# Natural Parallelism: The Paradigm of Matrix Multiplication Refined with Memory Locality, Data-Level, Thread-Level, and Process-Level Techniques

**Yohandi**
School of Data Science
The Chinese University of Hong Kong, Shenzhen
YOHANDI@LINK.CUHK.EDU.CN

## 1 Introduction

### 1.1 Background

In contemporary artificial intelligence, matrix multiplication is omnipresent, making it a foundational component of intricate computations. With the advent of architectures, optimizing matrix multiplication is more relevant than ever. The scope of this project will be solely focus on the most generalized form: dense matrix multiplication.

### 1.2 Objective

To improve a dense matrix multiplication function by implementing various optimization techniques. The function's computational performance will be enhanced through systematic optimization by addressing aspects of memory locality, data-level parallelism, thread-level parallelism, and process-level parallelism.

## 2 Parallel Programming Models and Their Computation Mechanisms

The optimization of matrix multiplication lies at the intersection of advanced hardware capabilities and sophisticated programming techniques. By utilizing various parallelism levels, the performance of matrix multiplication can be significantly boosted, ensuring optimal resource utilization and swift computational results. This section focuses on parallel programming models' distinctive computational mechanisms.

**Data-Level Parallelism (DLP):** At its core, DLP taps into the power of modern CPUs, particularly through SIMD (Single Instruction, Multiple Data) operations. Vectorized matrix multiplication stands as a quintessential example of DLP. In this model, rather than handling a single data point per instruction cycle, the CPU processes multiple data points simultaneously under a single instruction. By doing so, it optimally uses the data processing capabilities, minimizing idle CPU cycles and boosting overall performance. Furthermore, the efficiency of DLP isn't just limited to SIMD operations; by systematically organizing data, optimizing memory layout, and ensuring contiguous data access, one can significantly reduce computational bottlenecks, ensuring a smooth flow of data for parallel operations.

**Thread-Level Parallelism (TLP):** The advent of multi-core processors ushered in a new era for matrix multiplication optimization. TLP operates on the principle of distributing computational tasks across multiple threads, which can run concurrently on separate CPU cores. This concurrent execution allows for simultaneous calculation of matrix elements, resulting in a drastic reduction in overall computation time.

Moreover, GPUs, with their multitude of cores designed for parallel processing, offer a haven for TLP. GPU-accelerated matrix multiplication, for instance, can distribute matrix operations across thousands of threads, each working in harmony. This ensures that even the most massive matrices can be multiplied in a fraction of the time it would take a single-threaded operation.

**Process-Level Parallelism:** While threads can be seen as the smaller units of parallel computation, processes stand as more substantial, independent units of execution. Recursive matrix multiplication typifies process-level parallelism, where matrices are broken down into smaller chunks. Each chunk, or submatrix, is then processed independently, often on separate processors or even different machines in a cluster. This distributed approach, by its nature, promotes concurrent execution, as different processors or machines can work simultaneously without waiting for one another.

Furthermore, by breaking down matrices and focusing on smaller data chunks, memory usage is also inherently optimized. Each process can load its required submatrix into cache memory, ensuring quicker access times and reduced latency. However, a word of caution: as with most things in computing, there's a trade-off. As the matrices get too small, the overhead of managing multiple processes can outweigh the benefits of parallelism.

# 3 Refining Methods

The pursuit of optimizing matrix multiplication required a blend of several strategies, each designed to capitalize on the intrinsic nature of matrix structures and the capabilities of modern hardware. Here's a comprehensive breakdown of the methods employed:

## 3.1 Memory Locality and Cache Miss Minimization

### Loop Order Alteration:

One fundamental shift was reordering the nested loops from the traditional M-N-K sequence to a K-M-N pattern (i.e., transitioning from `for i j k` to `for k i j`). This change was not arbitrary. Matrices in memory are stored contiguously, and by reordering loops, memory access patterns are altered. The new sequence enhanced spatial locality, ensuring that successive operations accessed nearby memory locations. This, in turn, significantly reduced cache misses and ensured a more streamlined and efficient flow of data during multiplication.

### Tiled Matrix Multiplication:

Dividing matrices into smaller blocks or tiles brought forth another layer of optimization. When computation pivots around these bite-sized chunks, there's an inherent benefit to cache memory. The entire tile can often be accommodated within cache memory, making subsequent operations on the tile extremely swift due to the minimized latency associated with cache access.

### Direct Memory Access:

Array indices, although intuitive, introduce an overhead. To mitigate this, pointers were leveraged, offering a more direct route to memory addresses. This shift to constructs like `int *result_i = result[i]` and `const int *matrix2_k = matrix2[k]` ensured that the additional layer of indirection is bypassed, leading to quicker data access and modification.

### 3.2 Data-Level Parallelism with SIMD

Modern CPUs, with their advanced architectural designs, present a plethora of opportunities for optimization. One such avenue is SIMD. By processing multiple data elements simultaneously under a singular instruction, the computational throughput is substantially increased. To exploit this, the framework was structured to utilize a 512-bit register. In layman's terms, this allowed us to simultaneously conduct 16 separate operations. Orchestrating two such sets successively meant that a whopping 32 operations were synchronized, leading to a drastic reduction in overall computation time.

### 3.3 Thread-Level Parallelism with OpenMP

With a solid foundation of SIMD and memory-level optimizations in place, the introduction of OpenMP seemed the logical next step. OpenMP, with its high-level directives, offers a simplistic yet powerful means of incorporating thread-level parallelism. The specific pragma:

`pragma omp parallel for collapse(3) schedule(dynamic)`

was integrated, enabling the nested loops' parallelization. The `collapse(3)` directive is particularly intriguing, merging multiple loops into a singular, parallelizable entity, while the `schedule(dynamic)` ensures that loop iterations are dynamically allocated across threads, balancing the computational load and preventing any single thread from becoming a bottleneck.

### 3.4 Process-Level Parallelism with MPI

While previous strategies were primarily localized, MPI enabled us to extend the reach across multiple processors or even entire clusters, transforming a massive computational task into concurrently executed segments.

The synergy between SIMD, memory optimizations, and OpenMP was crucial. With MPI layered atop these refinements, it achieved a seamless fusion of local and distributed optimizations. In the specifics of matrix multiplication using MPI, operations were smartly partitioned. Each matrix segment was designated to a unique process, promoting efficient distribution and execution across available processors or nodes.

Our system's core constraint, limited to 32 cores, presented challenges and opportunities. It was paramount to find the perfect equilibrium between MPI processes and OpenMP threads. This balancing act led to configurations such as 1 MPI process using 32 OpenMP threads, 2 MPI processes each employing 16 OpenMP threads, and so on until 32 MPI processes with 1 OpenMP thread. Tweaking this balance ensured optimal load distribution without overburdening the available resources.

Inter-process communication was another focal point. Streamlined data transfer protocols ensured minimal latency between processes, fostering cohesion and efficiency. This intelligent blend of strategies, tailored to the system's specifications, elevated matrix multiplication performance, achieving faster computation times and more efficient resource utilization.

## 4 Experiment Results

In this section, the results of the computational experiments are presented, highlighting the performance improvements achieved after implementing various optimization models in matrix multiplication across two distinct matrix sizes along with the profile details such as cache miss rate, page fault rate, and *et cetera*.

## 4.1 Performance Improvements Highlight

The table below provides a view of these benchmarks, showcasing average and best execution times for each method and matrix size. Beyond the raw numbers, further analysis, including the calculation of speedup and efficiency, will be demonstrated with the aid of tables and figures in subsequent sections.

| **Methods** | **Cores** | **Matrices 1024x1024** | | **Matrices 2048x2048** | |
|---|---|---|---|---|---|
| | | **AVG (ms)** | **BEST (ms)** | **AVG (ms)** | **BEST (ms)** |
| Naïve | 1 | 8165 | - | 89250 | - |
| Memory Locality | 1 | 548.5 | 546 | 4574.5 | 4559 |
| SIMD + Memory Locality | 1 | 280.9 | 277 | 2364.8 | 2346 |
| OpenMP + SIMD + Memory Locality | 1 | 281.9 | 280 | 2404.3 | 2397 |
| | 2 | 256.5 | 255 | 2077.7 | 2073 |
| | 4 | 152.3 | 150 | 1184.5 | 1175 |
| | 8 | 91.8 | 87 | 645 | 614 |
| | 16 | 62.2 | 52 | 349.8 | 331 |
| | 32 | 43.5 | 42 | 200.1 | 193 |
| MPI + OpenMP + SIMD + Memory Locality | 1 x 32 | 50.3 | 47 | 225.5 | 220 |
| | 2 x 16 | 45.7 | 38 | 206.1 | 183 |
| | 4 x 8 | 46.7 | 34 | 222.2 | 185 |
| | 8 x 4 | 32.8 | 31 | 178.1 | 174 |
| | 16 x 2 | 30.5 | 27 | 167.7 | 165 |
| | 32 x 1 | 34.9 | 31 | 183.8 | 180 |

**Table 1: Performance benchmarks for matrix multiplication optimization models across two matrix sizes.**

| **Methods** | **Cores** | **Matrices 1024x1024** | | **Matrices 2048x2048** | |
|---|---|---|---|---|---|
| | | **Speedup** $S(p)$ | **Effic.** $E$ | **Speedup** $S(p)$ | **Effic.** $E$ |
| Memory Locality | 1 | 14.89 | 1489% | 19.51 | 1951% |
| SIMD + Memory Locality | 1 | 29.06 | 2906% | 37.73 | 3773% |
| OpenMP + SIMD + Memory Locality | 1 | 28.95 | 2895% | 37.13 | 3713% |
| | 2 | 31.83 | 1592% | 43.00 | 2150% |
| | 4 | 53.59 | 1340% | 75.31 | 1883% |
| | 8 | 88.95 | 1112% | 138.36 | 1730% |
| | 16 | 131.35 | 821% | 255.53 | 1597% |
| | 32 | 187.70 | 586% | 446.01 | 1394% |
| MPI + OpenMP + SIMD + Memory Locality | 1 x 32 | 162.31 | 507% | 396.02 | 1238% |
| | 2 x 16 | 178.75 | 559% | 433.19 | 1354% |
| | 4 x 8 | 174.86 | 547% | 402.31 | 1257% |
| | 8 x 4 | 249.08 | 778% | 501.68 | 1568% |
| | 16 x 2 | 267.54 | 836% | 534.64 | 1671% |
| | 32 x 1 | 234.10 | 732% | 485.93 | 1519% |

**Table 2: Speedup and efficiency for matrix multiplication optimization models compared to the naïve method across two matrix sizes.**

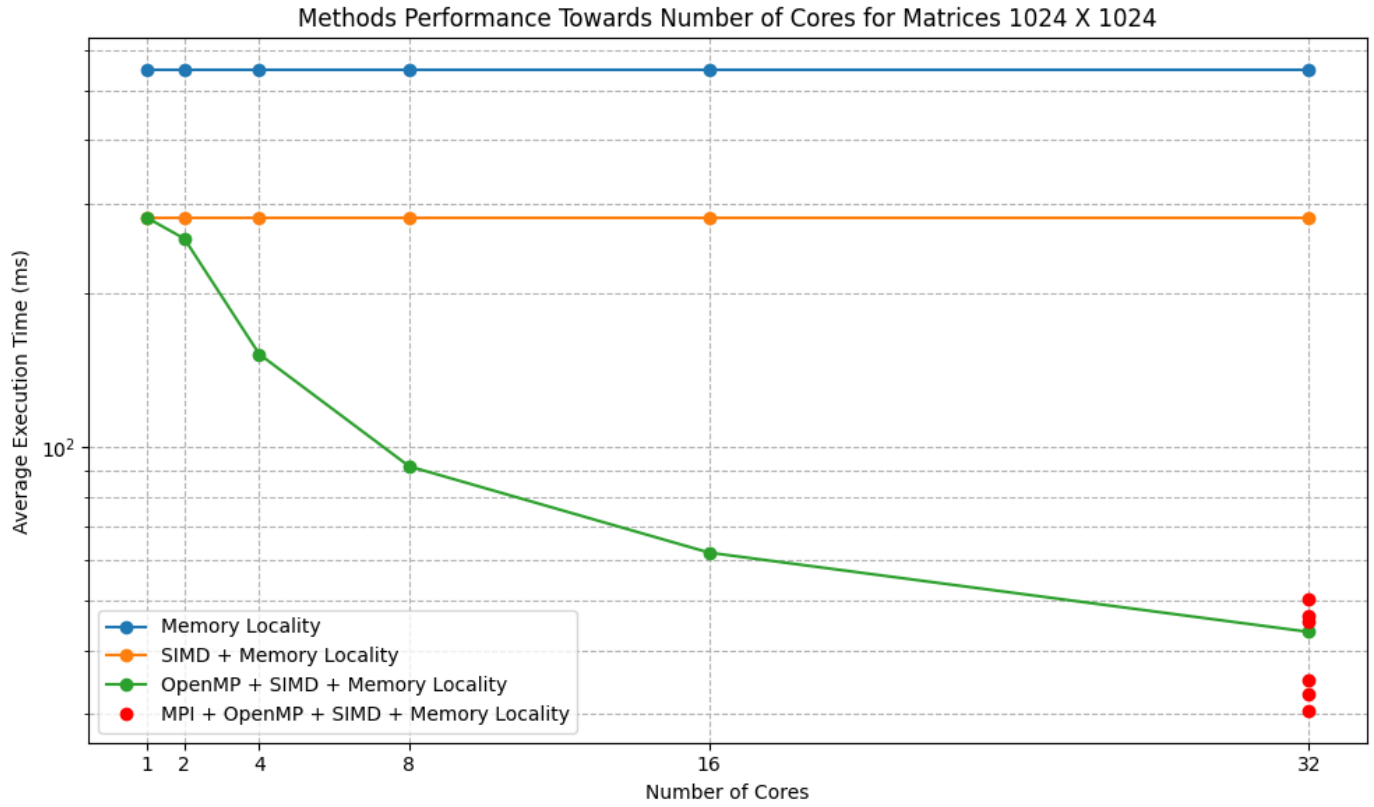


**Figure 1: Streamline graph showcasing all methods performance towards the number of cores when computing $1024 \times 1024$ matrices.**

## 4.2 Profiling Results

In the experimental evaluation, different parallelization methods with various numbers of cores and threads are profiled using perf. In the following sections,these profiling results for each parallelization method are presented.

| Cores | cpu-cycles:u | cache-misses:u | page-faults:u |
|---|---|---|---|
| 1 | 2,125,440,530 | 326,868 | 3,460 |

**Table 3: Profiling results for Memory Locality Matrix Multiplication.**

| Cores | cpu-cycles:u | cache-misses:u | page-faults:u |
|---|---|---|---|
| 1 | 1,245,281,245 | 179,698 | 3,459 |

**Table 4: Profiling results for SIMD + Memory Locality Matrix Multiplication.**

| Cores | cpu-cycles:u | cache-misses:u | page-faults:u |
|---|---|---|---|
| 1 | 1,249,363,574 | 160,747 | 3,516 |
| 2 | 1,841,480,324 | 181,018 | 3,525 |
| 4 | 2,020,925,558 | 2,641,180 | 3,530 |
| 8 | 2,226,097,400 | 2,993,250 | 3,551 |
| 16 | 2,580,138,772 | 2,589,169 | 3,583 |
| 32 | 3,257,574,157 | 2,272,633 | 3,621 |

**Table 5: Profiling results for OpenMP + SIMD + Memory Locality Matrix Multiplication.**

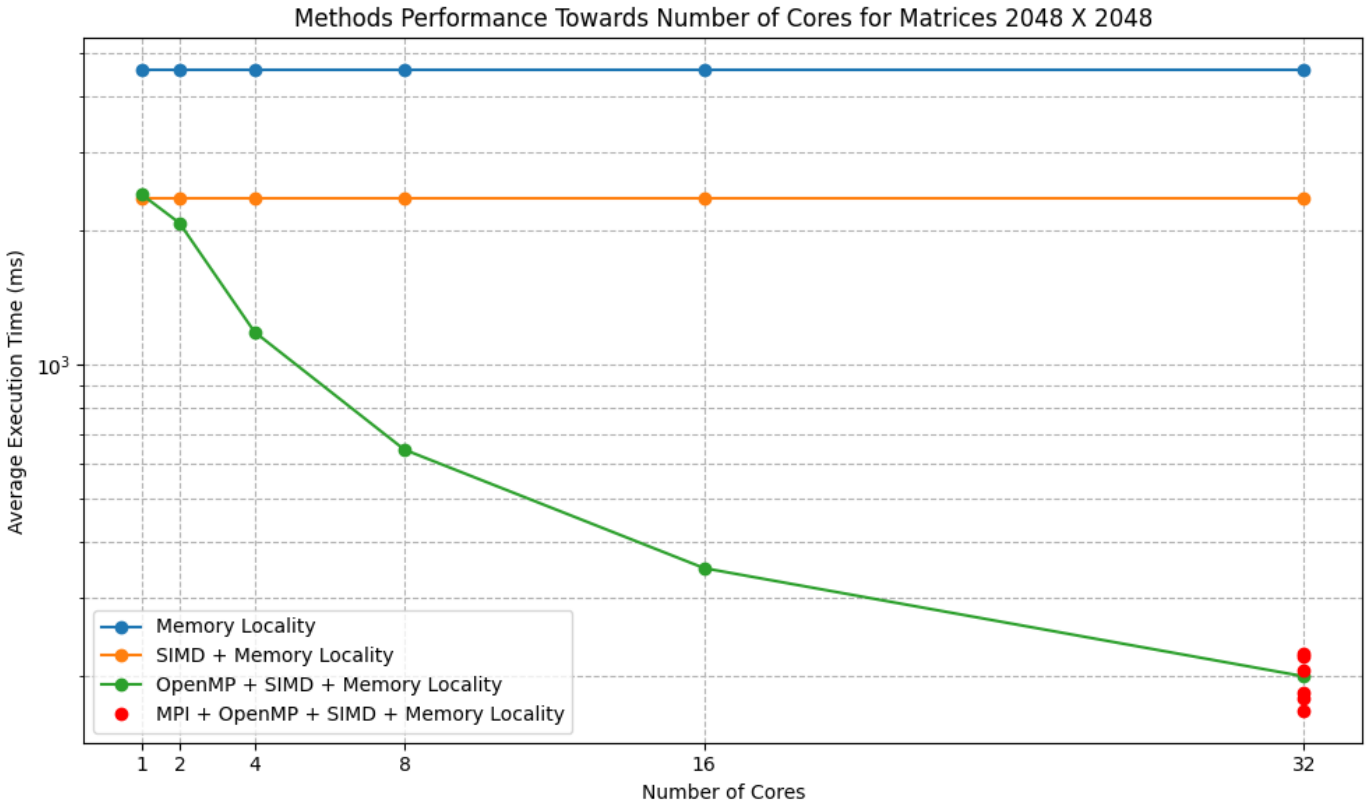


**Figure 2: Streamline graph showcasing all methods performance towards the number of cores when computing $2048 \times 2048$ matrices.**

| Processes | Threads | cpu-cycles:u | cache-misses:u | page-faults:u |
|---|---|---|---|---|
| 1 | 32 | 2,888,976,821 | 2,298,680 | 6,034 |
| 2 | 16 | 1,946,771,788 | 1,059,981 | 4,520 |
| 2 | 16 | 1,535,232,685 | 232,312 | 5,649 |
| 4 | 8 | 1,315,949,464 | 155,086 | 4,258 |
| 4 | 8 | 1,312,187,742 | 155,866 | 4,273 |
| 4 | 8 | 1,377,149,472 | 746,630 | 4,282 |
| 4 | 8 | 1,038,962,836 | 263,037 | 5,743 |
| 8 | 4 | 1,084,340,221 | 166,213 | 3,615 |
| ... | ... | ... | ... | ... |
| 16 | 2 | 952,271,171 | 209,341 | 3,571 |
| 16 | 2 | 680,104,939 | 408,470 | 7,441 |
| 32 | 1 | 1,452,787,889 | 211,929 | 3,572 |
| ... | ... | ... | ... | ... |
| 32 | 1 | 1,160,662,460 | 359,008 | 7,757 |

**Table 6: Profiling results for MPI + OpenMP + SIMD + Memory Locality Matrix Multiplication (truncated for brevity).**

## 4.3 GPU Performance

| | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 | Iteration 6 |
|---|---|---|---|---|---|---|
| CUDA | 4.81139 | 4.81424 | 4.81293 | 4.8089 | 4.8016 | 4.80659 |
| | Iteration 7 | Iteration 8 | Iteration 9 | Iteration 10 | AVG | BEST |
| | 4.80685 | 4.8113 | 4.79306 | 4.82032 | 4.808718 | 4.79306 |

**Table 7: CUDA's Performance in 10 Iterations on Matrices** $1024 \times 1024$

| | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 | Iteration 6 |
|---|---|---|---|---|---|---|
| CUDA | 38.4536 | 38.5557 | 38.4722 | 38.5607 | 38.462 | 38.4419 |
| | Iteration 7 | Iteration 8 | Iteration 9 | Iteration 10 | AVG | BEST |
| | 38.4603 | 38.4556 | 38.5532 | 38.4388 | 38.4854 | 38.4388 |

**Table 8: CUDA's Performance in 10 Iterations on Matrices** $2048 \times 2048$

# 5 Analyses and Findings

## 5.1 Impact of Optimizations

### 5.1.1 Performance of Naïve Implementation

The naïve implementation's sluggish performance for the $1024 \times 1024$ matrix multiplication, clocking at 8165ms, is a stark reminder of the inefficiencies latent within basic algorithms. Beyond the absence of parallelism, the naïve approach falls victim to cache misses and inefficient memory access patterns, translating to the bloated execution times observed. Without leveraging the specific capabilities of modern hardware or optimizing for memory hierarchies, such raw approaches are bound to underperform.

### 5.1.2 Memory Locality

With the loop order alteration from M-N-K to K-M-N, memory access patterns witnessed a dramatic shift. Given that matrices are stored contiguously in memory, this restructuring meant that consecutive operations accessed closely packed memory addresses. This strategic change, in conjunction with tiled matrix multiplication and direct memory access, brought down the computation time to 548.5ms, which is nearly 15 times faster than the naïve approach. It illustrates the dramatic effect memory access patterns and cache optimization techniques can have on performance, underscoring the importance of hardware considerations during algorithm design.

### 5.1.3 Superiority in Memory Locality and SIMD Operations

Data-level parallelism emerged as a game-changer. Combining memory locality with SIMD operations allowed us to harness the raw power of modern CPU architectures. Specifically, the ability to execute 32 synchronized operations are credited to 512-bit register utilization ensured that the matrix multiplication time plummeted to just 280.9ms. This is an impressive speedup of over 29 times when juxtaposed against the naïve algorithm, emphasizing the sheer computational boost that SIMD brings to the table.

### 5.1.4 Scalability Insights in OpenMP

OpenMP's introduction marked the foray into thread-level parallelism. Building on top of the foundational SIMD and memory locality optimizations, OpenMP's directives offered a simple yet potent mechanism to further expedite computations. The performance mirrored this, with the execution time dropping even further as more cores were incorporated. The massive speedup, peaking at 187.70 times

the naïve method on a 32-core configuration, demonstrates the compound benefits of combining data-level parallelism with thread-level parallelism. It's also noteworthy to mention the `schedule(dynamic)` directive's role, which ensured a balanced workload across threads, further optimizing performance.

#### 5.1.5 Hybrid Approaches: The Interplay of MPI, OpenMP, SIMD, and Memory Locality

MPI, representing process-level parallelism, was the final piece of the puzzle. This strategy, far from being a mere addition, was a synthesis of all previous optimizations on a grander scale. By allowing matrix multiplication tasks to span multiple processors or even entire clusters, MPI dramatically improved both execution times and resource efficiency. The highest speedup observed, a remarkable 267.54 for a $1024 \times 1024$ matrix, was achieved using a mix of 16 MPI processes and 2 OpenMP threads, reflecting the benefits of this hybrid approach. Moreover, the inter-process communication's efficiency, a pivotal aspect of MPI, guaranteed minimal latency, ensuring that the various processes worked harmoniously to achieve these stellar results.

### 5.2 Profiling Insights

Through the lens of profiling, it becomes evident that algorithmic and hardware-level optimizations can significantly influence performance metrics. While raw execution time is a critical metric, diving deeper into aspects like cache misses and CPU cycles gives us a holistic understanding of system behavior. It also underscores the importance of fine-tuning parallel implementations, as even within parallel approaches, nuances in thread and process distribution can have pronounced impacts on performance.

#### 5.2.1 Impact of Thread and Process Increase

It's clear from the profiling results that as the number of cores or threads increases, there's an associated increase in CPU cycles. This is consistent with the expectations, as more threads or processes will perform more operations, resulting in higher CPU cycle counts. However, it's important to note that the increase in CPU cycles doesn't always correspond to a proportional increase in execution time due to the parallel execution across cores.

#### 5.2.2 Cache Behavior with Scalability

The OpenMP + SIMD + Memory Locality results show an intriguing behavior regarding cache misses. When transitioning from 1 core to 4 cores, there's a massive increase in cache misses, going from 160,747 to a staggering 2,641,180. However, beyond this, as cores are increased further, the cache misses start to plateau or even slightly decrease. This could be attributed to an initial ramp-up phase where cache coherence mechanisms are heavily taxed, but as the system scales, cache access patterns stabilize.

#### 5.2.3 Interplay of MPI Processes and Threads

The results for the hybrid MPI + OpenMP + SIMD + Memory Locality approach are particularly revealing. When comparing configurations with the same total number of threads (e.g., 1 process with 32 threads vs. 2 processes with 16 threads each), the variance in CPU cycles and cache misses are observed. Notably, splitting computations across more MPI processes (and consequently fewer OpenMP threads per process) seems to lead to more efficient CPU cycle counts in several configurations. This suggests that distributing the workload among more separate processes can be more efficient than relying heavily on multithreading within a single process. The interplay between processes and threads in a hybrid parallelization approach is complex and can vary depending on the specifics of the hardware and software environment.

#### 5.2.4 Inherent Overheads

Page faults remain relatively stable across different parallelization methods and configurations, with only minor fluctuations. The slight increase in page faults as the number of cores or threads are ramped up can be attributed to the overheads of managing multiple threads or processes. Every new thread or process might introduce its own memory requirements, and as these numbers increase, the chances of encountering page faults can go up, especially if the system's physical memory starts to get saturated.

### 5.3 Comparative Analysis: GPU vs. Best CPU Methods [Extra Credits]

The data elucidates the stark difference in performance between GPU-accelerated matrix multiplication and the best CPU methods. Here's an analysis based on the results provided:

#### 5.3.1 Efficiency of GPU Acceleration

The numbers are unambiguously in favor of the GPU. For the matrix size of $1024 \times 1024$, the GPU's best time was approximately 4.79 ms, in stark contrast to the CPU's best time of 27 ms using the MPI + OpenMP + SIMD + Memory Locality method with 16 processes and 2 threads. This translates to nearly a 5.6x speedup when utilizing the GPU.

For the matrix size of $2048 \times 2048$, the disparity becomes even more pronounced. The GPU finished the task in just 38.44 ms, while the CPU's best time was 165 ms using the same method. This is a speedup of around 4.3x.

#### 5.3.2 Understanding the Performance Gap

The speedup is due to the inherently parallel architecture of the GPU, which is designed specifically to handle these types of matrix operations with thousands of simultaneous threads. CPUs, even with the most optimized methods and setups, have a more generalized architecture and are bound to have certain limitations when it comes to handling such massively parallel tasks.

Another factor is the overhead introduced by the hybrid methods in CPU. The MPI + OpenMP approach, while powerful, introduces certain overheads. GPUs, on the other hand, are designed with minimal overhead for parallel matrix operations.

#### 5.3.3 Implications and Considerations

While the results might tilt heavily in favor of GPUs, it's essential to take into account the power consumption, initial setup costs, and specific task requirements. GPUs might excel at specific tasks like matrix multiplication, but for a varied range of tasks, a hybrid CPU approach might be more versatile.

However, when performance is of the utmost priority, especially for large-scale matrix operations, the GPU clearly stands out as the superior choice.

## 6 Conclusion

Matrix multiplication stands as a cornerstone of modern computational paradigms, especially within the realm of artificial intelligence and deep learning. Its omnipresence in complex AI operations underscores the importance of achieving optimal performance.

The findings illuminate the stark differences in computational efficiency between a naïve approach and one that is finely tuned to hardware and software intricacies. The naïve implementation, while simple, displayed significant inefficiencies, primarily due to non-optimized memory access patterns and

the absence of parallelism. However, as the journey of optimization unfolded, it became evident how alterations in memory access patterns, harnessed through memory locality techniques, could achieve a significant reduction in computation time. The combination of memory locality and SIMD operations accentuated this optimization, unveiling the raw computational prowess of modern CPUs and pushing performance gains even further.

Delving into the world of thread-level parallelism with OpenMP, the study witnessed how leveraging multiple cores could amplify the results achieved by data-level parallelism. The results, quantifiable and significant, were a testament to the compounded effects of utilizing both parallelism types in tandem. Furthermore, the introduction of process-level parallelism through MPI, when combined with prior optimizations, represented the pinnacle of the computational efficiency endeavors. The culmination of these strategies achieved an astonishing 267.54 times speedup, a testament to the power of hybrid optimization techniques.

Profiling served as an indispensable tool throughout this endeavor. Beyond the sheer execution times, it unraveled deeper intricacies in system behavior. It shone a light on the effects of increasing threads and processes, cache behavior in scalable systems, and the subtle complexities that arise when threading and processes interplay in a hybrid environment. Such insights not only provide an understanding of the results but also furnish a roadmap for further refinements.

In sum, this project has not only showcased the paramount importance of optimizing matrix multiplication but has also delineated a systematic approach to achieve it. Through meticulous implementation and profiling, a clear trajectory from a basic, inefficient algorithm to a highly optimized, scalable solution was charted. It serves as a potent reminder that while the foundational algorithms remain unchanged, the ways they are implemented.

As the computational landscape continues to evolve, with emerging architectures and increasingly complex AI models, the lessons gleaned from this project will remain relevant. It underscores the perennial truth that optimization, rooted in a deep understanding of hardware and software, is key to harnessing the full potential of any system.

# Compilation and Execution Instructions

Within the submitted zip file, a folder project2 can be found, this will be the main project folder.

1. Navigate to the folder
2. Run run.sh or simply paste the below sequence of commands:

```
scancel -u {username} # to cancel submitted batch jobs
mkdir build
cd build
cmake ..
make -j4
cd ..
sbatch src/sbatch.sh
sbatch src/sbatch-perf.sh # to display profiling results
squeue # to check batch jobs queue
```

   By default, the above commands will execute the a multiplication between matrices in matrix5.txt and matrix6.txt. If one wish to test other matrices, one can simply modify the appropriate scripts located in src/sbatch.sh. Additional scripts for different matrices might also be found in the parent directory, ../. Make sure relevant matrices can be found in matrices/ directory.

3. Upon completion, one can find the experiment results in the directory: matrices/.

# Appendix

After executing run.sh, the directory structure of matrices is as follows:

```
bash-4.2$ find . -print | sed -e 's;[^/]*/;|____;g;s;____|; |;g'
.
|____matrix1.txt
|____matrix2.txt
|____matrix5.txt
|____matrix4.txt
|____matrix3.txt
|____matrix6.txt
|____matrix8.txt
|____matrix7.txt
|____result-locality.txt
|____result-simd.txt
|____result-openmp.txt
|____result-mpi.txt
|____result-naive.txt
|____result-cuda.txt
```

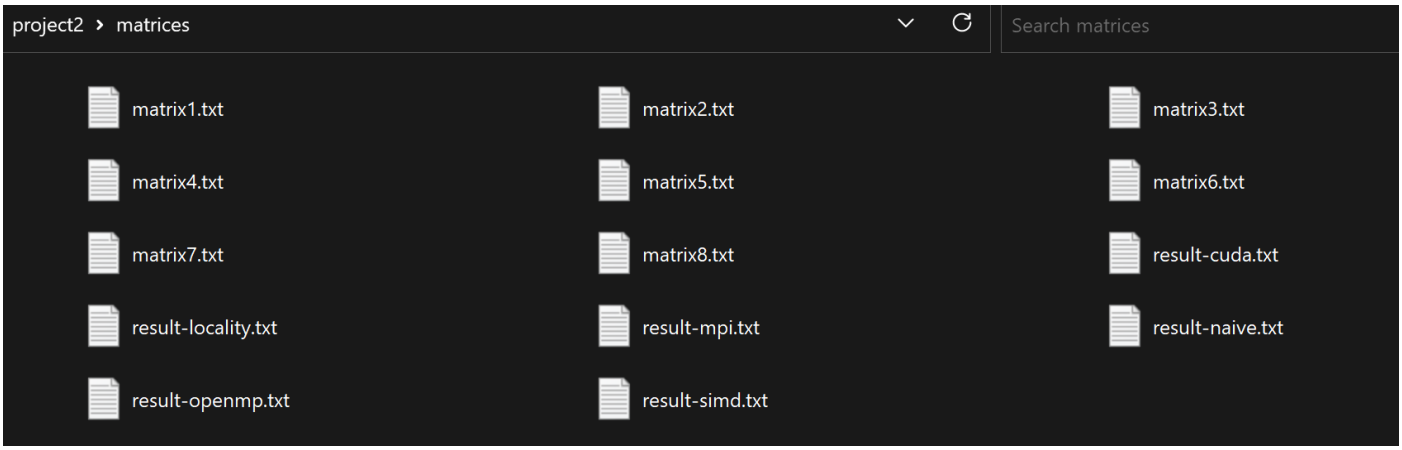


**Figure 3: matrices directory after performing matrix multiplication using all methods, including the GPU one.**

| Methods | Cores | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 | Iteration 6 | Iteration 7 | Iteration 8 | Iteration 9 | Iteration 10 | AVG | BEST |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Naïve | 1 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 |
| Memory Locality | 1 | 549 | 550 | 548 | 548 | 551 | 548 | 549 | 549 | 546 | 547 | 548.5 | 546 |
| SIMD + Memory Locality | 1 | 278 | 280 | 285 | 280 | 284 | 278 | 277 | 278 | 285 | 284 | 280.9 | 277 |
| OpenMP + SIMD + Memory Locality | 1 | 281 | 284 | 284 | 281 | 280 | 281 | 281 | 285 | 281 | 281 | 281.9 | 280 |
| | 2 | 257 | 257 | 259 | 256 | 256 | 256 | 258 | 255 | 256 | 255 | 256.5 | 255 |
| | 4 | 153 | 151 | 150 | 151 | 152 | 153 | 150 | 155 | 154 | 154 | 152.3 | 150 |
| | 8 | 89 | 87 | 90 | 96 | 90 | 98 | 88 | 89 | 98 | 93 | 91.8 | 87 |
| | 16 | 64 | 59 | 66 | 64 | 52 | 57 | 65 | 65 | 65 | 65 | 62.2 | 52 |
| | 32 | 42 | 43 | 45 | 42 | 43 | 46 | 42 | 46 | 44 | 42 | 43.5 | 42 |
| MPI + OpenMP + SIMD + Memory Locality | 1 x 32 | 52 | 48 | 52 | 49 | 53 | 50 | 52 | 51 | 47 | 49 | 50.3 | 47 |
| | 2 x 16 | 48 | 53 | 55 | 39 | 39 | 42 | 50 | 44 | 49 | 38 | 45.7 | 38 |
| | 4 x 8 | 51 | 51 | 39 | 61 | 37 | 51 | 71 | 34 | 36 | 36 | 46.7 | 34 |
| | 8 x 4 | 33 | 31 | 32 | 33 | 32 | 33 | 33 | 33 | 35 | 33 | 32.8 | 31 |
| | 16 x 2 | 29 | 29 | 29 | 30 | 44 | 28 | 28 | 28 | 33 | 27 | 30.5 | 27 |
| | 32 x 1 | 31 | 35 | 34 | 31 | 54 | 32 | 32 | 31 | 34 | 35 | 34.9 | 31 |

**Figure 4: All optimization methods' execution time comparison over 10 iterations for the $1024 \times 1024$ matrices (in milliseconds)**

| Methods | Cores | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 | Iteration 6 | Iteration 7 | Iteration 8 | Iteration 9 | Iteration 10 | AVG | BEST |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Naïve | 1 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 | 8165 |
| Memory Locality | 1 | 549 | 550 | 548 | 548 | 551 | 548 | 549 | 549 | 546 | 547 | 548.5 | 546 |
| SIMD + Memory Locality | 1 | 278 | 280 | 285 | 280 | 284 | 278 | 277 | 278 | 285 | 284 | 280.9 | 277 |
| OpenMP + SIMD + Memory Locality | 1 | 281 | 284 | 284 | 281 | 280 | 281 | 281 | 285 | 281 | 281 | 281.9 | 280 |
| | 2 | 257 | 257 | 259 | 256 | 256 | 256 | 258 | 255 | 256 | 255 | 256.5 | 255 |
| | 4 | 153 | 151 | 150 | 151 | 152 | 153 | 150 | 155 | 154 | 154 | 152.3 | 150 |
| | 8 | 89 | 87 | 90 | 96 | 90 | 98 | 88 | 89 | 98 | 93 | 91.8 | 87 |
| | 16 | 64 | 59 | 66 | 64 | 52 | 57 | 65 | 65 | 65 | 65 | 62.2 | 52 |
| | 32 | 42 | 43 | 45 | 42 | 43 | 46 | 42 | 46 | 44 | 42 | 43.5 | 42 |
| MPI + OpenMP + SIMD + Memory Locality | 1 x 32 | 52 | 48 | 52 | 49 | 53 | 50 | 52 | 51 | 47 | 49 | 50.3 | 47 |
| | 2 x 16 | 48 | 53 | 55 | 39 | 39 | 42 | 50 | 44 | 49 | 38 | 45.7 | 38 |
| | 4 x 8 | 51 | 51 | 39 | 61 | 37 | 51 | 71 | 34 | 36 | 36 | 46.7 | 34 |
| | 8 x 4 | 33 | 31 | 32 | 33 | 32 | 33 | 33 | 33 | 35 | 33 | 32.8 | 31 |
| | 16 x 2 | 29 | 29 | 29 | 30 | 44 | 28 | 28 | 28 | 33 | 27 | 30.5 | 27 |
| | 32 x 1 | 31 | 35 | 34 | 31 | 54 | 32 | 32 | 31 | 34 | 35 | 34.9 | 31 |

**Figure 5: All optimization methods' execution time comparison over 10 iterations for the $2048 \times 2048$ matrices (in milliseconds)**