role of parallelization techniques in numerical computing and providing empirical results, we aim to push the boundaries of computational mathematics and promote more efficient computing practices. The experimental results showcase significant speedups across different matrix sizes, with SIMD achieving up to 6.5x, OpenMP 3.2x, Hybrid OpenMP-SIMD 16.7x, and OpenCL delivering an impressive 32x speedup in MMM tasks.

This paper is structured as follows: Section II covers the relevant background information. Section III reviews the existing literature on matrix-matrix multiplication. Section IV presents our proposed approaches for implementing MMM on different CPU platforms. Section V provides details on the benchmarks, test machine, and experimental results. Finally, Section VI concludes the paper with key findings and conclusions.

2) Background Information

Matrix Multiplication

Matrix-matrix multiplication (MMM) refers to the operation where rows of matrix A are multiplied by columns of matrix B to generate matrix C [24]. There are several strategies for implementing this operation, including basic mapping, matrix transposition, loop interchange, and blocking, as illustrated in Fig 1. The performance of a basic MMM implementation can be significantly affected by the cost of fetching columns from matrix B. In contrast, the transposed method incurs only the additional cost of transposing matrix B. Both loop interchange and blocking methods offer superior data reuse compared to the basic and transposed approaches, resulting in improved performance.

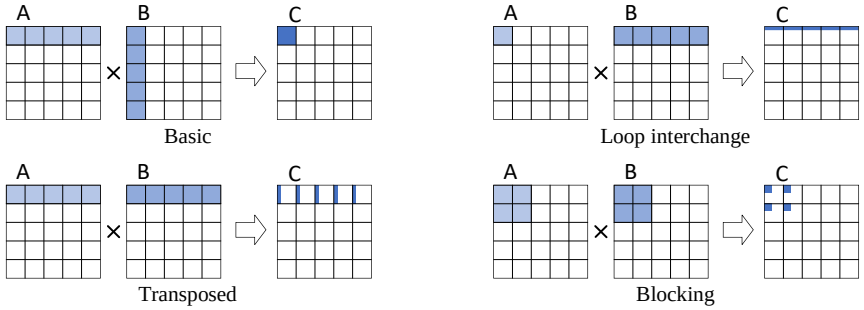

Figure 1. Different approaches to matrix-matrix multiplication: basic, transposed, loop interchange, and blocking

Due to its performance advantages, the loop interchange technique has been selected for our basic CPU implementation. In a straightforward MMM, where each element of a row from matrix A is multiplied with corresponding elements of a column from matrix B, the process of loading matrix B columns becomes a performance bottleneck. In contrast, using loop interchange allows individual elements from matrix A to be multiplied into columns of matrix B efficiently. The C code for both basic MMM and the loop interchange method is shown in Fig 2.

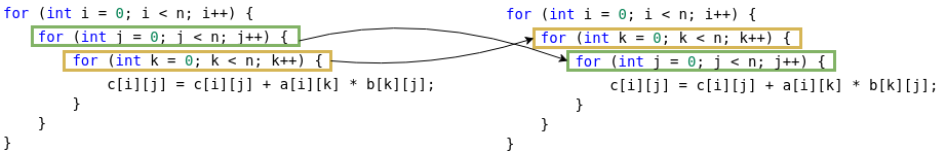

Figure 2. Example of loop interchange implementation [25]

In the loop interchange technique, every element of matrix A is processed once and multiplied with corresponding rows of matrix B, with the result stored in matrix C. This approach reduces cache misses and enhances data reuse. Each multiplication is added step-by-step to build the elements of matrix C. As shown in Fig 3, each element of matrix A is used only once, and all necessary multiplications are performed, with results stored in matrix C. The components of matrix C are computed incrementally, and intermediate results remain in the cache until a full row of matrix C is completed. The process repeats for each row of matrix A, continuing until all elements of matrix C are calculated according to Equation (1). This loop interchange implementation runs up to two times faster than transposed MMM and up to eight times faster than the basic MMM for our test matrix sizes.

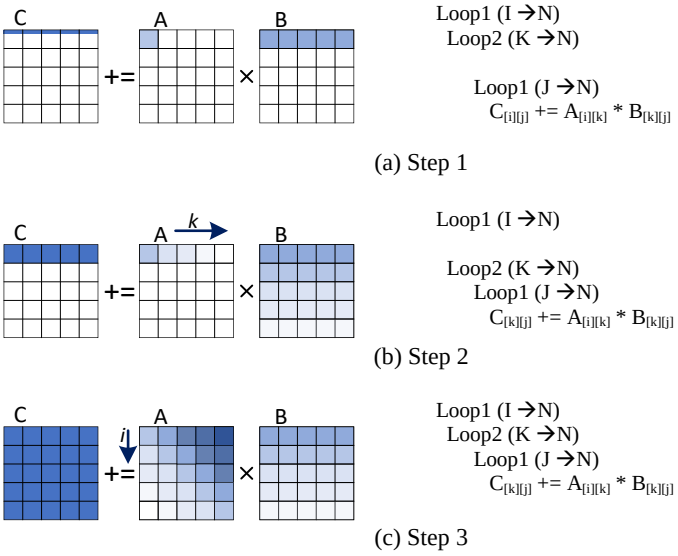

Figure 3. Step-by-step in-loop interchange mapping

$$C = \alpha AB + \beta C \quad (1)$$

Parallel Programming Approaches

Parallel programming models are frameworks and techniques that enable developers to design programs capable of utilizing multiple processors or computing resources simultaneously, thus improving computational efficiency. These models leverage modern hardware, such as multi-core processors and accelerators, to

enhance both performance and scalability. The suitability of each parallel programming model depends on factors like the nature of the computational task, the available hardware, and the programming language or tool that best supports parallelization.

Several parallel programming strategies exist, including instruction-level parallelism (ILP), data-level parallelism (DLP) with SIMD, thread-level parallelism (TLP) using OpenMP, and OpenCL for auto-vectorization. These models can be implemented across single-core and multi-core CPUs, providing flexibility for various hardware configurations [8-10; 13; 16; 26; 27]. The next subsection provides a more detailed exploration of SIMD.

SIMD

Single Instruction, Multiple Data (SIMD) is a parallel computing architecture classified under Flynn's taxonomy, enabling simultaneous processing of multiple data points using a single instruction. This architecture is supported by modern processors from Intel and AMD. The earliest SIMD instruction set for the x86 architecture was Intel's MMX, introduced in 1996, which was also referred to as the Multimedia Multiple Math extension. The MMX instruction set provided eight 64-bit registers (MM0 to MM7) that were primarily used for integer operations. SIMD has since become widely employed to boost the performance of matrix-matrix multiplication (MMM) [8-10; 12]. SIMD code can be written using three methods: inline assembly, intrinsic functions, and vector classes. While inline assembly grants full control over registers, it's not always ideal, particularly for newer compilers. As a result, in the SIMD version of MMM, intrinsic functions written in standard C have been utilized.

Following MMX, Intel introduced SSE (Streaming SIMD Extensions), which features eight 128-bit registers (XMM0–XMM7) for x86 systems and sixteen 128-bit registers (XMM0–XMM15) for x64 systems. SSE focuses more on single-precision floating-point operations than integer operations [16]. SSE expanded on the capabilities of MMX by supporting double-precision floating-point calculations in addition to wider integer operations. SSE has been used for efficient MMM implementations, particularly on hardware optimized for specific matrix sizes, minimizing DDR access and selecting parameters like tiling size based on cache configurations[8-10]. We also implemented SIMD MMM for Intel CPUs, combining it with OpenMP to achieve notable improvements on specific platforms. Further advancements in SIMD came with the introduction of the AVX (Advanced Vector Extensions) instruction set. AVX registers are twice as large as SSE registers, allowing the processing of eight single-precision or four double-precision floating-point numbers in a single instruction, compared to SSE, which handles half that number. AVX has been applied to small matrix sizes in experiments comparing Intel and Visual Studio compilers [9]. Results indicated that intrinsic function AVX code outperformed inline assembly for matrix sizes up to 1120. An illustration is provided in Fig 4, showing the total floating-point operations for vector addition using AVX and SSE.

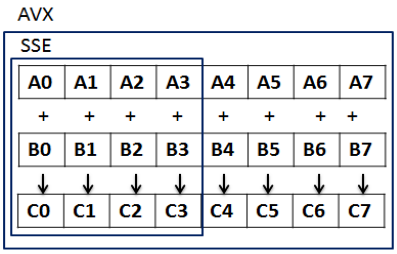

Figure 4. Total floating-point operations involved in vector addition using AVX and SSE, with A and B representing individual single-precision floating-point elements.

Both SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) are SIMD instruction sets developed by Intel for x86 and x86-64 processors. They allow multiple data elements to be processed simultaneously, improving performance for compute-heavy tasks. SSE is widely supported across a range of processors and is backward-compatible with older hardware, whereas AVX offers enhanced performance for applications that can take advantage of its wider SIMD width. A comparison of the SSE and AVX is shown in Table 1.

Table 1. A comparison of SSE and AVX

	SSE	AVX
Width	SSE instruction sets, starting from SSE1 to SSE4.2, provide SIMD operations with a width of 128 bits. SSE1, SSE2, SSE3, and SSSE3 provide 128-bit SIMD registers, while SSE4.1 and SSE4.2 introduced additional instructions.	AVX instruction sets, starting from AVX1 to AVX2 and AVX-512, provide SIMD operations with a width of 256 bits (AVX1 and AVX2) and 512 bits (AVX-512).
Number of Operands	SSE instructions operate on two operands at a time for 128-bit registers.	AVX instructions operate on three operands simultaneously for 256-bit and 512-bit registers.
Instruction Throughput	SSE is still valuable for older processors and systems not supporting AVX or AVX-512.	AVX can achieve higher throughput than SSE due to its wider SIMD width, enabling more data to be processed in a single instruction.
Software Compatibility	SSE is supported on a wide range of x86 and x86-64 processors, making it more compatible with older hardware and software.	AVX is supported on newer processors, starting from the Intel Sandy Bridge microarchitecture (AVX1) and Intel Haswell (AVX2) for 256-bit registers and on more recent processors for AVX-512.
Performance Impact	SSE is still relevant and valuable for applications that do not require the wider SIMD width provided by AVX.	AVX can provide significant performance improvements for applications that can efficiently utilize its wider SIMD width and additional instructions. It is particularly beneficial for tasks involving large-scale data parallelism.
Energy Efficiency	SSE's narrower SIMD width can be more energy-efficient in specific scenarios where processing smaller data sets and lower precision are sufficient.	AVX's wider SIMD width and additional operations consume more power, which may not be ideal for all applications.

OpenMP

OpenMP, or Open Multiprocessing, is an Application Programming Interface (API) designed for shared-memory parallel programming in C, C++, and Fortran [13; 27; 28]. It enables developers to parallelize their code by utilizing multicore

processors to improve performance and scalability. OpenMP works by introducing directives into the code to indicate sections that can be executed concurrently. The compiler uses these directives to generate multithreaded code that runs in parallel across multiple CPU cores [30]. The OpenMP execution model uses a master thread that forks a predetermined number of slave threads, distributing tasks among them, as shown in Fig 5. The master thread is referred to as thread ID 0.

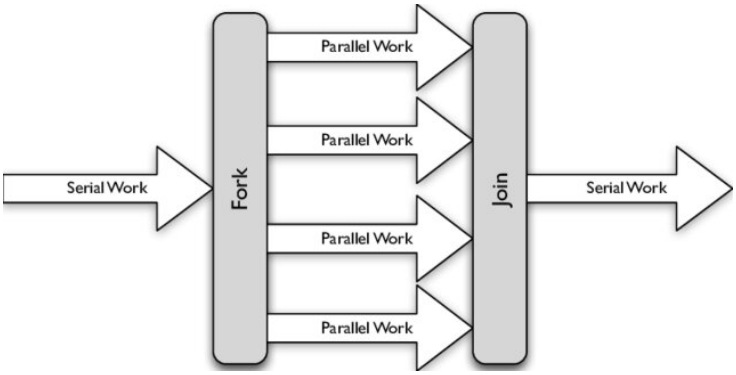

Figure 5. The fork-join model used for thread-level parallelism in OpenMP [29]

OpenMP provides a general programming framework, allowing sequential algorithms to be parallelized without major restructuring. It supports Single Process Multiple Data (SPMD) and Multiple Process Multiple Data (MPMD) paradigms [30], along with SIMD vectorization. OpenMP 4.0 introduced the "pragma omp simd" directive, making it easier for developers to vectorize code [31]. However, in this work, SSE and AVX in the SIMD MMM version are implemented using intrinsic functions, without the use of "pragma omp simd". The OpenMP 5.1 Specification [31], published by the OpenMP Architecture Review Board, defines the latest updates for parallel programming, improving performance and ease of use across a variety of architectures.

OpenCL

Open Computing Language (OpenCL), introduced by Apple and co-developed with companies such as AMD, Intel, and NVIDIA, is a framework designed for cross-platform parallel programming across CPUs and GPUs [26; 32]. OpenCL offers a unified programming environment that allows developers to write code portable across different architectures. It includes a C-like language for writing kernels (which execute in parallel) and APIs for managing hardware devices [33]. Open Computing Language (OpenCL) was developed collaboratively by Apple, AMD/ATI, Intel, and NVIDIA to support both CPU and GPU platforms. OpenCL consists of a host, typically the main CPU, and multiple devices, which may include both CPUs and GPUs. It incorporates an auto-vectorization methodology, enabling developers to write code that can run across various architectures. By utilizing OpenCL APIs, developers can interact with the platform, allowing for both low-level control and higher-level abstractions for ease of use. While many studies have focused on enhancing matrix-matrix multiplication (MMM) performance using SIMD, OpenMP, and OpenCL on CPU and GPU platforms, there is still a lack of

comprehensive evaluations comparing these parallel programming models, especially hybrid approaches [1; 4; 5; 8-13].

OpenCL shares similarities with CUDA in terms of parallelism management and data handling but relies more heavily on APIs, ensuring it can execute on all compatible processors without modifications [32; 33]. Various libraries, such as MAGMA, clAMDBLAS, and BOLT C++, leverage OpenCL for acceleration, and there are Java bindings available as well. Consequently, OpenCL is a cross-platform parallel computing API that defines a C-like language for writing programs, including a C99-based language for kernels and an API for controlling OpenCL-supported devices. Its portability across multiple processors, including CPUs, GPUs, and FPGAs, enables programmers to abstract hardware specifications, promoting a versatile programming environment for high-performance computing[34]. The Khronos OpenCL consortium provides the specification that outlines the programming architecture and the APIs necessary for application developers. OpenCL employs a data-parallel execution model characterized by fine granularity based on the target platform. It consists of two main components: kernels that execute in parallel and a host that is written in standard C/C++. Kernels are initiated by work-items, which are organized into work-groups. OpenCL's mapping strategy makes it particularly effective for implementing Basic Linear Algebra Subprograms (BLAS). Level 1 BLAS performs scalar, vector, and vector-vector operations; Level 2 BLAS handles matrix-vector operations; and Level 3 BLAS focuses on matrix-matrix operations, as represented in (1), where α and β are arbitrary scalars [11; 35]. Table 2 offers a comparative analysis of SIMD, OpenMP, and OpenCL regarding their scopes, purposes, and use cases.

Table 2. Comparison of the SIMD, OpenMP, and OpenCL in Scopes, Purposes, and Use Cases.

	SIMD	OpenMP	OpenCL
Purpose	SIMD is a parallel computing model that operates on multiple data elements using a single instruction. It is best suited for data-level parallelism, where the same operation is performed on multiple data elements simultaneously.	OpenMP is an API for shared memory parallel programming. It enables developers to parallelize loops and sections of code to take advantage of multiple CPU cores on a single machine.	OpenCL is a standard for parallel programming across heterogeneous platforms, including CPUs, GPUs, FPGAs, and other accelerators. It allows developers to write code that can run on various devices, utilizing their specific capabilities.
Scope	SIMD is typically implemented in hardware through vector processing units in modern CPUs and GPUs. It allows efficient execution of operations like element-wise vector additions, multiplications, and other arithmetic operations.	OpenMP focuses on multicore processors and shared memory systems. It simplifies writing parallel code by providing compiler directives and runtime library routines.	OpenCL supports shared memory parallelism (on CPUs) and data parallelism (on GPUs and accelerators). It provides a more flexible programming model than SIMD and OpenMP by targeting heterogeneous systems.
Use Cases	SIMD is commonly used in multimedia processing, scientific simulations, image and signal processing, and	OpenMP is widely used in general-purpose computing, numerical simulations, parallelizing iterative	OpenCL is commonly used in high-performance computing, scientific simulations, computer

	other applications where operations on large arrays or vectors can be parallelized.	algorithms, and tasks with fine-grained parallelism.	vision, machine learning, and any application that can benefit from parallel execution on diverse hardware.
Comparison Summary:	SIMD is hardware-level parallelism designed for vector processing units in CPUs and GPUs, best suited for data-level parallelism on single-core or multicore CPUs.	OpenMP is a shared memory parallel model for multicore CPUs, useful for parallelizing loops and sections of code on shared memory systems.	OpenCL is a heterogeneous parallel model that targets a wide range of devices (CPUs, GPUs, FPGAs) and supports shared memory and data parallelism, making it more versatile for heterogeneous systems.
Advantage	Faster execution for smaller matrices: When the matrices fit well within the processor registers, SIMD instructions can significantly accelerate computations compared to sequential execution. Simplicity: Implementing SIMD can be relatively straightforward, especially with modern compiler support for intrinsic functions.	Ease of use: OpenMP provides a relatively high-level interface for thread-level parallelism, making it accessible to programmers with moderate experience in parallel programming. Applicability: OpenMP is suitable for problems that can be decomposed into independent tasks with minimal communication overhead between threads. This includes matrix multiplication algorithms that can be partitioned into sub-matrices for individual threads to process.	High performance: OpenCL can leverage the powerful processing capabilities of GPUs, potentially offering significantly higher performance compared to CPU-based implementations, particularly for large-scale computations.
Applications	SIMD is well-suited for various data-parallel tasks where the same operation needs to be applied to independent data elements, including image processing, signal processing, and scientific computing.	OpenMP finds applications in various scientific computing, engineering simulations, and data analysis tasks that benefit from thread-level parallelism.	OpenCL is widely used in high-performance computing, image and video processing, machine learning, and other applications that demand intensive parallel computations.

Note that the choice between SIMD, OpenMP, and OpenCL depends on the specific application, hardware architecture, and parallelism requirements. In some cases, these approaches can be used together to optimize performance and achieve better parallelism.

The choice of a parallel programming model and its implementation can be influenced by the target hardware architecture. This research primarily focused on implementing these methods on x86-based CPU architectures with varying numbers of cores. While the concepts and principles apply generally, specific optimizations and considerations might be necessary for different hardware platforms, such as GPUs.

3) Literature Review

Jeong et al. [16] focus on evaluating the performance of SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) instruction sets. These

instruction sets enable parallel processing of data using SIMD (Single Instruction, Multiple Data) operations in modern computer processors. Their experiments assess the efficiency and speedup achieved by employing SSE and AVX instructions across various computational tasks, likely utilizing benchmark programs that rely heavily on vectorized computations to leverage the parallel processing capabilities of these instruction sets. Kelefouras et al. [8] employ parallel programming techniques, including SIMD for CPUs and CUDA (Compute Unified Device Architecture) for GPUs. Their exploration of optimizations, such as loop unrolling, data reordering, and cache management, significantly enhances MMM performance. The results demonstrate remarkable speedups on both CPU and GPU platforms, achieving up to 8x speedup on CPUs and up to 25x speedup on GPUs compared to conventional implementations. Similarly, Hassan et al. [9] investigate strategies to optimize MMM performance on Intel processors with AVX support, employing techniques like loop unrolling and data alignment. Their findings illustrate the positive impact of AVX on speedup compared to traditional matrix multiplication techniques, providing valuable insights into harnessing AVX instructions for improved performance on modern Intel-based platforms.

In a related study, Kelefouras et al. [10] propose a SIMD parallelism methodology to achieve data-level parallelism and optimize MMM computations, demonstrating significant speedup compared to conventional techniques on both single-core and multicore architectures. Gautier et al. [11] consider topology-aware heuristics on the Level-3 Basic Linear Algebra Subprograms (BLAS) library for multi-GPU platforms, evaluating their impact on the performance of MMM and other key BLAS routines across various multi-GPU systems. Their experimental results affirm the effectiveness of these heuristics in optimizing MMM and related operations. Matsumoto et al. [35] explore various strategies, including data layout optimizations, loop unrolling, and work-group size adjustments, to maximize the throughput of matrix multiplication on different hardware architectures. Their experimental evaluations demonstrate significant performance improvements across diverse computing devices. He et al. [27] investigate decomposition strategies and parallelization techniques to efficiently distribute the workload across multiple processors in a distributed-memory cluster, achieving significant speedup improvements for MMM. Ranka et al. [28] provide an in-depth exploration of GPU matrix multiplication techniques, discussing various algorithms tailored for GPUs. Their chapter presents experimental results and performance analyses to showcase the benefits and trade-offs of these approaches compared to conventional CPU-based methods. They also address challenges associated with GPU matrix multiplication, such as memory constraints and algorithm scalability, while outlining future research directions to enhance GPU-based matrix computations.

Cleverson et al. [36] offer a comprehensive comparative analysis of three widely used parallel programming frameworks: OpenACC, OpenMP, and CUDA. Their study evaluates the performance of these frameworks by implementing and benchmarking sequential and parallel algorithms across various hardware platforms, focusing on execution time, speedup, and scalability under different problem sizes and parallelization strategies. Salim et al. [13] conducted a comparative benchmarking study investigating the performance of matrix multiplication on multicore coprocessors and GPUs, providing valuable insights into the suitability of

each platform for specific computational requirements. In , the authors note that while hybrid parallel programming models[13; 27], such as MPI+OpenMP, can significantly improve overall performance for MMM[27], there are instances where SIMD outperforms OpenMP, suggesting the potential benefits of using them in tandem. Moreover, Pal et al. [14] introduce a novel accelerator architecture for efficiently performing sparse matrix multiplication, addressing the challenges posed by irregular data access patterns. Their proposed OuterSPACE accelerator leverages the outer product formulation to optimize sparse matrix multiplication, exploiting inherent sparsity to improve performance and energy efficiency. Finally, [37] discusses enhancements made to the matrix-matrix multiplication routine for ScaLAPACK, showcasing improved computation efficiency and resource utilization. Table 3 depicts the implementation of programming models on the CPU.

Table 3. Implementation of MMM on the CPU platform using the parallel programming approach in detail

Ref	CPU	Core-tread	Max (N)	Programming model	Applied Algorithms
[8]	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
[9]	Core i7-5600U	2-4	1120	C++ SIMD	Basic, transposed
[10]	Pentium core 2 E6550	2-2	4800	SIMD	Tiling and cache management
	core i7 2600k	4-8	4800	SIMD	Tiling and cache management
[11]	Core i7 3960X	6-12	8192	OpenCL	BLAS
	Xeon Phi 5110P	60-240	8192		
[35]	Intel Core i7 3960X	6-12	6144	OpenCL	Basic, transposed
	AMD Bulldozer FX-8150	8-8	6144		
[27]	Intel Pentium D 3.40GHz	2-2	2100	MPI MPI+ OpenMP	Blocking
[28]	Intel Xeon quad-core	4-8	4096	OpenMP	Loop parallelization
[36]	AMD Athlon II X2 270	2-2	834	OpenMP	Loop parallelization
[13]	Intel Xeon Phi 7120P	61-244	60000	OpenMP MPI+ OpenMP	Basic

Please note that on the CPU platform, "Core-thread" refers to the number of physical cores and the total supported threads, while "Max (N)" indicates the maximum matrix dimension utilized in the corresponding study.

Parallel Implementation of Matrix Multiplication

Matrix multiplication is a computationally intensive operation that is well-suited for parallelization, making use of modern multicore processors and parallel computing architectures. Various strategies exist for parallel implementation, with the performance gains from parallelization depending on factors such as matrix size, the number of threads, and the underlying hardware architecture. Efficient parallel

matrix multiplication hinges on load balancing and effective data management. This paper investigates various parameters, including matrix size, number of cores, and processor type:

- **Matrix Size:** The performance of matrix-matrix multiplication is evaluated across a range of matrix sizes. Experiments are conducted with matrices of different dimensions, including 256x256, 512x512, 1024x1024, 2048x2048, 4096x4096, and 6400x6400.
- **Number of Cores:** The paper discusses the number of physical cores present in each CPU platform used for experimentation, noting that the tested systems have either 2 or 4 physical cores. It also addresses the number of threads utilized in OpenMP implementations, indicating that a maximum of 8 threads is employed due to the limitations of certain CPU platforms.
- **Processor Type:** Detailed specifications of the CPU architectures of the tested systems are provided, including the CPU model (e.g., Intel® Core™ i3-2370M, Intel® Core™ i7-6700HQ, AMD Ryzen 5 3600, AMD Ryzen 7 3700X, Intel® Core™ i7-10700K), architecture (e.g., Sandy Bridge, Skylake), clock speed, cache sizes, memory bandwidth, and instruction set extensions (e.g., SSE, AVX, AVX 2.0).

Overall, this research paper offers valuable insights into how matrix size and, to some extent, the number of cores influence the performance of various matrix multiplication approaches. While processor type is noted, the primary focus is on the impact of the programming models themselves.

SIMD implementation of Matrix Multiplication

SIMD (Single Instruction, Multiple Data) programming facilitates parallel processing within a single CPU. Three methods can be employed for SIMD programming: inline assembly, intrinsic functions, and vector classes. In this study, the SIMD version of matrix-matrix multiplication (MMM) is implemented using intrinsic functions. Although inline assembly allows for optimal performance by managing each register individually, it is complex and requires careful handling of data transfers between CPUs and memory [16].

The utilization of SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) instructions enables parallel execution of data in the MMM kernel. By taking advantage of SSE and AVX capabilities, multiplication can be performed with a step size of four and eight floating-point operations, respectively, using intrinsic functions, with results stored in pre-allocated registers. However, the loop interchange algorithm proved ineffective for SSE and AVX, prompting the adoption of transposed matrix multiplication, which incorporates the transpose time into the kernel execution time.

In the kernel code, two registers (XMM0 and XMM1 for SSE, YMM0, and YMM1 for AVX) load elements from matrices A and B, while the summation of the multiplication results is stored in another register (XMM2 or YMM2). As the kernel executes a single instruction for multiple data elements, the resulting values are stored in an array of size 4 or 8. The final elements of matrix C are computed by

summing the elements of this array. The AVX implementation is exemplified in Fig. 6, which illustrates the multiplication of two 4x4 matrices. However, the use of a scalar loop to accumulate the final result from the AVX register creates a bottleneck, potentially limiting performance improvements compared to a fully vectorized implementation.

```
void mat_mult_avx(const float *A, const float *B, float *C) {
   // Initialize C to zero
   for (int i = 0; i < 16; i++) {
      C[i] = 0.0f;
   }
   // Perform matrix multiplication
   for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 4; j++) {
         // Load row of A
         __m256 a_row = _mm256_set_ps(A[i * 4 + 3], A[i * 4 + 2], A[i * 4 + 1],
A[i * 4]);
         // Calculate the dot product
         for (int k = 0; k < 4; k++) {
            // Load column of B
            __m256 b_col = _mm256_load_ps(&B[k * 4]);
            // Multiply and accumulate
            __m256 c_temp = _mm256_mul_ps(a_row, b_col);
            // Sum the results
            C[i * 4 + j] += _mm256_hadd_ps(c_temp, c_temp)[0]; // Horizontal
add
         } } }}
```

Figure 6. AVX intrinsic function code for matrix-matrix multiplication. This snippet employs AVX instructions to perform efficient element-wise multiplication within the nested loops of the matrix-matrix multiplication (MMM) process.

Multicore CPUs

When programming in a SIMD (Single Instruction, Multiple Data) style, the operation is still executed on a single CPU core. However, modern CPUs typically feature at least 2 to 8 physical cores, and with multi-threading capabilities, they can handle 4 to 16 or more threads simultaneously.

OpenMP

Open Multiprocessing (OpenMP) is an API designed for shared memory architectures that facilitate thread-level parallelism (TLP) for multicore systems. The tenth version of the matrix-matrix multiplication (MMM) is depicted in Fig. 7.

```
void mat_mult_omp(const float *A, const float *B, float *C, int N) {
   // Initialize C to zero in parallel
   #pragma omp parallel for collapse(2)
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
         C[i * N + j] = 0.0f;
      } }
```

```
// Perform matrix multiplication in parallel
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++) {
      for (int k = 0; k < N; k++) {
         C[i * N + j] += A[i * N + k] * B[k * N + j];
      } } } }
```

Figure 7. OpenMP Implementation of Matrix-Matrix Multiplication. This code snippet demonstrates how OpenMP efficiently utilizes parallel processing to execute matrix-matrix multiplication by distributing the computational workload across multiple threads, resulting in enhanced execution speed.

In OpenMP, the process of parallelizing a loop requires integrating the OpenMP library and using pragma directives with parameters designed specifically for loops. Granularity can be managed by modifying loop chunks through various scheduling types, including static and dynamic options. The parallelization itself depends largely on the OpenMP compiler, which is responsible for generating the parallel code, while the runtime system oversees the management of parallel threads and resources. In this implementation, the loop interchange model was utilized, focusing on parallelizing the outer loop.

OpenCL

In OpenCL, the host code is responsible for coordinating and queuing data transfers and kernel execution commands. The device code executes the kernel in an array of work items, organized into work-groups. This paper utilizes the Intel SGEMM library[38], which implements the BLAS (Basic Linear Algebra Subprograms) and evaluates its effects on CPUs, comparing it with the hybrid OpenMP-SIMD kernel code. The implementation includes basic, transposed, and blocking techniques, with a particular emphasis on finding suitable work-item and work-group dimensions to achieve optimal performance. We will use the method that is mentioned in Equ (2). To maximize automatic vectorization on selected platforms, work-item dimensions are set to 1 and 16, where 16 corresponds to the total number of floating-point operations executed in a SIMD manner. Additionally, work-group sizes must exceed eight due to the AVX support of the selected platforms.

Work-group (W/2, W/128), Work-item (16,1) (2)

Hybrid OpenMP-SIMD

The hybrid approach integrates OpenMP and SIMD instructions. In this version, each thread operating on the CPU fully utilizes SIMD functions, though the number of available registers is limited. OpenMP employs the fork-join method, wherein a master thread automatically distributes workloads across slave threads. Since SIMD vectorization has been supported since the release of OpenMP 4.0, the kernel code is partitioned into 4-8 tasks, allowing several threads to be allocated by the runtime system. This allocation can increase the register count to 8-16 for loading matrix elements, with each register capable of holding four to eight components, and 4-8

registers for storing multiplication results. Given that up to 16 registers are available in both SSE (for 64-bit systems) and AVX, a decline in performance is anticipated. Moreover, loading eight components necessitates significantly more bandwidth, which could introduce another bottleneck.

By combining data-level and thread-level parallelism, this hybrid approach has the potential to outperform either SIMD or OpenMP when used independently, especially for larger matrices that exceed the efficient register capacity. This method is particularly advantageous for compute-intensive tasks involving substantial datasets, such as matrix multiplication with large matrices.

3) Performance Evaluation

Environment and Hardware System

The performance of matrix-matrix multiplication (MMM) implemented using SSE, AVX, OpenMP, and OpenMP-SIMD programmed with intrinsic functions was evaluated and compared against the best-performing OpenCL version of MMM. All tested CPU platforms feature 256-bit wide vector units, enabling the execution of up to 16 single-precision floating-point operations. The specifications of the testbeds are detailed in Table 4. Square matrices filled with random floating-point numbers were used for the implementation results. The execution times for all code versions—single-core, SIMD, OpenMP, and hybrid OpenMP-SIMD—were measured using the Visual Studio 2013 (MSVC++) Profiler, in addition to the CPU clock () function. The Intel SDK 5.3 was employed for analyzing the OpenCL application. The minimum runtime was recorded for each selected system.

Table 4. Platform's specification. There are eight systems named PF#1 to PF#8.

Platforms	CPU	Cores	Core Clock	Memory Bandwidth
PF#1	Intel® Core™ i3-2370M	2	2400	21.3
PF#2	Intel® Core™ i3-5005U	2	200	21.3
PF#3	Intel® Core™ i7-2630QM	4	2900	25.6
PF#4	Intel® Core™ i7-6700HQ	4	3500	34.1
PF#5	AMD Ryzen 5 3600	6	3600	30.0
PF#6	Intel® Core™ i5-10400	6	2900	25.0
PF#7	AMD Ryzen 7 3700X	8	3600	36.0
PF#8	Intel® Core™ i7-10700K	8	3600	32.0

The CPU platforms utilized in our study exhibit significant differences in terms of processor generation, core count, clock speed, cache sizes, and memory bandwidth. Notably, platforms PF#1 to PF#8 offer enhanced capabilities through the AVX instruction set extensions.

Experimental Design and Setup

Our study involved extensive experiments designed to evaluate the performance of various parallel programming models for matrix-matrix multiplication (MMM) across different CPU platforms. The experimental design included several key components to ensure a rigorous evaluation and accurate comparison of the parallel implementations. Below are the detailed insights into our experimental methodology and setup:

- **Matrix Sizes:** We varied the sizes of the input matrices to assess the performance of parallel implementations across different problem dimensions. Specifically, we considered square matrices ranging from 256×256 to 6400×6400 to capture a wide spectrum of computational complexities and scalability behaviors.
- **Parallel Programming Models:** We implemented MMM using eight distinct parallel programming models: SIMD, OpenMP, Hybrid OpenMP-SIMD, and OpenCL. Each model was carefully selected to exploit specific parallelization techniques and effectively leverage the computational resources available on the CPU platforms.
- **Experimental Platforms:** Experiments were conducted on eight different CPU platforms, labeled PF#1 to PF#8. These platforms feature diverse hardware configurations, including varying numbers of physical cores, CPU architectures, and memory bandwidths. The detailed specifications for each platform can be found in Table 4 of our manuscript.
- **Experimental Procedure:** For each combination of matrix size, parallel programming model, and CPU platform, we executed multiple iterations of the MMM algorithm to ensure the robustness and reliability of the results. Specifically, each experiment was repeated 100 times to account for variability in execution times and to obtain statistically significant measurements.
- **Performance Metrics:** The primary performance metric measured was the execution time of each MMM implementation. Additionally, we calculated speedup values by comparing the execution time of the parallel implementations against a baseline sequential implementation. These metrics allowed us to quantitatively evaluate the effectiveness of each parallel programming model in accelerating MMM computations.

Experimental Results for Sequential Implementation of MMM

The loop interchange algorithm was implemented across eight testbeds, with results detailed in Table 5. For speedup comparisons, each parallel implementation of MMM was evaluated against the results from the sequential loop interchange algorithm. For the remainder of this paper, all parallel implementations from platforms PF#1 to PF#8 will be compared to the sequential results in Table 5. The experiment results investigate the execution times of the Sequential Loop Interchange Algorithm across eight CPU platforms, highlighting the impact of CPU architecture on performance in matrix-matrix multiplication tasks. As matrix sizes

increase, execution times demonstrate a substantial rise, confirming the theoretical complexity of matrix multiplication at O(n3). The performance comparison reveals that Platform PF#4, with its advanced Intel i7 CPU, achieves the best execution times across all tested matrix sizes, attributed to its higher clock speed and memory bandwidth. In contrast, the AMD Ryzen series exhibits superior performance at larger matrix sizes compared to Intel CPUs, suggesting efficiency in handling parallel tasks. The observed non-linear scaling behavior in execution times emphasizes the challenges associated with larger datasets, particularly in memory access. These findings underscore the importance of hardware selection and algorithm optimization, advocating for continued research into hybrid approaches and memory management strategies to enhance performance for computationally intensive applications.

The execution times provided in the table, do not directly facilitate a comparison of performance between different CPU platforms. To enable such comparisons, additional metrics, such as execution time per floating-point operation (FLOP/s) or a relative speedup analysis between platforms for specific matrix sizes, would be necessary.

Table 5. Execution time (in seconds) of the sequential loop interchange algorithm across eight CPU platforms on different Matrix sizes.

Platforms	Matrix size					
	256×256	512×512	1024×1024	2048×2048	4096×4096	6400×6400
PF#1	0.017	0.132	1.332	9.405	76.473	278.436
PF#2	0.015	0.125	1.131	9.006	72.507	270.857
PF#3	0.015	0.119	0.979	8.235	66.058	246.786
PF#4	0.013	0.093	0.750	6.031	48.05	180.2
PF#5	0.0143	0.1023	0.8250	6.6341	52.8550	198.22
PF#6	0.01365	0.09765	0.78750	6.33255	50.45250	189.21
PF#7	0.013	0.093	0.750	6.031	48.050	180.2
PF#8	0.01235	0.08835	0.71250	5.72945	45.64750	171.19

The experiments in our study were conducted multiple times to ensure comparability and reliability of results. Specifically, each experiment was repeated 100 times to mitigate the effects of variability and to achieve robust findings. This repetition aimed to minimize the influence of random fluctuations, thus providing reliable and consistent results. To accurately measure execution time, we employed various tools and algorithms tailored to the specific parallel programming models used in our study. For instance, when evaluating the performance of SIMD implementations, we utilized profiling tools offered by the compiler, such as Visual Studio 2013 (MSVC++) Profiler, to gauge the execution time of the code. For OpenMP and OpenCL implementations, we relied on performance analysis tools included in the Intel SDK 5.3 and the Intel OpenCL SGEMM library. To enhance the accuracy of our measurements, we adopted a rigorous approach that considered the minimum execution time observed across the 100 iterations. By selecting the average time recorded during the repeated experiments, we aimed to mitigate the influence of outliers or fluctuations in individual measurements, thus ensuring the

reliability and consistency of our results. Overall, the combination of repeated experiments and careful selection of the average execution time significantly contributed to the comparability and accuracy of the results obtained from our study.

Experimental Results for SIMD and OpenMP Implementation

Our analysis of the SIMD and OpenMP implementations, shown in Table 6, revealed distinct performance characteristics:

- **SSE (Streaming SIMD Extensions)**: The speedup results for SSE show moderate improvements, particularly in smaller matrix sizes. For instance, at a matrix size of 256, platforms PF#1 to PF#4 exhibit speedup values ranging from 3.0 to 3.25, indicating that SSE effectively utilizes data-level parallelism to enhance performance. However, as matrix sizes increase to 2048 and 6400, the speedup values stabilize around 2.3, suggesting diminishing returns with larger datasets. This indicates that while SSE is beneficial for smaller matrices, it may not leverage the full potential of available resources in larger computations.
- **AVX (Advanced Vector Extensions)**: AVX demonstrates superior speedup metrics compared to SSE, especially as the matrix size grows. For matrix size 256, speedups reach up to 6.5 on PF#4, and at 2048, they remain robust at around 2.86. The increased register width and enhanced instruction set of AVX contribute significantly to its ability to perform operations in fewer cycles. Notably, even for larger matrices, the performance remains competitive, which aligns with the expectation that AVX can handle greater computational loads efficiently.
- **OpenMP (Open Multi-Processing)**: OpenMP shows varied performance, particularly excelling in larger matrix sizes. The speedup values for OpenMP across platforms PF#3 and PF#4 at 2048 and 6400 are particularly noteworthy, reaching up to 3.81 and 3.29, respectively. This suggests that OpenMP effectively harnesses multi-threading capabilities, which are beneficial in computationally intensive tasks like matrix-matrix multiplication (MMM). The significant speedups observed highlight the importance of leveraging parallelization at the thread level, especially for large data sets.

The experiment indicates that while all three parallel computing models (SSE, AVX, and OpenMP) offer performance enhancements over traditional sequential implementations, their effectiveness varies based on matrix size and CPU architecture. SSE is more advantageous for smaller matrices, AVX consistently outperforms its advanced capabilities, and OpenMP excels in multi-threaded environments with larger datasets. This comprehensive analysis underscores the importance of selecting the appropriate parallelization strategy based on specific computational requirements and hardware configurations to achieve optimal performance in high-performance computing tasks.

Table 6. Speedup comparison for SSE, AVX, and OpenMP relative to each system's best sequential implementation on different Matrix sizes.

Matrix Size	Type	PF#1	PF#2	PF#3	PF#4	PF#5	PF#6	PF#7	PF#8
256	SSE	3.1	3	3	3.25	3.2	3.0	3.4	3.5
	AVX	3.4	5	5	6.5	5.5	5.2	6.0	6.3
	OMP	**1.21**	**1.31**	**2.5**	**3.25**	**2.8**	**2.5**	**3.0**	**3.2**
2048	SSE	2.3	2.48	2.5	2.6	2.7	2.6	2.8	2.9
	AVX	2.3	2.65	2.8	2.86	2.9	2.7	3.0	3.1
	OMP	**1.67**	**1.63**	**3.2**	**3.81**	**3.0**	**2.8**	**3.3**	**3.5**
6400	SSE	2.21	2.48	2.61	2.27	2.6	2.5	2.7	2.9
	AVX	2.16	2.4	2.61	2.65	2.7	2.6	2.9	3.0
	OMP	**2.02**	**1.55**	**3**	**3.29**	**3.2**	**2.9**	**3.5**	**3.8**

Note that the speedup values are specific to each system's sequential baseline, meaning lower speedup does not necessarily correspond to a higher execution time on that system.

SSE and AVX comparison

Table 7 indicates that the assembly codes generated for SSE and AVX are comparable in terms of the number of operations. A deeper analysis reveals that latency is a critical factor affecting the performance of both SSE and AVX. Since the algorithm used does not address the memory bottleneck in matrix-matrix multiplication (MMM), memory operation latency becomes increasingly significant as matrix sizes grow larger. Although AVX features twice the register size and calculation capacity compared to SSE, it also experiences slower memory operations. This combination of slower memory access and the higher number of arithmetic operations in each iteration leads to a decline in performance for larger matrices.

Table 7. The number of assembly operations for the SSE and AVX version of MMM.

	Memory Operations			Mathematical Operations		
	Type	Num. of	Latency	Addition	Multiplication	Latency
AVX	Y, m256	14	3	9	12	6
	m256, Y	13	3			
	Y, m256 (Double Precision)	16	4	10	8	7
	m256, Y (Double Precision)	15	4			
	Y, m256 (Aligned)	14	2	8	6	5
	m256, Y (Aligned)	13	2			
	Y, m256 (Unaligned)	17	5	11	10	8
	m256, Y (Unaligned)	16	5			
SSE	X, m128	14	2	5	8	6
	m128, X	14	3			
	X, m128 (Double Precision)	16	3	6	9	7

	m128, X (Double Precision)	15	3			
	X, m128 (Aligned)	14	2	5	6	5
	m128, X (Aligned)	14	2			
	X, m128 (Unaligned)	18	4	7	11	9
	m128, X (Unaligned)	17	4			

"NUM. of" indicates the count of operations, where X and Y represent 128-bit and 256-bit registers. Memory operations, denoted as m128 and m256, refer to 128-bit and 256-bit memory interactions, which can occur either as (register, memory) or (memory, register) transfers.

The AVX implementation typically performs a greater number of operations compared to SSE, which can be attributed to its capacity to process wider data elements (256 bits versus 128 bits for SSE). Despite this increased operational throughput, the latency associated with most operations tends to be lower for AVX than for SSE, suggesting that AVX may achieve faster execution on hardware architectures optimized for AVX instructions.

However, it's essential to consider that the specific number of multiplication operations and their associated latencies can vary between AVX and SSE based on the type of memory operations being performed. This discrepancy means that while AVX may excel in certain computational scenarios, its advantages can be diminished in cases where memory bandwidth and latency become bottlenecks, particularly with larger matrices in matrix-matrix multiplication (MMM). Thus, the performance benefits of AVX can be context-dependent, highlighting the need for careful optimization and tuning in real-world applications.

Hybrid OpenMP-SIMD and OpenCL Result

In multicore CPUs, as illustrated in Fig 8, we implemented and compared hybrid methods, OpenMP-SSE and OpenMP-AVX, with Intel's OpenCL SGEMM [38]. While OpenCL is known to perform better with a hybrid CPU-GPU setup [39], this paper focuses solely on CPU performance, and integrated GPU results were excluded. SIMD vectorization, supported since OpenMP 4.0, allowed us to apply #pragmaompsimd for vectorizing our MMM code. The intention was to compare the performance of two SIMD technologies, where AVX, theoretically twice as fast as SSE, was expected to yield significantly different results. However, experimental findings showed this was not the case. While OpenCL delivered the best overall performance, all three methods showed performance degradation as matrix size increased.

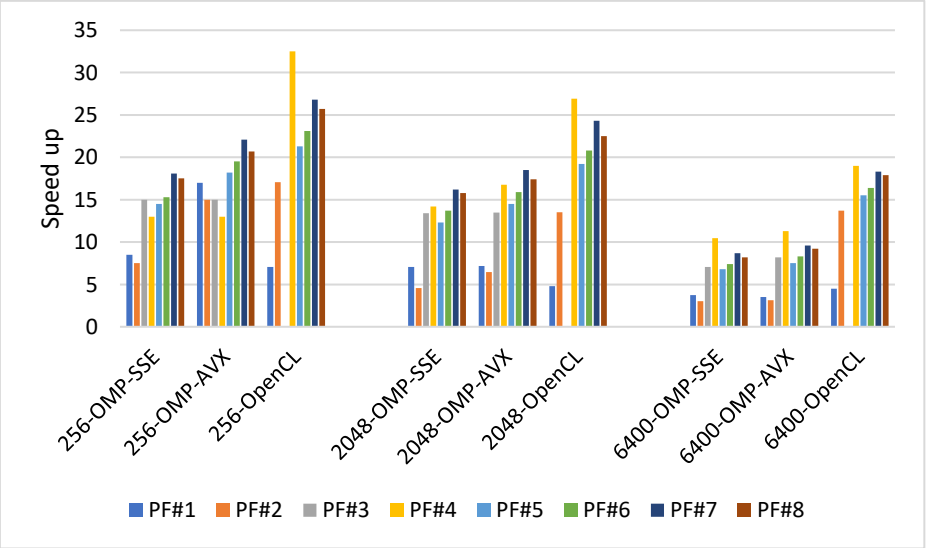

Figure 8. A performance comparison of the hybrid OpenMP-SIMD approach and OpenCL across eight chosen platforms. The graph illustrates the speedup achieved by the hybrid method and OpenCL relative to the best sequential execution time on each respective system.

The performance comparison of hybrid OpenMP-SIMD (using SSE and AVX) and OpenCL across various platforms reveals distinct trends. For smaller matrices (256x256), both SSE and AVX optimizations show good speedup, with newer platforms (PF#5 to PF#8) exhibiting greater improvements due to their advanced hardware. AVX, with its 256-bit registers, benefits more from modern processors, especially those with higher core counts and memory bandwidth. As matrix sizes grow (2048x2048, 6400x6400), performance declines across all systems due to memory bottlenecks, but newer platforms continue to show better results, particularly with AVX optimizations. OpenCL performs best on larger matrices, particularly on platforms with higher memory bandwidth, confirming its efficiency in handling data-heavy tasks. Overall, modern hardware significantly boosts the speedup of SIMD and OpenCL optimizations, especially for larger matrices, underscoring the importance of platform architecture in determining performance in matrix-matrix multiplication tasks.

Although maximizing performance was not the primary goal of this paper, the highest performance for MMM is shown in Fig 9. The results indicate that programming languages with fine granularity are more suited for MMM, and larger registers may not significantly impact memory-bound applications. Matrix multiplication, with a complexity of ($2n^3$ FLOPS) [9], was measured in terms of GFlops (giga-floating point operations per second) in Fig 9, representing the total floating-point operations executed per second on each platform. While OpenCL showed superior performance, two key considerations emerged. First, Intel's OpenCL SGEMM employs a blocking algorithm that can mitigate memory bandwidth limitations. Second, the CPUs used in this study supported no more than

eight threads, suggesting that OpenMP performance could improve on systems with better CPU capabilities.

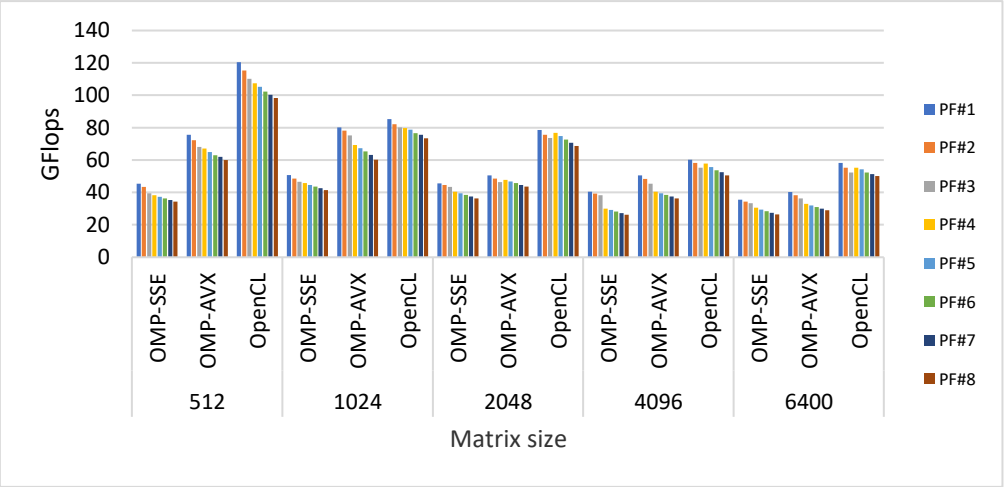

Figure 9. Comparative performance analysis of hybrid OpenMP-SIMD across benchmark systems.

Overall, OpenCL demonstrates the highest performance across all tested matrix sizes, with its efficiency remaining relatively stable or even experiencing slight improvements as the matrix size increases. In contrast, the performance of both OpenMP implementations (OMP-SSE and OMP-AVX) shows initial improvements with increasing matrix sizes but begins to exhibit limitations at larger dimensions. For OMP-SSE, performance tends to plateau or even decrease slightly for larger matrices (4096 and 6400). This trend suggests potential constraints in leveraging SSE instructions or OpenMP parallelization effectively for these larger workloads. On the other hand, OMP-AVX exhibits more significant improvements than OMP-SSE, particularly for medium-sized matrices (1024 and 2048). However, when dealing with very large matrices (6400), OMP-AVX's performance declines more substantially than that of OMP-SSE. This drop may be attributed to factors such as increased thread management overhead or limitations in effectively utilizing AVX instructions with extensive data sets. The results imply that while OpenCL is the preferred choice for maximizing performance (especially for larger matrices on system PF#1) the effectiveness of OpenMP implementations (both SSE and AVX) is highly dependent on matrix size. Consequently, OpenMP may be beneficial for specific use cases, particularly when considering its relative ease of use compared to OpenCL.

Discussion on the Comparative Performance Analysis of Hybrid OpenMP-SIMD Across Benchmark Systems

The comparative performance analysis of hybrid OpenMP-SIMD across various platforms (PF#1 to PF#8) highlights the significant improvements achieved through the implementation of SIMD (Single Instruction, Multiple Data) techniques alongside OpenMP (Open Multi-Processing) for matrix-matrix multiplication

(MMM). As seen in the data, the Gigaflops per second (GFlops) metrics indicate substantial performance differences between the two SIMD approaches—SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions)—as well as OpenCL across the selected platforms.

On examining the results, it is evident that the best performance is observed on platforms equipped with higher core counts and better memory bandwidth, especially in the case of OpenCL. For instance, PF#1 and PF#2 consistently outperform others in GFlops for all matrix sizes, leveraging their architecture to optimize SIMD operations effectively. The AVX implementations show a marked improvement over SSE, particularly in larger matrix sizes where AVX's 256-bit registers can process more data simultaneously compared to the 128-bit registers used in SSE. This trend is visible across all platforms, with AVX yielding significantly higher performance metrics, particularly in the matrix sizes of 2048x2048 and 6400x6400.

In contrast, the performance of OpenCL stands out, particularly in larger matrices, indicating its capacity to handle complex computations efficiently through optimized memory management and parallel processing capabilities. The consistent results across PF#5 to PF#8 illustrate that as system specifications improve, so perform the hybrid approaches. For example, PF#8, despite having the lowest GFlops in some cases, shows competitive results, particularly in the higher matrix sizes due to its advanced architecture and memory management.

Moreover, the speedup comparisons reflect that all platforms benefit from SIMD optimizations, although the extent of this benefit varies. The observed trends confirm that while both SSE and AVX provide enhancements over sequential implementations, the gains are more pronounced with AVX, particularly as the computational workload increases. The hybrid approaches demonstrate that combining these advanced techniques not only optimizes performance but also allows for better utilization of modern multicore architectures.

Performance Comparison of the Proposed Approach

Given that the CPU used in the PF#3 system closely resembles that utilized in prior research [9], comprehensive results are presented in Fig 10. Both systems feature Intel Broadwell family CPUs, with the PF#3 system operating at 2GHz compared to 3.2GHz in previous work. Each system comprises two cores and four threads, supporting the same instruction set. The results indicate that hybrid OpenMP-AVX outperforms in all selected matrix sizes. For a fair comparison with the most recent work utilizing AVX intrinsic functions, the GFlops from [9] were approximated and compared to the PF#3 system results due to their similar CPU architecture. The best performance reported in previous work [9] was 8.16 GFlops, while our hybrid OpenMP-AVX implementation achieved 25.48 GFlops in optimal conditions. This comparison highlights that relying solely on SIMD to enhance scientific applications may not be the most effective strategy.

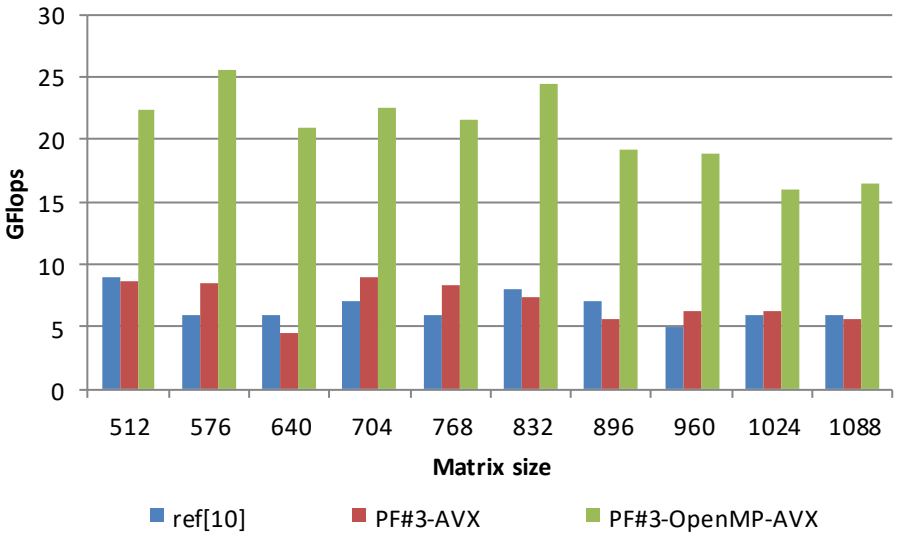

Figure 10. Comparison of the hybrid OpenMP-AVX with the most recent work on AVX intrinsic function code.

Furthermore, the speedup of PF#3-OpenMP-AVX compared to PF#3-AVX varies significantly with matrix size. For smaller matrices (512, 576, and 640), PF#3-OpenMP-AVX achieves considerable speedups, ranging from 3.04 to 4.97. This substantial speedup is likely due to the relatively low overhead of OpenMP thread management compared to the computational workload for these smaller matrices. However, as matrix sizes increase (704 and above), the speedup for PF#3-OpenMP-AVX drops considerably. In certain instances, it even performs slower than PF#3-AVX (e.g., at a matrix size of 704). This decline is likely caused by the heightened overhead of thread management and communication for larger matrices, which can negate the advantages of parallelization. In summary, while OpenCL provides superior performance, the hybrid OpenMP-AVX approach shows promising results, particularly for smaller matrices. However, challenges arise for larger matrices, where the management overhead can diminish the benefits of parallelization.

Performance Dependence on Matrix Size

The performance of various parallel programming models for matrix-matrix multiplication (MMM) is significantly influenced by the size of the matrices involved. The following trends and explanations have been observed:

- **Smaller Matrices:** For smaller matrices that fit conveniently within processor registers, SIMD (Single Instruction, Multiple Data) instructions perform exceptionally well. SIMD excels by executing the same operation on multiple data elements simultaneously, resulting in faster execution than sequential processing. However, as matrices increase in size and do not entirely fit within the available registers, the

effectiveness of SIMD diminishes. This is because data must be frequently loaded and unloaded from memory, hindering overall performance.

- **Larger Matrices:** In the case of larger matrices, OpenMP (Open Multi-Processing) provides performance benefits by distributing tasks across multiple threads. This distribution allows for effective processing, although its efficiency is influenced by various factors:
- **Memory Access Patterns:** For larger matrices, how data is accessed from memory can create bottlenecks. Optimizing algorithm design can help mitigate these challenges to some extent.
- **Communication Overhead:** As the number of threads increases, communication overhead among them can become significant, particularly for very large matrices.

The Hybrid OpenMP-SIMD approach combines the advantages of SIMD and OpenMP. By leveraging data-level parallelism within threads (via SIMD) and thread-level parallelism across cores (via OpenMP), this method generally leads to better performance across all matrix sizes. For smaller matrices, it benefits from SIMD instructions, while for larger matrices, it utilizes thread-level parallelism from OpenMP.

OpenCL (Open Computing Language) also demonstrates the potential for enhanced performance, particularly for larger matrices, especially when executed on platforms with powerful GPUs. The superior number of processing cores and higher memory bandwidth of GPUs allow OpenCL to effectively exploit these resources for parallel processing. However, some challenges remain:

- **Complexity:** Implementing efficient OpenCL code can be more complicated than using CPU-centric models like OpenMP, primarily due to the need to manage data transfer between CPU and GPU memory.
- **Overhead:** Setting up and managing execution on the GPU can introduce overhead, which may not be beneficial for very small matrices.

4)Conclusion

In this study, we investigated various parallel programming models to optimize matrix-matrix multiplication (MMM) on CPU platforms, addressing the inherent challenges of scalability and memory bandwidth. Our comprehensive experiments evaluated the performance of SIMD, OpenMP, Hybrid OpenMP-SIMD, and OpenCL across a range of matrix sizes. The results demonstrated that SIMD instruction sets, particularly SSE and AVX, excelled with smaller matrices by effectively utilizing data-level parallelism. However, their performance benefits diminished as matrix sizes increased due to memory access limitations. In contrast, the Hybrid OpenMP-SIMD approach proved to be more effective overall, leveraging both thread-level and data-level parallelism, leading to significant performance improvements across various matrix sizes.

OpenCL emerged as a strong contender, particularly for larger matrices, by harnessing the power of heterogeneous computing environments. Its ability to exploit the parallel processing capabilities of GPUs offered substantial performance gains, though the complexity of implementation and overhead considerations must be taken into account.

Ultimately, our findings underscore the importance of selecting the appropriate parallel programming model based on matrix size and computational requirements. The Hybrid OpenMP-SIMD and OpenCL approaches stand out as the most effective strategies for optimizing MMM performance, particularly for larger datasets. Future research should explore further hybrid methodologies, enhance memory access optimization, and address scalability issues to fully leverage the potential of modern CPU architectures and GPU resources. Through these efforts, we aim to advance the efficiency and effectiveness of computational tasks in parallel computing.

References

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

[2] Xu, H., Zhu, X., Wang, Q., & Liu, J. (2022, 18-20 Dec. 2022). *Efficiently Executing Sparse Matrix-Matrix Multiplication on General Purpose Digital Single Processor*. 2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys), https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00035

[3] Ghasempour Balagafshe, R., Akoushideh, A., & Shahbahrami, A. (2022). Matrix-matrix multiplication on graphics processing unit platform using tiling technique. *Indonesian Journal of Electrical Engineering and Computer Science*, *28*(2), 8. https://doi.org/10.11591/ijeecs.v28.i2.pp1012-1019

[4] Bustio-Martínez, L., Cumplido, R., Letras, M., Hernández-León, R., Feregrino-Uribe, C., & Hernández-Palancar, J. (2021). FPGA/GPU-based Acceleration for Frequent Itemsets Mining: A Comprehensive Review. *ACM Comput. Surv.*, *54*(9), Article 179. https://doi.org/10.1145/3472289

[5] Jo, Y.-Y., Kim, S.-W., & Bae, D.-H. (2014). *GPU-based matrix multiplication methods for social networks analysis* Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems, Towson, Maryland.https://doi.org/10.1145/2663761.2664192

[6] Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., R. Ruiz, F. J., Schrittwieser, J., Swirszcz, G., Silver, D., Hassabis, D., & Kohli, P. (2022). Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, *610*(7930), 47-53. https://doi.org/10.1038/s41586-022-05172-4

[7] Akoushideh, A., Shahbahrami, A., & Joe Afshany, A. (2024). Parallelization of license plate localization on GPU platform. *Multimedia Tools and Applications*, *83*(1), 2551-2564. https://doi.org/10.1007/s11042-023-15656-8

[8] Kelefouras, V., Kritikakou, A., Mporas, I., & Kolonias, V. (2016). A high-performance matrix–matrix multiplication methodology for CPU and GPU architectures. *The Journal of Supercomputing*, *72*(3), 804-844. https://doi.org/10.1007/s11227-015-1613-7

[9] Hassan, S. A., Hemeida, A. M., & Mahmoud, M. M. M. (2016). Performance Evaluation of Matrix-Matrix Multiplications Using Intel's Advanced Vector Extensions

(AVX). *Microprocess. Microsystems*, *47*, 369-374. https://doi.org/10.1016/j.micpro.2016.10.002

[10] Kelefouras, V., Kritikakou, A., & Goutis, C. (2014). A Matrix–Matrix Multiplication methodology for single/multi-core architectures using SIMD. *The Journal of Supercomputing*, *68*(3), 1418-1440. https://doi.org/10.1007/s11227-014-1098-9

[11] Gautier, T., & Lima, J. V. F. (2021, 2021-11-19). *Evaluation of two topology-aware heuristics on level-3 BLAS library for multi-GPU platforms*. PAW-ATM 2021 - 4th Annual Parallel Applications Workshop, Alternatives To MPI+X, Saint Louis, United States. https://hal.inria.fr/hal-03363275

[12] Aroon, M. A., Ismail, A. F., Matsuura, T., & Montazer-Rahmati, M. M. (2010). Performance studies of mixed matrix membranes for gas separation: A review. *Separation and Purification Technology*, *75*(3), 229-242. https://doi.org/https://doi.org/10.1016/j.seppur.2010.08.023

[13] Salim, M., Akkirman, A. O., Hidayetoglu, M., & Gurel, L. (2015, 1-4 July 2015). *Comparative benchmarking: matrix multiplication on a multicore coprocessor and a GPU*. 2015 Computational Electromagnetics International Workshop (CEM), https://doi.org/10.1109/CEM.2015.7237429

[14] Pal, S., Beaumont, J., Park, D. H., Amarnath, A., Feng, S., Chakrabarti, C., Kim, H. S., Blaauw, D., Mudge, T., & Dreslinski, R. (2018, 24-28 Feb. 2018). *OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator*. 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), https://doi.org/10.1109/HPCA.2018.00067

[15] Goto, K., & Geijn, R. A. v. d. (2008). Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, *34*(3), Article 12. https://doi.org/10.1145/1356052.1356053

[16] Jeong, H., Kim, S., Lee, W., & Myung, S.-H. (2012). Performance of SSE and AVX Instruction Sets. *ArXiv*. https://doi.org/10.48550/arXiv.1211.0820

[17] Oo, N. Z., & Chaikan, P. (2022, 4-7 May 2022). *Fast Blockwise Matrix-Matrix Multiplication Using AVX and Prefetching on Shared Memory*. 2022 13th Asian Control Conference (ASCC), https://doi.org/10.23919/ASCC56756.2022.9828222

[18] Chen, X., Gao, Y., Shang, H., Li, F., Xu, Z., Liu, X., & Chen, D. (2022). Increasing the Efficiency of Massively Parallel Sparse Matrix-Matrix Multiplication in First-Principles Calculation on the New-Generation Sunway Supercomputer. *IEEE Transactions on Parallel and Distributed Systems*, *33*(12), 4752-4766. https://doi.org/10.1109/TPDS.2022.3202518

[19] Keatkaew, T., Woradit, K., & Champrasert, P. (2022, 5-8 July 2022). *Hybrid Vectorization and Parallelization for Matrix-Matrix Multiplication on Multi-core Platform*. 2022 37th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC), https://doi.org/10.1109/ITC-CSCC55581.2022.9894915

[20] *Intel® oneAPI Math Kernel Library*. Microsoft. https://www.intel.com

[21] Corporation, I. *Intel® 64 and IA-32 Architectures Optimization*. Intel Corporation. https://www.intel.com/

[22] Chapman, B., Jost, G., & Pas, R. v. d. (2007). *Using OpenMP; Portable Shared Memory Parallel Programming*. The MIT Press. https://mitpress.mit.edu/9780262533027/using-openmp/

[23] Stone, J. E., Gohara, D., & Shi, G. (2010). OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Comput Sci Eng*, *12*(3), 66-72. https://doi.org/10.1109/mcse.2010.69

[24] Balagafshe, R. G., Akoushideh, A., & Shahbahrami, A. (2022). Matrix-matrix multiplication on graphics processing unit platform using tiling technique. *Indonesian Journal of Electrical Engineering and Computer Science*, *28*(2), 1012-1019. https://doi.org/https://doi.org/10.11591/ijeecs.v28.i2.pp1012-1019

[25] Bogosavljević, I. (2021). *Memory Access Pattern and Performance: the Example of Matrix Multiplication*. Johnny's Software Lab. Retrieved May 20 from https://johnnysswlab.com/memory-access-pattern-and-performance-the-example-of-matrix-multiplication/

[26] Memeti, S., Li, L., Pllana, S., Kołodziej, J., & Kessler, C. (2017). *Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption* Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing, Washington, DC, USA.https://doi.org/10.1145/3110355.3110356

[27] He, L., Shen, W., Li, Y., Shi, A., & Zhao, D. (2010, 28-31 May 2010). *MPI+OpenMP Implementation and Results Analysis of Matrix Multiplication Based on Rowwise and Columnwise Block-Striped Decomposition of the Matrices*. Third International Joint Conference on Computational Science and Optimization, https://doi.org/10.1109/CSO.2010.123

[28] Li, J., Ranka, S., & Sahni, S. (2013). In *GPU matrix multiplication*. Chapman-Hall/CRC Press. https://doi.org/10.1007/978-1-4613-9692-5_3

[29] Beckingsale, D. A. (2015). *Towards Scalable Adaptive Mesh Refinement on Future Parallel Architectures* [Doctor of Philosophy, The University of Warwick]. https://webcat.warwick.ac.uk/record=b2827209~S1

[30] Fang, J., Huang, C., Tang, T., & Wang, Z. (2020). Parallel programming models for heterogeneous many-cores: a comprehensive survey. *CCF Transactions on High Performance Computing*, *2*(4), 382-400. https://doi.org/10.1007/s42514-020-00039-4

[31] *OpenMP 5.1 Specification*. (2021). The OpenMP ARB (Architecture Review Boards). https://www.openmp.org/specifications/

[32] Bertoni, C., Kwack, J., Applencourt, T., Ghadar, Y., Homerding, B., Knight, C., Videau, B., Zheng, H., Morozov, V., & Parker, S. (2020, 18-22 May 2020). *Performance Portability Evaluation of OpenCL Benchmarks across Intel and NVIDIA Platforms*. 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), https://doi.org/10.1109/IPDPSW50202.2020.00067

[33] Banger, R., Bhattacharyya, B., & Bhattacharyya, K. (2013). *OpenCL Programming by Example*. Packt Publishing. https://books.google.com/books?id=1O80ngEACAAJ

[34] Waidyasooriya, H. M., Hariyama, M., & Uchiyama, K. (2017). *Design of FPGA-Based Computing Systems with OpenCL*. Springer. https://doi.org/10.1007/978-3-319-68161-0

[35] Matsumoto, K., Nakasato, N., & Sedukhin, S. G. (2012, 10-16 Nov. 2012). *Performance Tuning of Matrix Multiplication in OpenCL on Different GPUs and CPUs*. 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, https://doi.org/10.1109/SC.Companion.2012.59

[36] Cleverson, L., Ledur, D., Zeve, C., & Anjos, D. (2013). *Comparative Analysis of OpenACC, OpenMP and CUDA using Sequential and Parallel Algorithms* 11th Workshop on Parallel and Distributed Processing (WSPPD), UFRGS (Porto Alegre).

[37] Rizwan, M., Jung, E., Park, Y., Choi, J., & Kim, Y. (2022, 5-8 Dec. 2022). *Optimization of Matrix-Matrix Multiplication Algorithm for Matrix-Panel Multiplication on Intel KNL*. 2022 IEEE/ACS 19th International Conference on Computer Systems and Applications (AICCSA), https://doi.org/10.1109/AICCSA56895.2022.10017947

[38] *General Matrix Multiply - Intel® SDK for OpenCL™ Applications*. Intel. https://software.intel.com

[39] Mittal, S., & Vetter, J. S. (2015). A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.*, *47*(4), Article 69. https://doi.org/10.1145/2788396