Performance Study of Large-Scale Square Matrix Multiplication Algorithms on Various Parallel Platforms

Tuong Nguyen Ha
nguyen.hatuong0107@hcmut.edu.vn
Ho Chi Minh University of Technology (HCMUT), VNU-HCM
Ho Chi Minh, Vietnam

Quoc Khai Phan
khai.phanquoc@hcmut.edu.vn
Ho Chi Minh University of Technology (HCMUT), VNU-HCM
Ho Chi Minh, Vietnam

Quynh Huong Vu
huong.vuquynh0611@hcmut.edu.vn
Ho Chi Minh University of Technology (HCMUT), VNU-HCM
Ho Chi Minh, Vietnam

Figure 1: Snapshots taken during algorithm benchmarking. The first image shows the computer screen in the heat of performance evaluation. The second and third images highlight the fan-based cooling system on System 2. The last two images show Tuong Nguyen Ha placing his MacBook inside a 10 m^2 home refrigerator. The takeaway: home-field advantage and a proper cooling system matter a lot, not only in research, but also in surviving this course.

Abstract

This report investigates the performance characteristics of square matrix multiplication across three computational paradigms: **shared memory**, **distributed memory**, and **GPU acceleration**. Four implementations are examined, including a sequential baseline, a cache aware blocked algorithm, a recursive fork join formulation, and a recursive Strassen based variant. CPU oriented implementations rely on OpenMP or OpenMPI, while GPU experiments are conducted on an NVIDIA RTX 2080 Super. An extensive empirical evaluation is performed, covering threshold tuning for recursive methods as well as detailed analyses of cache behavior, memory traffic, and parallel scalability.

Several notable observations emerge from the experimental study. The custom GPU kernel demonstrates up to 2.7× speedup over PyTorch for matrices of identical dimensions. On a Xeon E5 2680 v3 system within the HCMUT SuperNode environment, matrices of size 16384×16384 are multiplied within 35 seconds using shared memory parallelism alone. Additionally, more than 1000 performance plots are generated to support the analysis.

CCS Concepts

• **Computing methodologies** → **Shared memory algorithms**; **Distributed algorithms**; *Concurrent algorithms*; • **Theory of computation** → **Design and analysis of algorithms**; • **Computer systems organization** → *Parallel architectures*; *Distributed architectures.*

Keywords

Square Matrix-Matrix Multiplication, Strassen Algorithm, Fork-Join Pool Model, Cache Oblivious Algorithm, GPU, Shared Memory Algorithm, Distributed Memory Algorithm

1 Introduction

Matrix multiplication constitutes a core computational primitive in scientific computing, numerical linear algebra, data analytics, and modern machine learning systems. A wide range of applications, including partial differential equation solvers, graph analytics, signal processing pipelines, and deep neural network training, rely heavily on efficient dense matrix multiplication. Consequently, the performance characteristics of this operation often dominate overall application runtime and resource utilization.

Over several decades, extensive research efforts have been devoted to improving matrix multiplication performance from both algorithmic and architectural perspectives. On the algorithmic side, classical cubic time formulations have been complemented by asymptotically faster methods such as Strassen's algorithm and its successors, which reduce the number of arithmetic operations at the expense of increased memory traffic and algorithmic complexity. On the architectural side, optimizations such as cache aware blocking, loop reordering, SIMD vectorization, and parallel decomposition have been developed to better exploit modern memory hierarchies and multicore processors. More recently, specialized accelerators including GPUs and TPUs have further reshaped the performance landscape by offering massive parallelism and high memory bandwidth.

[0]The code is available at GEMM Repository

Despite the availability of highly optimized numerical libraries such as OpenBLAS, Intel MKL, cuBLAS, and high level frameworks such as PyTorch, achieving portable and predictable performance across different hardware platforms remains a nontrivial task. Optimized libraries often rely on architecture specific heuristics and opaque auto tuning mechanisms, which can obscure the relationship between algorithmic structure and observed performance. As a result, a clear understanding of how algorithm design interacts with cache behavior, memory bandwidth, synchronization overhead, and parallel scalability continues to be an important objective in both **education** and **system oriented research**.

The contributions of this report are summarized as follows:

- A systematic experimental comparison of square matrix multiplication is conducted across three computational paradigms, namely shared memory distributed memory, and GPU accelerated execution, under a unified evaluation framework.
- Four representative implementations are analyzed, including a sequential baseline, a cache aware blocked algorithm, a recursive fork join formulation, and a recursive Strassen based approach, enabling a detailed study of algorithmic across different hardware architectures.
- An extensive empirical evaluation is performed, incorporating threshold tuning for recursive algorithms as well as in depth analyses of cache behavior, memory traffic, and parallel scalability.
- A custom GPU tiling kernel is evaluated against a widely used high level framework, providing practical insights into the performance gap between hand optimized implementations and general purpose libraries.

This report is organized as follows. Related Work, presented in Section 2, surveys prior research on matrix multiplication optimizations and parallel formulations. Section 3 introduces the computational models and hardware assumptions used throughout the study. The experimental methodology is described in Section 4. Section 5 reports experimental results and provides a detailed performance analysis across all evaluated platforms.

2 Related Work

The study of fast matrix multiplication has progressed through several major milestones over more than five decades, beginning with foundational work in the late 1960s. Winograd's early contribution in 1968 [25] provided techniques to reduce arithmetic operations in inner products, offering ideas that directly influenced later fast matrix multiplication algorithms. The field underwent a revolution in 1969 when Strassen [22] demonstrated that the classical $O(N^3)$ algorithm is not optimal, introducing a divide-and-conquer method achieving $O(N^{\log_2 7}) \approx O(N^{2.81})$. This breakthrough launched the modern study of algebraic complexity.

Winograd subsequently proved in 1971 [26] that multiplying two 2×2 matrices requires at least seven multiplications, establishing the optimality of Strassen's base case. Over the following decade, increasingly sophisticated tensor-based constructions were developed, including Schönhage's 1981 refinement [18], which reduced the computational complexity of matrix multiplication to $O(N^{2.522})$.

A major methodological shift occurred with the Coppersmith-Winograd framework [3], which exploited algebraic structures of higher-order tensors to reduce the asymptotic complexity. Successive refinements by Stothers [21] and Vassilevska Williams [24] pushed the computational exponent below 2.38, culminating in the state-of-the-art complexity of $O(N^{2.38})$ as achieved by Alman and Vassilevska Williams [1]. Although these algorithms set asymptotic records, their extreme algebraic complexity and large constant factors render them impractical for real-world computations.

Beyond asymptotics, a complementary line of research has sought constant-factor improvements and practical performance gains. Karstadt and Schwartz [15] reduced the leading multiplicative constant of the 2×2 base-case from 6 to 5 while preserving the overall asymptotic behavior. More recently, Schwartz and Vaknin [19] introduced pebbling-based communication optimizations, achieving speedups over DGEMM for matrix sizes starting at the value 96.

A significant contemporary development is the emergence of machine learning for algorithm discovery. DeepMind's AlphaTensor framework [6] formulated fast matrix multiplication as a reinforcement learning problem over tensor decomposition spaces. AlphaTensor not only rediscovered Strassen's algorithm but also uncovered novel schemes over both real and finite fields, optimized for specific hardware cost models. These results highlight the potential of reinforcement learning to navigate algorithmic design spaces far beyond human intuition.

Alongside these discoveries, a substantial body of work has examined hardware-aware fast algorithms. These include cache-optimized Strassen-Winograd variants and adaptive hybrid methods that combine classical and fast multiplication based on threshold heuristics. GPU-accelerated implementations have also been explored, including tensor-core variants. In addition, multicore and distributed-memory parallelizations have been developed, emphasizing communication avoidance.

Despite these advances, fast matrix multiplication algorithms remain difficult to deploy in production environments. State-of-the-art asymptotic methods are impractical due to complexity, while practical divide-and-conquer algorithms such as Strassen achieve strong real-world performance only when carefully tuned for hardware characteristics. The present study follows this practical perspective, evaluating Strassen's algorithm under multicore parallelism and different memory hierarchies across multiple systems.

3 Preliminaries

3.1 Atomic Instruction

Atomic operations are those that complete as an indivisible unit, ensuring that no intervening actions can occur during their execution. For example, the protocol under discussion assumes that a write miss can be detected, acquire the bus, and receive a coherence response as a single atomic action[1].

The following snippet illustrates several representative atomic instructions available on the Intel x86-64 architecture:

```
; Examples of atomic operations on Intel x86-64
lock  addq     %rax, (%rbx)        ; Atomic integer addition
lock  cmpxchgq %rdx, -8(%rbp)      ; Compare-and-swap (CAS)
```

[1] See Subsection "An Example Protocol" in Section 5.2, "Centralized Shared-Memory Architectures", in [13].

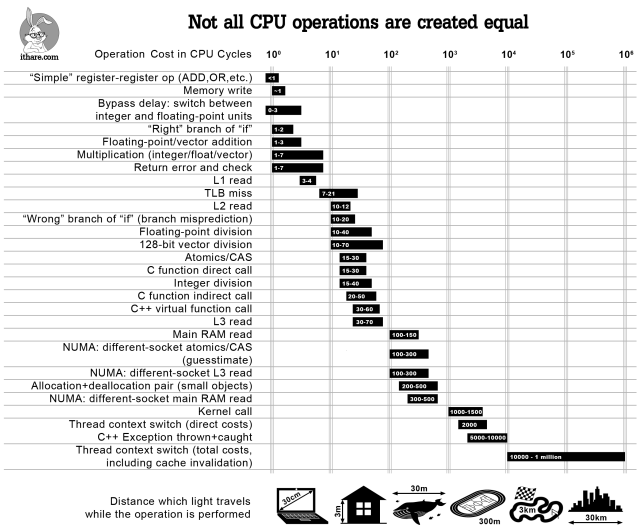


Figure 2: Not all CPU operations are created equal. Figure illustrates the range of instruction throughput (cycle/instruction) on various CPU. (http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/)

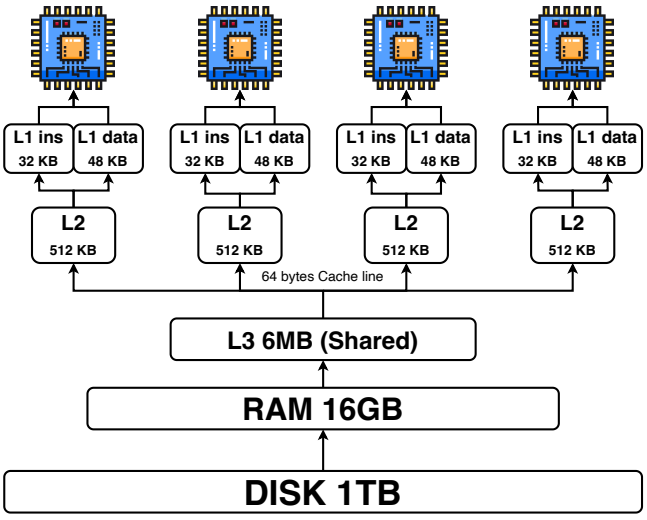


Figure 3: Memory hierarchy of the Intel(R) Core(TM) i5-1038NG7 CPU @ 2.00GHz processor in the 2020 MacBook Pro. Each core integrates a private Level 1 cache, consisting of a 32 KB instruction cache and a 48 KB data cache, as well as a private 512 KB Level 2 cache. The last-level cache Level 3 (LLC) is a 6 MB shared cache. While these caches are on-chip structures, the system also includes 16 GB of SDRAM as main memory and a 1 TB SSD as secondary storage.

3.2 Cache-Aware/Oblivious Algorithms

Modern processors rely heavily on multi-level cache hierarchies to bridge the performance gap between fast CPU cores and relatively slow main memory (RAM, secondary memory, e.g.). The efficiency of many algorithms therefore depends not only on the number of arithmetic operations they perform but also on how effectively they exploit the cache subsystem.

Figure 2 illustrates the throughput (cycles per instruction) of several basic operations on modern processors. Based on this data, we can observe that an L3 cache misses typically propagates to an L2 and L1 miss as well, resulting in a long memory access latency. The total stall penalty can reach approximately 100 clock cycles, whereas basic arithmetic operations such as addition or multiplication require less than one clock cycle. This substantial discrepancy highlights the critical importance of designing memory-efficient algorithms, as memory stalls can easily dominate the overall execution time even when the computational workload itself is relatively small.

> An algorithm to be **cache aware** if it contains parameters (set at either compile-time or runtime) that can be tuned to optimize the cache complexity for the particular cache size and length of cache block. Otherwise, the algorithm is **cache oblivious** [7, 8, 10].

Both cache-aware and cache-oblivious algorithms enhance cache efficiency. Cache-aware algorithms exploit detailed knowledge of the cache to achieve **optimal** performance, whereas cache-oblivious algorithms operate independently of cache parameters and still attain **asymptotically optimal** results.

3.3 Cache Misses

Modern processors are typically equipped with multiple levels of cache (Figure 3 shows an example of Intel core i5 1028NG7 memory system), and each core operates on data at the granularity of bytes rather than individual bits. When data moves across cache levels, however, it is transferred in cache lines rather than as isolated bytes. When a core requires a piece of data, it first searches the closest cache level. If the data is not found, the core consults the translation lookaside buffer abbreviated as TLB [13][2] (see Figure 4). If the entry is also absent there, the request continues to higher memory levels. This phenomenon is known as a **cache misses**. Cache misses are commonly classified into four major categories: ***cold misses***, ***capacity misses***, ***conflict misses***, and ***coherence misses***. Figure 5 presents the hierarchical taxonomy, in which the coherence category is further divided into *true sharing* and *false sharing*.

A ***cold miss*** (or *compulsory miss*) occurs when a memory block is accessed for the first time, and thus must be fetched from a lower level of the hierarchy. Although unavoidable in principle, its impact can be reduced through software-directed techniques such as compiler-guided prefetching or out-of-order execution, as well as hardware mechanisms like speculative prefetching.

A ***capacity miss*** arises when a cache does not have enough space to hold the entire working set of a program. Even with an optimal replacement policy[3], previously used blocks may be evicted

[2] See Section 2.6, "Putting It All Together: Memory Hierarchies in the ARM Cortex-A53 and Intel Core i7 6700", in [13]

[3] Do not confuse with **cache placement policy**. Common examples include Least Recently Used (LRU), Most Recently Used (MRU), and Least Frequently Used (LFU).

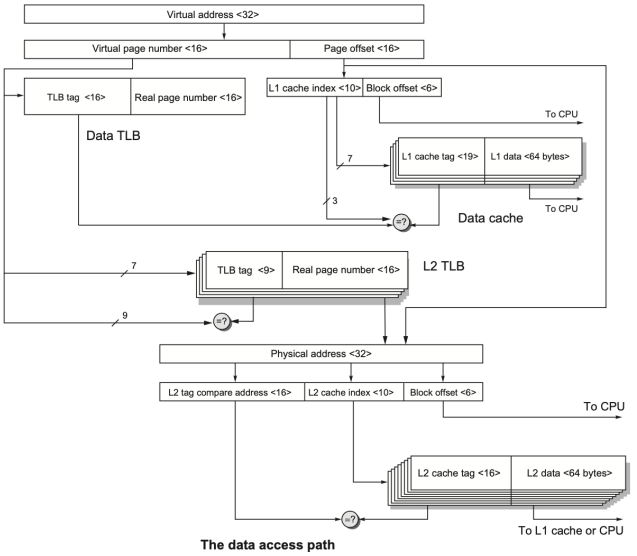


Figure 4: This figure illustrates the data path followed when a core issues a request for a 32-bit virtual address, which is used to index both the TLB and the cache hierarchy. Valid bits and protection bits for the TLB and cache structures are omitted for clarity.

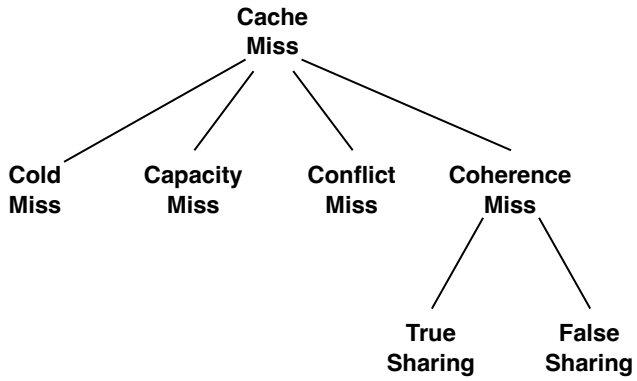


Figure 5: Cache misses taxonomy on multi-core processor.

prematurely and must be reloaded later when they are referenced again.

A ***conflict miss*** (see Subsection 5.2) occurs due to cache placement restrictions[4]. Such misses typically arise in direct-mapped and set-associative caches, where multiple memory addresses are constrained to map to the same cache set. As a result, a particular set may become full even though other sets in the cache still contain empty slots.

Figure 6 illustrates this effect for an L1-L2 hierarchy in which both levels use direct mapping. Cache lines in L2 are grouped into

[4]Do not confuse with **cache replacement policy**. This topic related to direct-mapped, fully associative, and set-associative caches correspond to different cache placement organizations.

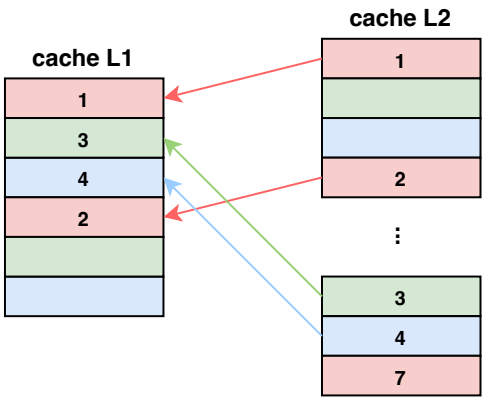


Figure 6: Illustration of address mapping from the L2 cache to the L1 cache under a direct-mapped placement policy. In this example, the red cache line 7 in L1 remains unused.

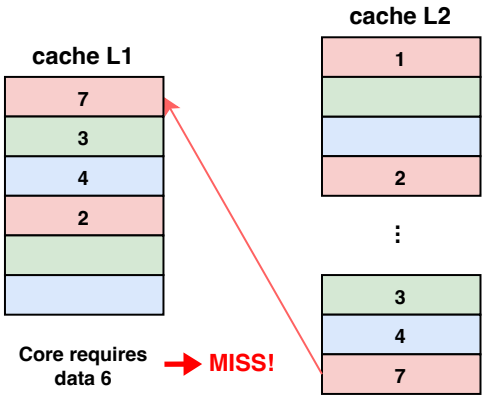


Figure 7: Although the L1 cache still contains unused entries, the required red cache line 7 must occupy its mapped position, causing cache line 1 to be evicted. If cache line 1 is later needed again, this replacement results in a conflict miss.

three color-coded regions (red, blue, and green), each representing the set of L2 lines that map to a specific position in L1. Although numerous L1 entries remain unused, all entries corresponding to the red region have already been occupied. When the processor accesses another address belonging to the red region, the new line must replace an existing red-mapped line in L1 (Figure 7), potentially according to a replacement policy such as least frequently used. If the evicted line is needed again soon, this replacement results in a *conflict miss*.

Coherence misses arise specifically from interactions among processors in shared-memory multiprocessors. Before diving into *true sharing* and *false sharing*.

A simple scenario is considered in Figures 8–10, which depict a representative **coherence misses** on a dual-core system implementing the MSI (Modified, Shared, Invalid) cache-coherence protocol. In Figure 8, each core contains a private cache backed by a shared lower-level memory. A cache line consisting of two words,

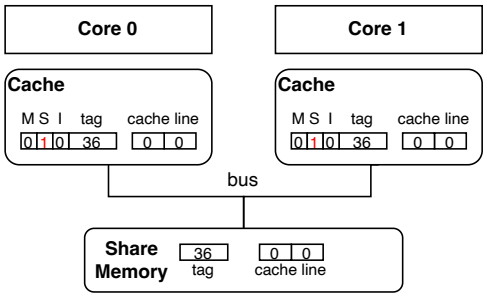


Figure 8: A dual-core system implements the MSI (Modified, Shared, Invalid) cache-coherence protocol. Each core contains a private cache backed by a shared lower-level memory. A cache line consisting of two words, identified by tag value 36, is initially present in both private caches in the *Shared* state, represented by the MSI vector (0,1,0).

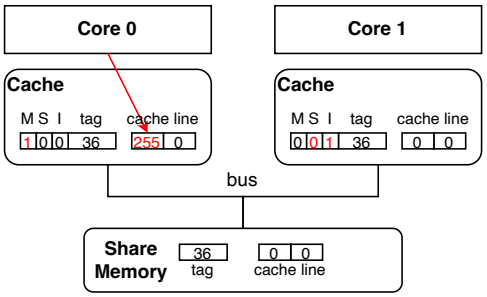


Figure 9: When Core 0 writes 255 to the first word of the line, its copy transitions to the *Modified* state (1,0,0). The coherence protocol invalidates the corresponding copy in Core 1, whose state becomes (0,0,1).

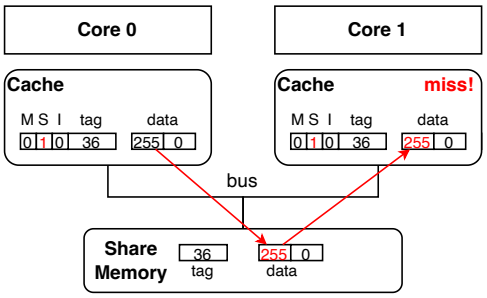


Figure 10: When Core 1 attempts to read the same cache line, the access results in a coherence misses because its local copy is invalid. To satisfy the request, the coherence controller retrieves the updated data from the modified copy, writes it back to the shared cache, and forwards it to Core 1. Both caches then hold the line in the *Shared* state (0,1,0).

identified by tag value 36, is initially present in both private caches in the *Shared* state, represented by the MSI vector (0,1,0).

When Core 0 writes 255 to the first word of the line, its copy transitions to the *Modified* state (1,0,0). The coherence protocol invalidates the corresponding copy in Core 1, whose state becomes (0,0,1) (see Figure 9). Later, when Core 1 attempts to read the same cache line, the access results in a **coherence misses** because its local copy is invalid. To satisfy the request, the coherence controller retrieves the updated data from the modified copy, writes it back to the shared cache, and forwards it to Core 1. Both caches then hold the line in the *Shared* state (0,1,0), as shown in Figure 10.

Coherence misses comprise two distinct categories *true sharing* and *false sharing* [13][5]. *True sharing* occurs when multiple processors actually share and modify the same data. In an invalidation-based protocol, a write to a shared block triggers invalidations in other caches. Subsequent reads of the modified word by another processor result in misses that carry updated data, representing genuine communication. *False sharing*, by contrast, occurs when processors access different words that reside in the same cache line. Because coherence protocols operate at cache-line granularity, a write to one word may invalidate the entire line, causing misses on data that are logically independent. These misses do not transfer new information and would disappear under a word-granular coherence mechanism.

3.4 Floating-Point Inaccuracy in Matrix Multiplication

Matrix multiplication using `float32` arithmetic inevitably introduces rounding errors due to the limited precision of IEEE 754 single-precision floating-point representation. Each multiplication step generates a rounding error, and the subsequent accumulation of partial sums further amplifies this effect. For large N, the number of accumulation operations grows quadratically, making numerical instability increasingly observable.

During our initial experiments, we observed discrepancies that at first appeared to be implementation bugs. However, upon detailed inspection, we confirmed that these deviations were caused by floating-point accumulation error rather than incorrect logic. To mitigate this issue, we employed the Kahan summation algorithm (see Algorithm 1) in selected base-case implementations. Kahan summation compensates for lost low-order bits during addition and therefore significantly improves numerical stability in scenarios involving long accumulation chains[6].

3.5 Race Condition

> Where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable count. To make such a guarantee, we require that the processes be synchronized in some way. [20]

[5] See Section 5.3, "Performance of Symmetric Shared-Memory Multiprocessors," in [13].

[6] The authors initially spent a considerable amount of time debugging the implementation, mistakenly assuming a logic error when the root cause was floating-point inaccuracy.

Algorithm 1: Kahan Dot Product for `float32` Arithmetic

Input: $\mathbf{a}, \mathbf{b}$ vectors of length N
Output: $s = \sum_{k=0}^{N-1} a_k \cdot b_k$

Procedure *KahanDot*($\mathbf{a}, \mathbf{b}$)
  $s \leftarrow 0.0$; `// accumulated sum`
  $c \leftarrow 0.0$; `// compensation for lost low-order bits`
  for $k = 0$ **to** $N - 1$ **do**
    $y \leftarrow a_k \cdot b_k - c$;
    $t \leftarrow s + y$;
    $c \leftarrow (t - s) - y$;
    $s \leftarrow t$;
  end
  return s;

A **race condition** occurs when multiple execution contexts access and manipulate shared data concurrently, and the final outcome depends on the particular interleaving of these operations. When two or more processes modify a shared variable without proper coordination, the resulting value becomes nondeterministic. Preventing such anomalies requires explicit synchronization mechanisms that ensure controlled access to shared memory and impose well-defined ordering constraints on concurrent operations.

Early attempts to address race conditions relied on *software-based* techniques. Classical algorithms such as Peterson's solution illustrate that mutual exclusion can be achieved using only shared variables and structured busy waiting. Although elegant in principle, such techniques implicitly assume sequentially consistent memory behavior. Contemporary processors employ out-of-order execution, speculative evaluation, and compiler reordering, which invalidate these assumptions and render purely software-based methods unreliable on modern hardware. Higher-level constructs such as locks, semaphores, and monitors therefore encapsulate the necessary synchronization logic while depending internally on *hardware-supported* atomicity.

Modern processors provide low-level primitives that form the foundation of reliable synchronization. Atomic instructions such as `test_and_set`, `compare_and_swap`, or `fetch_and_add` allow a read-modify-write sequence to be executed as an indivisible operation. These primitives are indispensable for implementing spinlocks, mutexes, and lock-free data structures [7] that must operate correctly even in the presence of concurrent access from multiple cores. In addition to atomicity, contemporary multiprocessor systems must also enforce ordering constraints on memory operations. Architectures differ in their consistency guarantees. Strongly ordered systems make all memory modifications visible in program order across processors, whereas weakly ordered systems permit both the processor and compiler to reorder loads and stores for performance reasons, potentially delaying visibility of writes across cores.

[7] My tutor, Thanh Dang Diep (Ong trum Parallel Computing cua Viet Nam), who conducted his doctoral research on non-blocking data structures for distributed-memory systems [5], did not require or request that this work be cited. This statement is made voluntarily. I swear!

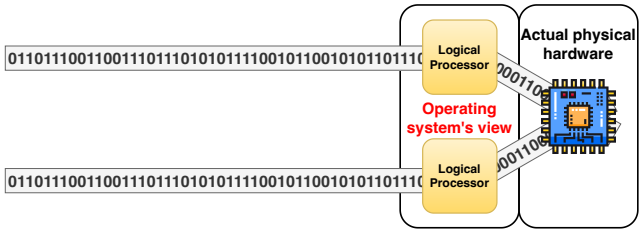


Figure 11: From operating system point of view, `0` and `1` are instruction stream of a process executing on hardware threads, however, those cores actually logical (virtual) core.

To control such reordering, hardware provides memory barriers, also known as memory fences, which explicitly constrain the relative ordering of memory operations. A memory barrier guarantees that all accesses preceding it complete before any subsequent accesses are issued. This property is essential when constructing correct lock-free algorithms or enforcing consistent views of shared data in the presence of weak memory models. Modern operating systems and language runtimes combine these hardware primitives with software abstractions so that programmers can reason about synchronization without directly managing low-level ordering semantics.

In contemporary systems, therefore, the prevention of race conditions requires a coordinated interplay between software-level design and hardware-level guarantees. Correct synchronization depends not only on mutual exclusion but also on ensuring the visibility and ordering of memory operations across processors. Robust concurrent programs rely on both aspects: high-level constructs that express the intended coordination pattern and low-level mechanisms that enforce atomicity and consistency at the architectural level.

3.6 Thread Scheduling

A **thread** represents the fundamental unit of execution scheduled on a physical core, or on a logical core in processors that support simultaneous multithreading (SMT) [8], such as hyper-threading (see Figures 11). While software can create an arbitrary number of **"software thread"**, (e.g. to wait for I/O operations or to handle requests from external devices as commonly seen in server applications), the number of **"hardware threads"** is fixed and determined by the underlying architecture of the system. For instance, the Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz provides eight cores and eight hardware threads (one thread per core), indicating the absence of hyper-threading. In contrast, the Intel(R) Core(TM) i5-1038NG7 CPU @ 2.00GHz 3 exposes four cores and eight hardware threads (two threads per core), reflecting support for hyper-threading.

In the Linux kernel [23], the term **task** is employed, partly due to the absence of a strict distinction, to refer interchangeably to a process or a thread. A task may represent a single-threaded process (illustrated on the left side of Figure 12), an individual thread within a multithreaded process (right side of Figure 12), or even a kernel thread [9]. The process control block (PCB) is implemented as

[8] Intel website
[9] kernel thread is also a "software thread"

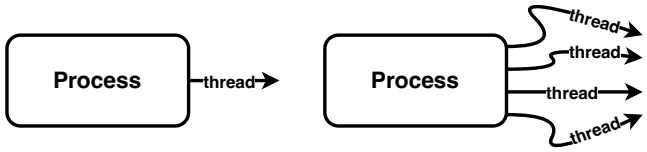


Figure 12: Illustration of a single-threaded process (left) and threads within a multithreaded process (right).

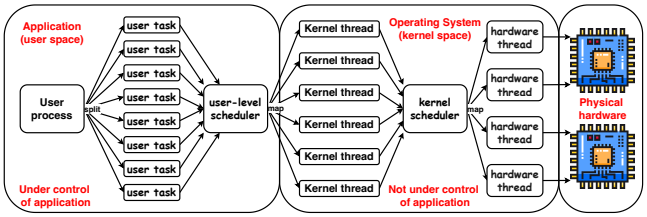


Figure 13: A single process can create multiple user tasks ("software threads"), and these tasks are first scheduled by the user-level scheduler before being mapped onto kernel threads. Once the mapping occurs, the operating system's kernel scheduler takes over and schedules these kernel threads onto the available hardware contexts ("hardware threads").

the structure `task_struct`, defined in `include/linux/sched.h`. Scheduling decisions are based on the "schedulability" of tasks, an abstract property determined by factors such as task priority, cache and core locality, and other architecture-specific metrics.

When multiple software threads is created by a process, these threads are initially scheduled by a user-level scheduler. During this stage, each software thread is mapped to a corresponding kernel thread, as illustrated in the first block of Figure 13. All of these operations are carried out entirely in user space. Once the mapping is completed, the resulting kernel threads are handled by the kernel scheduler, which determines their execution order and placement on the underlying hardware resources. Ultimately, each kernel thread is mapped onto a hardware thread, as shown in the second block of Figure 13.

4 Methodology

In this section, we present the algorithms and parallelization strategies used to implement Square Matrix Multiplication (from now on, we will call `MatMul`) across multiple computing paradigms. We start with the classical **sequential algorithm** and progressively extend it to **parallel models**, including *shared-memory systems* (SMS), *distributed-memory clusters* (DMS), *GPU-accelerated computing*, and a *hybrid approach* combining SMS and DMS. For each method, we describe the computation model, provide pseudo code, and discuss the theoretical performance characteristics, including computational complexity, communication overhead, and expected scalability.

Algorithm 2: Sequential Matrix Multiplication (ijk Order)

Input: A, B are $n \times n$ matrices
Output: $C = A \times B$
Initialize: $C[i, j] \leftarrow 0$ for all i, j
Procedure *MatMul_IJK(A, B)*
  $n \leftarrow dim(C)$;
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $temp \leftarrow 0$;
      for $k \leftarrow 1$ **to** n **do**
        $temp \leftarrow temp + A[i, k] \cdot B[k, j]$;
      end
      $C[i, j] \leftarrow temp$;
    end
  end
  return C ;

Algorithm 3: Sequential Matrix Multiplication (ikj Order)

Input: A, B are $n \times n$ matrices
Output: $C = A \times B$
Initialize: $C[i, j] \leftarrow 0$ for all i, j
Procedure *MatMul_IKJ(A, B)*
  $n \leftarrow dim(C)$;
  for $i \leftarrow 0$ **to** $n - 1$ **do**
    for $k \leftarrow 0$ **to** $n - 1$ **do**
      $lhs \leftarrow A[i, k]$;
      for $j \leftarrow 0$ **to** $n - 1$ **do**
        $C[i, j] \leftarrow C[i, j] + lhs \cdot B[k, j]$;
      end
    end
  end
  return C ;

4.1 Sequential Matrix Multiplication

4.1.1 Classical Algorithm. We consider two loop-ordering variants of matrix multiplication: `ijk` (Algorithm 2) and `ikj` (Algorithm 3), where `i` and `j` index the row of A and the column of B, respectively, and `k` iterates over the corresponding elements of row `i` in A and column `j` in B.

At first glance, the two multiplication orders `ijk` and `ikj` appear nearly identical, as both perform n^3 multiplications and $n^2(n-1)$ additions. Thus, their time complexity is the same: $O(n^3)$.

Although the `ijk` ordering is the more straightforward formulation, its memory-access pattern is significantly less cache-friendly. As illustrated in Figure 14, a matrix is typically stored in memory as a contiguous block in row-major order. However, under the `ijk` traversal, the algorithm repeatedly revisits elements of matrix B in a non-contiguous pattern. This leads to poor spatial locality, especially when matrix B does not fit entirely within the cache, and consequently results in a higher cache-miss rate (see 3.3).

In contrast, the `ikj` ordering is considerably more cache-friendly. As shown in Figure 15, the traversal reuses the value $A[i, k]$, loaded

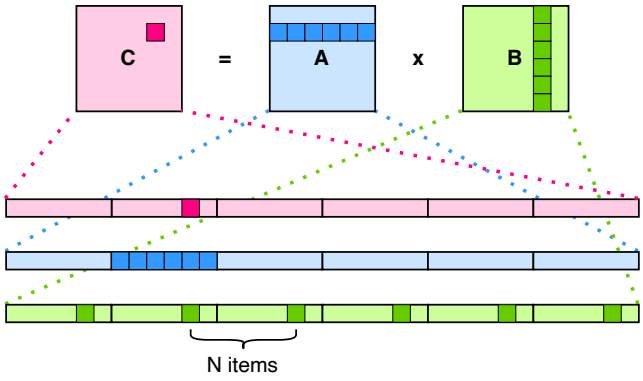


Figure 14: Illustration of the memory access pattern for the `ijk` loop ordering. The figure shows that elements of matrix B are accessed non-contiguously, leading to poor spatial locality and higher cache misses rates.

once into a register, while streaming through a contiguous row-major block of $B[k, *]$. This improves spatial locality for matrix B and reduces cache-miss rates when updating row `i` of the output matrix C.

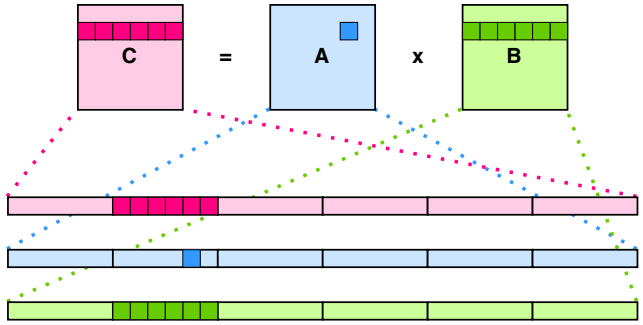


Figure 15: Illustration of the memory access pattern for the `ikj` loop ordering. The figure highlights how the value $A[\mathbf{i}, \mathbf{k}]$ is reused while accessing a contiguous row segment $B[\mathbf{k}, *]$, resulting in better spatial locality and fewer cache misses when updating row `i` of the output matrix C.

4.1.2 Strassen Sequential Matrix Multiplication. In this section, we present Strassen Algorithm in Sequential version (Algorithm 4) Strassen's algorithm, introduced in 1969 [22], has complexity

$$\Theta\left(n^{\lg(7)}\right),$$

where $\lg(7) \approx 2.807$. By contrast, the classical matrix multiplication algorithm requires

$$\Theta(n^3)$$

operations.

Correctness of Strassen's Algorithm - Backward Proof

We first recall the classical 2×2 matrix multiplication. Let

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \qquad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}.$$

Algorithm 4: Sequential Strassen Matrix Multiplication

Input: A, B are $n \times n$ matrices; threshold T
Output: $C = A \times B$

Procedure *StrassenSeq*(C, A, B, n, T)
  `// Base case: switch to classical multiplication`
  if $n \le T$ **then**
    $C \leftarrow$ MatMul_IKJ(A, B);
    return;
  end
  `// Split matrices into four subblocks`
  Partition $A \rightarrow A_{00}, A_{01}, A_{10}, A_{11}$;
  Partition $B \rightarrow B_{00}, B_{01}, B_{10}, B_{11}$;
  `// Allocate seven Strassen intermediate matrices`
  Allocate $M_1..M_7$ (size $n/2 \times n/2$);
  Allocate temporary blocks T_1, T_2;
  `// Compute the 7 Strassen multiplications`
  $T_1 \leftarrow A_{00} + A_{11}$;
  $T_2 \leftarrow B_{00} + B_{11}$;
  StrassenSeq$(M_1, T_1, T_2, n/2, T)$;
  $T_1 \leftarrow A_{10} + A_{11}$;
  StrassenSeq$(M_2, T_1, B_{00}, n/2, T)$;
  $T_2 \leftarrow B_{01} - B_{11}$;
  StrassenSeq$(M_3, A_{00}, T_2, n/2, T)$;
  $T_2 \leftarrow B_{10} - B_{00}$;
  StrassenSeq$(M_4, A_{11}, T_2, n/2, T)$;
  $T_1 \leftarrow A_{00} + A_{01}$;
  StrassenSeq$(M_5, T_1, B_{11}, n/2, T)$;
  $T_1 \leftarrow A_{10} - A_{00}$;
  $T_2 \leftarrow B_{00} + B_{01}$;
  StrassenSeq$(M_6, T_1, T_2, n/2, T)$;
  $T_1 \leftarrow A_{01} - A_{11}$;
  $T_2 \leftarrow B_{10} + B_{11}$;
  StrassenSeq$(M_7, T_1, T_2, n/2, T)$;
  `// Combine the results`
  $C_{00} \leftarrow M_1 + M_4 - M_5 + M_7$;
  $C_{01} \leftarrow M_3 + M_5$;
  $C_{10} \leftarrow M_2 + M_4$;
  $C_{11} \leftarrow M_1 - M_2 + M_3 + M_6$;
  Assemble C from $C_{00}, C_{01}, C_{10}, C_{11}$;

The standard multiplication $C = A \times B$ is

$$C = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

Strassen's algorithm replaces the eight scalar multiplications above by seven product:

$$\begin{aligned}
M_1 &= (a+d)(e+h),\\
M_2 &= (c+d)e,\\
M_3 &= a(f-h),\\
M_4 &= d(g-e),\\
M_5 &= (a+b)h,\\
M_6 &= (c-a)(e+f),\\
M_7 &= (b-d)(g+h).
\end{aligned}$$

The entries of C are then reconstructed as:

$$C = \begin{bmatrix} c_{11} = M_1 + M_4 - M_5 + M_7 & c_{12} = M_3 + M_5 \\ c_{21} = M_2 + M_4 & c_{22} = M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

Direct verification for 2×2 matrices. We now prove that these formulas agree with the classical product for arbitrary scalars a, b, c, d, e, f, g, h.

For c_{11} we use the definitions of M_1, M_4, M_5, M_7 and obtain

$$\begin{aligned}
c_{11} &= M_1 + M_4 - M_5 + M_7\\
&= (ae + ah + de + dh) + (dg - de) - (ah + bh) + (bg + bh - dg - dh)\\
&= ae + \underbrace{ah - ah}_{=0} + \underbrace{de - de}_{=0} + \underbrace{dh - dh}_{=0} + \underbrace{dg - dg}_{=0} + bg + \underbrace{bh - bh}_{=0}\\
&= ae + bg.
\end{aligned}$$

This coincides with the classical formula $c_{11} = ae + bg$.

Similarly, for c_{12} we have

$$\begin{aligned}
c_{12} &= M_3 + M_5\\
&= a(f-h) + (a+b)h\\
&= af - ah + ah + bh\\
&= af + bh
\end{aligned}$$

which agrees with the standard expression $c_{12} = af + bh$.

For c_{21} we obtain

$$\begin{aligned}
c_{21} &= M_2 + M_4\\
&= (c+d)e + d(g-e)\\
&= ce + de + dg - de\\
&= ce + dg
\end{aligned}$$

which matches $c_{21} = ce + dg$.

Finally, for c_{22} we get

$$\begin{aligned}
c_{22} &= M_1 - M_2 + M_3 + M_6\\
&= (a+d)(e+h) - (c+d)e + a(f-h) + (c-a)(e+f)\\
&= (ae + ah + de + dh) - (ce + de)\\
&\quad + (af - ah) + (ce + cf - ae - af)\\
&= \underbrace{ae - ae}_{=0} + \underbrace{ah - ah}_{=0} + \underbrace{de - de}_{=0} + dh + \underbrace{ce - ce}_{=0} + cf\\
&= cf + dh,
\end{aligned}$$

which coincides with the classical formula $c_{22} = cf + dh$.

We have therefore shown that for arbitrary scalars a, b, c, d, e, f, g, h, Strassen's scheme produces the same four entries c_{ij} as the standard 2×2 matrix product AB.

Arithmetic Operation Reduction and Efficiency

The standard algorithm for multiplying two $n \times n$ matrices requires n^3 scalar multiplications and $n^3 - n^2$ scalar additions, for a total arithmetic operation count of $2n^3 - n^2$.

In Strassen's paper [22], introduced an algorithm stated for square matrices—based on a new way of multiplying 2×2 matrices using only 7 multiplications and 18 additions/subtractions.

If one level of Strassen's algorithm is applied to a 2×2 block matrix whose elements are $(n/2) \times (n/2)$ submatrices, and the standard algorithm is used for the seven block matrix multiplications, the total arithmetic operation count becomes

$$7\left(2\left(\frac{n}{2}\right)^3 - \left(\frac{n}{2}\right)^2\right) + 18\left(\frac{n}{2}\right)^2 = \frac{7}{4}n^3 + \frac{11}{4}n^2.$$

The ratio of this operation count to that required by the standard algorithm is therefore

$$\frac{7n^3 + 11m^2}{8n^3 - 4n^2}$$

which approaches $7/8$ as $n \to \infty$, implying that for sufficiently large matrices, a single level of Strassen's construction yields a 12.5% improvement over the classical matrix multiplication algorithm.

Although Strassen's algorithm reduces the number of multiplications and improves the complexity, the increase in additions and subtractions can still lead to significant practical overhead. We remark that while the complexity is not affected by these extra operations, reducing them can still have substantial practical performance benefits.

4.2 Shared-Memory Parallel Matrix Multiplication

In this subsection, the sequential `ikj`-based matrix multiplication algorithm is parallelized on a shared-memory architecture. The initial approach employs a straightforward work-sharing strategy, denoted as the **vanilla** version. The terminology emphasizes the simplicity of the method, analogous to vanilla ice cream, which serves as a classic and baseline flavor. Subsequently, a **Fork-Join** model is introduced, representing a cache-oblivious algorithm. Finally, the **Strassen** algorithm is presented as an advanced method for matrix multiplication.

4.2.1 *Vanilla Parallel Matrix Multiplication Algorithm.*

Rule of Thumb: Parallelize outer loops rather than inner loops. Parallelizing the inner `k`- and `j`-loops introduces significant synchronization and scheduling overhead (see sec. 3.6), which becomes disproportionately large because the amount of work per iteration is extremely small. Moreover, inner-loop parallelization breaks the cache locality benefits of the `ikj` traversal, as multiple threads would compete for nearby cache lines in B and C, leading to false sharing (see Section. 3.3) and a higher cache-miss rate. These effects collectively make inner-loop parallelism inefficient on shared-memory systems.

Why is the `k`*-loop not parallelized in this design?* Parallelizing the `k`-loop causes multiple threads to update the same row elements of the output matrix $C[\mathtt{i}, *]$ simultaneously, producing data races (see sec. 3.5) unless atomic operations (see sec. 3.1) or explicit synchronization are used. Both options add heavy contention and memory

Algorithm 5: Parallel Matrix Multiplication on Shared-Memory (Vanilla version)

Input: A, B are $n \times n$ matrices
Output: $C = A \times B$

```
1  Procedure VanillaMatMul(C, A, B)
2      n ← dim(C) ;
3      for i ← 0 to n − 1 do in parallel
4          for k ← 0 to n − 1 do
5              a ← A[i, k] ;
6              for j ← 0 to n − 1 do
7                  C[i, j] ← C[i, j] + a · B[k, j] ;
8              end
9          end
10     end
11     return C;
```

traffic. Additionally, each k-iteration writes to the same small subset of cache lines in C, so threads would repeatedly evict one another's partial sums, degrading spatial and temporal locality. The resulting false sharing eliminates any potential speedup.

Why is the `j`*-loop not parallelized in this design?* Although iterations of the `j`-loop write to distinct columns of C, the loop body performs only a single fused multiply-add utilizing a value already loaded into a register.Consequently, the iteration granularity becomes excessively fine, preventing effective amortization of synchronization and scheduling overheads (see Sec. 3.6). Furthermore, multiple threads concurrently traversing adjacent columns access cache lines that are physically close, increasing memory traffic and creating false sharing effects. As a result, parallelizing the `j`-loop leads to slower execution rather than improvement.

4.2.2 Fork-Join Parallel Matrix Multiplication Algorithm. Early implementations of high-performance matrix multiplication initially did not exploit tiling (or blocking) effectively, due to the complexity of managing data across memory hierarchies. However, with the emergence of multi-level cache architectures, the tiling technique became a practical and critical optimization for improving data locality.

Figure 16 illustrates the computational pattern of the resulting tiled matrix C. In this figure, the $N \times N$ matrix is partitioned into square blocks of size $bs \times bs$, each identified by block indices (`ih`, `jh`). Within each block, individual elements are indexed by (`il`, `jl`), and the arrows indicate the order in which elements are accessed during the computation, highlighting the traversal pattern that maximizes cache reuse.

To understand intuitively why blocking improves cache locality, consider the two illustrative examples in Figures 17 and 18, which show the number of memory accesses required to compute n elements of a row of C. Without tiling, we need n writes to C, n reads from A, and n^2 reads from B, for a total of $n^2 + 2n$ memory operations. With tiling using a block size $k = \sqrt{n}$ (chosen so that the example still computes n elements of C), we need n writes to C, $n\sqrt{n}$ reads from A, and $n\sqrt{n}$ reads from B, for a total of $n\left(1 + \frac{2}{\sqrt{n}}\right)$

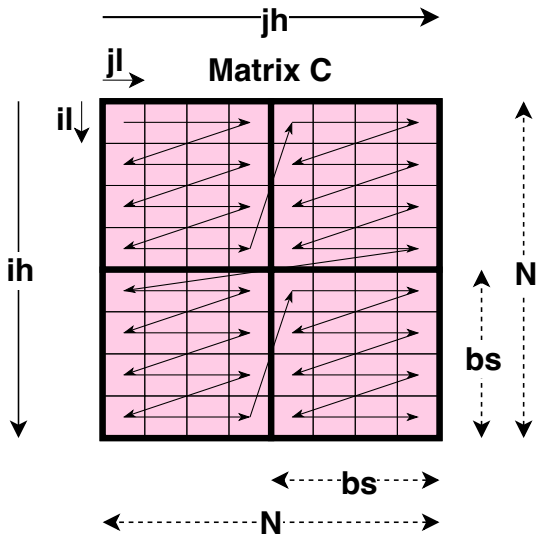


Figure 16: $N \times N$ matrix is partitioned into square blocks of size $bs \times bs$, each identified by block indices (`ih`, `jh`). Within each block, individual elements are indexed by (`il`, `jl`), and the arrows indicate the order in which elements are accessed during the computation, highlighting the traversal pattern that maximizes cache reuse.

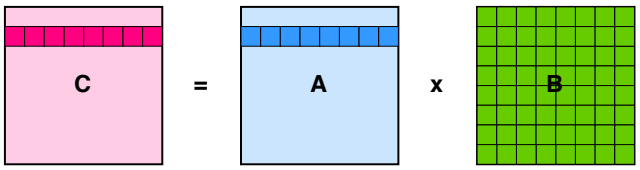


Figure 17: Memory access pattern when computing a row of C without tiling. Each iteration loads an entire row of A and an entire column of B, resulting in $n^2 + 2n$ total memory accesses.

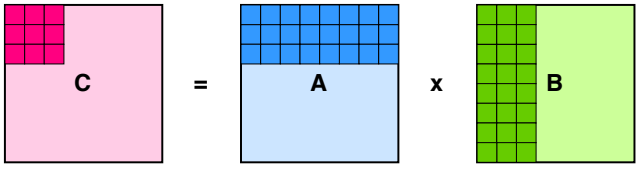


Figure 18: Memory access pattern when computing a row of C using tiling with block size $k = \sqrt{n}$ (chosen so that the example still computes n elements of C). Each tile reuses data inside the block, reducing the number of memory accesses to $n\left(1 + \frac{2}{\sqrt{n}}\right)$.

memory operations. This demonstrates that tiling reduces memory traffic considerably when computing n elements of C.

Algorithm 6: Tiled Matrix Multiplication

Input: A, B are $n \times n$ matrices, bs is block size
Output: $C = A \times B$

```
Procedure TiledMatMul(C, A, B, bs)
    n ← dim(C) ;
    for ih ← 0 to n − 1 step bs do in parallel
        for jh ← 0 to n − 1 step bs do in parallel
            for kh ← 0 to n − 1 step bs do
                for il ← 0 to bs − 1 do
                    for kl ← 0 to bs − 1 do
                        lhs ← A[ih + il, kh + kl] ;
                        for jl ← 0 to bs − 1 do
                            c ← C[ih + il, jh + jl] ;
                            C[ih + il, jh + jl] ←
                              c + lhs · B[kh + kl, jh + jl] ;
                        end
                    end
                end
            end
        end
    end
    return C;
```

This approach was formalized in Level-3 BLAS [10] (called "BLAS3 below" [11]) and has since been studied extensively as a model for cache-aware algorithm design (see Sec. 3.2). Algorithm 6 provides a pseudo-code representation of the tiled computation strategy illustrated in the figure.

However, modern processors include multiple cache levels of various sizes. Since loop-based tiling achieves locality only for a single cache level per chosen block size, it effectively introduces one hyperparameter per cache level. Thus, if a system has three cache levels, achieving optimal performance may require tuning three block-size parameters, which require 12 for-loops to implement. Such tuning becomes burdensome when designing libraries intended to run efficiently across a wide variety of hardware architectures.

Between roughly 1996 and 1999, Charles E. Leiserson (see Figure 19) and his students introduced the concept of cache-oblivious algorithms [9, 10][11] (see Sec. 3.2). Instead of relying on loop based tiling, cache-oblivious algorithms employ a recursive divide and conquer strategy that implicitly forms blocks. This paradigm has been applied to many algorithms, including matrix multiplication, which will be referred to as the Fork–Join algorithm in what follows. Unlike loop based tiling, which requires tuning three hyperparameters and often demands detailed knowledge of the memory hierarchy, Fork–Join requires tuning only a single parameter, namely the threshold used as the base case for recursion. Although it does not guarantee fully optimal cache behavior, it achieves performance close to optimal (asymptotically optimal) and reduces

[10]The adoption of tiling in BLAS3 was pioneered by Berry et al. (1986), Calahan (1986), and Gallivan, Jalby, and Meier (1986), who addressed performance limitations posed by multi-level memory hierarchies.
[11]The correctness of the algorithm is accepted in the literature; see Frigo et al. (1999, 2012).

Figure 19: Charles E. Leiserson The G.O.A.T

the tuning problem from a three dimensional search space to a one dimensional one.

In practice, the threshold should be chosen carefully to avoid both extremes. Setting the threshold too small increases the recursion depth, which introduces significant overhead from function calls and scheduling (see Subsection 3.6). Conversely, setting the threshold too large reduces the benefits of recursive decomposition and can cause blocks to exceed the cache capacity, resulting in frequent cache misses. Optimal performance is typically achieved by balancing these factors, selecting a threshold that minimizes recursion overhead while keeping working blocks small enough to fit efficiently in the cache. Empirical tuning on the target system is usually necessary to determine this optimal value.

4.2.3 Strassen Matrix Multiplication Algorithm. In this section, we present Strassen Algorithm in Shared Memory version using OMP (Algorithm 8).

Key Ideas in Parallelism. Because the computations of M_1 through M_7 in Strassen's algorithm are independent, we decided to parallelize these seven subproblems. However, this approach introduces a drawback: Strassen requires substantial temporary workspace, especially when allocating intermediate matrices for each recursive call. To mitigate this overhead, we use private temporary buffers T_1 and T_2 for each task, and immediately release these buffers once the task finishes.

Key Idea in Implementation. Consider the cutoff criterion: "The performance of Strassen's algorithm varies across architectures, so an effective cutoff criterion must be adaptable" [14]. In our work, we determine the cutoff threshold empirically by benchmarking Strassen's algorithm while varying the number of hardware threads available on our system.

4.3 GPU-Accelerated Matrix Multiplication

The proposed approach implements dense matrix multiplication on the GPU using a tiled computation strategy combined with asynchronous host-device execution. Given two square matrices $A, B \in \mathbb{R}^{N \times N}$, the computation of the output matrix $C = A \times B$ is

Algorithm 7: Parallel Matrix Multiplication on Shared-Memory (Fork–Join)

Input: A, B are $n \times n$ matrices
Output: $C = A \times B$

Procedure *ForkJoinMatMul*$(C, A, B, threshold)$
  // Fork–Join parallel region
  parallel region;
    single Compute$(C, A, B, threshold)$;
Procedure *Compute*$(C, A, B, threshold)$
  $N \leftarrow \dim(C)$;
  if $N \leq threshold$ **then**
    SeqMultiply(C, A, B);
    return;
  end
  Create temporary matrices T^0, T^1 of size $N \times N$;
  Divide A, B, C into M_{ij} // $i, j \in \{0, 1\}$;
  Divide T^0, T^1 into M^i_{jk} // $i, j, k \in \{0, 1\}$;
  for $i = 0$ *to* 2 **do**
    for $j = 0$ *to* 2 **do**
      for $k = 0$ *to* 2 **do**
        spawn Compute$(T^i_{jk}, A_{ji}, B_{ik}, threshold)$;
      end
    end
  end
  sync // wait for tasks;
  MatrixAddition(C, T_0, T_1);

Algorithm 8: Task-Based Parallel Strassen Multiplication (Shared Memory)

Input: A, B ($n \times n$ matrices), threshold T
Output: $C = A \times B$

Procedure *StrassenOmp*(C, A, B)
  $N \leftarrow \dim(C)$;
  if $N \leq T$ **then**
    IKJ_Multiply(C, A, B); **return**
  end
  // Partition matrices into 4 blocks each
  $A_{00}, A_{01}, A_{10}, A_{11} \leftarrow \text{split}(A)$;
  $B_{00}, B_{01}, B_{10}, B_{11} \leftarrow \text{split}(B)$;
  $C_{00}, C_{01}, C_{10}, C_{11} \leftarrow \text{split}(C)$;
  // Allocate temporary matrices $M_1, \ldots, M_7$
  spawn StrassenOmp$(M_1, A_{00} + A_{11}, B_{00} + B_{11})$;
  spawn StrassenOmp$(M_2, A_{10} + A_{11}, B_{00})$;
  spawn StrassenOmp$(M_3, A_{00}, B_{01} - B_{11})$;
  spawn StrassenOmp$(M_4, A_{11}, B_{10} - B_{00})$;
  spawn StrassenOmp$(M_5, A_{00} + A_{01}, B_{11})$;
  spawn StrassenOmp$(M_6, A_{10} - A_{00}, B_{00} + B_{01})$;
  spawn StrassenOmp$(M_7, A_{01} - A_{11}, B_{10} + B_{11})$;
  // Combine submatrices
  $C_{00} \leftarrow M_1 + M_4 - M_5 + M_7$;
  $C_{01} \leftarrow M_3 + M_5$;
  $C_{10} \leftarrow M_2 + M_4$;
  $C_{11} \leftarrow M_1 - M_2 + M_3 + M_6$;
  $C \leftarrow \text{combine}(C_{00}, C_{01}, C_{10}, C_{11})$;

decomposed into independent subproblems that are executed in parallel on the GPU.

As shown in Algorithm 9, the **GPU kernel** adopts a two dimensional grid of thread blocks, where each block contains $TILE \times TILE$ threads and is responsible for computing a corresponding tile of the output matrix. Each thread computes a single element $C_{i,j}$ by accumulating partial results across multiple phases. To reduce global memory access latency and increase data reuse, two shared memory buffers are allocated per thread block to store tiles of the input matrices. This shared memory region is private to each thread block and is accessible only by threads within the same block, thereby enforcing memory isolation across blocks while enabling efficient intra-block data reuse. During each phase, threads cooperatively load contiguous submatrices from global memory into shared memory, followed by a synchronization barrier to ensure data consistency. The partial dot product is then computed using the data stored in shared memory and accumulated in a register. This process is repeated until all tiles along the inner matrix dimension are processed, after which the final result is written back to global memory.

On the **host side**, the execution flow, described in Algorithm 10, is designed to overlap data transfer and computation using multiple CUDA streams. Device memory buffers are allocated for the input and output matrices, and the input data are partitioned into contiguous chunks according to the number of available CPU cores. Each chunk is asynchronously transferred to the device using a dedicated

Algorithm 9: Tiled GPU Matrix Multiplication Kernel

Input: Matrices A, B of size $N \times N$
Output: Matrix $C = L \times R$

Kernel TiledKernel$(C, A, B, TILE)$
  $N \leftarrow matrixsize$;
  $ty \leftarrow$ threadIdx.y;
  $tx \leftarrow$ threadIdx.x;
  $i \leftarrow$ blockIdx.$y \cdot$ blockDim.$y + ty$;
  $j \leftarrow$ blockIdx.$x \cdot$ blockDim.$x + tx$;
  Allocate shared TileA$[TILE][TILE]$;
  Allocate shared TileB$[TILE][TILE]$;
  $acc \leftarrow 0$;
  for $p \leftarrow 0$ **to** $N/TILE - 1$ **do**
    TileA$[ty][tx] \leftarrow A[i,\ p * TILE + tx]$;
    TileB$[ty][tx] \leftarrow B[p * TILE + ty,\ j]$;
    Synchronize all threads;
    for $k \leftarrow 0$ **to** $TILE - 1$ **do**
      $acc \leftarrow acc +$ TileA$[ty][k] \cdot$ TileB$[k][tx]$;
    Synchronize all threads;
  $C[i, j] \leftarrow acc$;

stream, allowing concurrent data movement. A synchronization

Algorithm 10: Tiled GPU Matrix Multiplication API

Input: Matrices A, B of size $N \times N$
Output: Matrix $C = L \times R$
Host TiledMatmul(C, A, B)
  $TILE \leftarrow$ tile width;
  $P \leftarrow$ number of CPU cores;
  Allocate device buffers G_A, G_B, G_C;
  Create streams $S_1, \ldots, S_P$;
  Divide A and B into P contiguous chunks;
  for $s \leftarrow 1$ **to** P **do**
    Async copy chunk s of A to G_A using S_s;
    Async copy chunk s of B to G_B using S_s;
  Create compute stream S_c;
  Record event after copies; wait on S_c;
  Launch TiledKernel on grid $(\lceil N/TILE \rceil, \lceil N/TILE \rceil)$ with block $(TILE, TILE)$ on S_c;
  Async copy $G_C \rightarrow C$ using S_c;
  Wait for S_c; destroy streams and free memory;

event ensures that kernel execution begins only after all required data transfers are completed. The tiled kernel is then launched with a grid configuration of $(\lceil N/TILE \rceil, \lceil N/TILE \rceil)$ and a block size of $(TILE, TILE)$. After kernel execution, the result matrix is asynchronously copied back to host memory, and a final synchronization guarantees correctness before releasing GPU resources.

By combining shared-memory tiling, fine-grained thread-level parallelism, and asynchronous execution, the proposed methodology effectively exploits the GPU memory hierarchy and parallel architecture, minimizing global memory traffic and achieving high computational throughput for large-scale matrix multiplication.

4.4 Distributed-Memory Matrix Multiplication

In this section, we present two MPI-based implementations for matrix multiplication on distributed-memory systems: a baseline approach and a grid-based approach. The baseline method distributes rows of the first matrix among worker processes, while the grid-based method organizes processes into a two-dimensional process grid to improve data locality and reduce communication overhead. In addition, we implement the Strassen algorithm and discuss its distributed-memory execution strategy.

4.4.1 Baseline MPI Matrix Multiplication on Distributed-Memory. The idea behind the baseline MPI matrix multiplication (see Figure 20) is to distribute the rows of the first matrix A among multiple worker processes. Each worker process computes its assigned rows of the resulting matrix C by multiplying its portion of A with the entire matrix B with the algorithm matrix multiplication (see Algorithm 3) IKJ loop ordering for better cache performance., which is broadcasted to all workers. Finally, the master process collects the computed rows from each worker to assemble the final result matrix C.

This approach can work with any number of MPI processes, making it flexible for various distributed-memory systems. However, because each worker needs access to the entire matrix B, the

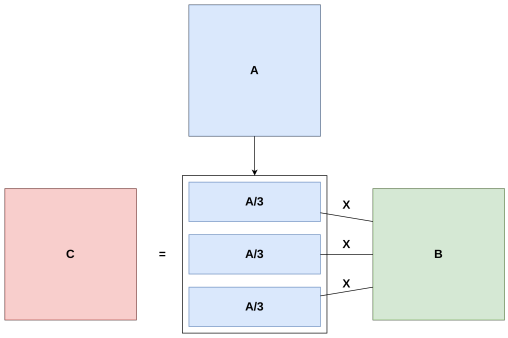


Figure 20: Baseline MPI Matrix Multiplication on Distributed-Memory

Algorithm 11: MPI Matrix Multiplication on Distributed-Memory (Baseline)

Input: Matrices A, B of size $n \times n$
Output: Matrix $C = A \times B$
Procedure *MPIMatMulBaseline(C, A, B)*
  $rank \leftarrow$ MPI_Comm_rank(); $size \leftarrow$ MPI_Comm_size();
  if $rank = 0$ **then**
    `// Master Process`
    Distribute rows of A to workers;
    MPI_Bcast(B);
    $C_{local} \leftarrow A_{local} \times B$ `// Compute local portion;`
    Collect results from workers;
    return C;
  else
    `// Worker Process`
    Receive assigned rows A_{local} from master;
    MPI_Bcast(B);
    $MatMul\_IKJ(C_{local}, A_{local}, B)$;
    Send C_{local} to master;
  end

communication overhead can become significant, especially for large matrices or a high number of processes. In the experiments section, we can see that this baseline method can just work up to matrix size is 2^{14} due to MPI communication limits.

4.4.2 Grid MPI Matrix Multiplication on Distributed-Memory. The Algorithm 12 employs a 2D grid distribution of matrices across MPI processes, where each process computes a block of the resulting matrix C by multiplying corresponding blocks of matrices A and B (see Figure 21). The local block multiplications are performed using a recursive strategy, switching to direct multiplication when the submatrix size falls below a specified threshold.

This approach improves scalability by distributing both matrices A and B across processes, reducing the communication overhead compared to the baseline method. However, it requires the number of MPI processes to be a perfect square to form a 2D grid, which may limit flexibility in certain distributed-memory environments.

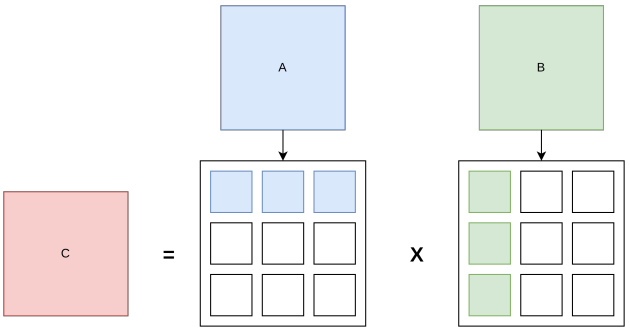


Figure 21: 2D Grid MPI Matrix Multiplication where each process computes a block of resulting matrix C by multiplying corresponding blocks of matrices A and B.

Moreover, the overhead resides in the splitting and gathering of submatrices among processes.

4.4.3 Strassen's Algorithm MPI Matrix Multiplication on Distributed-Memory. In this section, we present Strassen Matrix Multiplication Algorithm in Distributed-Memory version using MPI (see Algorithm 13). At first glance, load balancing appears to be the primary concern, leading us to consider the master-worker model for implementing Strassen's algorithm. However, during implementation we encountered a far more critical issue: the communication cost inherent in Strassen's recursive structure.

In a naive master-worker approach, each worker repeatedly sends a request-for-task message to the master, receives the required submatrices or linear combinations needed for computing one of the seven Strassen products, and then returns the resulting block M_i to the master. This repeated exchange of data becomes extremely expensive. As shown in prior work, communication costs dominate the execution time of fast matrix multiplication algorithms such as Strassen when deployed on distributed-memory systems [2].

In our implementation, we experimentally evaluated this master-worker design. The execution time was overwhelmingly dominated by communication overhead, confirming that this model is unsuitable for Strassen in a distributed environment.

Given this observation, we reconsidered the nature of workload distribution in Strassen. Under optimistic assumptions, the seven subproblems of Strassen have nearly identical computational cost, enabling a more predictable and balanced distribution of work. Therefore, a *static scheduling* [16] strategy, where each process is assigned a fixed Strassen subproblem of the recursion-becomes a far more appropriate choice. This approach avoids repeated communication during task assignment, reduces latency, making it better suited for achieving scalable performance.

At the top level of Strassen's algorithm, the computation is decomposed into seven independent subproblems corresponding to the intermediate matrices $M_1, \ldots, M_7$.

Case 1: 2-7 processes. Each of the seven Strassen subproblems is guaranteed to receive at least one process. When fewer than seven processes are available, some processes take responsibility for multiple subproblems. Work is assigned from M_1 to M_7 until all processes are mapped. After assignment, each process independently executes the full recursive computation of its designated subproblem using `StrassenSeq` 4. (Algorithm 14)

Case 2: 8-14 processes. Ensure that each of the seven subproblems has at least one dedicated process (Compute in Algorithm 15).

Algorithm 12: Grid MPI Matrix Multiplication on Distributed-Memory

Input: A, B are $n \times n$ matrices, THRESHOLD
Output: $C = A \times B$

Procedure *MPIMatMulGrid(C, A, B, THRESHOLD)*
  $T \leftarrow THRESHOLD$;
  $rank \leftarrow$ MPI_Comm_rank(), $size \leftarrow$ MPI_Comm_size();
  `// size must be a perfect square`
  $gridSize \leftarrow \sqrt{size}$;
  if $rank = 0$ **then**
    for *each worker process* (i, j) *in grid* **do**
      Send $A_{i,k}$ and $B_{k,j}$ blocks for all k;
    end
    $\sum_{k=0}^{gridSize-1} RecursiveMultiply(C_{0,0}, A_{0,k}, B_{k,0}, T)$;
    for *each worker process* (i, j) *in grid except (0,0)* **do**
      Receive $C_{i,j}$ block;
    end
    return C;
  else
    `// Worker Process` (r, c)
    $r \leftarrow rank/gridSize$;
    $c \leftarrow rank \bmod gridSize$;
    $C_{local} \leftarrow \mathbf{0}$;
    for $k \leftarrow 0$ **to** $gridSize - 1$ **do**
      Receive $A_{r,k}$ and $B_{k,c}$ from master;
      $C_k \leftarrow RecursiveMultiply(C_k, A_{r,k}, B_{k,c}, T)$;
      $C_{local} \leftarrow C_k$;
    end
    Send C_{local} to master;
  end
`// Recursive Matrix Multiplication`
Procedure *RecursiveMultiply(C, A, B, THRESHOLD)*
  $T \leftarrow THRESHOLD$;
  if $size(A) \leq T$ **then**
    `// Base case: direct multiplication`
    return $MatMul\_IKJ(C, A, B)$;
  end
  `// Divide into 4 submatrices and recurse`
  $C_{00} \leftarrow RecursiveMultiply(A_{00}, B_{00}, T) +$
   $RecursiveMultiply(A_{01}, B_{10}, T)$;
  $C_{01} \leftarrow RecursiveMultiply(A_{00}, B_{01}, T) +$
   $RecursiveMultiply(A_{01}, B_{11}, T)$;
  $C_{10} \leftarrow RecursiveMultiply(A_{10}, B_{00}, T) +$
   $RecursiveMultiply(A_{11}, B_{10}, T)$;
  $C_{11} \leftarrow RecursiveMultiply(A_{10}, B_{01}, T) +$
   $RecursiveMultiply(A_{11}, B_{11}, T)$;

Algorithm 13: MPI-Based Strassen Matrix Multiplication (1–16 Processes)

Input: A, B are $n \times n$ matrices, THRESHOLD
Output: $C = A \times B$

Procedure *MPI_Strassen($C, A, B, N, threshold$)*
  $rank \leftarrow$ MPI_Comm_rank();
  $p \leftarrow$ MPI_Comm_size();
  `// Broadcast input dimension and matrices`
  MPI_Bcast(N);
  MPI_Bcast(A);
  MPI_Bcast(B);
  `// Base case: fallback to sequential multiplication`
  if $N \leq threshold$ **then**
    if $rank = 0$ **then**
      $C \leftarrow$ SequentialMultiply(A, B);
    end
    MPI_Bcast(C);
    return;
  end
  `// Split matrices into four blocks`
  Partition $A \rightarrow A_{00}, A_{01}, A_{10}, A_{11}$;
  Partition $B \rightarrow B_{00}, B_{01}, B_{10}, B_{11}$;
  Allocate $M_{1..7}$ (size $N/2 \times N/2$);
  `// Case 1: Use Group 1 (direct task mapping)`
  if $2 \leq p \leq 7$ **then**
    StrassenMPIGroup1($M_{1..7}$, A, B, N);
  end
  `// Case 2: Use Group 2 (two-process subgroups)`
  else if $8 \leq p \leq 14$ **then**
    StrassenMPIGroup2($M_{1..7}$, A, B, N);
  end
  `// Case 3: Use Group 3 (two/three-process subgroups)`
  else
    StrassenMPIGroup3($M_{1..7}$, A, B, N);
  end
  `// Combine all 7 Strassen products (root only)`
  if $rank = 0$ **then**
    $C_{11} \leftarrow M_3 + M_6 - M_2 + M_1$;
    $C_{00} \leftarrow M_4 + M_7 - M_5 + M_1$;
    $C_{10} \leftarrow M_2 + M_4$;
    $C_{01} \leftarrow M_3 + M_5$;
    Assemble C from blocks $C_{00}, C_{01}, C_{10}, C_{11}$;
  end
  return C;

Algorithm 14: StrassenMPIGroup1: Subgroup for 2–7 Processes

Input: Matrix quadrants of A, B; Output blocks $M_{1..7}$
Output: Each M_i computed by one MPI process

Procedure *StrassenMPIGroup1($M_{1..7}, A, B, N$)*
  rank $\leftarrow$ MPI_Comm_rank();
  p $\leftarrow$ MPI_Comm_size();
  Partition $A \rightarrow A_{00}, A_{01}, A_{10}, A_{11}$;
  Partition $B \rightarrow B_{00}, B_{01}, B_{10}, B_{11}$;
  for $i \leftarrow 1$ **to** 7 **do**
    if $(i-1) \bmod p = rank$ **then**
      if $i = 1$ **then**
        $T_1 \leftarrow A_{00} + A_{11}$;
        $T_2 \leftarrow B_{00} + B_{11}$;
        StrassenSeq(M_1, T_1, T_2);
      else if $i = 2$ **then**
        $T_1 \leftarrow A_{10} + A_{11}$;
        StrassenSeq(M_2, T_1, B_{00});
      else if $i = 3$ **then**
        $T_2 \leftarrow B_{01} - B_{11}$;
        StrassenSeq(M_3, A_{00}, T_2);
      else if $i = 4$ **then**
        $T_2 \leftarrow B_{10} - B_{00}$;
        StrassenSeq(M_4, A_{11}, T_2);
      else if $i = 5$ **then**
        $T_1 \leftarrow A_{00} + A_{01}$;
        StrassenSeq(M_5, T_1, B_{11});
      else if $i = 6$ **then**
        $T_1 \leftarrow A_{10} - A_{00}$;
        $T_2 \leftarrow B_{00} + B_{01}$;
        StrassenSeq(M_6, T_1, T_2);
      else
        $T_1 \leftarrow A_{01} - A_{11}$;
        $T_2 \leftarrow B_{10} + B_{11}$;
        StrassenSeq(M_7, T_1, T_2);
      end
      if $rank \neq 0$ **then**
        MPI_Send(M_i, root);
      end
    end
  end
  if $rank = 0$ **then**
    Receive all $M_1..M_7$;
  end

The remaining processes are again distributed from M_1 through M_7, ensuring that no subproblem is left without computational resources. For each M_i, a subgroup is formed to compute the recursive branches of Strassen for that subproblem. The subgroup root sends the resulting block M_i back to rank 0.

Case 3: 15-16 processes. When at least 15 processes are available, each subproblem M_i is guaranteed to have at least two dedicated processes. Any additional processes (15th and 16th) are assigned to assist with the first subproblems, M_1 and M_2. For each M_i, a subgroup performs the recursive computation cooperatively. The

Algorithm 15: StrassenMPIGroup2: Subgroups for 8-14 Processes

Input: Matrix quadrants of A, B; Output blocks $M_{1..7}$
Output: Each M_i computed by one 2-process subgroup

Procedure *StrassenMPIGroup2*($M_{1..7}, A, B, N$)
- rank ← MPI_Comm_rank();
- task ← ($rank$ mod 7);
- Define a 2-process subgroup for M_{task+1};
- subcomm ← MPI_Comm_create_group(group(task));
- subrank ← MPI_Comm_rank(subcomm);
- groupsize ← MPI_Comm_size(subcomm);
- **if** *subrank = 0* **then**
  - Compute $M_1..M_4$ using StrassenSeq;
- **else**
  - Compute $M_5..M_7$ using StrassenSeq;
- **end**
- **if** *subrank = 0* **then**
  - MPI_Send(M_{task+1}, root);
- **end**
- **if** *rank = 0* **then**
  - Receive all $M_1..M_7$;
- **end**
- MPI_Comm_free(subcomm);

subgroup1 root returns the corresponding block to rank 0 (see Algorithm 16).

4.5 Hybrid Parallel Matrix Multiplication

In this section, we present three hybrid matrix multiplication algorithms that combine MPI for distributed-memory parallelism and OpenMP for shared-memory parallelism within each compute node. The first algorithm is a baseline hybrid approach that extends the standard MPI matrix multiplication by incorporating OpenMP parallel regions for local computations. The second algorithm is a grid-based hybrid approach that builds upon the grid MPI matrix multiplication, utilizing OpenMP to parallelize the local block multiplications.

4.5.1 Baseline Hybrid Matrix Multiplication. Baseline Hybrid Matrix Multiplication (see Algorithm 17) combines the standard MPI matrix multiplication approach with OpenMP for parallelizing local computations within each MPI process. The master process distributes rows of matrix A to worker processes, broadcasts matrix B, and each process computes its local portion of the resulting matrix C with the VanillaMatMul (Algorithm 5). Finally, the master process collects the results from all workers.

Like the baseline MPI method, this hybrid approach is flexible and can work with any number of MPI processes. However, it still suffers from significant communication overhead due to the need for each worker to access the entire matrix B. Therefore, this approach can just work up to matrix size is 2^{14} due to MPI communication limits, similar to the baseline MPI method.

Algorithm 16: StrassenMPIGroup3: Subgroups for 15–16 Processes

Input: Quadrants of A, B; Output blocks $M_{1..7}$
Output: Each M_i computed by either a 3-process or 2-process subgroup

Procedure *StrassenMPIGroup3*($M_{1..7}, A, B, N$)
- rank ← MPI_Comm_rank();
- task ← ($rank$ mod 7);
- Define a subgroup (size 2 or 3) for M_{task+1};
- subcomm ← MPI_Comm_create_group(group(task));
- subrank ← MPI_Comm_rank(subcomm);
- groupsize ← MPI_Comm_size(subcomm);
- `// Case 2: subgroup has 2 ranks`
- **if** *groupsize = 2* **then**
  - **if** *subrank = 0* **then**
    - Compute $M_1..M_4$ via StrassenSeq;
  - **else**
    - Compute $M_5..M_7$ via StrassenSeq;
  - **end**
  - **if** *subrank = 0* **then**
    - MPI_Send(M_{task+1}, root);
  - **end**
  - MPI_Comm_free(subcomm);
  - **return**;
- **end**
- `// Case 3: subgroup has 3 ranks`
- **if** *groupsize = 3* **then**
  - **if** *subrank = 0* **then**
    - Compute M_1 and M_2 via StrassenSeq;
  - **else if** *subrank = 1* **then**
    - Compute M_3, M_4, M_5;
  - **end**
  - **else**
    - Compute M_6 and M_7;
  - **end**
  - **if** *subrank = 0* **then**
    - MPI_Send(M_{task+1}, root);
  - **end**
- **end**
- **if** *rank* = 0 **then**
  - Receive all $M_1..M_7$;
- **end**
- MPI_Comm_free(subcomm);

4.5.2 Grid Hybrid Matrix Multiplication. The idea behind the grid hybrid matrix multiplication (see Algorithm 18) is to organize MPI processes in a 2D grid, similar to the grid MPI approach. Each process is responsible for computing a block of the resulting matrix C by multiplying corresponding blocks of matrices A and B. Within each process, we utilize OpenMP to parallelize the local block multiplications using a recursive strategy. When the submatrix size falls below a specified threshold, we switch to direct multiplication using

Algorithm 17: Baseline Hybrid Matrix Multiplication on Distributed-Memory with Shared-Memory Nodes

Input: Matrices A, B of size $n \times n$, THRESHOLD
Output: Matrix $C = A \times B$

Procedure *HybridMatMulBaseline(C, A, B, THRESHOLD)*
  $rank \leftarrow$ MPI_Comm_rank(), $size \leftarrow$ MPI_Comm_size();
  if $rank = 0$ **then**
    `// Master: Distribute + Compute + Collect`
    Distribute rows of A to workers;
    MPI_Bcast(B);
    $VanillaMatMul(C_{local}, A_{local}, B)$;
    Collect results from workers;
    return C;
  else
    `// Worker: Receive + Compute + Send`
    Receive A_{local} from master;
    MPI_Bcast(B);
    $VanillaMatMul(C_{local}, A_{local}, B)$;
    Send C_{local} to master;
  end

OpenMP parallel for loops to leverage shared-memory parallelism effectively.

Like the grid MPI method, this hybrid approach improves scalability by distributing both matrices A and B across processes, thereby reducing communication overhead. However, it still requires the number of MPI processes to be a perfect square to form a 2D grid, which may limit flexibility in certain distributed-memory environments. The additional overhead comes from the splitting and gathering of submatrices among processes, as well as the management of OpenMP parallel regions within each process.

4.5.3 Strassen Hybrid Matrix Multiplication. In the Hybrid Strassen approach, the task decomposition strategy proposed in Section 4.4.3 is retained, while the number of MPI processes is restricted to the range 2–8. Instead of using the Sequential Strassen (Algorithm 4), each assigned subproblem is computed using the Shared-Memory Strassen version (Algorithm 8)[12].

5 Experiments

This section presents the experimental results, interprets the observed performance, and provides insights into the algorithms discussed in the Methodology (Section 4). The experiments are organized into multiple subsections that cover different aspects of the evaluation. Subsection 5.1 describes the experimental setup. Subsection 5.2 investigates conflict misses in memory access. Subsection 5.3 analyzes performance on shared-memory systems, including the behavior of the addition operator (Subsubsection 5.3.1), tuning of the optimal threshold (Subsubsection 5.3.2), sequential execution (Subsubsection 5.3.3), and multi-core execution (Subsubsection 5.3.4). Subsection 5.4 reports results on GPU-accelerated systems. Subsection 5.5 and Subsection 5.6 present evaluations

[12]Algorithm `StrassenHybridGroup1` is similar to `StrassenMPIGroup1` (Algorithm 14), with the only difference being that `StrassenSeq` is replaced by `StrassenOmp`.

Algorithm 18: Grid Hybrid Matrix Multiplication on Distributed-Memory with Shared-Memory Nodes

Input: A, B are $n \times n$ matrices, THRESHOLD
Output: $C = A \times B$

Procedure *HybridMatMulGrid(C, A, B, THRESHOLD)*
  $T \leftarrow THRESHOLD$;
  $rank \leftarrow$ MPI_Comm_rank(), $size \leftarrow$ MPI_Comm_size();
  $gridSize \leftarrow \sqrt{size}$ `// Require size to be perfect square`
  if $rank = 0$ **then**
    for *each worker process* (i, j) *in grid* **do**
      Send $A_{i,k}$ and $B_{k,j}$ blocks for all k;
    end
    $\sum_{k=0}^{gridSize-1} HybridRecursive(C_{0,0}, A_{0,k}, B_{k,0}, T)$;
    for *each worker process* (i, j) *in grid except (0,0)* **do**
      Receive $C_{i,j}$ block;
    end
    return C;
  else
    `// Worker Process` (r, c)
    $r \leftarrow rank / gridSize$;
    $c \leftarrow rank \bmod gridSize$;
    $C_{local} \leftarrow \mathbf{0}$;
    for $k \leftarrow 0$ **to** $gridSize - 1$ **do**
      Receive $A_{r,k}$ and $B_{k,c}$ from master;
      $C_k \leftarrow ForkJoinMatMul(C_k, A_{r,k}, B_{k,c}, T)$;
      $C_{local} \leftarrow C_k$;
    end
    Send C_{local} to master;
  end

on distributed-memory and hybrid distributed-memory systems, respectively.

5.1 Experimental Setup

The shared-memory experiments were performed on three heterogeneous systems: a Lenovo ThinkPad T480 (System 1), the Cong Tu Vu C6 compute cluster (System 2), and the HCMUT Super Node cluster (System 3). The system configurations of a single node used in the experiments are summarized in Table 1.

5.2 Conflict Miss

The Level 1 data cache of the Intel i7–9700 processor consists of 64-byte cache lines and is 8-way set-associative, yielding a total of 64 cache sets (Table 1). Consequently, the cache-line address format contains **52 tag bits**, **6 set-index bits**, and **6 block-offset bits**, as illustrated in subfigure (I) of Figure 22. Consider a 512×512 matrix stored in a contiguous one-dimensional array. Since each element occupies 4 bytes, the starting address of row i is simply $i \cdot 2^{11}$, meaning that its binary representation corresponds to the row index shifted left by 11 bits. The lower 11 bits are always zero due to alignment, while the higher-order bits encode the row index.

Algorithm 19: StrassenHybridGroup2: Subgroups for 8-14 Processes

Input: Matrix quadrants of A, B; Output blocks $M_{1..7}$
Output: Each M_i computed by one 2-process subgroup

Procedure *StrassenHybridGroup2*($M_{1..7}, A, B, N$)
- rank ← MPI_Comm_rank();
- task ← ($rank$ mod 7);
- Define a 2-process subgroup for M_{task+1};
- subcomm ← MPI_Comm_create_group(group(task));
- **if** *subcomm = NULL* **then**
  - **return** ; `// Rank does not participate`
- **end**
- subrank ← MPI_Comm_rank(subcomm);
- groupsize ← MPI_Comm_size(subcomm);
- `// Case: subgroup has only 1 rank`
- **if** *groupsize = 1* **then**
  - Compute M_{task+1} using StrassenOmp;
  - MPI_Send(M_{task+1}, root);
  - MPI_Comm_free(subcomm);
  - **return**;
- **end**
- `// Case: subgroup has 2 ranks`
- **if** *subrank = 0* **then**
  - Compute partial blocks for $M_1..M_4$ using StrassenOmp;
- **else**
  - Compute partial blocks for $M_5..M_7$ using StrassenOmp;
- **end**
- **if** *subrank = 0* **then**
  - MPI_Send(M_{task+1}, root);
- **end**
- **if** *rank = 0* **then**
  - Receive all $M_1..M_7$;
- **end**
- MPI_Comm_free(subcomm);

Algorithm 20: Strassen Hybrid Matrix Multiplication on Distributed-Memory with Shared-Memory Nodes (1-16 Processes)

Input: A, B are $n \times n$ matrices, THRESHOLD
Output: $C = A \times B$

Procedure *HybridStrassen*($C, A, B, N, threshold$)
- $rank$ ← MPI_Comm_rank();
- p ← MPI_Comm_size();
- `// Broadcast input dimension and matrices`
- MPI_Bcast(N);
- MPI_Bcast(A);
- MPI_Bcast(B);
- `// Base case: fallback to sequential multiplication`
- **if** $N \leq threshold$ **then**
  - **if** $rank = 0$ **then**
    - C ← Matmul_IKJ(A, B);
  - **end**
  - MPI_Bcast(C);
  - **return**;
- **end**
- `// Split matrices into four blocks`
- Partition $A \rightarrow A_{00}, A_{01}, A_{10}, A_{11}$;
- Partition $B \rightarrow B_{00}, B_{01}, B_{10}, B_{11}$;
- Allocate $M_{1..7}$ (size $N/2 \times N/2$);
- `// Case 1: Use Group 1 (direct task mapping)`
- **if** $2 \leq p \leq 7$ **then**
  - StrassenHybridGroup1($M_{1..7}, A, B, N$);
- **end**
- `// Case 2: Use Group 2 (two-process subgroups)`
- **else if** $8 \leq p \leq 14$ **then**
  - StrassenHybridGroup2($M_{1..7}, A, B, N$);
- **end**
- `// Combine all 7 Strassen products (root only)`
- **if** $rank = 0$ **then**
  - $C_{11} \leftarrow M_3 + M_6 - M_2 + M_1$;
  - $C_{00} \leftarrow M_4 + M_7 - M_5 + M_1$;
  - $C_{10} \leftarrow M_2 + M_4$;
  - $C_{01} \leftarrow M_3 + M_5$;
  - Assemble C from blocks $C_{00}, C_{01}, C_{10}, C_{11}$;
- **end**
- **return** C;

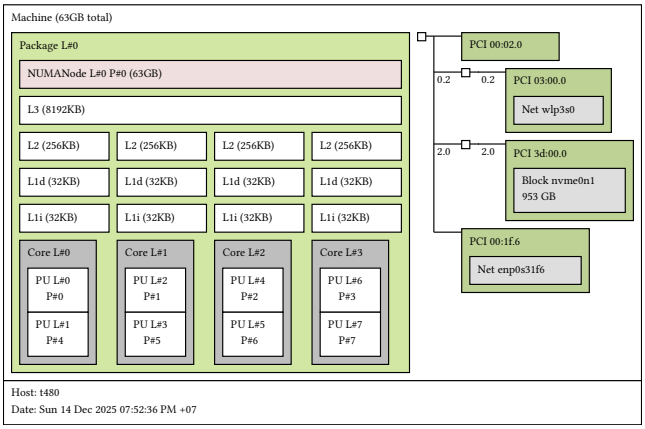
Subfigures (II), (III), and (IV) illustrate the starting addresses of the first element of row 0, row 1, and row i, respectively.

Figure 23 shows how the cache lines of matrix B (represented as cells) are assigned set indices according to the 6 set-index bits in subfigure (I). Because the L1 cache of System 2 contains 64 sets, the set index repeats every 64 cache lines that is, every two matrix rows. Each group of two rows is annotated in red to indicate the 8-way associativity of the target cache sets. However, the IJK traversal pattern accesses matrix B column-wise. Therefore, the cache lines whose set indices are 0 and 32 are brought into the cache first and most frequently, causing extremely unbalanced set utilization.

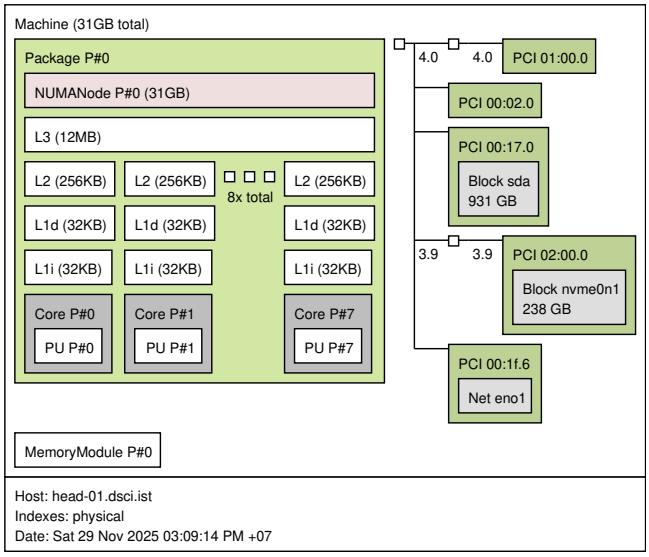
Figure 24 illustrates the state of the L1 cache when the first cache line of row 16 of matrix B is accessed. At this moment, almost all cache sets remain unused except for sets 0 and 32, whose 8 ways are fully occupied. When the cache line of row 16 is fetched, it replaces the line in *set 0* and *way 0* changing the entry from $(0, 0)$ to $(8, 0)$, highlighted in yellow in the figure. This replacement pattern persists, due to repeated conflicts in only two heavily used sets, every new row access evicts a previously loaded cache line. As a result, when the computation reaches C[0,1], *all* cache lines of B needed for the previous computation have already been evicted, forcing the processor to reload the entire column from memory.

Table 1: System specifications of single compute node for experiments

Specification	**Lenovo ThinkPad T480** (System 1)	**Cong Tu Vu C6 Cluster** (System 2)	**HCMUT Super Node Cluster** (System 3)
System type	Personal laptop	Compute server	Compute server
Compiler	g++ 13.3.0 (Ubuntu)	`clang++` 18.1.8 (Red Hat)	`clang++` 14.0.0 (Ubuntu)
Operating System	Ubuntu 24.04 LTS	Rocky Linux 8.10 (Green Obsidian)	Ubuntu 22.04.5 LTS
Processor	Intel Core i5-8250U @ 1.60GHz	Intel Core i7-9700 @ 3.00GHz	Intel Xeon E5-2680 v3 @ 2.50GHz
Frequency	Set constant 3.0 GHz	Set constant 3.6 GHz	Constant 2.5 GHz
Cores / Threads per	4 cores / 8 threads	8 cores / 8 threads	8 cores / 8 threads
Hyper Threading [13]	Yes	No	No
RAM	64 GB	32 GB	16 GB
Cache hierarchy / Associative way	L1i: 32 KB (private) / 8 L1d: 32 KB (private) / 8 L2: 256 KB (private) / 4 L3: 8 MB (shared) / 16	L1i: 32 KB (private) / 8 L1d: 32 KB (private) / 8 L2: 256 KB (private) / 4 L3: 12 MB (shared) / 12	L1i: 32 KB (private) / 8 L1d: 32 KB (private) / 8 L2: 4096 KB (private) / 8 L3: 16 MB (shared) / 16
Accelerator	None	RTX 2070 Super (8GB VRAM)	None
OpenMP / MPI	4.5 / 4.1.6	5.1 / 4.1.1	4.5 / 4.0.0
Python / Numpy	3.12.3 / 2.3.5	3.13.9 / 2.3.5	3.10.12 / 2.2.6



System 1: Intel(R) Core(TM) CPU i7-8650U @ 4.2GHz processor in the Lenovo ThinkPad T480 - khaiphan personal laptop



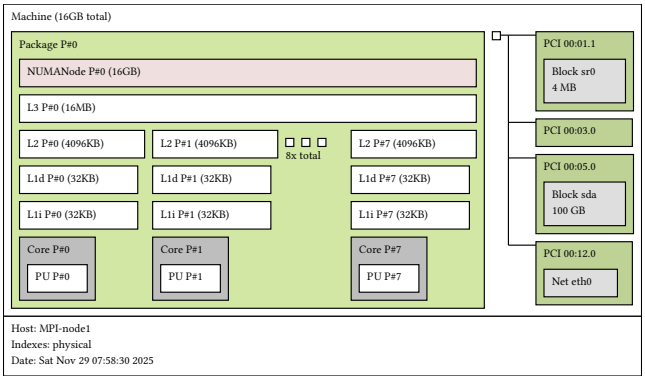
System 2: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz - C6 Cong Tu Vu Cluster

This leads to severe performance degradation caused by continuous conflict misses.

This severe conflict-miss behavior occurs only when the matrix dimension is a power of two using IJK order. For non-power-of-two dimensions, the lower address bits are no longer all zero after the left shift (as the row index increments), these bits vary and distribute the cache-line mappings across more sets, greatly reducing conflict misses. This effect is clearly observed in practice. When $N = 512 = 2^9$, the execution time is nearly twice as slow as when $N = 513$. The additional low-order bits introduced at $N = 513$ break the alignment pattern responsible for concentrated set mappings, yielding substantially fewer conflict misses. While Figure 25 presents empirical results for matrix sizes in the range $[497, 527]$, highlighting the sharp performance degradation at power-of-two dimensions under the IJK multiplication order, Figure 26 do not show such degradation for the IKJ order, which accesses rows of B instead of columns.

5.3 Shared Memory System

The experimental evaluation of square matrix multiplication on shared-memory systems was performed on the **three** hardware platforms described in Section 5.1. Unless otherwise stated, the baseline used for computing speedup ratios is the sequential implementation employing the same loop order (either `IJK` or `IKJ`) as the corresponding parallel algorithm being compared. When a parallel



System 3: Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz - HCMUT Super Node Cluster

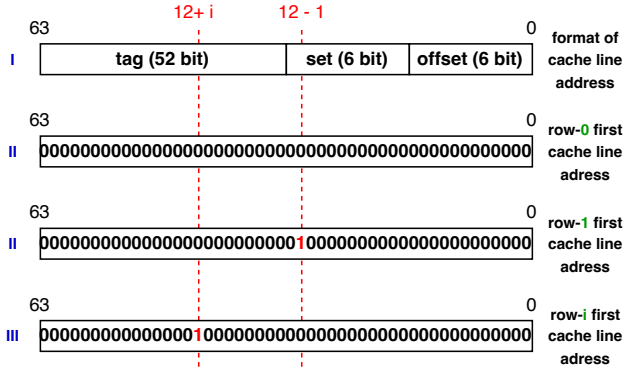


Figure 22: Bit representation of first and i-th row of a matrix size 512×512 with the assumption that first element is located at byte address 0.

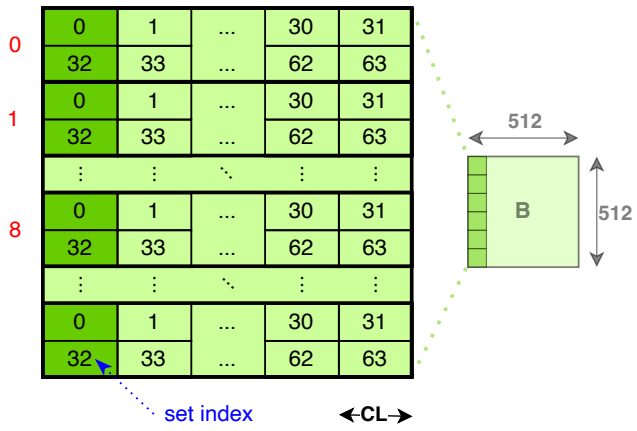


Figure 23: The cache lines of matrix B (represented as cells) are assigned set indices according to the 6 set-index bits in subfigure (I).

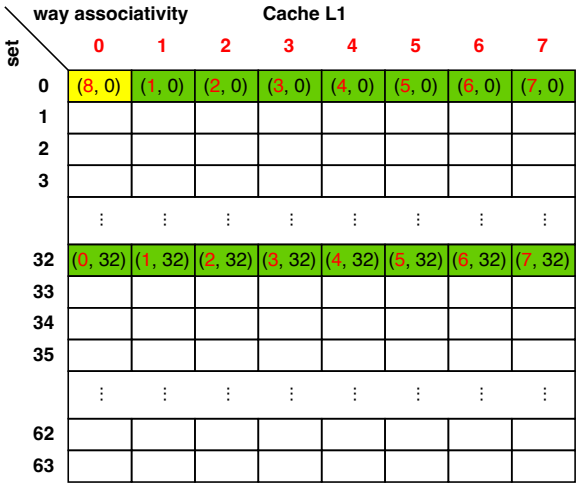


Figure 24: L1 cache state during access to row 16 of matrix B, showing repeated conflict misses caused by contention in sets 0 and 32, where a cache line replacement from $(0, 0)$ to $(8, 0)$ triggers eviction of previously loaded data.

algorithm includes its own base-case sequential kernel, that kernel is also used as the baseline for the relevant measurements.

Several engineering optimizations were applied uniformly across all implementations. All matrices are stored in a single contiguous one-dimensional buffer aligned to 64 bytes, enabling the compiler to exploit **implicit SIMD vectorization** and to benefit from `#pragma omp simd`[14]. All recursive routines operate on references to regions within the same underlying buffer, which is often called **matrix views**, to avoid excessive memory allocation. Temporary buffers are allocated only when strictly necessary and released immediately to prevent memory leaks and maintain predictable performance.

5.3.1 Addition Operator. Matrix addition is required as a subroutine for many divide and conquer multiplication algorithms. Since adding an (N, N) matrix is equivalent to adding a vector of length N^2, we first benchmarked a vector-add kernel under multi-core execution in order to assess whether explicit AVX2 SIMD instructions could provide benefits beyond OpenMP or compiler auto-vectorization. Figure 27 shows the execution time and speedup on the 8-core on System 2. Surprisingly, both small and very large vector sizes exhibit almost **no performance difference between sequential and parallel execution**, regardless of the number of cores used[15]. We are currently unable to provide a definitive explanation for this *"bell shape"* behavior.

[14]Using explicit SIMD intrinsics such as `immintrin.h` (Intel) or `arm_neon.h` (ARM) would likely yield higher performance, but this was not pursued due to time constraints of the assignment.

[15]The same phenomenon was observed on other machines with different core counts; see the supplementary material for details

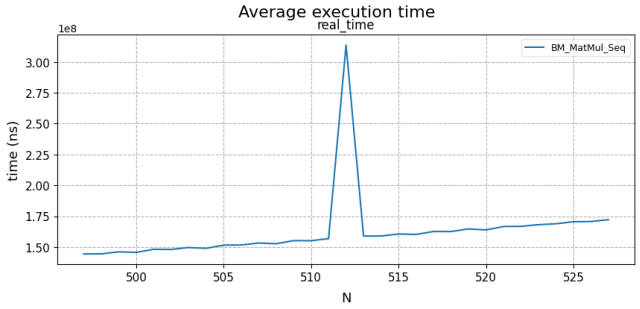


Figure 25: Performance comparison of matrix multiplication using IJK order for matrix sizes from 497 to 527, highlighting the performance drop at power-of-two size 512.

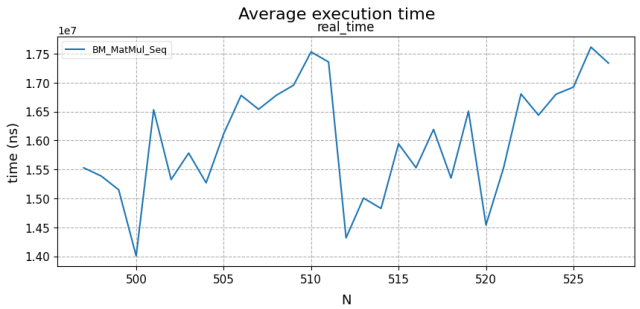


Figure 26: Performance comparison of matrix multiplication using IKJ order for matrix sizes from 497 to 527, highlighting the absence of performance drop at power-of-two size 512.

Figure 28 presents the corresponding results for matrix addition using three parallel variants[16], again on 8 cores. All variants exhibit the same behavior that parallel execution provides little to no improvement and, in some cases, is indistinguishable from the sequential baseline.

Based on these observations, all subsequent multiplication experiments in this paper use the sequential matrix-add implementation as the underlying addition operator. We also note that *NumPy* [12] performs matrix addition on a single core by default, which is consistent with our findings.

5.3.2 Tuning the Optimal Threshold. This section aims to identify the optimal granularity threshold for divide-and-conquer algorithms through exhaustive brute-force search. Although this method is computationally expensive and lacks analytical sophistication, it provides a comprehensive characterization of hardware behavior and yields a reliable empirical choice of threshold.

The experimental results for the **plain fork–join** approach, Algorithm 7, whose outcomes are shown in Figures 29, together with those for the **Strassen fork–join** variant, Algorithm 8, illustrated in Figures 30, are visualized as heatmaps. The vertical axis represents the threshold T, the horizontal axis the matrix size N, and

[16] *vanilla* OpenMP parallelfor, *ForkJoin* with optimal threshold, and a combined hybrid approach called GODDAMNNNN

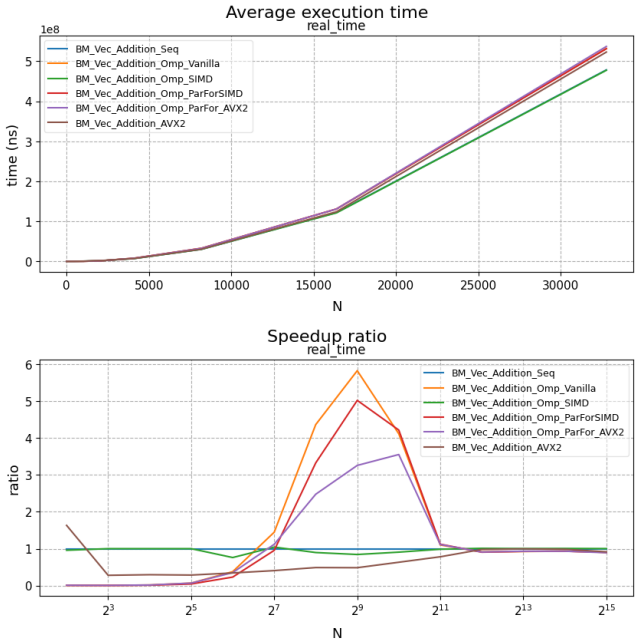


Figure 27: Execution time and speedup of vector addition using 8 cores on System 2.

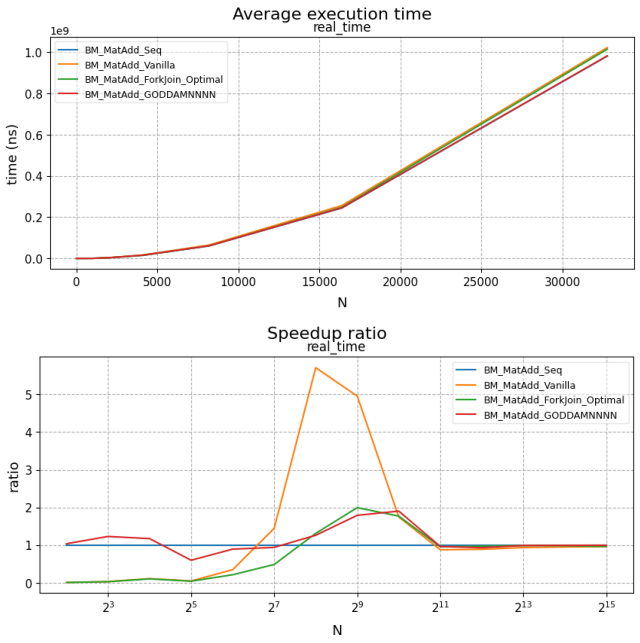


Figure 28: Execution time and speedup of matrix addition using 8 cores on System 2.

each cell encodes the speedup relative to the slowest configuration for that particular N. A **black**, staircase-shaped boundary line separates the parallel region ($T < N$) from the sequential region ($T \geq N$), with its step-like pattern reflecting the discrete nature of

the threshold values. For each column, the yellow cell marks the globally fastest configuration, while the green cell indicates the fastest configuration restricted to the parallel region. Thresholds as small as $T = 2$ consistently appear among the slowest, as their overly fine granularity introduces significant overhead; therefore, $T = 2$ is always treated as a baseline. Consequently, whenever the yellow and green cells **coincide** within a column, that threshold can be interpreted as the **optimal choice** for the corresponding matrix dimension N.

As demonstrated in Section 5.2, the `IKJ` ordering outperforms `IJK` due to better memory locality, while adding Kahan (see Subsection 3.4) summation to `IJK` makes the kernel even slower because of compensation bookkeeping and additional floating-point operations. This establishes the performance ranking:

$$\text{IJK + Kahan} \;<\; \text{IJK} \;<\; \text{IKJ},$$

where "<" means "slower than."

Because a slower base case becomes expensive earlier in the recursion, the algorithm should switch to it sooner in order to avoid excessive parallel overhead. This produces the following experimentally determined optimal thresholds.

Optimal thresholds for plain fork–join (FJ).

$$T^{\text{FJ}}_{\text{opt}}(\text{IJK + Kahan}) = 16, \qquad T^{\text{FJ}}_{\text{opt}}(\text{IJK}) = 32, \qquad T^{\text{FJ}}_{\text{opt}}(\text{IKJ}) = 64.$$

Optimal thresholds for fork–join Strassen (Str).

$$T^{\text{Str}}_{\text{opt}}(\text{IJK + Kahan}) = 16, \qquad T^{\text{Str}}_{\text{opt}}(\text{IJK}) = 32, \qquad T^{\text{Str}}_{\text{opt}}(\text{IKJ}) = 128.$$

These results highlight an intriguing distinction between the two recursive schemes. Although all three base-case kernels preserve the same relative performance ordering, Strassen's recursion magnifies the separation between their optimal thresholds. Most notably, the optimal threshold for the fast `IKJ` kernel increases dramatically from $T = 64$ in the plain fork–join setting to $T = 128$ under Strassen's framework. This shift reflects the algorithm's more aggressive subdivision. When the base-case kernel is fast, Strassen can profitably recurse to smaller subproblems, thereby favoring a larger threshold before falling back to the classical multiplication kernel.

Taken together, a key observation emerges consistently across both experiment groups (plain fork–join and fork–join Strassen): **the slower the base-case kernel, the smaller the optimal threshold**. This inverse relationship between base-case efficiency and threshold choice becomes even more pronounced in deeply recursive algorithms such as Strassen's method.

5.3.3 Sequential Performance. First, experiments are conducted using a single CPU core. The experiments are performed on System 2. Figure 31 presents the execution time and speedup of the sequential implementations. When $N = 2^8$, the Strassen algorithm is nearly twice as fast as the baseline IKJ implementation. As the problem size increases to $N = 2^{14}$, Strassen achieves an approximate sevenfold speedup over IKJ.

This behavior is expected, since Strassen's algorithm reduces the number of multiplications from eight to seven per recursion level. Due to memory limitations, experiments for $N = 2^{15}$ cannot

[16]The vanilla parallel implementation is not evaluated in this setting, as execution on a single core is equivalent to sequential execution.

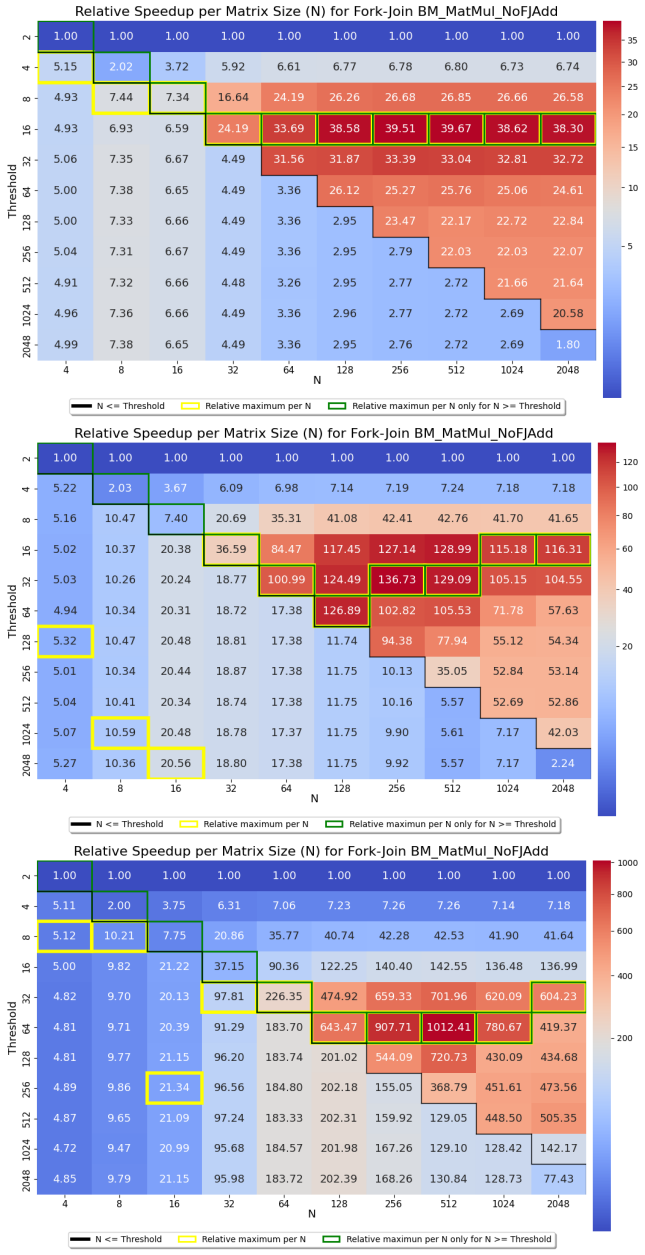


Figure 29: Threshold tuning for fork–join (Algorithm 7) with IJK + Kahan, IJK, and IKJ base-case using 8 cores on System 2, respectively.

be conducted. Therefore, it remains unclear whether the observed speedup trend continues for larger problem sizes.

5.3.4 Multi-core Performance. We begin by comparing the vanilla `IKJ` (Algorithm 5) and Fork–Join (Algorithm 7) implementations against the base-case `IKJ` kernel (Algorithm 3), **excluding Strassen** since all of these algorithms have cubic time complexity $O(N^3)$. Informally speaking, we observe superlinear speedups across all core counts on System 2 (Figures 32). Remarkably, similar behavior

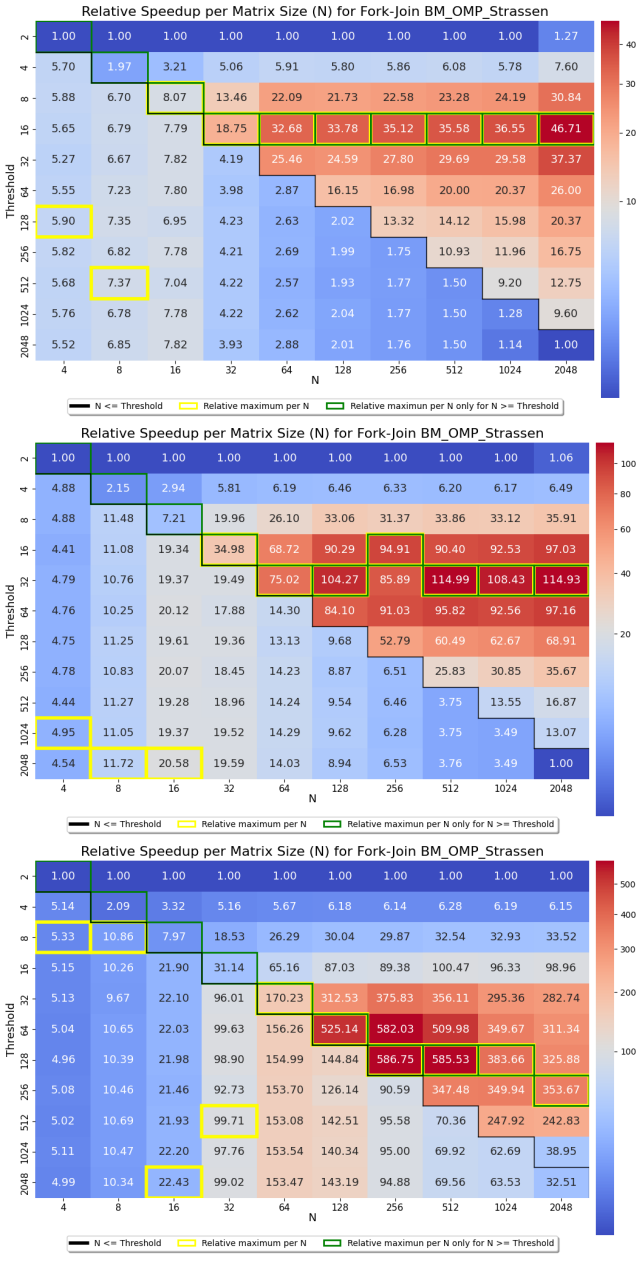


Figure 30: Threshold tuning for fork–join Strassen (Algorithm 8) with IJK + Kahan, IJK and IKJ base-case using 8 cores on System 2, respectively.

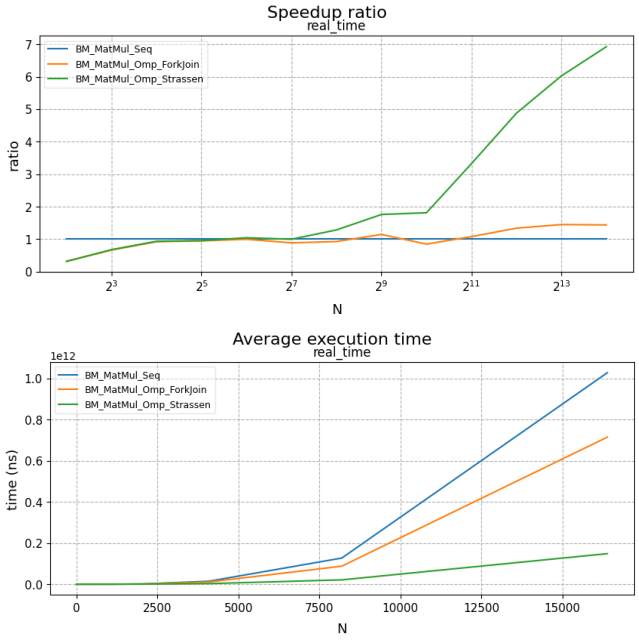


Figure 31: Execution time and speedup of sequence matrix multiplication on System 2.

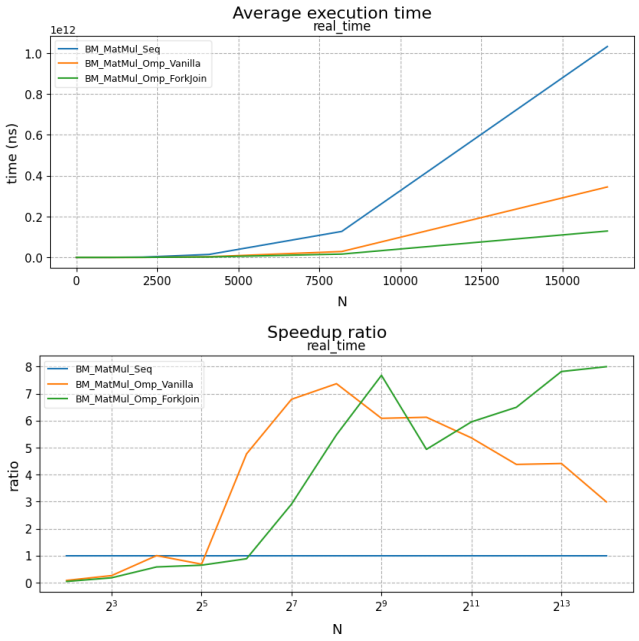


Figure 32: Execution time and speedup of matrix multiplication using 8 cores on System 2.

also appears on System 3 (Figures 33). For matrix sizes in the range $[2^{10}, 2^{14}]$, the execution becomes more than 20× **faster** when using 8 cores, and up to 5× **faster** even with only 2 cores. This phenomenon can be attributed to improved cache utilization [17]. As the workload is distributed among multiple cores, each core operates on a smaller subset of data that fits more effectively within its local cache, thereby reducing memory access latency and enhancing overall performance.

Next, we compare these algorithms against **Strassen's method** (Algorithm 4). Despite being a historically "ancient" algorithm, Strassen performs extraordinarily well. **Figures 34**, **35**, and **36** correspond to Systems 1, 2, and 3, respectively. For the largest

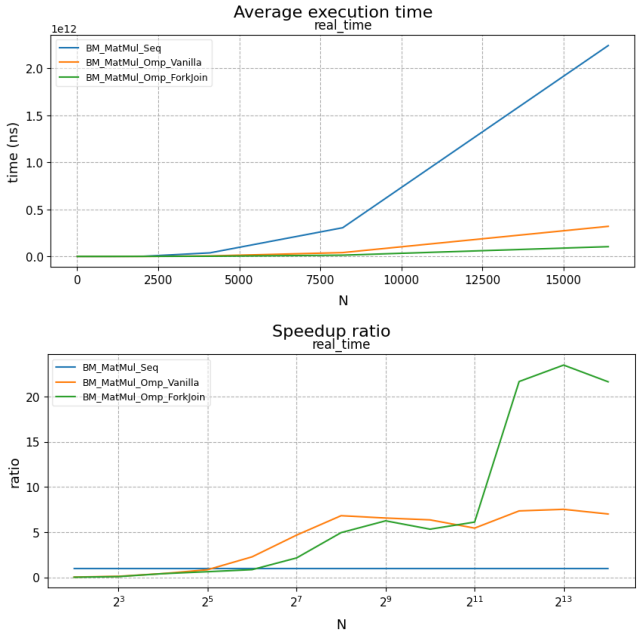


Figure 33: Execution time and speedup of matrix multiplication using 8 cores on System 3.

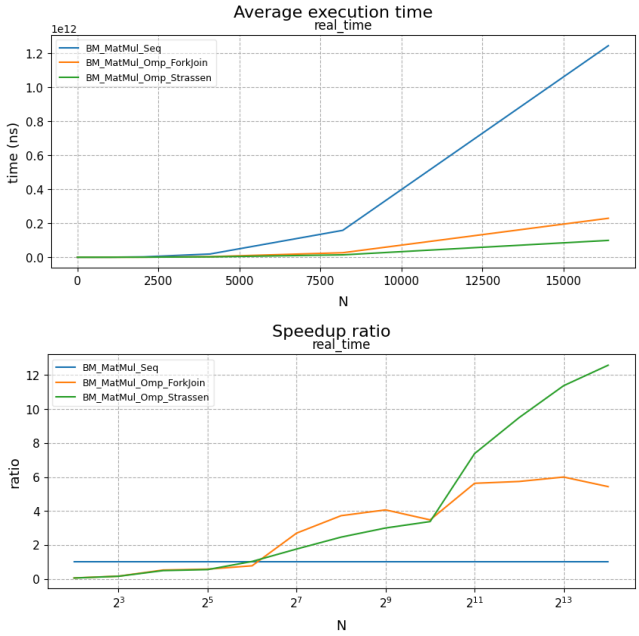


Figure 34: Execution time and speedup of matrix multiplication including Strassen using 8 cores on System 1.

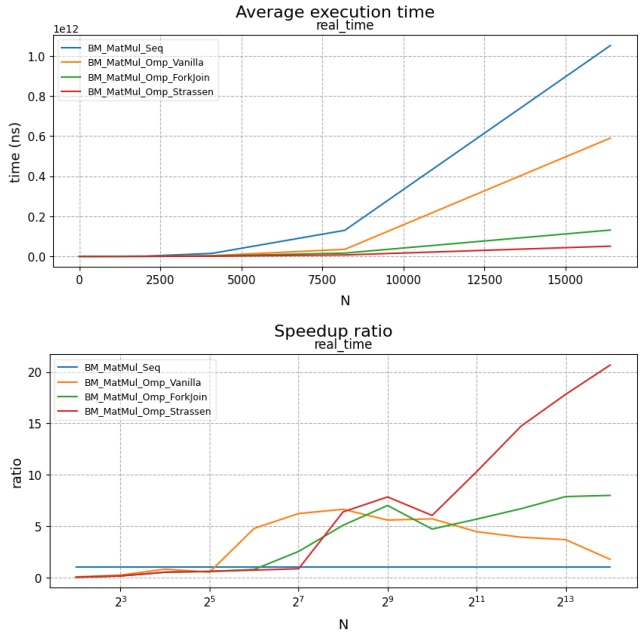


Figure 35: Execution time and speedup of matrix multiplication including Strassen using 8 cores on System 2.

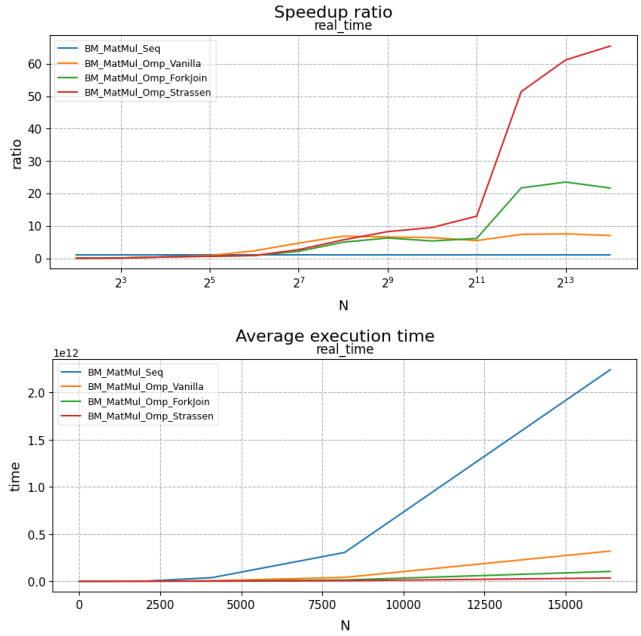


Figure 36: Execution time and speedup of matrix multiplication including Strassen using 8 cores on System 3.

benchmarked size, $N = 2^{14}$, the fastest configurations across all machines are consistently the 8-core Strassen runs. The exact execution times for Strassen and for `NumPy` on each system are summarized in Table 2.

A noteworthy observation emerges when comparing Systems 2 and 3. Although Strassen is markedly faster on System 3 than on System 2, NumPy exhibits nearly identical performance on the two machines. This contrast underscores the leak of author knowledge

Table 2: Execution times for Strassen (8 cores) and NumPy at $N = 2^{14}$ on three hardware systems.

System	Strassen (8 cores)	NumPy
System 1	98.77 s	109.52 s
System 2	50.86 s	14.21 s
System 3	34.27 s	14.57 s

regarding NumPy's internal optimizations, which likely include architecture-specific enhancements that mitigate performance disparities across different hardware configurations.

5.4 GPU Accelerated System

This section presents the experimental evaluation of the CUDA-based implementations on System 2. The experiments are divided into two benchmarks: (i) a comparison between the proposed tiled CUDA implementation (Algorithm 9) and the OpenMP fork–join approach (Algorithm 7), and (ii) a comparison between the tiled CUDA implementation and the GPU-based *PyTorch* implementation.

The experiments are conducted with matrix sizes ranging from $N \in [4, 16384]$. Larger matrix sizes are desirable for further evaluation; however, experiments beyond this range are constrained by the available GPU memory (VRAM). The reported execution times include both data transfers from host memory (RAM) to device memory (VRAM) and the corresponding transfers back to the host, thereby capturing end-to-end performance.

The comparison between tiled CUDA and the OpenMP fork–join implementation using 8 CPU cores is not strictly fair due to the fundamental differences between CPU and GPU architectures. Nevertheless, this benchmark is included to demonstrate the importance of assigning computational workloads to the most suitable hardware. As shown in Figure 37, once the matrix size becomes sufficiently large, the CUDA implementation significantly outperforms the CPU-based approach. In particular, for $N = 2^{14}$ (i.e., 16384), the tiled CUDA implementation achieves a speedup of approximately 236× over the CPU baseline.

A more balanced comparison is provided between the tiled CUDA implementation and the GPU-based *PyTorch* implementation. This comparison evaluates two GPU-resident solutions under similar execution conditions. As illustrated in Figure 38, the proposed **tiled CUDA implementation outperforms *PyTorch* by up to** 2.7×. This result indicates that the custom kernel effectively exploits GPU architectural features beyond those leveraged by the general-purpose *PyTorch* implementation.

5.5 Distributed Memory System

We observe that the baseline MPI matrix multiplication (Fig. 39) achieves noticeable speedup as the number of MPI processes increases; however, the speedup degrades once the number of processes exceeds eight—the number of physical cores on each node. This degradation is caused by context-switching overhead when more processes are spawned than available physical cores.

[16] version 2.9.0+cu126

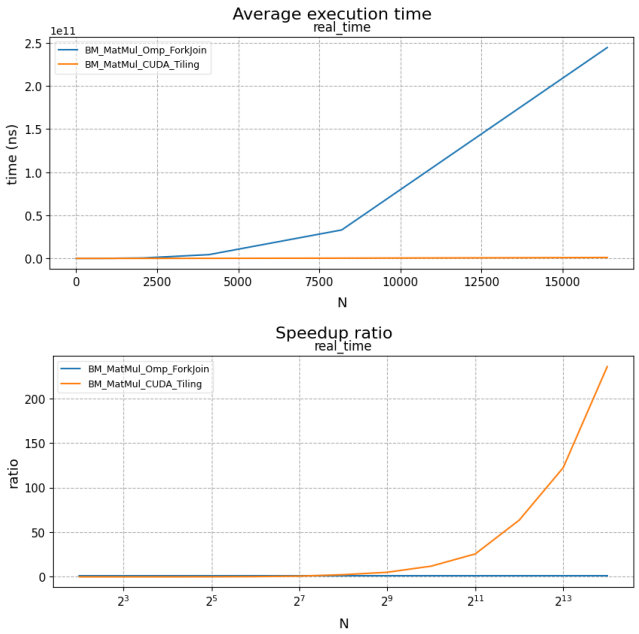


Figure 37: GPU tiling vs OMP Fork-Join matrix multiplication

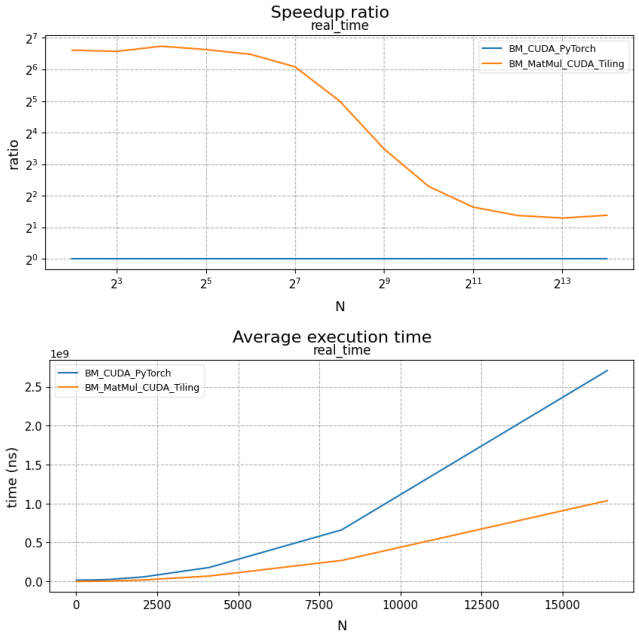


Figure 38: GPU tiling vs OMP Fork-Join matrix multiplication

When the matrix size reaches 2^{11} or larger, the computation time begins to dominate the communication cost, resulting in more substantial speedup. Furthermore, these results highlight the importance of data locality in distributed-memory systems: the baseline MPI method requires each worker to access the entire matrix B,

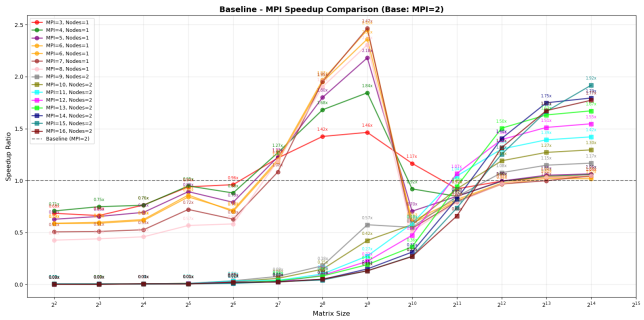


Figure 39: Speedup of Baseline MPI Matrix Multiplication versus number of MPI processes on Distributed-Memory system

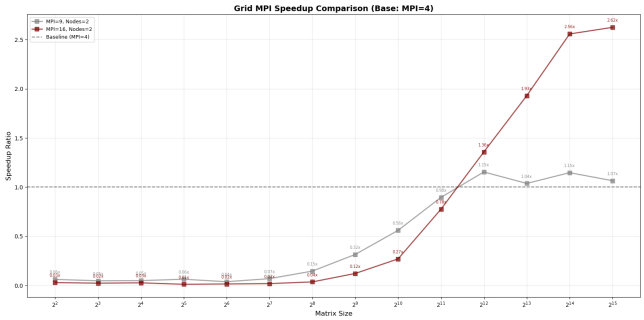


Figure 40: Speedup of Grid MPI Matrix Multiplication versus number of MPI processes on Distributed-Memory system

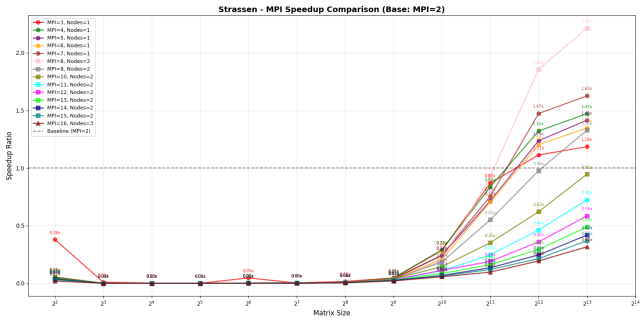


Figure 41: Speedup of Strassen MPI Matrix Multiplication versus number of MPI processes on Distributed-Memory system

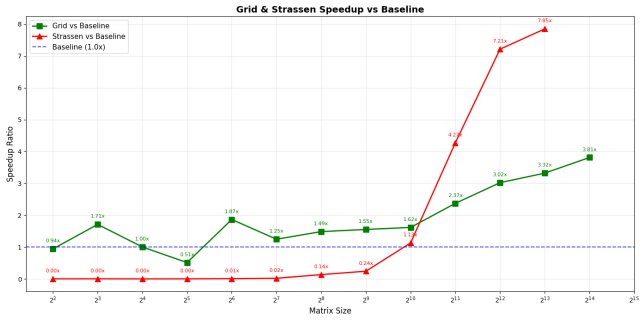


Figure 42: Execution Time of MPI Matrix Multiplication on Distributed-Memory system

which introduces significant communication overhead, especially for large matrices.

The grid MPI matrix multiplication (Fig. 40) demonstrates improved scalability compared to the baseline method. By distributing both matrices A and B across processes, the grid approach reduces communication overhead, leading to better performance as the number of MPI processes increases. However, this is only true when the matrix size is large enough (i.e., 2^{12} or larger) for computation to dominate communication costs. When it comes to smaller matrices, the overhead of splitting and gathering submatrices among processes can outweigh the benefits of reduced communication, resulting in less pronounced speedup and we can observe that in the figure the small matrices (smaller than 2^{12}) works best in case of numer of process is four. Which can be explained by the fact that with four processes, the data locality is maximized in comparison to other configurations.

As the result (Fig. 41), Strassen's algorithm shows limited benefit for small matrix sizes, where communication and management overhead dominate the execution time. As the matrix size increases, Strassen begins to outperform the baseline, with noticeable speedup improvements starting around 2^{10}. The best performance is achieved with a moderate number of MPI processes (around 7-8), indicating an optimal balance between parallelism and communication overhead. We choose 8 processes for other comparison.

As the benchmark (Figure 42), MPI-based Strassen performs poorly for small matrix sizes due to the overhead of recursion and communication that cannot be amortized at small scales. As the matrix size increases, MPI Strassen significantly outperforms, achieving much higher speedup. This result confirms that the computational complexity reduction of Strassen becomes beneficial only for large problem sizes.

Dask - MPI Library. All experiments were conducted using Dask version 2025.11.0 and Distributed version 2025.11.0, configured with 12 single-threaded workers, each allocated 4 GB of RAM (for a total of 48 GB of cluster memory), on a C6 cluster 2, in order to ensure fair resource usage and configuration when compared with other MPI-based algorithms.

Figure 43 indicate that the benchmark results do not exhibit a globally optimal chunk size applicable to all matrix dimensions. Instead, each matrix size corresponds to a different chunk size that yields the best performance [4], and therefore no fixed chunk size can be considered optimal for all matrix multiplication workloads in Dask. Based on this observation, all subsequent benchmarks of the implemented algorithms employ the empirically determined optimal chunk size for each matrix size to ensure fair and representative performance evaluation.

From Figure 44, Dask achieves the highest speedup for large matrix sizes, outperforming the other implemented algorithms when an appropriate chunk size is selected. However, Dask's performance

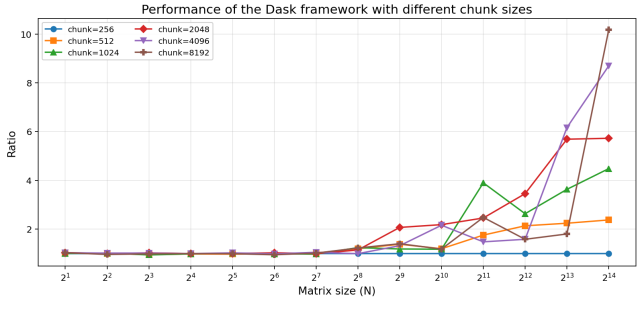


Figure 43: Speedup of Dask MPI Matrix Multiplication versus size of chunk on Distributed-Memory system

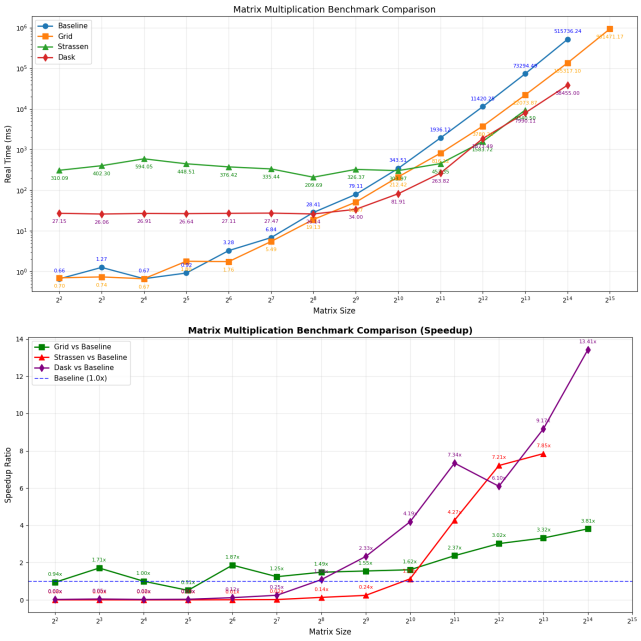


Figure 44: Execution Time and Speedup of MPI Matrix Multiplication (with Dask) on Distributed-Memory system

is less stable as the problem size varies, exhibiting noticeable fluctuations across different matrix dimensions. Although Dask can deliver high performance for large workloads, its effectiveness strongly depends on parameter tuning and is less robust when applied across a wide range of matrix sizes.

5.6 Hybrid Distributed Memory System

This experiment takes the best of three configurations in Baseline, Grid and Strassen Hybrid Matrix Multiplication (Section 4.5.1, Section 4.5.2 and Section 4.5.3) to evaluate their performance on a hybrid distributed-memory system with shared-memory nodes. Where Baseline's number of processes is 2 and Grid's number of processes is 4, both with a threshold of 64. While Strassen's number of processes is 8 and threshold of 32. Observing from Figure 45, we can see that both algorithms achieve significant speedup as the

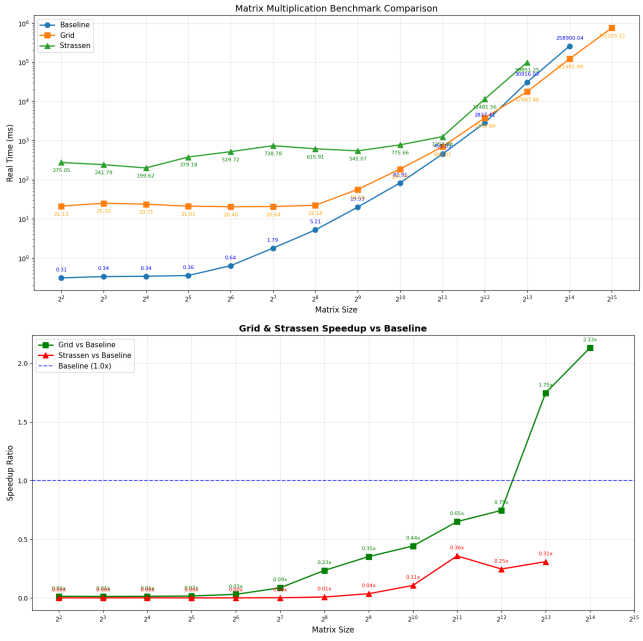


Figure 45: Execution Time and Speedup of Hybrid Matrix Multiplication on Hybrid Distributed-Memory system with Shared-Memory nodes

number of MPI processes increases. However, the Grid Hybrid Matrix Multiplication outperforms the Baseline approach, especially as the matrix size increases (larger than 2^{12}). This improvement arises from the grid-based data distribution, which enhances data locality and significantly reduces inter-process communication.

Although Strassen theoretically reduces the number of multiplications and increases computational throughput, it introduces additional communication and memory traffic due to the extra matrix additions and subtractions required to compute the seven intermediate products $M_1 \dots M_7$. These operations increase pressure on memory bandwidth and generate additional data movement across MPI processes, which becomes particularly costly in a distributed hybrid setting.

In contrast, the Grid method maintains stable communication patterns and does not incur the added traffic caused by Strassen's linear-combination steps. As a result, the Grid Hybrid Matrix Multiplication achieves better performance scalability than the Baseline method, while also avoiding the communication overhead intrinsic to Strassen's algorithm.

Dask - Hybrid Library. All experiments were conducted using Dask version 2025.11.0 and Distributed version 2025.11.0, configured with 3 workers, each running 4 threads and allocated 16 GB of RAM (for a total of 48 GB of cluster memory), on a C6 cluster, in order to ensure fair resource usage and configuration when compared with other Hybrid algorithms.

Based on the MPI benchmark results, we implement a hybrid approach using Dask, and the results (see Figure 46) demonstrate higher efficiency in matrix multiplication compared to our implemented algorithms.

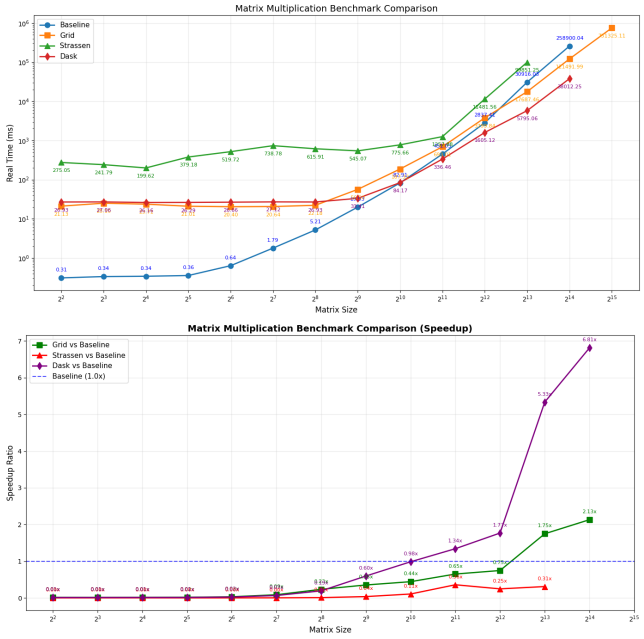


Figure 46: Execution Time and Speedup of Hybrid Matrix Multiplication (with Dask) on Hybrid Distributed-Memory system with Shared-Memory nodes

6 Conclusion

This report has presented a systematic experimental study of square matrix multiplication across multiple computing paradigms, including shared-memory systems, GPU-accelerated execution, distributed-memory clusters, and hybrid architectures. We implemented and evaluated a range of algorithms—from the classical sequential IKJ baseline and cache-aware blocked methods to recursive fork–join formulations and Strassen's asymptotically faster algorithm—under a unified evaluation framework spanning three distinct hardware configurations.

Our investigation began with an analysis of cache behavior, where we demonstrated that the IJK loop order suffers severe conflict misses at power-of-two matrix dimensions due to concentrated cache-set utilization, while the IKJ order avoids this pathology entirely. This finding underscores the critical importance of memory access patterns in achieving high performance.

In shared-memory experiments, we observed superlinear speedups exceeding 20× on 8 cores for matrix sizes in the range $[2^{10}, 2^{14}]$, attributable to improved cache utilization when workloads are distributed across multiple cores. Strassen's algorithm consistently delivered the fastest execution times for large matrices, achieving approximately 7× speedup over the sequential IKJ baseline at $N = 2^{14}$. Notably, our 8-core Strassen implementation outperformed NumPy on certain systems, highlighting the potential of algorithm-architecture co-optimization.

For GPU-accelerated execution, our custom tiled CUDA kernel achieved speedups of approximately 236× over the CPU-based fork–join implementation at $N = 2^{14}$ and outperformed PyTorch by up to 2.7×, demonstrating that hand-optimized GPU kernels can substantially exceed the performance of general-purpose frameworks.

In distributed-memory experiments using MPI, the grid-based approach outperformed the baseline method by exploiting improved data locality and reduced communication overhead. MPI-based Strassen showed limited benefit for small matrices due to communication costs but delivered significant speedups for large problem sizes. Experiments with Dask revealed that while it can achieve the highest speedup for large matrices with appropriate chunk sizes, its performance is sensitive to parameter tuning and less robust across varying matrix dimensions.

The hybrid experiments combining MPI with OpenMP revealed that the grid-based distribution strategy outperforms both the baseline and Strassen-based hybrid approaches. Although Strassen reduces arithmetic operations, the additional memory traffic from intermediate matrix additions and subtractions introduces communication overhead that diminishes its advantage in distributed settings.

In summary, this study demonstrates that achieving optimal matrix multiplication performance requires careful consideration of algorithmic structure, memory hierarchy behavior, and communication patterns tailored to the target architecture. Future work could explore adaptive algorithms that dynamically select between classical and Strassen multiplication based on problem size and system characteristics, extend these approaches to heterogeneous computing environments combining CPUs and GPUs, and investigate more sophisticated communication-avoiding algorithms for distributed systems.

7 Contribution

Each group member is responsible for carrying out the experimental studies associated with their assigned theoretical components.

Table 3: Team contribution

Name	Contribution	Percent
Tuong Nguyen Ha 2250013	Project & report structure, Shared memory system (except for Strassen) & CUDA. Section 1, 2, 3.	100%
Quoc Khai Phan 2252339	MPI Baseline, Hybrid Baseline, MPI Grid, and Hybrid Grid matrix multiplication Algorithms.	100%
Quynh Huong Vu 2252286	All Strassen Implementation. Dask MPI and Dask hybrid.	100%

Acknowledgments

I would like to express my sincere gratitude to the Advanced Institute of Interdisciplinary Science and Technology (iST) for providing the computational resources used in this study, and my special thanks go to Duc-Phuong Doan-Ngo, Cong Tu Vu, and Phuc Hung Dinh from the High Performance Computing Laboratory (HPC) for their invaluable support, as well as to the anonymous contributors

whose comments and feedback have greatly improved the quality of this work.

References

[1] Josh Alman and Virginia Vassilevska Williams. 2021. A Refined Laser Method and Faster Matrix Multiplication. *Proceedings of the 2021 ACM–SIAM Symposium on Discrete Algorithms (SODA)* (2021), 522–539. doi:10.1137/1.9781611976465.32

[2] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Communication-optimal parallel algorithm for Strassen's matrix multiplication. *Journal of the ACM (JACM)* 59, 6 (2012), 1–23. doi:10.1145/2395116.2395121

[3] Don Coppersmith and Shmuel Winograd. 1990. Matrix Multiplication via Arithmetic Progressions. *Journal of Symbolic Computation* 9, 3 (1990), 251–280. doi:10.1016/S0747-7171(08)80013-2

[4] Dask Development Team. 2024. Dask Array Best Practices. https://docs.dask.org/en/latest/array-best-practices.html. Section: Select a good chunk size.

[5] Thanh-Dang Diep, Phuong Hoai Ha, and Karl Fürlinger. 2023. A general approach for supporting nonblocking data structures on distributed-memory systems. *J. Parallel and Distrib. Comput.* 173 (2023), 48–60. doi:10.1016/j.jpdc.2022.11.006

[6] Alhussein Fawzi, Matej Balog, Angelica Chen Huang, Andrew Brock, Laurent Sifre, Pawel Lichocki, Pablo Samuel Castro Jose Olmos, Kimberly Feldman, Andre Barreto, Thomas Keck, Pushmeet Kohli, Stephen McAleer, Søren Larsen, Omar Fawzi, Nando de Freitas, Oriol Vinyals, and David Silver. 2022. Discovering Faster Matrix Multiplication Algorithms with Reinforcement Learning. *Nature* 610, 7930 (2022), 47–53. doi:10.1038/s41586-022-05172-4

[7] M Frigo, CE Leiserson, H Prokop, and S Ramachandran. [n. d.]. Cache-oblivious algorithms, Extended abstract. Lab for Computer Science. MIT (May 1999). http://supertech. lcs. mit. edu/cilk/papers/abstracts ….

[8] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE, 285–297.

[9] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE, 285–297.

[10] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Trans. Algorithms* 8, 1, Article 4 (Jan. 2012), 22 pages. doi:10.1145/2071379.2071383

[11] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed H. Sameh. 1988. Impact of Hierarchical Memory Systems On Linear Algebra Algorithm Design. *Int. J. High Perform. Comput. Appl.* 2, 1 (March 1988), 12–48. doi:10.1177/109434208800200103

[12] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *nature* 585, 7825 (2020), 357–362.

[13] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[14] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Turnbull. 1996. Implementation of Strassen's Algorithm for Matrix Multiplication. In *Proceedings of Supercomputing '96 (SC'96)*. IEEE.

[15] Elaye Karstadt and Oded Schwartz. 2017. Matrix Multiplication, a Little Faster. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (Washington, DC, USA) *(SPAA '17)*. Association for Computing Machinery, New York, NY, USA, 101–110. doi:10.1145/3087556.3087579

[16] Kaushik Ravindran. 2007. *Task Allocation and Scheduling of Concurrent Applications to Multiprocessor Systems.* Technical Report. Electrical Engineering and Computer Sciences, University of California at Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-149.html

[17] Sasko Ristov, Radu Prodan, Marjan Gusev, and Karolj Skala. 2016. Superlinear speedup in HPC systems: Why and when?. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 889–898.

[18] A. Schönhage. 1981. Partial and Total Matrix Multiplication. *SIAM J. Comput.* 10, 3 (1981), 434–455. doi:10.1137/0210032 arXiv:https://doi.org/10.1137/0210032

[19] Oded Schwartz and Noa Vaknin. 2023. Pebbling Game and Alternative Basis for High Performance Matrix Multiplication. *SIAM Journal on Scientific Computing* 45, 6 (2023), C277–C303. doi:10.1137/22M1502719 arXiv:https://doi.org/10.1137/22M1502719

[20] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. 2021. *Operating System Concepts* (10 ed.). John Wiley & Sons, Nashville, TN.

[21] Andrew James Stothers. 2010. *On the Complexity of Matrix Multiplication.* Ph. D. Dissertation. University of Edinburgh.

[22] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.* 13, 4 (Aug. 1969), 354–356. doi:10.1007/bf02165411

[23] Linus Torvalds. [n. d.]. GitHub - torvalds/linux: Linux kernel source tree — github.com. https://github.com/torvalds/linux.git. [Accessed 27-11-2025].

[24] Virginia Vassilevska Williams. 2012. Multiplying Matrices Faster Than Coppersmith–Winograd. *Proceedings of the 44th ACM Symposium on Theory of Computing (STOC)* (2012), 887–898. doi:10.1145/2213977.2214056

[25] S. Winograd. 1968. A New Algorithm for Inner Product. *IEEE Trans. Comput.* C-17, 7 (1968), 693–694. doi:10.1109/TC.1968.227420

[26] S. Winograd. 1971. On multiplication of 2 × 2 matrices. *Linear Algebra Appl.* 4, 4 (1971), 381–388. doi:10.1016/0024-3795(71)90009-7

Received December 17 2025