VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PROGRAMMING INTERGRATION PROJECT

SEMESTER 251 ACADEMIC YEAR 2025-2026

JOURNEY TO SC2031: FUNDAMENTALS OF PARALLEL AND HIGH-PERFORMANCE COMPUTING

MAJOR: COMPUTER SCIENCE

SUPERVISOR: PhD. DIEP THANH DANG

——o0o——

STUDENT 1: HA TUONG NGUYEN - 2250013

HO CHI MINH CITY, January 2026

Declaration of Authenticity

We declare that we solely conducted this specialized project, under the supervision of PhD. Diep Thanh Dang at the Faculty of Computer Science and Engineering, Vietnam National University - Ho Chi Minh City University of Technology.

We have taken care to properly acknowledge and document all external sources and references used in the project.

If there is any instance of plagiarism, we are ready to accept the consequences. Ho Chi Minh City University of Technology - Vietnam National University HCMC will not be held responsible for any copyright violations that may have occurred during my research.

Ho Chi Minh City, January 2026

Ha Tuong Nguyen,

Acknowledgement

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. I am especially grateful to Professor Thoai Nam, Head of the iST Laboratory, for his continuous support of research students and his generosity in resource allocation. I also thank Dr. Diep Thanh Dang for his valuable guidance and academic mentorship. My appreciation further extends to Duc-Phuong Doan-Ngo, Cong Tu Vu, and Phuc Hung Dinh from the High Performance Computing Laboratory for their technical assistance.

I would like to acknowledge my 8th-grade English teacher, Ms. Vu Tinh Van, whose encouragement played an important role in shaping my academic aspirations and motivating my pursuit of a research-oriented career. I also thank my friends at the Big Data Club and iST for their companionship during coursework and examinations, which indirectly provided the time and motivation needed to complete this work.

Finally, I am deeply grateful to my family for their unconditional support throughout my studies. I also wish to express my sincere thanks to Nguyen Dao Thuy Duong for her constant encouragement, patience, and understanding during the long hours devoted to this project.

Abstract

This report is submitted as part of the Programming Integration Project, formally registered under the topic *"Work Stealing on GPUs"*. Rather than presenting a narrowly focused investigation of a single algorithmic technique, the report aims to consolidate fundamental concepts, system-level knowledge, and practical experimentation in the broader domains of Parallel Computing and High-Performance Computing (HPC).

This report is submitted as part of the Programming Integration Project and is formally registered under the topic "Work Stealing on GPUs". Rather than presenting a narrowly focused investigation of a single algorithmic technique, the report aims to consolidate fundamental concepts, system-level knowledge, and practical experimentation in the broader domains of Parallel Computing and High-Performance Computing (HPC).

The primary objective of this work is to establish a solid theoretical and experimental foundation covering CPU and GPU architectures, memory hierarchies, cache behavior, synchronization mechanisms, and parallel scheduling strategies. Through a combination of architectural analysis and empirical evaluation, particularly using matrix multiplication workloads implemented with sequential, OpenMP, and CUDA-based approaches, the report demonstrates how low-level design decisions influence performance on modern multicore and manycore systems. Due to its exploratory and integrative nature, the report does not follow a single linear research narrative. Instead, it is structured as a curated collection of concepts, experiments, and observations that reflect the learning trajectory required for advanced HPC research. This work is intended to serve as a preparatory step toward future, more specialized research contributions in parallel runtime systems and GPU scheduling, with long-term aspirations of participating in the Supercomputing (SC) conference.

Contents

1 Introduction **1**

2 Central Processing Unit (CPU) **3**

2.1 CPU Architecture 3

2.2 Threads vs Processes 4

2.3 Hardware/Software Threads and Scheduling 5

2.4 Hyper-Threading (Simultaneous Multi-Threading) 6

3 Memory System on CPU **8**

3.1 Memory Hierarchy 8

3.2 Cache Memory 9

3.3 Cache Placement Policies 11

3.4 Cache Replacement Policies 12

3.5 Cache Misses 12

3.6 Race Conditions and Memory Consistency 14

4 Cache Friendly Programming **16**

4.1 Data Structure Alignment 16

4.2 Cache Aware vs Oblivious Algorithms 17

5 Graphics Processing Unit (GPU) **20**

5.1 Fundamental Differences: CPU vs. GPU 20

5.2 GPU Macro-Architecture 21

5.3 The Streaming Multiprocessor (SM) 23

5.4 Execution Units and Resources 23

5.5 Thread Hierarchy and Execution Model 25
5.6 Memory Hierarchy 26
5.6.1 Memory Spaces 27
5.6.2 Hardware Implementation 28

6 Superlinear Speedup 29

7 Matrix Multiplication Algorithms and Optimizations 33
7.1 Matrix Multiplication Introduction 33
7.2 Matrix Multiplication Related Work 34
7.3 Numeric Stable with Kahan Summation 36
7.4 Sequential Matrix Multiplication 36
7.5 Parallel Matrix Multiplication on Shared-Memory Systems 38
7.6 Matrix Multiplication on GPU 44

8 Matrix Multiplication on CPU and GPU Experiments 47
8.1 Experimental Setup 47
8.2 Conflict Miss 47
8.3 Shared Memory System 51
8.3.1 Addition Operator 51
8.3.2 Tuning the Optimal Threshold 52
8.3.3 Sequential Performance 55
8.3.4 Multi-core Performance 55
8.4 GPU Accelerated System 56

9 Conclusion 58

References 58

List of Figures

2.1 Architecture of a single Intel Skylake core [1]. 4
2.2 Cycle costs for different CPU operations [10]. 5
2.3 Illustration of a single-threaded process (left) and threads within a multithreaded process (right). 6
2.4 Mapping from user-level software threads to kernel threads, and finally to physical hardware threads. 6
2.5 A single physical core exposing two hardware threads via Hyper-Threading technology. 7
2.6 Interleaved execution of two threads on a dual-threaded core to hide memory latency [22]. 7

3.1 Memory hierarchy of the Intel(R) Core(TM) i5-1038NG7 CPU. The system features private L1 and L2 caches, a shared L3 cache, main memory, and SSD storage. 9
3.2 A cache line is a contiguous block of memory transferred between different levels of the memory hierarchy. Typical cache line sizes range from 32 to 256 bytes. 9
3.3 Data path diagram illustrating the interaction between the core, TLB, and cache hierarchy during a memory request for a 32-bit virtual address [12]. 10
3.4 Impact of cache size and associativity (1-way to 8-way) on relative access time [12]. 11
3.5 Impact of cache size and associativity on relative energy consumption per read [12]. 11
3.6 Taxonomy of cache misses in multi-core processors. 13

3.7 Illustration of conflict misses in a direct-mapped cache. 13
3.8 Illustration of a coherence miss caused by a write–invalidate protocol. . 14

4.1 Impact of field ordering on memory layout and internal padding. 17
4.2 Visualizing the three fundamental rules of data structure alignment. . . 18

5.1 CPU (Latency Oriented) vs. GPU (Throughput Oriented) [16]. 21
5.2 Macro-architecture of the NVIDIA GA100 GPU, highlighting GPCs and TPCs [16]. 22
5.3 Internal architecture of a Streaming Multiprocessor (SM) showing the Quad-Partition design [16]. 24
5.4 The GPU Thread Hierarchy: From Grid to Warp [16]. 25
5.5 Mapping of CUDA Software Threads to Hardware Resources [27]. . . . 26
5.6 Logical Data Access Levels and Memory Hierarchy. 27

6.1 Visualization of a non-structure persistent algorithm, where parallel execution reduces the total number of computational steps, enabling superlinear speedup. 30
6.2 Taxonomy of superlinear speedup, distinguishing algorithmic (non-persistent) and systemic (persistent) causes. Based on the classification by Ristov et al. Sasko Ristov et al. “Superlinear Speedup in HPC Systems: why and when?” In: *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*. Vol. 8. FedCSIS 2016. IEEE, Oct. 2016, pp. 889–898. DOI: 10.15439/2016f498. URL: http://dx.doi.org/10.15439/2016F498. 31

7.1 Memory access pattern for `ijk` ordering. Elements of matrix B are accessed non-contiguously, leading to poor spatial locality and higher cache miss rates. 39
7.2 Memory access pattern for `ikj` ordering. The value $A[\mathtt{i},\mathtt{k}]$ is reused while accessing a contiguous row segment of B, resulting in excellent spatial locality. 39

7.3 $N \times N$ 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. 41
7.4 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. 42
7.5 Memory access pattern using tiling with block size $k = \sqrt{n}$. Each tile reuses data inside the block, reducing memory accesses to $n\left(1 + \frac{2}{\sqrt{n}}\right)$. 42
7.6 Charles E. Leiserson The G.O.A.T 42

8.1 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. 48
8.2 The cache lines of matrix B (represented as cells) are assigned set indices according to the 6 set-index bits in subfigure (I). 50
8.3 Illustration of cache misses. 50
8.4 Impact of loop ordering on cache conflict misses in naive matrix multiplication for matrix sizes ranging from 497 to 527. 50
8.5 Execution time and speedup of vector addition using 8 cores on System 8.1. 52
8.6 Execution time and speedup of matrix addition using 8 cores on System 8.1. 52
8.7 Threshold tuning for fork–join (Algorithm 6) with IJK+Kahan, IJK, and IKJ base cases using 8 cores on System 8.1. 54
8.8 Execution time and speedup of sequential matrix multiplication on System 8.1. 55
8.9 Execution time and speedup of matrix multiplication using 8 cores on System 8.1. 56
8.10 Execution time and speedup of matrix multiplication using 8 cores on System 8.2. 56
8.11 GPU tiling vs OMP Fork–Join matrix multiplication. 57
8.12 GPU tiling vs OMP Fork–Join matrix multiplication in PyTorch. 57

List of Tables

8.1 System specifications of single compute node for experiments 49

Chapter 1

Introduction

The rapid evolution of multicore CPUs and massively parallel GPUs has fundamentally reshaped the landscape of modern computing. Achieving high performance on such systems requires not only algorithmic parallelism, but also a deep understanding of hardware architecture, memory hierarchies, synchronization costs, and scheduling strategies. As a result, High Performance Computing (HPC) has become an inherently interdisciplinary field that combines computer architecture, systems programming, and parallel algorithms.

This report is developed as part of the Programming Integration Project, with an initial registered topic of *"Work Stealing on GPUs"*. However, effective research on advanced scheduling mechanisms such as work stealing presupposes strong familiarity with lower level architectural and system concepts. Consequently, the scope of this report has been intentionally broadened to emphasize foundational knowledge and practical experimentation, rather than a single specialized contribution.

The report is organized to progressively build this foundation. It begins with an overview of CPU and GPU architectures, including hardware and software threading models, memory hierarchies, cache organizations, and execution units. Fundamental issues in parallel programming, such as race conditions, synchronization primitives, atomic operations, and scheduling policies, are then examined to illustrate common performance pitfalls and correctness challenges.

A substantial portion of the report is devoted to empirical studies using matrix multiplication as a representative workload. Sequential implementations, OpenMP based ap-

proaches, and CUDA based designs are analyzed to investigate cache behavior, conflict misses, data alignment effects, threshold tuning, and scalability across multiple cores and GPUs. These experiments serve as concrete demonstrations of how theoretical concepts manifest in real systems.

Rather than proposing a novel algorithm or runtime system, this report aims to integrate theory and practice into a coherent knowledge base. The resulting work provides a stepping stone toward future research on dynamic scheduling and work stealing mechanisms for GPUs and heterogeneous systems.

Chapter 2

Central Processing Unit (CPU)

This chapter provides a comprehensive overview of the Central Processing Unit (CPU), serving as the foundational knowledge for understanding high-performance computing. We begin by examining the microarchitecture of modern processors, highlighting the complexity of execution units and the varying costs of CPU operations in Section 2.1. Subsequently, we distinguish between the fundamental units of execution (processes and threads) within the context of the Linux kernel (Section 2.2). The chapter further explores the critical mapping between user-level software threads and physical hardware resources in Section 2.3, concluding with an analysis of Simultaneous Multi-Threading (Hyper-Threading) technology and its role in mitigating memory latency in Section 2.4.

2.1 CPU Architecture

Modern CPU architectures are highly complex, comprising various components designed to execute instructions efficiently. Figure 2.1 illustrates the microarchitecture of a single core in the Intel Skylake processor family. This block diagram highlights the intricate data paths, execution units, and cache hierarchy that facilitate high-performance computing [1].

Notably, not all CPU operations incur the same computational cost. As depicted in Figure 2.2, different instructions require varying numbers of clock cycles to complete. For instance, simple arithmetic operations are significantly faster than complex floating-point calculations or memory access operations [10].

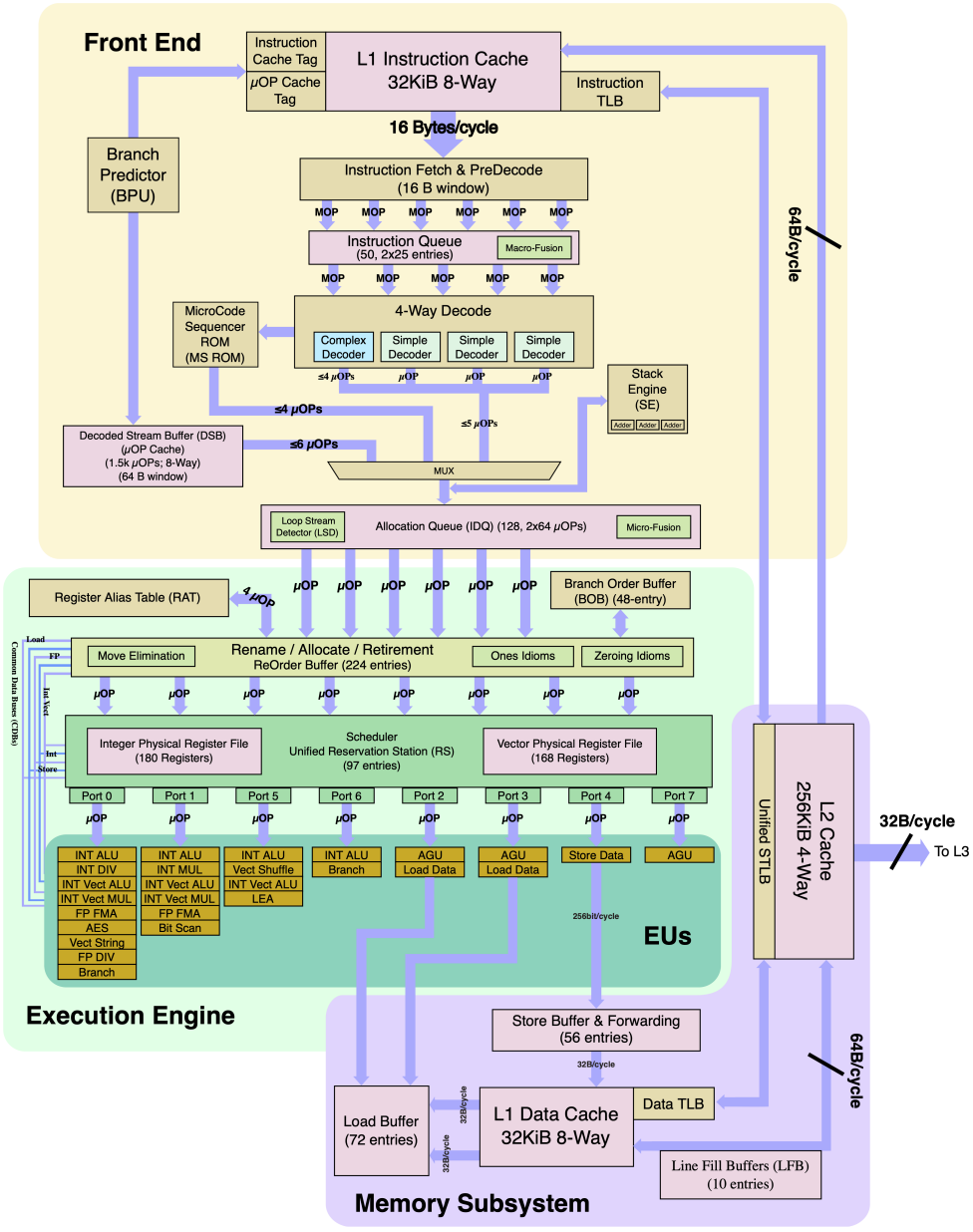


Figure 2.1: Architecture of a single Intel Skylake core [1].

2.2 Threads vs Processes

In the Linux kernel, the term **task** is often used interchangeably to refer to either a process or a thread, primarily due to the absence of a strict distinction in the kernel's internal representation [26]. The Process Control Block (PCB) is implemented as the `task_struct` structure, defined in `include/linux/sched.h`. Scheduling decisions rely on the "schedulability" of tasks—an abstract property determined by factors such as task priority, cache and core locality, and other architecture-specific metrics.

A task generally represents a single-threaded process or an individual thread within a

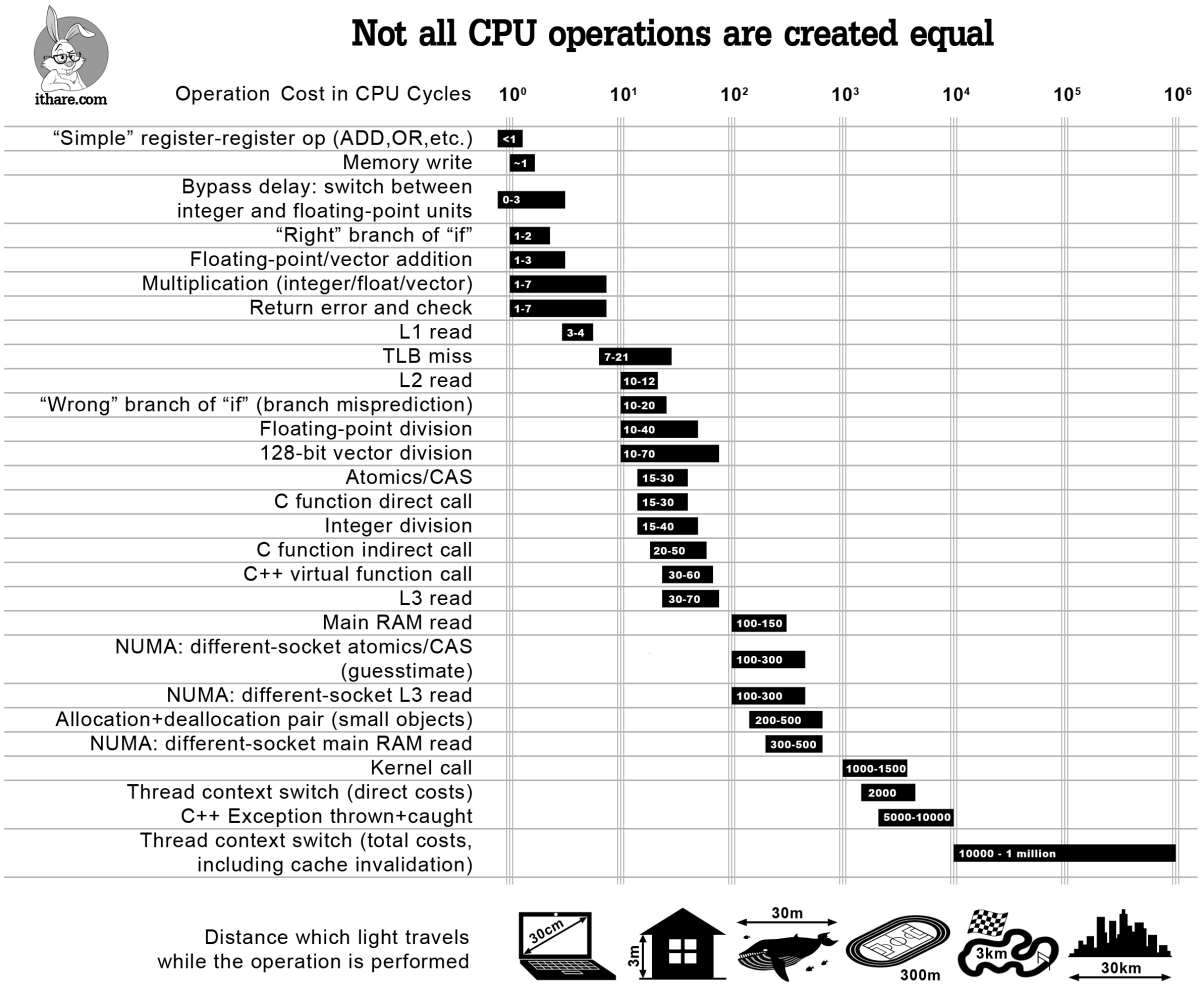


Figure 2.2: Cycle costs for different CPU operations [10].

multithreaded process. As illustrated in Figure 2.3, a single-threaded process (left) consists of a single stream of execution, whereas a multithreaded process (right) comprises multiple threads sharing the same process resources.

2.3 Hardware/Software Threads and Scheduling

A **thread** represents the fundamental unit of execution scheduled on a physical core, or on a logical core in processors that support Simultaneous Multi-Threading (SMT), such as Hyper-Threading. While software can create an arbitrary number of **"software threads"** (e.g., to wait for I/O operations or handle requests from external devices, as commonly seen in server applications), the number of **"hardware threads"** is fixed and determined by the underlying system architecture [13].

For instance, the Intel Core i7-9700 CPU (3.00GHz) features eight cores and eight

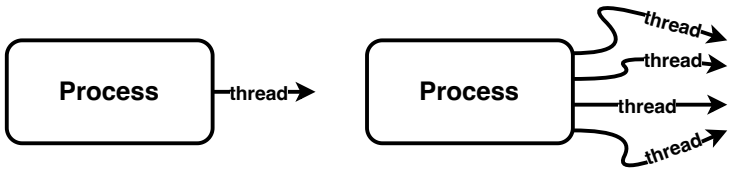


Figure 2.3: Illustration of a single-threaded process (left) and threads within a multithreaded process (right).

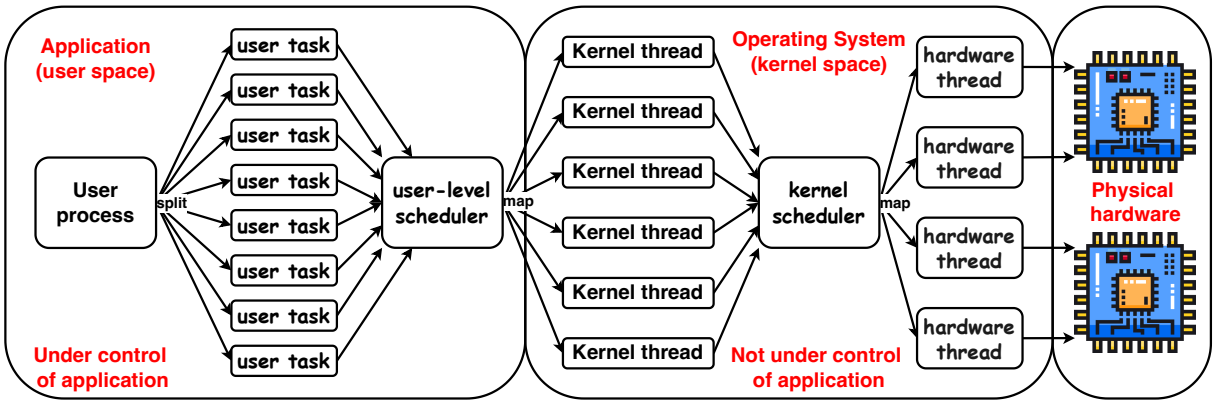


Figure 2.4: Mapping from user-level software threads to kernel threads, and finally to physical hardware threads.

hardware threads (one thread per core), indicating the absence of Hyper-Threading. In contrast, the Intel Core i5-1038NG7 CPU (2.00GHz) exposes four cores and eight hardware threads (two threads per core), reflecting support for Hyper-Threading.

When multiple software threads are created by a process, they are typically managed by a threading library or runtime environment. In most modern operating systems, each software thread is mapped to a corresponding kernel thread. Once this mapping is established, the kernel scheduler determines their execution order and placement on the underlying hardware resources, effectively mapping each kernel thread onto a hardware thread (see Figure 2.4).

2.4 Hyper-Threading (Simultaneous Multi-Threading)

Hyper-Threading Technology is a hardware feature that allows multiple execution threads to run simultaneously on each physical core [13]. From the operating system's perspective, these hardware threads appear as distinct logical (virtual) cores, as shown in Fig-

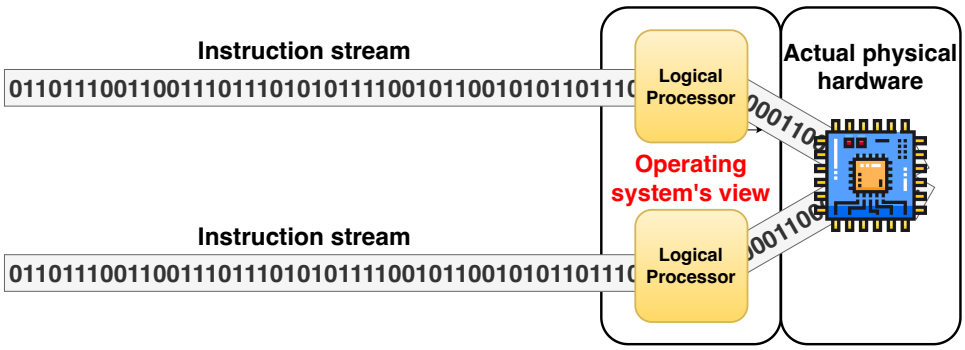


Figure 2.5: A single physical core exposing two hardware threads via Hyper-Threading technology.

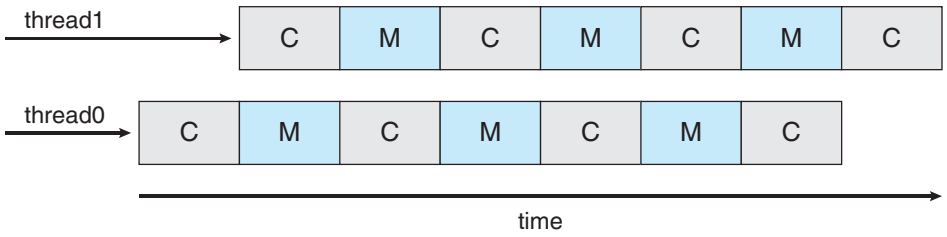


Figure 2.6: Interleaved execution of two threads on a dual-threaded core to hide memory latency [22].

ure 2.5.

The primary motivation for Hyper-Threading is to mitigate the latency caused by **memory stalls**. A memory stall occurs when the processor, which operates significantly faster than the memory subsystem, is forced to pause while waiting for data retrieval (due to cache misses, page faults, etc.). This latency can potentially waste up to 50% of execution time [22].

As illustrated in Figure 2.6, multithreaded cores (supporting ≥ 2 hardware threads) allow the hardware to context-switch to another thread whenever the current thread stalls. This interleaved execution ensures better utilization of the physical core's execution units.

However, Hyper-Threading is not without drawbacks. The performance benefits are often negligible when executing computationally light or non-intensive tasks. Furthermore, it has been reported to increase power consumption [15] and can lead to **cache contention**. This phenomenon occurs when threads competing for shared cache resources frequently evict each other's data, potentially degrading overall performance [22].

Chapter 3

Memory System on CPU

This chapter explores the fundamental architecture and design principles of modern computer memory systems, which are critical for bridging the performance gap between high-speed processors and slower storage. We begin by examining the **Memory Hierarchy** in Section 3.1, illustrating how multiple levels of cache (L1, L2, LLC) interact with main memory to optimize access latency. Subsequent sections delve into the mechanics of **Cache Memory** in Section 3.2, analyzing the trade-offs between associativity, hit rates, and energy consumption, as well as the specific **Cache Placement Policies** (Section 3.3) and **Cache Replacement Policies** (Section 3.4) that govern data management. The discussion extends to the classification of **Cache Misses** in Section 3.5, which includes cold, capacity, conflict, and coherence misses, and addresses the complexities of shared-memory multiprocessing, such as **Race Conditions** and memory consistency in Section 3.6. Finally, we investigate the practical impact of **Data Structure Alignment** in Section 4.1 regarding compiler padding, memory layout, and program efficiency.

3.1 Memory Hierarchy

Modern processors rely on a complex memory hierarchy to balance the disparity between processor speed and memory latency. For example, as illustrated in Figure 3.1, the Intel Core i5-1038NG7 processor exemplifies this design by integrating multiple levels of on-chip cache. Each core possesses a private Level 1 (L1) cache divided into a 32 KB instruction cache and a 48 KB data cache, along with a larger 512 KB private Level 2 (L2)

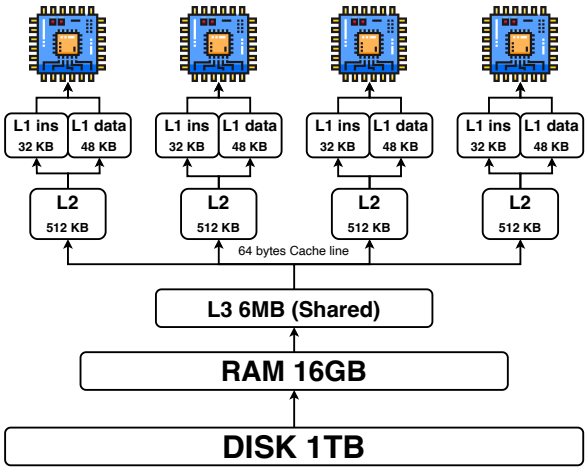


Figure 3.1: Memory hierarchy of the Intel(R) Core(TM) i5-1038NG7 CPU. The system features private L1 and L2 caches, a shared L3 cache, main memory, and SSD storage.

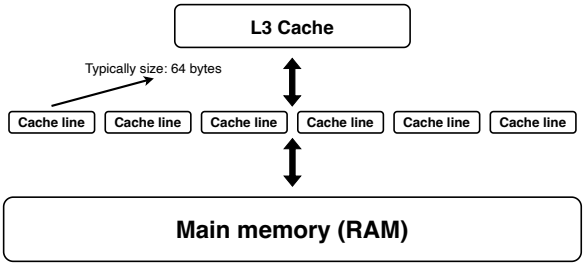


Figure 3.2: A cache line is a contiguous block of memory transferred between different levels of the memory hierarchy. Typical cache line sizes range from 32 to 256 bytes.

cache. These cores share a substantial 6 MB Level 3 (LLC) cache. While the processor operates on data at byte granularity, data transfer between these memory levels occurs in fixed-size blocks known as cache lines (see Figure 3.2).

When the processor issues a memory request, it initiates a search in the closest cache level. If the data is absent, the core consults the Translation Lookaside Buffer (TLB), a specialized cache that expedites virtual-to-physical address translation [12]. Figure 3.3 depicts this data path, showing how a 32-bit virtual address indexes both the TLB and the cache hierarchy simultaneously. If the requested entry is not found in the TLB or the caches, the request propagates to main memory (SDRAM) or secondary storage (SSD).

3.2 Cache Memory

Design decisions in cache architecture involve critical trade-offs between hit rates, access latency, and energy consumption. A key design parameter is the degree of associativity, which dictates how many distinct cache lines a specific memory block can map to. While increasing associativity generally reduces conflict misses, it introduces penalties in both speed and power due to the increased complexity of the hardware required for tag comparisons and data selection.

Figure 3.4 illustrates the relationship between cache size, associativity, and access

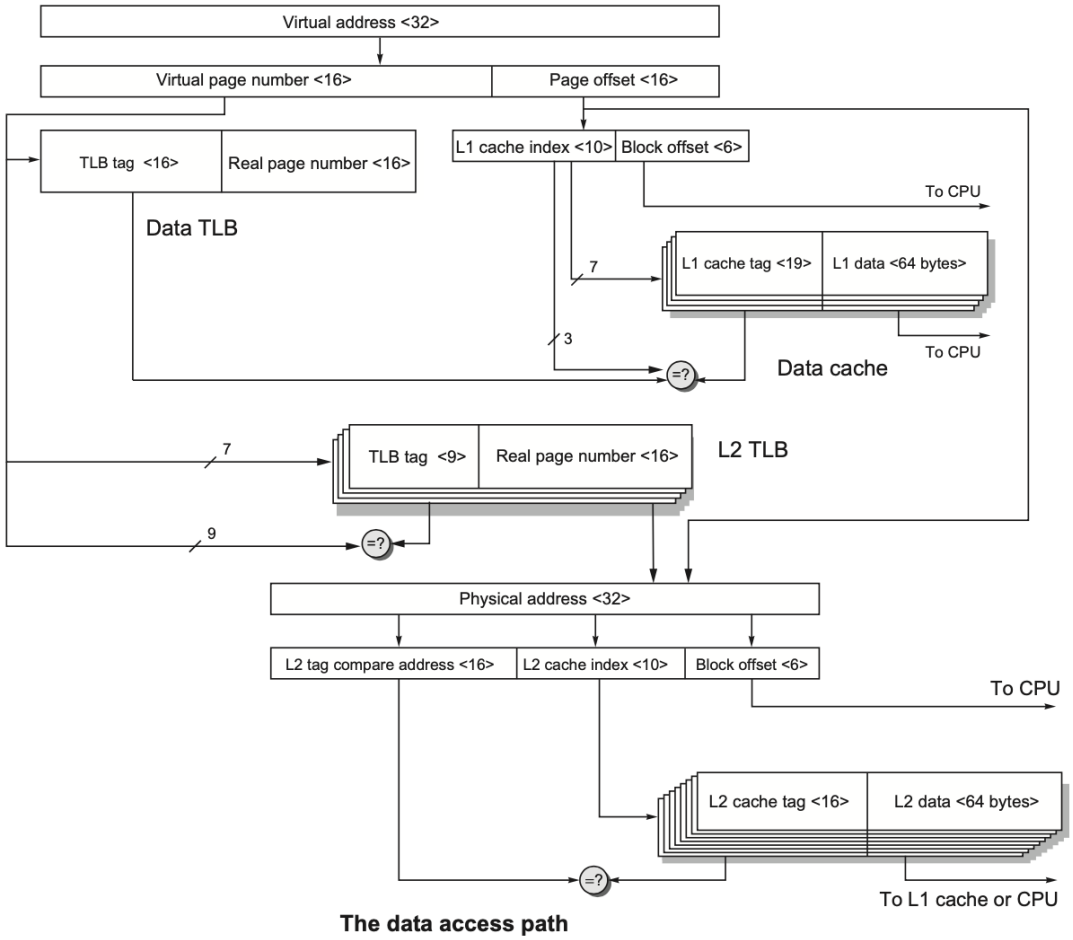


Figure 3.3: Data path diagram illustrating the interaction between the core, TLB, and cache hierarchy during a memory request for a 32-bit virtual address [12].

time. As the associativity increases from direct-mapped (1-way) to 8-way set associative, the relative access time rises noticeably across all cache sizes [12]. This latency increase is attributed to the additional logic gates (multiplexers and comparators) needed to process multiple ways simultaneously.

Furthermore, power efficiency is a major constraint in modern processor design. Figure 3.5 depicts the relative energy consumption per read operation. There is a sharp increase in energy usage as associativity grows; for instance, an 8-way associative cache consumes significantly more energy per read than a 1-way equivalent [12]. This is because highly associative caches often must read tags and data arrays from multiple ways to determine a hit, whereas a direct-mapped cache accesses a single specific location. Consequently, architects must strictly balance the performance gains from reduced miss rates against the costs of slower access times and higher energy expenditure.

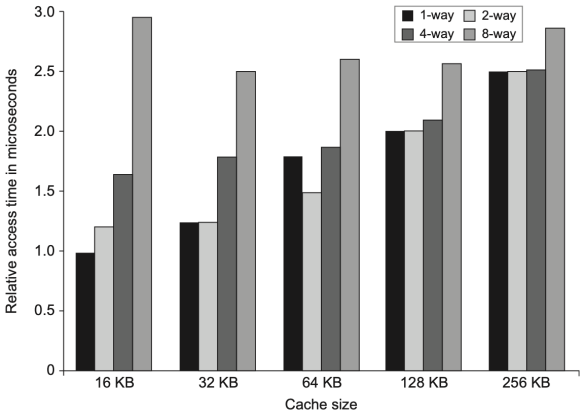


Figure 3.4: Impact of cache size and associativity (1-way to 8-way) on relative access time [12].

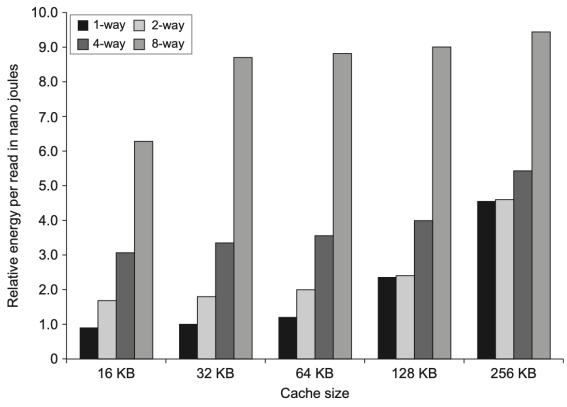


Figure 3.5: Impact of cache size and associativity on relative energy consumption per read [12].

3.3 Cache Placement Policies

Cache placement policies dictate where a specific memory block can reside within the cache structure. The three primary organizations are direct-mapped, set-associative, and fully associative caches. A direct-mapped cache offers the advantage of low search logic cost and consequently low energy consumption; however, it often suffers from lower hit rates due to conflict misses. Conversely, a fully associative cache allows a block to be placed in any location, maximizing hit rates and cache utilization, but this comes at the cost of high energy consumption due to the complexity of the parallel search and comparison logic [12].

Set-associative caches provide a trade-off between these two extremes. By dividing the cache into sets where a block can be placed in any way within a specific set, this policy balances implementation complexity with hit rate performance. While it mitigates some conflict misses found in direct-mapped designs, it requires hardware to implement valid bits and is still susceptible to conflicts when a specific set becomes full. Other specialized policies, such as two-way skewed associative and pseudo-associative caches, attempt to further optimize this behavior [12].

3.4 Cache Replacement Policies

When a cache miss occurs and the target set is full, the controller must select an existing block to evict to make room for the new data. This decision is governed by cache replacement policies. Random policies select a victim block arbitrarily, which is simple to implement but fails to exploit data locality. Queue-based policies, such as First-In-First-Out (FIFO), evict the oldest block regardless of how recently it was accessed.

More sophisticated approaches rely on historical access patterns. Recency-based policies, most notably Least Recently Used (LRU), assume that recently accessed data is likely to be reused soon, and therefore evict the block that has not been used for the longest time. Alternatively, Frequency-based policies (LFU) track how often a block is accessed and evict the least frequently used items. These policies aim to minimize the miss rate by preserving the data that constitutes the current working set of the application.

3.5 Cache Misses

A cache miss occurs when the data requested by the processor is not found in the cache, necessitating a fetch from a lower memory level. These misses are categorized into a taxonomy as shown in Figure 3.6. A **cold miss** (or compulsory miss) happens when a block is accessed for the very first time. While strictly unavoidable, its latency can be masked using hardware or software prefetching techniques. A **capacity miss** occurs when the cache is simply too small to contain the entire working set of the program, causing valid blocks to be evicted and subsequently re-fetched [12].

Conflict misses arise from the restrictions of the placement policy, particularly in direct-mapped and set-associative caches. As illustrated in Figure 3.7a, multiple memory addresses may map to the same specific cache location. In this example, the L2 cache lines are color-coded, where all lines in the red region map to the same slot in the L1 cache. Even if other slots in the L1 cache are empty (such as line 7), a new request for a red address must replace the existing red entry (line 1), as shown in Figure 3.7b. This contention causes a conflict miss if the evicted line is needed again shortly after.

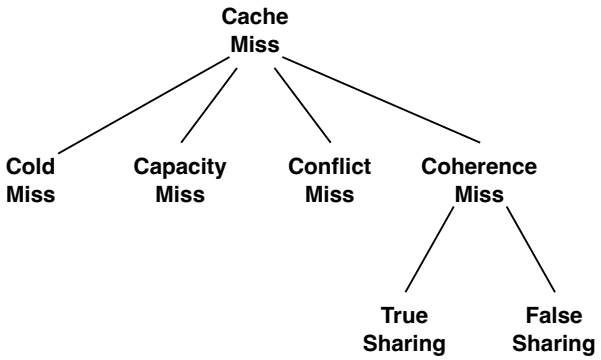
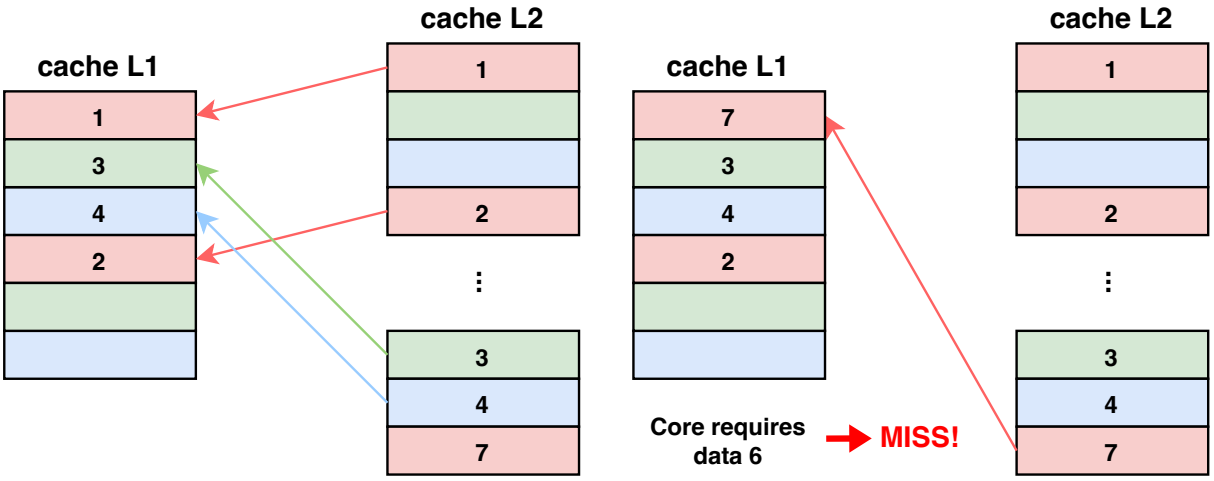


Figure 3.6: Taxonomy of cache misses in multi-core processors.

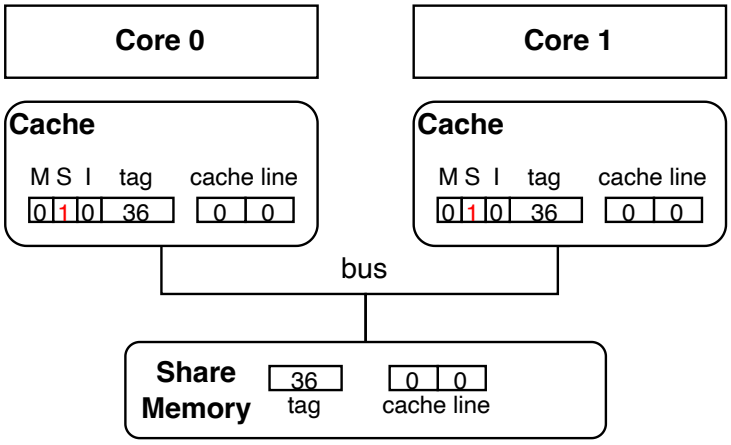


(a) Address mapping in a direct-mapped cache. The red cache lines in L2 all map to the same index in L1.

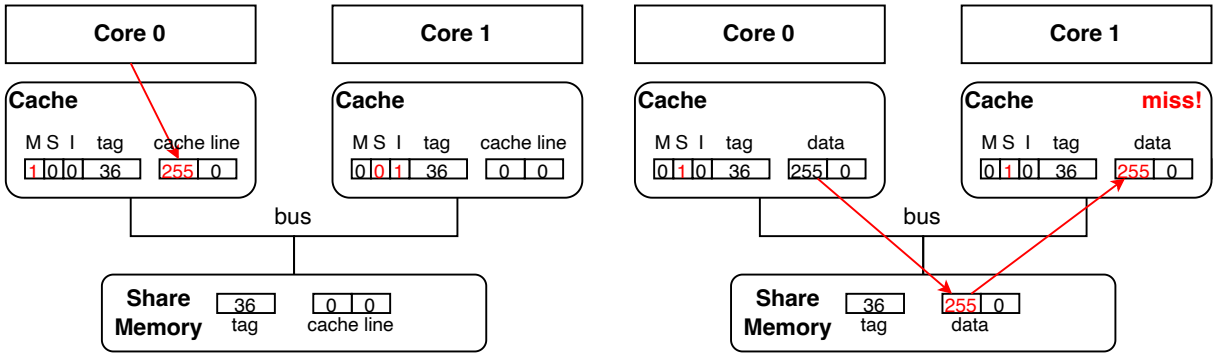
(b) A conflict miss scenario where a new red line evicts the existing red line, despite available space elsewhere.

Figure 3.7: Illustration of conflict misses in a direct-mapped cache.

In multiprocessor systems, **coherence misses** occur due to the interaction between cores. Under the MSI (Modified, Shared, Invalid) protocol, a write operation by one core invalidates copies of that data in other cores. Figure 3.8a shows an initial state where two cores share a cache line. When Core 0 writes to this line (Figure 3.8b), the protocol invalidates Core 1's copy. Consequently, when Core 1 attempts to read this data (Figure 3.8c), it encounters a coherence miss and must fetch the updated value. This category is further divided into *true sharing*, where cores access the same shared variable, and *false sharing*, where cores access independent variables that coincidentally reside on the same cache line.



(a) Initial state: both cores hold the cache line (tag 36) in the Shared state.



(b) Write and invalidate

(c) Read after invalidation

Figure 3.8: Illustration of a coherence miss caused by a write–invalidate protocol.

3.6 Race Conditions and Memory Consistency

A race condition arises when multiple execution contexts concurrently access and manipulate shared data, leading to a nondeterministic outcome that depends on the specific interleaving of operations. To prevent such anomalies and ensure data integrity, explicit synchronization mechanisms are required to impose ordering constraints and control access to shared memory [23].

While early solutions relied on software-based algorithms like Peterson's solution, these are often unreliable on modern hardware due to out-of-order execution and compiler optimizations. Consequently, contemporary systems utilize hardware-supported atomic primitives. Instructions such as `lock addq` or `lock cmpxchgq` (Compare-and-Swap) on the x86-64 architecture allow read-modify-write sequences to execute as in-

divisible units. These primitives are the building blocks for high-level synchronization constructs like mutexes and lock-free data structures. Additionally, memory barriers (fences) are employed to enforce consistency models by preventing the reordering of memory operations across the barrier, ensuring that all cores observe a consistent view of shared data.

Chapter 4

Cache Friendly Programming

4.1 Data Structure Alignment

The layout of data structures in memory significantly impacts performance and memory usage due to alignment requirements. To illustrate this, consider a naive structure definition, often referred to as the `SoyDev` layout in informal contexts. This structure contains alternating integer (4 bytes) and character (1 byte) fields. A common misconception is that the size of this structure is simply the sum of its members (10 bytes). However, due to data alignment rules, compilers insert invisible padding bytes to ensure that each member resides on a natural address boundary.

In this unoptimized layout, shown in Figure 4.1a, the compiler adds 3 bytes of padding after the first character to align the subsequent integer, and another 3 bytes at the end to align the structure size to a multiple of the largest member. This results in a total size of 16 bytes. By contrast, an optimized `ChadDev` layout, which reorders members by size (grouping integers together and characters together), eliminates the internal padding. As depicted in Figure 4.1b, this reduces the total size to 12 bytes, improving cache efficiency and reducing memory footprint.

Compilers strictly adhere to three fundamental rules to ensure efficient memory access and hardware compatibility. These rules dictate how data members are arranged within a structure and how the structure itself is sized, as illustrated comprehensively in Figure 4.2.

The first rule is the **Natural Alignment of Each Member**. This rule mandates that a

(a) Naive structure with internal padding.

(b) Optimized structure with reduced padding.

Figure 4.1: Impact of field ordering on memory layout and internal padding.

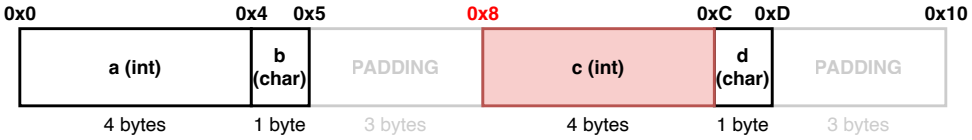
data type of size N must be stored at a memory address that is strictly divisible by N. For instance, on a standard architecture, a 4-byte integer is typically required to start at an address divisible by 4. As shown in Figure 4.2a, failing to meet this requirement results in misaligned access. While some processors handle misalignment gracefully at the cost of extra performance cycles, others may trigger hardware exceptions that terminate the program.

The second rule involves **Padding Between Members** to satisfy the natural alignment constraints of subsequent fields. When a member with a small alignment requirement is followed by a member with a larger requirement, the compiler automatically inserts unused ”padding” bytes between them. A classic example, depicted in Figure 4.2b, involves a 1-byte character followed by a 4-byte integer. To ensure the integer begins at a 4-byte aligned address, the compiler injects three bytes of padding after the character. This guarantees that the CPU can fetch the integer in a single aligned memory transaction.

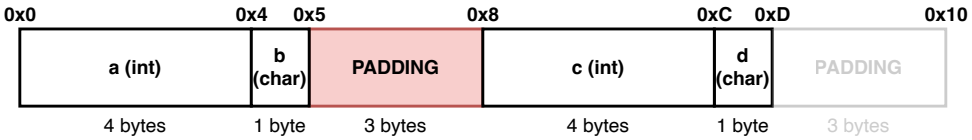
The third rule governs **Struct-Level Alignment**, specifically concerning the total size of the structure. The size of the entire struct must be padded to be a multiple of the largest alignment requirement of any of its members. This tail padding is crucial for array indexing. As demonstrated in Figure 4.2c, without this final padding, the second element in an array of structs would immediately follow the data of the first, potentially causing its members to start at misaligned addresses. By rounding up the total size, the compiler ensures that every element in an array maintains proper alignment constraints.

4.2 Cache Aware vs Oblivious Algorithms

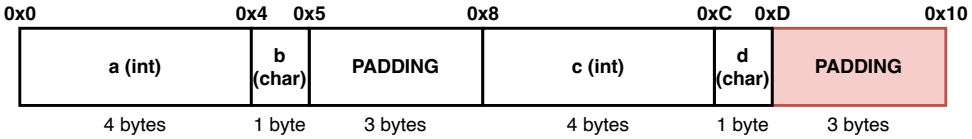
Modern processors rely heavily on multi-level cache hierarchies to bridge the performance gap between fast CPU cores and relatively slow main memory (RAM, secondary



(a) Rule 1: Natural alignment requires data types to start at addresses divisible by their size.



(b) Rule 2: Padding bytes are inserted between members to ensure the subsequent member is correctly aligned.



(c) Rule 3: The total struct size is padded to be a multiple of the largest member's alignment to support array indexing.

Figure 4.2: Visualizing the three fundamental rules of data structure alignment.

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 2.2 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, 5, 6].

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.

Chapter 5

Graphics Processing Unit (GPU)

While the previous chapter explored the Central Processing Unit (CPU) as a latency-optimized processor designed for complex logic, branching, and serial execution, this chapter shifts focus to the Graphics Processing Unit (GPU). Originally architected for the fixed-function pipeline of graphics rendering, modern GPUs have evolved into throughput-oriented, massive parallel processors capable of general-purpose computing (GPGPU). Based on the NVIDIA Ampere architecture (specifically the GA100 implementation), this chapter dissects the GPU's hierarchical structure, execution units, and memory organization to elucidate how it achieves high-performance computing capabilities through massive parallelism.

5.1 Fundamental Differences: CPU vs. GPU

The architectural divergence between CPUs and GPUs stems from their conflicting design philosophies, specifically the trade-off between **Latency** and **Throughput**. A CPU is designed to minimize the latency of execution for a single thread. It accomplishes this by dedicating a vast amount of transistor budget to large multi-level caches (L1, L2, L3) and complex control logic mechanisms such as branch prediction and out-of-order execution. These features allow the CPU to handle complex, serial instruction streams efficiently and minimize the stall time of a single thread, but they significantly limit the number of concurrent execution threads available on the silicon die.

In stark contrast, a GPU is designed to maximize throughput. As illustrated in Fig-

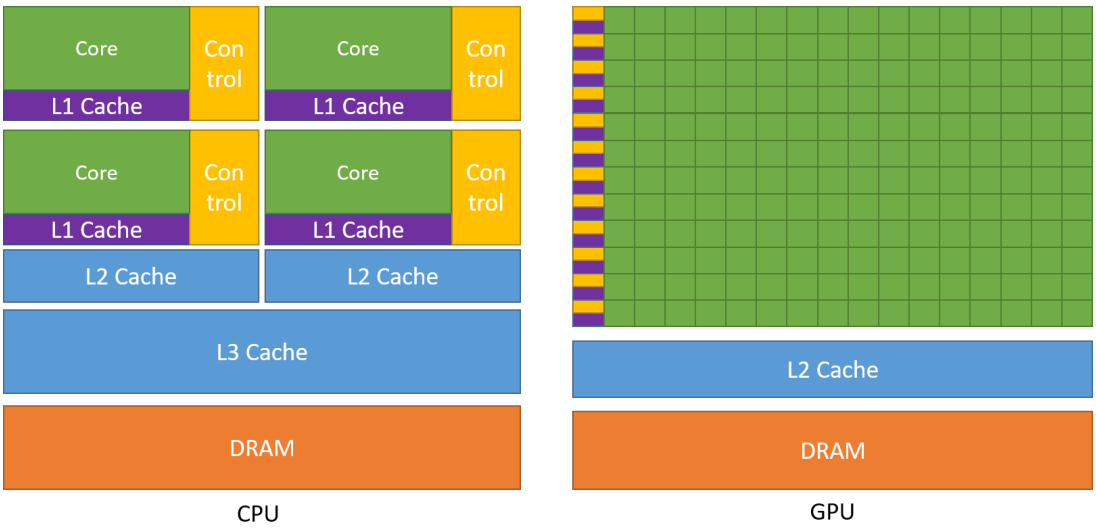


Figure 5.1: CPU (Latency Oriented) vs. GPU (Throughput Oriented) [16].

ure 5.1, the GPU devotes the majority of its die area to thousands of Arithmetic Logic Units (ALUs) rather than flow control or large caches. This design philosophy enables the GPU to hide memory latency not through large caches, but through **massive hardware multi-threading**. The GPU architecture relies on the concept of saturation; by maintaining thousands of active threads, the hardware can instantly switch context to another thread whenever the current one stalls waiting for a long-latency operation (such as a global memory fetch). This rapid, zero-overhead context switching effectively keeps the computational hardware busy at all times, masking the latency of individual threads behind the aggregate throughput of the entire system [16].

5.2 GPU Macro-Architecture

Modern NVIDIA GPUs, such as the GA100, function as a scalable array of processors structured hierarchically. Figure 5.2 depicts this high-level organization, which connects several key components via a high-speed crossbar interconnect to a shared L2 Cache and High-Bandwidth Memory (HBM).

At the macroscopic level, the workload distribution is managed by the **GigaThread Engine**, a global scheduler that distributes thread blocks to the available processing clusters. The highest level of the execution hierarchy is the **Graphics Processing Cluster (GPC)**. The GPC acts as a self-contained partition, housing all resources necessary for

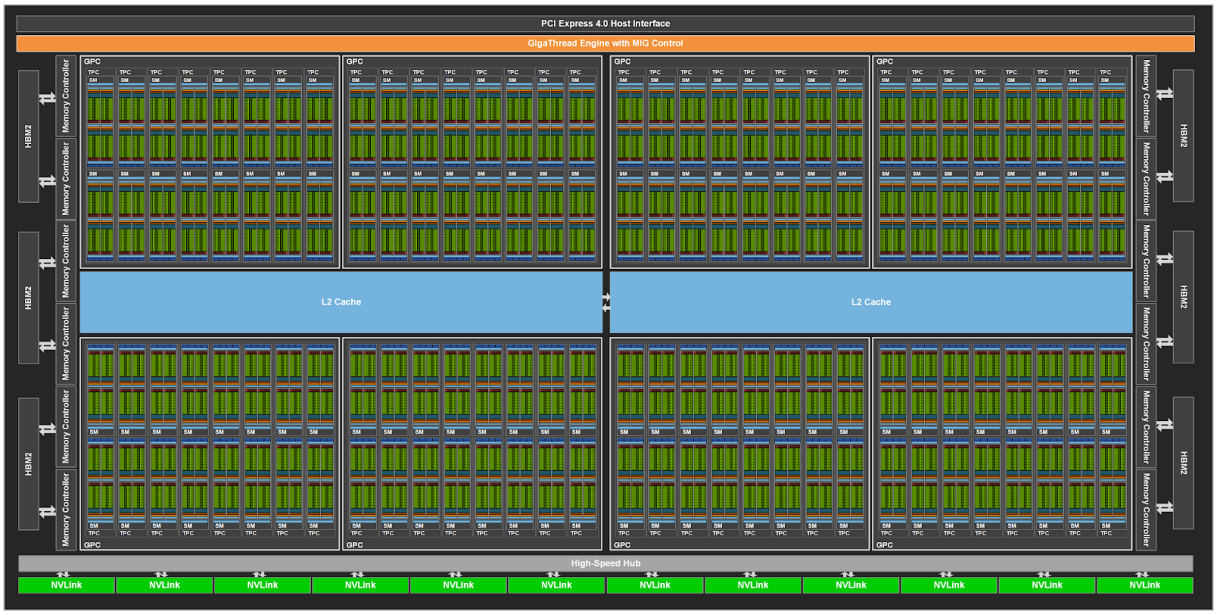


Figure 5.2: Macro-architecture of the NVIDIA GA100 GPU, highlighting GPCs and TPCs [16].

rasterization, shading, and texture processing in graphics workloads. In the context of compute workloads, GPCs serve as the primary containers for distributing thread blocks, ensuring that related tasks are physically grouped to optimize resource usage.

Within each GPC reside multiple **Texture Processing Clusters (TPC)**. The TPC serves as an intermediate architectural level that typically houses two or more Streaming Multiprocessors (SMs) along with dedicated texture processing units known as Poly-Morph Engines. This nesting structure ensures that execution units share local resources efficiently while maintaining modularity.

Connecting these clusters are the **L2 Cache and Memory Controllers**. Unlike the L1 cache which is private to each SM, the L2 cache is a unified memory bank shared across all GPCs. It functions as the central point of data coherency and synchronization before data reaches the device memory (VRAM), facilitating efficient data exchange between different parts of the GPU without accessing the slower off-chip memory. This hierarchy allows the GPU architecture to be highly modular; performance can be scaled simply by increasing the number of GPCs and TPCs on the silicon die without fundamentally changing the underlying architecture.

5.3 The Streaming Multiprocessor (SM)

The fundamental building block of the NVIDIA GPU is the **Streaming Multiprocessor (SM)**. The computational power of a GPU is largely determined by the number of SMs it contains and the architectural efficiency of their internal organization. The SM is analogous to a CPU core but designed to execute hundreds of threads concurrently.

Modern SM architectures, such as those found in Ampere and Hopper generations, employ a **Quad-Partition Design**. As shown in Figure 5.3, the SM is divided into four symmetrical processing blocks, or sub-partitions. Each sub-partition operates as an independent processing unit equipped with a dedicated **L0 Instruction Cache**, a **Warp Scheduler**, a dispatch unit capable of issuing instructions per clock cycle, a set of execution cores (CUDA Cores, Tensor Cores), and a dedicated portion of the Register File (64 KB per partition).

This partitioning strategy is critical for performance as it mitigates the bottleneck of a monolithic scheduler. By subdividing the resources, the SM can execute multiple warps concurrently across different partitions. For instance, while one partition executes an INT32 instruction for address calculation, another can simultaneously execute a Tensor Core operation for matrix math, thereby maximizing instruction-level parallelism and ensuring high utilization of the available silicon area.

5.4 Execution Units and Resources

Within each SM sub-partition, several specialized execution units work in tandem to process the instruction stream. Understanding the interplay between these components is crucial for developers aiming to optimize kernel performance.

The control logic is driven by the **Warp Scheduler**, which plays a central role in managing "Warps"—groups of 32 threads that execute in lock-step according to the Single Instruction, Multiple Threads (SIMT) model. The scheduler's primary responsibility is to monitor the state of all resident warps, select a warp that is ready to execute (i.e., not waiting for memory or synchronization), and dispatch its next instruction to the available cores. This mechanism allows the GPU to tolerate long-latency operations by simply

Figure 5.3: Internal architecture of a Streaming Multiprocessor (SM) showing the Quad-Partition design [16].

scheduling other warps while waiting.

Supporting these operations is the **Register File**, a massive on-chip memory (approximately 256 KB per SM in Ampere). Unlike CPU registers which are scarce resources, the large register file in a GPU allows it to hold the context state of thousands of threads simultaneously. This vast storage enables **zero-overhead context switching**, allowing the scheduler to instantly switch from a stalled warp to a ready warp without the expensive save/restore operations typical of CPU context switches. However, the register file is a limited resource; excessive register usage per thread can limit the number of active warps (Occupancy), potentially reducing the GPU's ability to hide latency.

The computational work is handled by various specialized core types. **CUDA Cores** constitute the standard arithmetic logic units, comprising FP32 and FP64 units for floating-point arithmetic, alongside INT32 units for integer math and addressing logic. For mod-

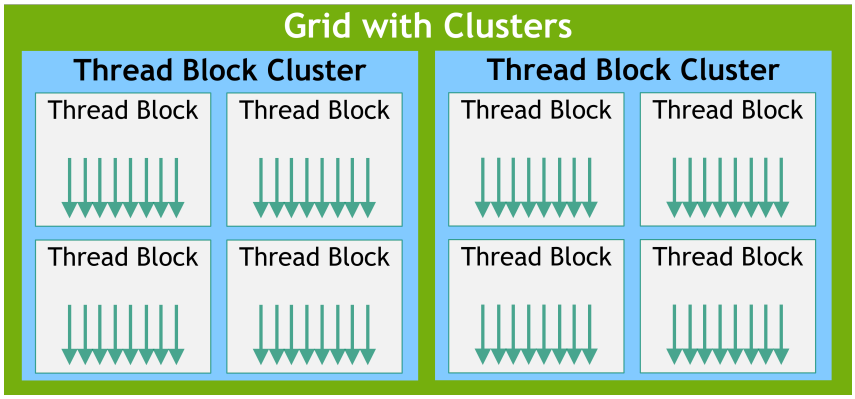


Figure 5.4: The GPU Thread Hierarchy: From Grid to Warp [16].

ern AI and deep learning workloads, the SM employs **Tensor Cores**. First introduced in Volta and enhanced in Ampere, these specialized units are designed to perform matrix multiply-and-accumulate (MMA) operations ($D = A \times B + C$) in a single clock cycle. Tensor Cores support mixed-precision computing, significantly accelerating matrix-heavy algorithms compared to standard CUDA cores.

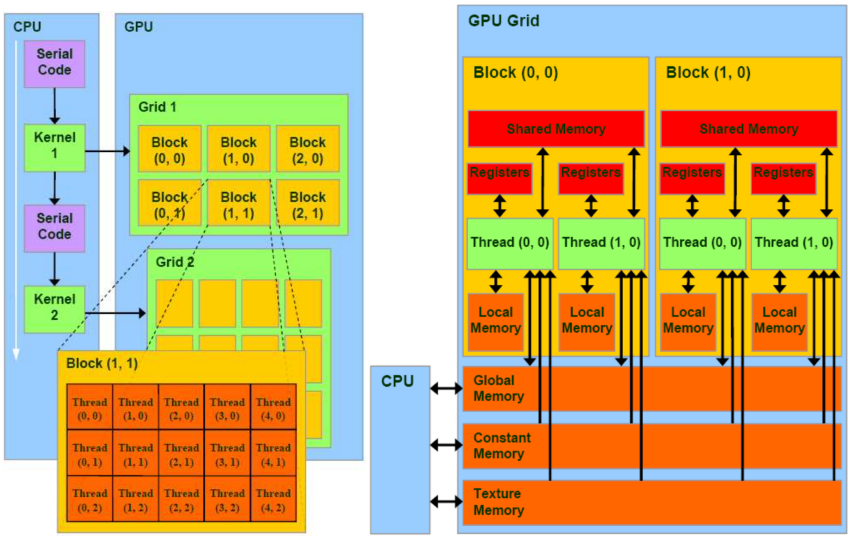
Additionally, **Special Function Units (SFUs)** are included to handle computationally expensive transcendental functions such as sine, cosine, square root, and reciprocal interpolation. Finally, **Load/Store Units (LD/ST)** manage the data transfer between the register file and the memory hierarchy, handling complex addressing calculations and cache line requests.

5.5 Thread Hierarchy and Execution Model

The GPU executes software kernels using a massive number of threads. To manage this scale, NVIDIA defines a logical hierarchy that maps directly to the hardware components described above (Figure 5.4).

At the top of this logical hierarchy is the **Grid**, representing all threads generated by a single kernel launch. A Grid maps to the entire GPU device. To optimize data locality, the Grid is subdivided into **Block Clusters** (a feature introduced in Hopper/Ampere). Block Clusters group thread blocks that are guaranteed to be scheduled on GPCs with close physical proximity, enabling efficient synchronization and data sharing via Distributed Shared Memory.

Beneath the cluster level is the **Thread Block**, a group of threads that can cooperate



(a) Thread Hierarchy in CUDA (b) CUDA Memory Spaces

Figure 2.3: CUDA execution and memory models(KIRK, 2010)

Figure 5.5: Mapping of CUDA Software Threads to Hardware Resources [27].

via Shared Memory and synchronize execution using barriers. From a scheduling perspective, a Thread Block is the atomic unit of resource allocation; it is always scheduled on a single SM and resides there until completion. The hardware further decomposes the block into **Warps**, which are the fundamental units of execution. A Warp consists of 32 threads executing the same instruction on different data (SIMT). Finally, the **Thread** represents the smallest logical unit, responsible for processing a single data element.

The relationship between the software definition and hardware execution is illustrated in Figure 5.5. While developers write code for individual threads within blocks, the hardware schedules and executes them as Warps on SMs. Understanding this mapping is vital because performance issues often arise when the software structure does not align well with the hardware reality, such as when threads within a warp diverge (execution divergence) or when memory accesses are not aligned.

5.6 Memory Hierarchy

Effective data management is often the bottleneck in GPU applications. The GPU memory hierarchy consists of several distinct spaces with varying scopes, lifetimes, and caching behaviors, as illustrated in Figure 5.6.

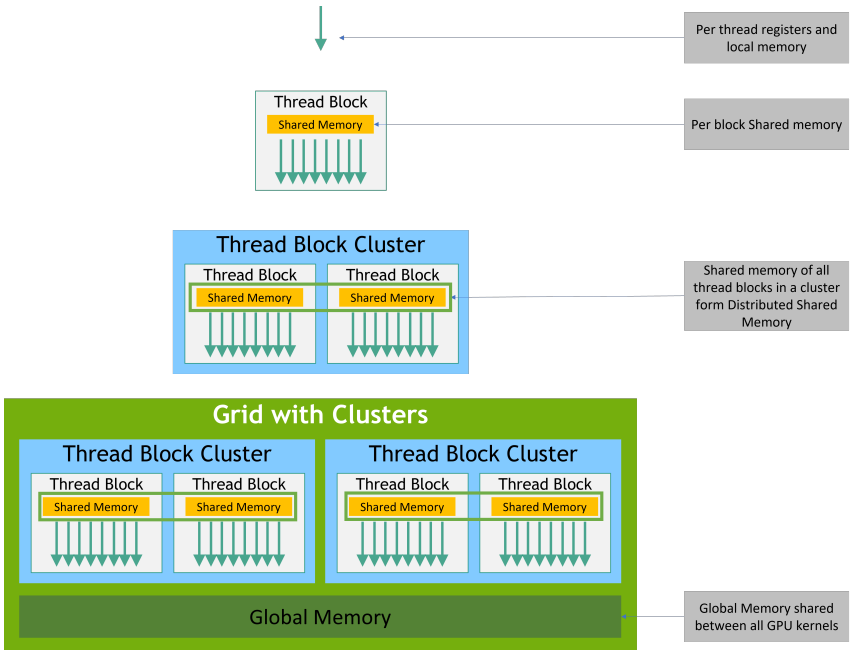


Figure 5.6: Logical Data Access Levels and Memory Hierarchy.

5.6.1 Memory Spaces

The largest and slowest memory space is **Global Memory**, residing in High-Bandwidth Memory (HBM) or GDDR VRAM. It is persistent across kernel launches and accessible by all threads in the grid. Since accessing Global Memory incurs high latency (hundreds of clock cycles), the hardware attempts to combine memory requests from threads in a warp into fewer transactions, a process known as **Memory Coalescing**. If threads access memory in a scattered pattern, the bandwidth is utilized inefficiently, leading to performance degradation.

In contrast to Global Memory, **Shared Memory** is a high-speed, on-chip programmable cache physically located within the SM. It is visible to all threads within a block (or cluster) and has a lifetime equal to that of the block. Because it offers bandwidth significantly higher than Global Memory and lower latency, it is often utilized as a user-managed cache to reduce off-chip memory access. However, Shared Memory is divided into banks, and simultaneous access to the same bank by different threads can cause **Bank Conflicts**, forcing accesses to be serialized. Complementing this is **Distributed Shared Memory**, an architectural enhancement allowing thread blocks within a cluster to perform direct load/store operations on each other's shared memory segments, facilitating efficient data exchange without traversing Global Memory.

Other specialized spaces include **Local Memory**, which, despite its name, resides in off-chip global memory and is used for thread-private data like register spills. Additionally, **Constant and Texture Memory** are read-only spaces backed by specific hardware caches, optimized for broadcasting values or 2D spatial locality access, respectively.

5.6.2 Hardware Implementation

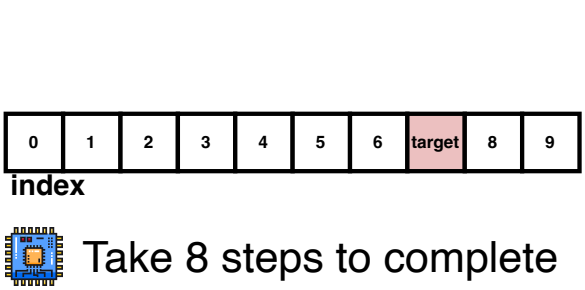
The physical implementation of these logical spaces resides within the SM, primarily relying on the **Register File** for the fastest private storage. In modern architectures like Ampere, the **L1 Data Cache and Shared Memory** share a unified on-chip memory block (e.g., 192 KB per SM in GA100). This unified design allows developers or the driver to configure the partition size dynamically based on workload requirements, allocating more space to Shared Memory for cache-heavy algorithms or more to L1 for general caching.

Chapter 6

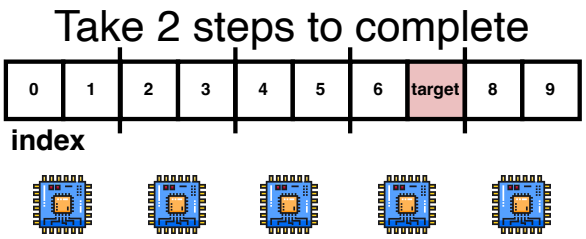
Superlinear Speedup

Superlinear speedup refers to the phenomenon in which the observed speedup of a parallel program exceeds the number of processing units employed, i.e., $S(p) > p$. Although this behavior appears to contradict classical performance models such as Amdahl's Law, it has been rigorously studied and shown to arise naturally under specific algorithmic and architectural conditions. A correct interpretation of superlinear speedup requires analyzing not only execution time but also the total amount of computational work performed. Following the perspective introduced by Yuan and Chi, algorithms can be classified according to their structure persistence, which characterizes how the total number of computational steps changes when parallelized Yuan Shi. "Reevaluating Amdahl's law and Gustafson's law". In: *Computer Sciences Department, Temple University (MS: 38-24)* (1996), p. 25.

Two major classes of algorithms are relevant in this context: non-structure persistent (NSP) algorithms and structure persistent (SP) algorithms. In NSP algorithms, there exists at least one input for which the total number of computational steps executed in parallel is strictly smaller than that required by the sequential execution. This reduction in total work is typically enabled by early termination behavior, where one processing element discovers a solution before others complete their assigned tasks, allowing the remaining processors to stop prematurely. A canonical example is linear search, in which multiple processors examine disjoint regions of the search space. If the target element is found early by one processor, the remaining processors do not need to continue searching. Figure 6.1 illustrates this effect by contrasting a sequential execution with a parallel



(a) Sequential linear search on a single core.



(b) Parallel linear search where the target is found early.

Figure 6.1: Visualization of a non-structure persistent algorithm, where parallel execution reduces the total number of computational steps, enabling superlinear speedup.

execution, showing that the parallel version completes in fewer collective steps due to early discovery of the target.

In contrast, structure persistent algorithms are characterized by the fact that the total number of computational steps remains constant for a given input, independent of the number of processors used. In such algorithms, parallelism does not remove or skip any operations, but instead redistributes a fixed amount of work among multiple processing units. Vector addition and matrix addition are representative examples, since every element must be processed exactly once regardless of execution order or degree of parallelism.

Although parallel execution can significantly reduce wall clock time by dividing the workload across processors, the overall computational volume remains unchanged. As a consequence, any performance improvement achieved by structure persistent algorithms is bounded by the available parallel resources. These algorithms therefore cannot exhibit superlinear speedup as a result of algorithmic effects alone, since no reduction in total work is possible.

Based on these observations, superlinear speedup can be further classified according to its underlying cause. Algorithmic, or non-persistent, superlinear speedup arises from a genuine reduction in total computational work, as observed in NSP algorithms. Persistent, or systemic, superlinear speedup, on the other hand, occurs even when the total workload is fixed, and must therefore be attributed to architectural and system-level effects rather than algorithmic shortcuts. This taxonomy, summarized in Figure 6.2, follows the classification proposed by Ristov et al., which distinguishes between algorithm-

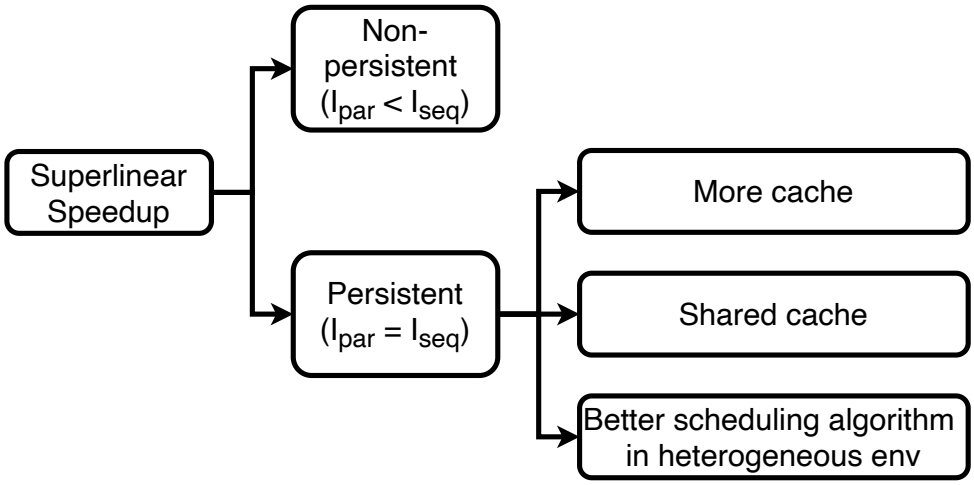


Figure 6.2: Taxonomy of superlinear speedup, distinguishing algorithmic (non-persistent) and systemic (persistent) causes. Based on the classification by Ristov et al. Sasko Ristov et al. "Superlinear Speedup in HPC Systems: why and when?" In: *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*. Vol. 8. FedCSIS 2016. IEEE, Oct. 2016, pp. 889–898. DOI: 10.15439/2016f498. URL: http://dx.doi.org/10.15439/2016F498.

driven and system-driven sources of superlinearity Sasko Ristov et al. "Superlinear Speedup in HPC Systems: why and when?" In: *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*. Vol. 8. FedCSIS 2016. IEEE, Oct. 2016, pp. 889–898. DOI: 10.15439/2016f498. URL: http://dx.doi.org/10.15439/2016F498.

In structure persistent algorithms, superlinear speedup is most commonly explained by improvements in memory-system efficiency rather than reductions in computational work. As the number of processing units increases, the effective resources available per unit of work may also increase. One prominent factor is the aggregate cache capacity provided by multiple cores. When a problem's working set no longer fits into the cache of a single core but fits into the combined caches of multiple cores, memory accesses that would otherwise go to main memory are satisfied from faster cache levels. This effect reduces memory access latency and lowers the effective cycles per instruction dominated by memory stalls. Additionally, shared caches enable data reuse among cores, further improving spatial and temporal locality.

Beyond cache capacity, system coupling and scheduling decisions also play a significant role in persistent superlinear speedup. The degree of coupling between cores,

determined by the interconnect and cache coherence mechanisms, influences how efficiently data is shared and how often cache lines are evicted. In heterogeneous systems that combine CPUs and GPUs, intelligent scheduling and task placement can reduce contention and balance memory traffic, thereby increasing overall efficiency. In such cases, the observed superlinear speedup reflects not a violation of theoretical bounds, but a shift in the effective cost model of computation, where each unit of work is executed more efficiently as parallel resources increase.

In summary, superlinear speedup emerges from two fundamentally different mechanisms. In non-structure persistent algorithms, it is enabled by a reduction in total computational work due to early termination or favorable execution paths. In structure persistent algorithms, it arises from architectural effects that improve memory behavior and resource utilization. Understanding this distinction is essential for correctly interpreting experimental results in parallel computing and high-performance computing systems, particularly in the presence of deep memory hierarchies and heterogeneous architectures.

Chapter 7

Matrix Multiplication Algorithms and Optimizations

7.1 Matrix Multiplication 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.

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 on shared memory distributed memory.
- Four representative implementations are analyzed, including a sequential baseline, a cache aware blocked algorithm, and a recursive fork join formulation, 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.

7.2 Matrix Multiplication 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 [29] 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 [25]

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 [30] 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 [19], 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 [24] and Vassilevska Williams [28] 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 [2]. 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 [14] 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 [20] 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 [4] 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 thresh-

old 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.

7.3 Numeric Stable with Kahan Summation

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[1].

7.4 Sequential Matrix Multiplication

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.

[1] 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: **a**, **b** vectors of length N

Output: $s = \sum_{k=0}^{N-1} a_k \cdot b_k$

Procedure *KahanDot(***a**, **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;

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: $\mathcal{O}(n^3)$.

Although the `ijk` ordering is the more straightforward formulation, its memory-access pattern is significantly less cache-friendly. As illustrated in Figure 7.1, 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 3.5).

In contrast, the `ikj` ordering is considerably more cache-friendly. As shown in Figure 7.2, the traversal reuses the value $A[\texttt{i}, \texttt{k}]$, loaded once into a register, while streaming through a contiguous row-major block of $B[\texttt{k}, *]$. This improves spatial locality for matrix B and reduces cache-miss rates when updating row `i` of the output matrix C.

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

```
1  Procedure MatMul_IJK(A, B)
2      n ← dim(C) ;
3      for i ← 1 to n do
4          for j ← 1 to n do
5              temp ← 0 ;
6              for k ← 1 to n do
7                  temp ← temp +
                     A[i, k] · B[k, j] ;
8              end
9              C[i, j] ← temp ;
10         end
11     end
12     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

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

7.5 Parallel Matrix Multiplication on Shared-Memory Systems

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.

Rule of Thumb: Parallelize outer loops rather than inner loops. Parallelizing the inner `k`- and `j`-loops introduces significant synchronization and scheduling overhead (see

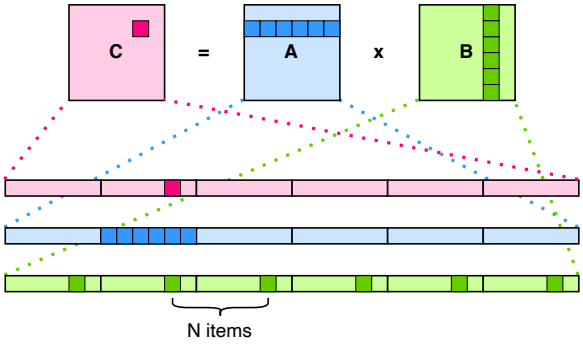


Figure 7.1: Memory access pattern for `ijk` ordering. Elements of matrix B are accessed non-contiguously, leading to poor spatial locality and higher cache miss rates.

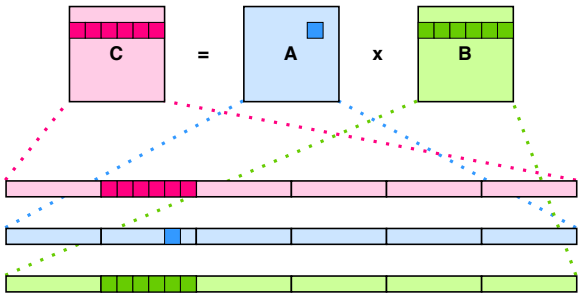


Figure 7.2: Memory access pattern for `ikj` ordering. The value $A[\texttt{i},\texttt{k}]$ is reused while accessing a contiguous row segment of B, resulting in excellent spatial locality.

sec. 2.3), 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. 3.5) 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 $C[\texttt{i},*]$ simultaneously, producing data races (see sec. 3.6) unless atomic operations (see sec. 3.6) or explicit synchronization are used. Both options add heavy contention and memory 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. 2.3). 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

Algorithm 4: 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;
```

slower execution rather than improvement.

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 7.3 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 $(\texttt{ih}, \texttt{jh})$. Within each block, individual elements are indexed by $(\texttt{il}, \texttt{jl})$, 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 7.4 and 7.5, 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

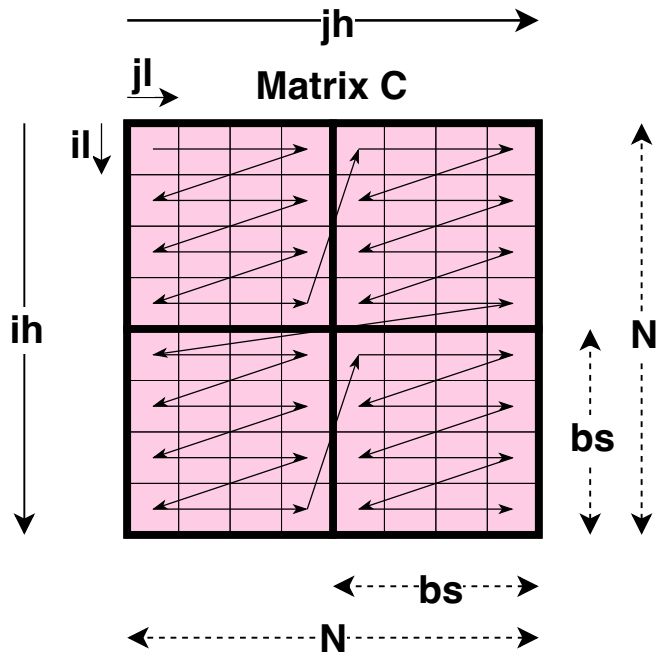


Figure 7.3: $N \times N$ matrix is partitioned into square blocks of size $bs \times bs$, each identified by block indices $(\texttt{ih}, \texttt{jh})$. Within each block, individual elements are indexed by $(\texttt{il}, \texttt{jl})$, and the arrows indicate the order in which elements are accessed during the computation, highlighting the traversal pattern that maximizes cache reuse.

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 $n\left(1 + \frac{2}{\sqrt{n}}\right)$ memory operations. This demonstrates that tiling reduces memory traffic considerably when computing n elements of C.

This approach was formalized in Level-3 BLAS [2] (called "BLAS3 below" [9]) and has since been studied extensively as a model for cache-aware algorithm design (see Sec. 4.2). Algorithm 5 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

[2]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.

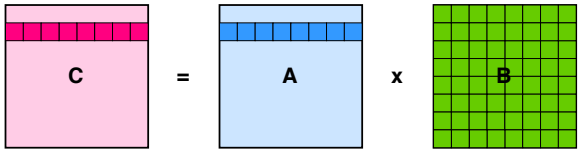


Figure 7.4: Memory access pattern when computing a row of C without tiling. Each iteration loads an entire row of A and an entire column of B, resulting in $n^2 + 2n$ total memory accesses.

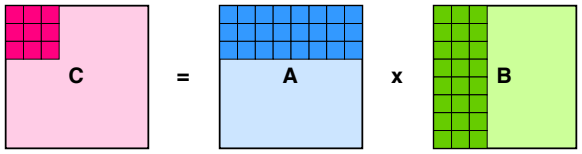


Figure 7.5: Memory access pattern using tiling with block size $k = \sqrt{n}$. Each tile reuses data inside the block, reducing memory accesses to $n\left(1 + \frac{2}{\sqrt{n}}\right)$.

Figure 7.6: Charles E. Leiserson The G.O.A.T

when designing libraries intended to run efficiently across a wide variety of hardware architectures.

Between roughly 1996 and 1999, Charles E. Leiserson (see Figure 7.6) and his students introduced the concept of cache-oblivious algorithms [6, 8][3] (see Sec. 4.2). 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 the tuning problem from a three dimensional search space to a one dimensional one.

[3] The correctness of the algorithm is accepted in the literature; see Frigo et al. (1999, 2012).

Algorithm 5: Tiled Matrix Multiplication

Input: A, B ($n \times n$), bs (block size)

Output: $C = A \times B$

```
1  Procedure TiledMatMul(C, A, B, bs)
2      n ← dim(C)
3      for ih ← 0 to n − 1 step bs do in parallel
4          for jh ← 0 to n − 1 step bs do in parallel
5              for kh ← 0 to n − 1 step bs do
6                  for il ← 0 to bs − 1 do
7                      for kl ← 0 to bs − 1 do
8                          lhs ← A[ih + il, kh + kl]
9                          for jl ← 0 to bs − 1 do
                               C[ih + il, jh + jl] += lhs · B[kh + kl, jh + jl]
10                     end
11                 end
12             end
13         end
14     end
15     return C
```

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 2.3). 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.

Algorithm 6: Parallel Matrix Multiplication (Fork–Join)

Input: A, B $(n \times n)$

Output: $C = A \times B$

Procedure *ForkJoinMatMul*$(C, A, B, thres)$

    // Fork--Join parallel region

    parallel region { **single** Compute$(C, A, B, thres)$ }

Procedure *Compute*$(C, A, B, thres)$

    $N \leftarrow \dim(C)$

    if $N \leq thres$ **then return** SeqMultiply(C, A, B)

    Create temporary T^0, T^1 $(N \times N)$

    Divide A, B, C, T^0, T^1 into 2×2 blocks (M_{ij})

    for $i, j, k \in \{0, 1\}$ **do**

        spawn Compute$(T^i_{jk}, A_{ji}, B_{ik}, thres)$

    end

    sync // wait for tasks

    MatrixAddition(C, T_0, T_1)

7.6 Matrix Multiplication on GPU

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

As shown in Algorithm 7, 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

Algorithm 7: Tiled GPU Matrix Multiplication Kernel

Input: Matrices A, B of size $N \times N$

Output: Matrix $C = L \times R$

```
1  Kernel TiledKernel(C, A, B, TILE)
2      N ← matrixsize;
3      ty ← threadIdx.y;
4      tx ← threadIdx.x;
5      i ← blockIdx.y · blockDim.y + ty;
6      j ← blockIdx.x · blockDim.x + tx;
7      Allocate shared TileA[TILE][TILE];
8      Allocate shared TileB[TILE][TILE];
9      acc ← 0;
10     for p ← 0 to N/TILE − 1 do
11         TileA[ty][tx] ← A[i, p * TILE + tx];
12         TileB[ty][tx] ← B[p * TILE + ty, j];
13         Synchronize all threads;
14         for k ← 0 to TILE − 1 do
15             acc ← acc + TileA[ty][k] · TileB[k][tx];
16         Synchronize all threads;
17     C[i, j] ← acc;
```

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 8, 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 stream, allowing concurrent data movement. A synchronization event ensures that kernel execution begins only after

Algorithm 8: 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 $\left(\lceil N/TILE \rceil, \lceil N/TILE \rceil\right)$ 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;

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.

Chapter 8

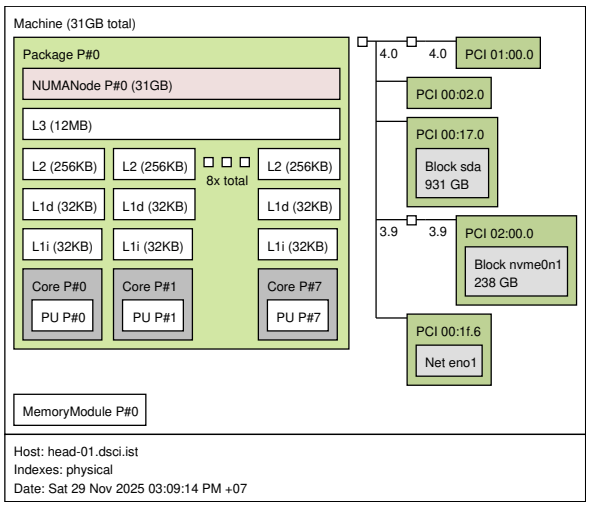
Matrix Multiplication on CPU and GPU Experiments

8.1 Experimental Setup

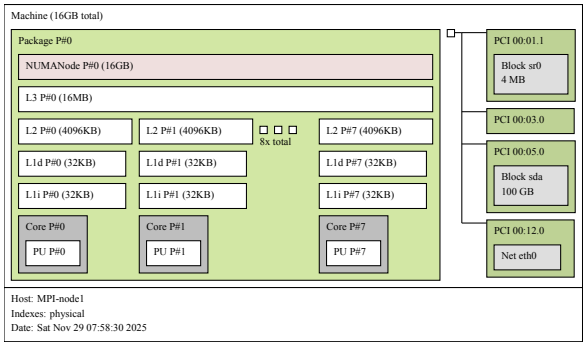
The shared-memory experiments were performed on two heterogeneous systems: the Cong Tu Vu C6 compute cluster (System 8.1), and the HCMUT Super Node cluster (System 8.2). The system configurations of a single node used in the experiments are summarized in Table 8.1.

8.2 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 8.1). 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 8.1. 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. Subfigures (II), (III), and (IV) illustrate the starting addresses of the first element of row 0, row 1, and row i, respectively.



System 8.1: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz - C6 Cong Tu Vu Cluster



System 8.2: Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz - HCMUT Super Node Cluster

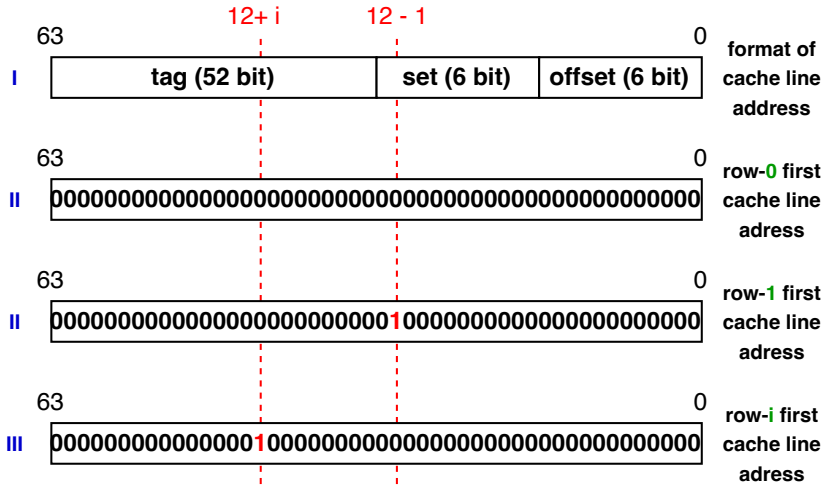


Figure 8.1: Bit representation of first and i-th row of a matrix size 512×512 with the assumption that first element is located at byte address 0.

Figure 8.2 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 8.1 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 8.3 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

Table 8.1: System specifications of single compute node for experiments

Specification	**Cong Tu Vu C6 Cluster** (System 8.1)	**HCMUT Super Node Cluster** (System 8.2)
System type	Compute server	Compute server
Compiler	`clang++` 18.1.8 (Red Hat)	`clang++` 14.0.0 (Ubuntu)
Operating System	Rocky Linux 8.10 (Green Obsidian)	Ubuntu 22.04.5 LTS
Processor	Intel Core i7-9700 @ 3.00GHz	Intel Xeon E5-2680 v3 @ 2.50GHz
Frequency	Set constant 3.6 GHz	Constant 2.5 GHz
Cores / Threads per	8 cores / 8 threads	8 cores / 8 threads
Hyper Threading [1]	No	No
RAM	32 GB	16 GB
Cache hierarchy / Associative way	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	RTX 2070 Super (8GB VRAM)	None
OpenMP / MPI	5.1 / 4.1.1	4.5 / 4.0.0
Python / Numpy	3.13.9 / 2.3.5	3.10.12 / 2.2.6

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. 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.

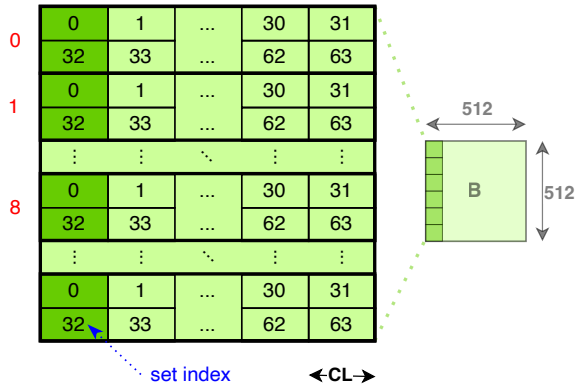


Figure 8.2: The cache lines of matrix B (represented as cells) are assigned set indices according to the 6 set-index bits in subfigure (I).

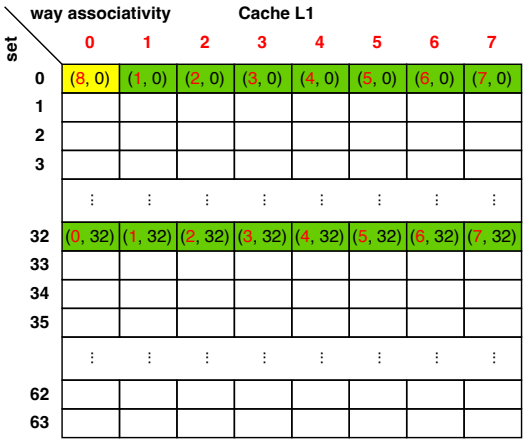
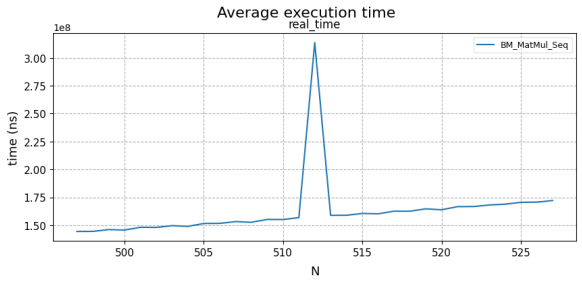
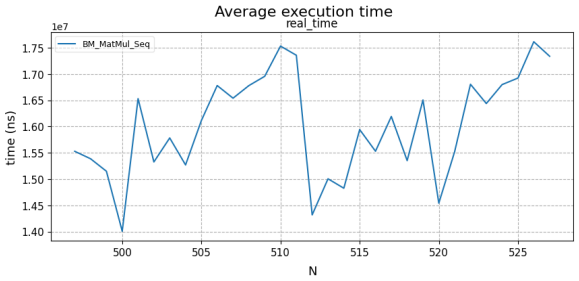


Figure 8.3: Illustration of cache misses.



(a) IJK order: performance drop observed at the power-of-two matrix size (512) due to conflict misses.



(b) IKJ order: no significant performance degradation at size 512, indicating reduced conflict misses.

Figure 8.4: Impact of loop ordering on cache conflict misses in naive matrix multiplication for matrix sizes ranging from 497 to 527.

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 8.4a 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 8.4b do not show such degradation for the IKJ order, which accesses rows of B instead of columns.

8.3 Shared Memory System

The experimental evaluation of square matrix multiplication on shared-memory systems was performed on the **three** hardware platforms described in Section 8.1. 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 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`[2]. 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.

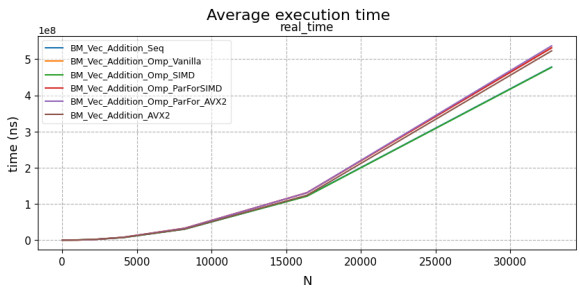
8.3.1 Addition Operator

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 8.5 shows the execution time and speedup on the 8-core on System 8.1. 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[3]. We are currently unable to provide a definitive explanation for this *"bell shape"* behavior.

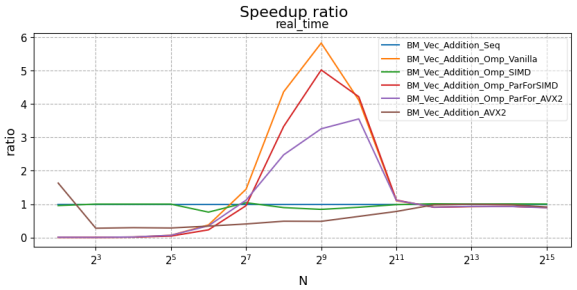
Figure 8.6 presents the corresponding results for matrix addition using three parallel

[2]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.

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

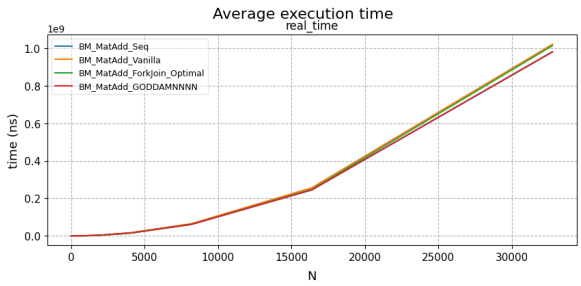


(a) Execution time

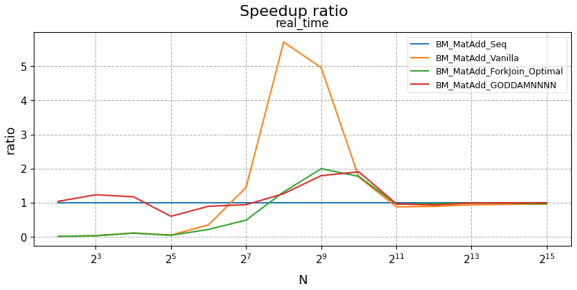


(b) Speedup

Figure 8.5: Execution time and speedup of vector addition using 8 cores on System 8.1.



(a) Execution time



(b) Speedup

Figure 8.6: Execution time and speedup of matrix addition using 8 cores on System 8.1.

variants[4], 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* [11] performs matrix addition on a single core by default, which is consistent with our findings.

8.3.2 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.

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

The experimental results for the **plain fork–join** approach, Algorithm 6, whose outcomes are shown in Figures 8.7, together with those for the **Strassen fork–join** variant. The vertical axis represents the threshold T, the horizontal axis the matrix size N, and 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 8.2, the `IKJ` ordering outperforms `IJK` due to better memory locality, while adding Kahan (see Subsection 7.3) 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.

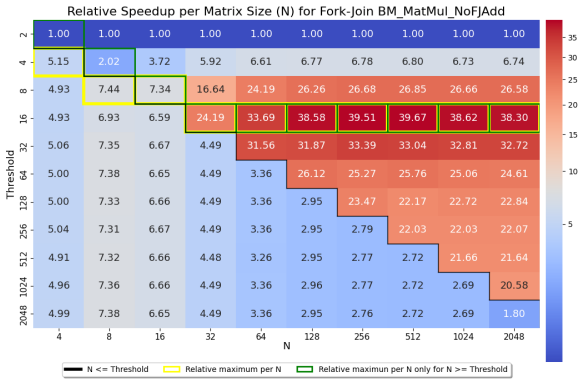
Optimal thresholds for plain fork–join (FJ).

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

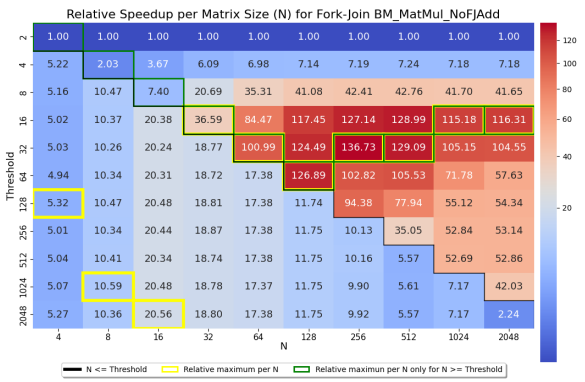
Optimal thresholds for fork–join Strassen (Str).

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

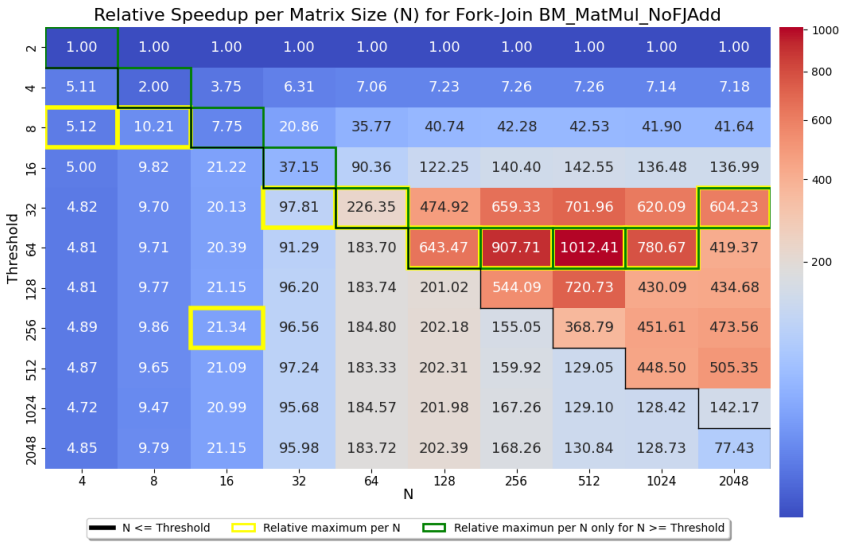
These results highlight an intriguing distinction between the two recursive schemes. Although all three base-case kernels preserve the same relative performance ordering, Strassen's recursion magnifies the separation between their optimal thresholds. Most



(a) IJK + Kahan



(b) IJK

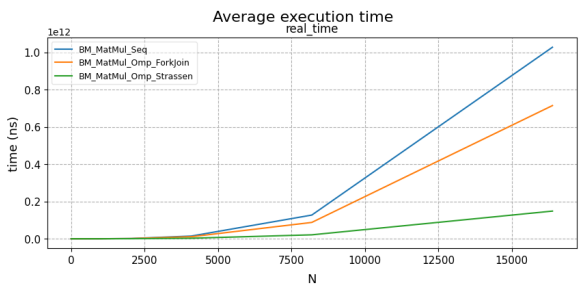


(c) IKJ

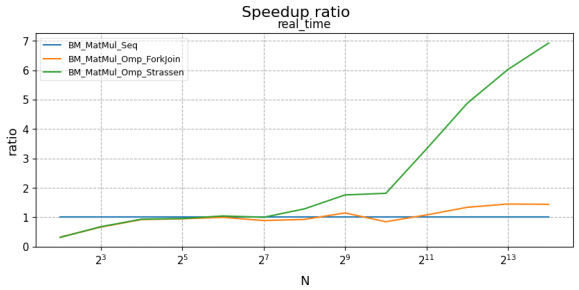
Figure 8.7: Threshold tuning for fork–join (Algorithm 6) with IJK+Kahan, IJK, and IKJ base cases using 8 cores on System 8.1.

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.



(a) Execution time



(b) Speedup

Figure 8.8: Execution time and speedup of sequential matrix multiplication on System 8.1.

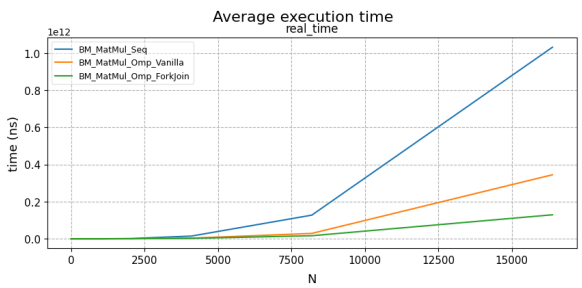
8.3.3 Sequential Performance

First, experiments are conducted using a single CPU core. The experiments are performed on System 8.1. Figure 8.8 presents the execution time and speedup of the sequential implementations.

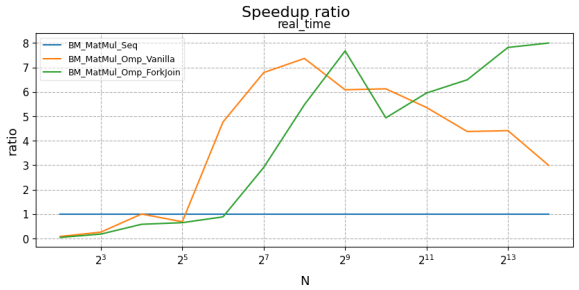
8.3.4 Multi-core Performance

We begin by comparing the vanilla `IKJ` (Algorithm 4) and Fork–Join (Algorithm 6) 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 8.1 (Figures 8.9). Remarkably, similar behavior also appears on System 8.2 (Figures 8.10). For matrix sizes in the range $[2^{10}, 2^{14}]$, the execution becomes more than $20\times$ **faster** when using 8 cores, and up to $5\times$ **faster** even with only 2 cores. This phenomenon can be attributed to improved cache utilization [18]. 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.

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

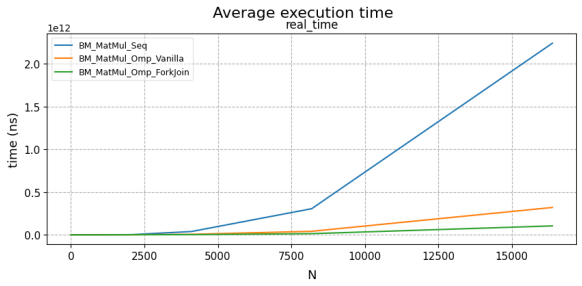


(a) Execution time

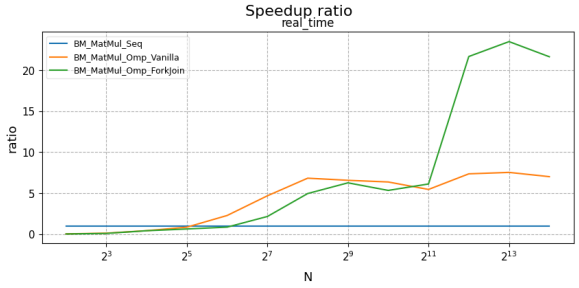


(b) Speedup

Figure 8.9: Execution time and speedup of matrix multiplication using 8 cores on System 8.1.



(a) Execution time



(b) Speedup

Figure 8.10: Execution time and speedup of matrix multiplication using 8 cores on System 8.2.

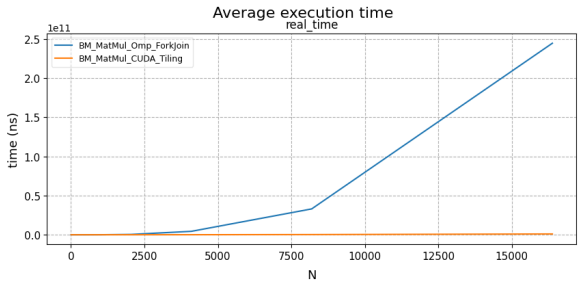
8.4 GPU Accelerated System

This section presents the experimental evaluation of the CUDA-based implementations on System 8.1. The experiments are divided into two benchmarks: (i) a comparison between the proposed tiled CUDA implementation (Algorithm 7) and the OpenMP fork–join approach (Algorithm 6), and (ii) a comparison between the tiled CUDA implementation and the GPU-based $PyTorch$ implementation.

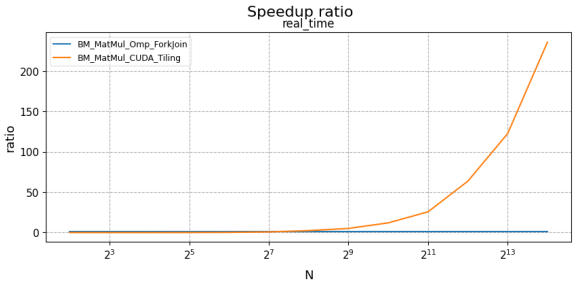
The experiments are conducted with matrix sizes ranging from $N \in [4, 16384]$. Larger matrix sizes are desirable for further evaluation; however, experiments beyond this range are constrained by the available GPU memory (VRAM). The reported execution times include both data transfers from host memory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

[4] version 2.9.0+cu126

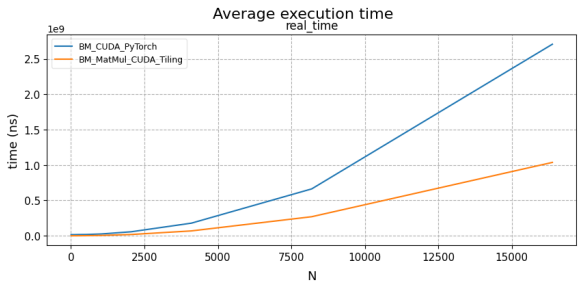


(a) Execution time

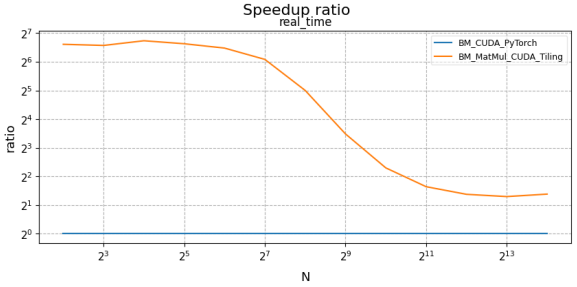


(b) Speedup

Figure 8.11: GPU tiling vs OMP Fork–Join matrix multiplication.



(a) Execution time



(b) Speedup

Figure 8.12: GPU tiling vs OMP Fork–Join matrix multiplication in PyTorch.

ing 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 8.11, 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 $236\times$ 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 8.12, the proposed **tiled CUDA implementation outperforms $PyTorch$ by up to $2.7\times$**. This result indicates that the custom kernel effectively exploits GPU architectural features beyond those leveraged by the general-purpose $PyTorch$ implementation.

Chapter 9

Conclusion

This report has presented a comprehensive exploration of foundational concepts in Parallel Computing and High-Performance Computing, motivated by the long-term objective of studying advanced scheduling mechanisms such as work stealing on GPUs. By integrating architectural analysis with hands-on experimentation, the work emphasizes the critical role of hardware-aware programming in achieving scalable performance.

Through systematic discussions of CPU and GPU architectures, memory hierarchies, cache behavior, synchronization, and scheduling, the report highlights how performance bottlenecks often arise from low-level system interactions rather than purely algorithmic limitations. Experimental evaluations based on matrix multiplication workloads further demonstrate the impact of loop ordering, cache conflicts, data alignment, and parallel execution models across sequential, OpenMP, and CUDA implementations.

While this study does not claim a novel research contribution, its value lies in the consolidation of essential knowledge and practical insights required for advanced HPC research. The breadth-first approach adopted in this report reflects the multifaceted learning process necessary to reason effectively about modern parallel systems.

Future work will build upon this foundation by narrowing the focus toward dynamic scheduling strategies on GPUs, including work-stealing and hybrid CPU–GPU execution models. With a stronger understanding of architectural constraints and performance trade-offs, subsequent research can more confidently pursue specialized contributions suitable for publication at leading venues such as the Supercomputing (SC) conference.

References

[1] https://en.wikichip.org/w/images/7/7e/skylake_block_diagram.svg. [Accessed 02-01-2026].

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

[3] Don Coppersmith and Shmuel Winograd. “Matrix Multiplication via Arithmetic Progressions”. In: *Journal of Symbolic Computation* 9.3 (1990), pp. 251–280. doi: 10.1016/S0747-7171(08)80013-2.

[4] Alhussein Fawzi et al. “Discovering Faster Matrix Multiplication Algorithms with Reinforcement Learning”. In: *Nature* 610.7930 (2022), pp. 47–53. doi: 10.1038/s41586-022-05172-4.

[5] M Frigo et al. “Cache-oblivious algorithms, Extended abstract. Lab for Computer Science”. In: MIT (May 1999). http://supertech. lcs. mit. edu/cilk/papers/abstracts …

[6] Matteo Frigo et al. “Cache-Oblivious Algorithms”. In: *ACM Trans. Algorithms* 8.1 (Jan. 2012). issn: 1549-6325. doi: 10.1145/2071379.2071383. url: https://doi.org/10.1145/2071379.2071383.

[7] Matteo Frigo et al. “Cache-oblivious algorithms”. In: *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE. 1999, pp. 285–297.

[8] Matteo Frigo et al. “Cache-oblivious algorithms”. In: *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*. IEEE. 1999, pp. 285–297.

[9] Kyle Gallivan et al. “Impact of Hierarchical Memory Systems On Linear Algebra Algorithm Design”. In: *Int. J. High Perform. Comput. Appl.* 2.1 (Mar. 1988), pp. 12–48. issn: 1094-3420. doi: 10.1177/109434208800200103. url: https://doi.org/10.1177/109434208800200103.

[10] “No Bugs” Hare. *Infographics: Operation Costs in CPU Clock Cycles - IT Hare on Soft.ware — ithare.com*. http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/. [Accessed 07-08-2025]. Sep 12, 2016.

[11] Charles R Harris et al. “Array programming with NumPy”. In: *nature* 585.7825 (2020), pp. 357–362.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. isbn: 0128119055.

[13] Intel. *What Is Hyper-Threading?* url: https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html (visited on 07/19/2025).

[14] Elaye Karstadt and Oded Schwartz. “Matrix Multiplication, a Little Faster”. In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’17. Washington, DC, USA: Association for Computing Machinery, 2017, pp. 101–110. isbn: 9781450345934. doi: 10.1145/3087556.3087579. url: https://doi.org/10.1145/3087556.3087579.

[15] Chip Mulligan. *ARM is no fan of HyperThreading*. Aug. 2006. url: https://web.archive.org/web/20090906005322/http://www.theinquirer.net/inquirer/news/1037948/arm-fan-hyperthreading (visited on 07/19/2025).

[16] *NVIDIA Ampere Architecture In-Depth | NVIDIA Technical Blog — developer.nvidia.com*. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/. [Accessed 14-09-2025]. May 14, 2020.

[17] Sasko Ristov et al. “Superlinear Speedup in HPC Systems: why and when?” In: *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems*. Vol. 8. FedCSIS 2016. IEEE, Oct. 2016, pp. 889–898. doi: 10.15439/2016f498. url: http://dx.doi.org/10.15439/2016F498.

[18] Sasko Ristov et al. “Superlinear speedup in HPC systems: Why and when?” In: *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2016, pp. 889–898.

[19] A. Schönhage. “Partial and Total Matrix Multiplication”. In: *SIAM Journal on Computing* 10.3 (1981), pp. 434–455. doi: 10.1137/0210032. eprint: https://doi.org/10.1137/0210032. url: https://doi.org/10.1137/0210032.

[20] Oded Schwartz and Noa Vaknin. “Pebbling Game and Alternative Basis for High Performance Matrix Multiplication”. In: *SIAM Journal on Scientific Computing* 45.6 (2023), pp. C277–C303. doi: 10.1137/22M1502719. eprint: https://doi.org/10.1137/22M1502719. url: https://doi.org/10.1137/22M1502719.

[21] Yuan Shi. “Reevaluating Amdahl’s law and Gustafson’s law”. In: *Computer Sciences Department, Temple University (MS: 38-24)* (1996), p. 25.

[22] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating System Concepts*. 10th ed. Nashville, TN: John Wiley & Sons, Feb. 2021.

[23] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating System Concepts*. 10th ed. Nashville, TN: John Wiley & Sons, Feb. 2021.

[24] Andrew James Stothers. “On the Complexity of Matrix Multiplication”. PhD thesis. University of Edinburgh, 2010.

[25] Volker Strassen. “Gaussian elimination is not optimal”. In: *Numerische Mathematik* 13.4 (Aug. 1969), pp. 354–356. issn: 0945-3245. doi: 10.1007/bf02165411. url: http://dx.doi.org/10.1007/BF02165411.

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

[27] JULIO TOSS. “Work Stealing Inside GPUs”. In: (2011). url: https://lume.ufrgs.br/handle/10183/36890.

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

[29] S. Winograd. “A New Algorithm for Inner Product”. In: *IEEE Transactions on Computers* C-17.7 (1968), pp. 693–694. doi: 10.1109/TC.1968.227420.

[30] S. Winograd. “On multiplication of 2 × 2 matrices”. In: *Linear Algebra and its Applications* 4.4 (1971), pp. 381–388. issn: 0024-3795. doi: https://doi.org/10.1016/0024-3795(71)90009-7. url: https://www.sciencedirect.com/science/article/pii/0024379571900097.