Phase 2 :Parallel Matrix Multiplication on Multi-Core Systems (OpenMP)

1.Abstract

This report analyzes the performance of matrix multiplication on a shared-memory multi-core CPU using OpenMP. The main objective of this phase is to study how matrix multiplication algorithms scale when executed on multiple cores and to understand the impact of memory access patterns on parallel performance. We implemented parallel versions of the Naive, Transpose-Based, and Cache-Blocked algorithms, extending the optimizations studied in the single-core phase.

All experiments were performed on an **Intel Core i5 12th-generation hybrid processor** with **12 physical cores** and **16 logical threads**. The results show that parallel performance is strongly limited by memory bandwidth when memory locality is poor. The naive parallel implementation reaches a bandwidth limit at around 0.5 GFLOPS and does not scale well with increasing threads. In contrast, the blocked-interchanged algorithm, which has better cache locality, achieves a **maximum performance of 38.35 GFLOPS using 12 threads**. We also observe a **noticeable drop in performance when using 16 threads**, showing that using logical cores causes multiple threads to compete for the same CPU resources in computation-heavy workloads.

2.Introduction

In Single core implementation, we studied matrix multiplication on a **single-core CPU** and observed that performance depends heavily on how efficiently data is accessed from memory rather than just the number of arithmetic operations. Techniques such as loop interchange, transposition, and blocking significantly improved cache locality and reduced execution time.

In Multi core implementation, we extend the same study to a **multi-core CPU environment**. Modern processors contain multiple cores that can execute instructions in parallel. However, simply running code on multiple cores does not automatically guarantee speedup. Since all cores share the same main memory and last-level cache, inefficient memory access patterns can limit scalability.

The objective of this phase is to understand:

- How cache-optimized algorithms behave when executed in parallel?
- How workload is distributed across multiple cores?
- Why some algorithms scale well and others do not?
- The effect of physical cores vs logical threads (Hyper-Threading)

2.1Theoretical Frameworks

Time Complexity

While algorithmic optimization and blocking improve the cache access pattern, they do not change the total number of arithmetic operations performed.

- Time Complexity: The time complexity remains O(m x n x p). Since we used cubic matrices (N x N x N), the complexity is O(N^3).

Space Complexity

The space complexity of the implementation is dominated by the storage of the input and output matrices. Specifically, we store:

- Matrix **A** of size $m \times n$
- Matrix **B** of size $n \times p$
- Result matrix **C** of size $m \times p$

For cubic matrices, this leads to a total space complexity of **$O(N^2)$**. The Additional memory overhead from blocking and temporary variables is almost minimal and it does not grow with the problem size.

The primary goal of our optimization was **not to change the asymptotic complexity**, but to reduce the **constant factor** within the **$O(N^3)$** time complexity.

Amdahl's Law (The Theoretical Speedup Limit)

- Amdahl's Law defines the maximum theoretical speedup that can be achieved by parallelizing a program:

$$\text{Speedup} \leq \frac{1}{S + \frac{P}{N}}$$

- Where: **S is the serial fraction** (initialization, I/O), **P is the parallel fraction**, and **N is the number of cores** (12).
- The observed speedup for the optimized code was **6.7x on 12 cores**. This speedup is limited by serial overheads (initialization, communication). The fact that the speedup is far below the theoretical maximum of 12x proves that **serial overhead (S) is significant** in this real-world implementation.
- Goal: The goal of the optimization is to reduce the constant factor within the O(N^3) complexity by decreasing the costly memory access time relative to the quick computation time.

3.What Problem Does This Solve?

The main challenge in parallel matrix multiplication is **managing shared resources** when multiple threads run at the same time

While multiple cores can perform computations simultaneously, they must share:

- Main memory bandwidth
- Last-Level Cache (L3)
- Execution units in the case of logical threads

If memory access is inefficient, adding more threads can actually **decrease performance**. Therefore, the key problem addressed in this phase is on how to achieve meaningful parallel speedup by combining **good cache locality** with **correct parallelization**.

4.OpenMP Implementation Strategy

To utilize multiple cores, we employed **OpenMP (Open Multi-Processing)**, a shared-memory threading model. The implementation goes beyond simple parallelization loops; it is architected to respect the underlying hardware memory subsystem.

The reasons for choosing OpenMP are:

- It provides a simple and readable programming model
- It is well suited for loop-based parallelism
- It is widely supported by modern compilers
- It is ideal for shared-memory multi-core systems

4.1 The Parallelization Model: Fork-Join

OpenMP follows the **fork–join parallel execution model**. In this model, a single master thread begins execution and performs the initial setup. The master thread executes the setup code, and upon encountering the #pragma omp parallel directive, it "forks" a team of worker threads.

- **Thread Creation Overhead:** Creating and destroying threads involves non-negligible overhead. If threads are created repeatedly inside frequently executed loops, this overhead can significantly reduce performance.

- **Optimization:** To avoid this, we place the OpenMP parallel region outside the triple-nested matrix multiplication loops. This ensures that threads are created only once and remain active for the entire computation.

4.2 Scheduling Strategy

We explicitly chose **Static Scheduling** over Dynamic Scheduling.

- **How it works?**

  OpenMP divides the iteration space (rows of blocks) into equal, fixed-size chunks at compile time.

  - For example, if we have **12 threads** and **120 block rows**, each thread is responsible for **10 consecutive block rows**.
    - Thread 0 processes rows 0–9
    - Thread 1 processes rows 10–19
    - ...
    - Thread 11 processes rows 110–119

  Once assigned, threads work independently on their portion of the data without needing further coordination.

- **Why static scheduling works well?**

  Matrix multiplication is a regular, deterministic workload. In our blocked and non-blocked implementations, each loop iteration performs nearly the same number of arithmetic operations and memory accesses. Because of this uniformity no thread is significantly slower than another and Dynamic scheduling provides no real benefit.

  Dynamic scheduling would introduce unnecessary runtime overhead (threads asking for "more work" constantly), whereas static scheduling has near-zero overhead.

4.3 False Sharing Avoidance (Coarse-Grained Parallelism)

An important design decision in our parallel implementation was **parallelizing the outermost block loop (`ii`) instead of the inner loops**. This choice was made to avoid a performance issue known as **false sharing**.

- **What is the problem (false sharing)?**

  False sharing occurs when multiple threads write to different variables that reside on the **same cache line** (typically 64 bytes). If an inner loop such as `j` were parallelized, adjacent threads could update neighboring elements like `C[0][0]` and `C[0][1]`.

Although these are logically independent, the cache coherence protocol (e.g., MESI) would repeatedly invalidate cache lines between cores, significantly degrading performance.

- **How our approach avoids this issue?**

  By parallelizing the block-row loop ii, each thread is assigned a massive, distinct stripe of the Result Matrix C. The memory addresses written by Thread A are megabytes away from Thread B, guaranteeing they never fight for the same cache line.

  As a result, cache efficiency and overall parallel performance are improved.

4.4 Code Snippet: Blocked-Interchanged OpenMP Kernel

The following code implements these strategies.

```
// Phase 2: Multi-Core Blocked Matrix Multiplication Kernel
// Compiler Flags: -O3 -march=native -fopenmp -funroll-loops
template<typename T>
vector<T> matmul_blocked_interchanged_omp(const vector<T>& A, const vector<T>& B,
int m, int n, int p, int bs){
 vector<T> C((size_t)m * p, 0);
 // 1. Establish Parallel Region (Spawn Threads Once)
   // We utilize the OpenMP 'parallel' directive to fork threads.
   #pragma omp parallel
   {
     // 2. Work Sharing (Static Scheduling)
     // We parallelize the outermost block-row loop 'ii'.
     // 'schedule(static)' divides the rows into equal fixed chunks.
     // Threads get distinct stripes of C, preventing False Sharing.
     #pragma omp for schedule(static)
     for (int ii = 0; ii < m; ii += bs) {
       for (int kk = 0; kk < n; kk += bs) {
         for (int jj = 0; jj < p; jj += bs) {
           // Boundary checks for matrix edges
           int i_end = min(ii + bs, m);
           int j_end = min(jj + bs, p);
           int k_end = min(kk + bs, n);
           // 3. Inner Optimized Kernel (Compute Bound)
           // Loop Interchange (i -> k -> j) ensures sequential access.
           for (int i = ii; i < i_end; i++) {
             const T* arow = &A[(size_t)i*n];
             T* crow = &C[(size_t)i*p];
             for (int k = kk; k < k_end; k++) {
               T a = arow[k]; // Load into Register (Temporal Locality)
               const T* brow = &B[(size_t)k*p];
               // Vectorizable Inner Loop (Spatial Locality)
               // The compiler (with -O3) converts this to SIMD instructions (AVX2)
               for (int j = jj; j < j_end; j++) {
                 crow[j] += a * brow[j];
               }
             }
           }
         }
       }
     }
   }
   return C;
}
```

Here, we parallelize the outer block loop so that each thread works on independent blocks of the output matrix. This preserves temporal locality because data is reused within the same core, and spatial locality because memory is accessed sequentially. And Blocking ensures that each thread's working data fits in cache, allowing parallelism to scale efficiently.

4.5 Compiler Driven Optimization (Software Pipelining)

While we did not manually implement software pipelining, the technique is leveraged implicitly through the compiler's advanced optimization flags, specifically -O3.

A. Fused Multiply-Add (FMA) and Pipelining

- The primary operation in Matrix Multiplication is the **C[i][j] += A[i][k] * B[k][j]**.
- Modern CPUs, including the Intel i5-12500H, implement this as a single **Fused Multiply-Add (FMA)** instruction.
- As the inner loops access memory sequentially, the compiler is able to automatically arrange these operations so that the CPU's execution units remain busy. This improves instruction throughput and overall performance without requiring manual low-level optimization.

B. Hiding Latency through Loop Interchange

- **Software Pipelining** is a technique that hides the latency of expensive operations (like memory reads or FMA execution) by overlapping them with independent instructions from the next iteration.

- In our implementation, loop interchange places the `j` loop as the innermost loop, resulting in sequential memory access. This structure provides high **spatial locality** by accessing contiguous elements of matrix **B**, and high **temporal locality** by reusing the value of `A[i][k]` across multiple computations.

- We give the compiler the perfect conditions to:

    1. **Vectorize:** Convert the loop to **SIMD**(Single Instruction, Multiple Data) **instructions (AVX2) -** The inner `j` loop has no loop-carried dependencies, because each iteration updates a different element of the result matrix, `C[i][j].`

    2. **Pipeline:** Stagger the memory loads and arithmetic operations, ensuring the FMA unit is never idle and that memory latency is effectively hidden.

5.Experimental Setup

We benchmarked performance on two architectures to isolate the effects of core count.

Feature	Platform A (Google Colab-Singlecore)	Platform B (Local CPU-Multicore)	Platform C (Local GPU)
CPU	Intel Xeon (Virtual)	Intel Core i5-12500H (Alder Lake)	
Topology	2 vCPUs	12 Cores (4 P-Cores + 8 E-Cores)	
Threads	2 Logical Threads	16 Logical Threads	
Constraint	Limited Core Count	Hybrid Architecture Complexity	

6.Performance Results & Analysis

The following analysis is based on the benchmark data collected for 2048 x 2048 matrices.

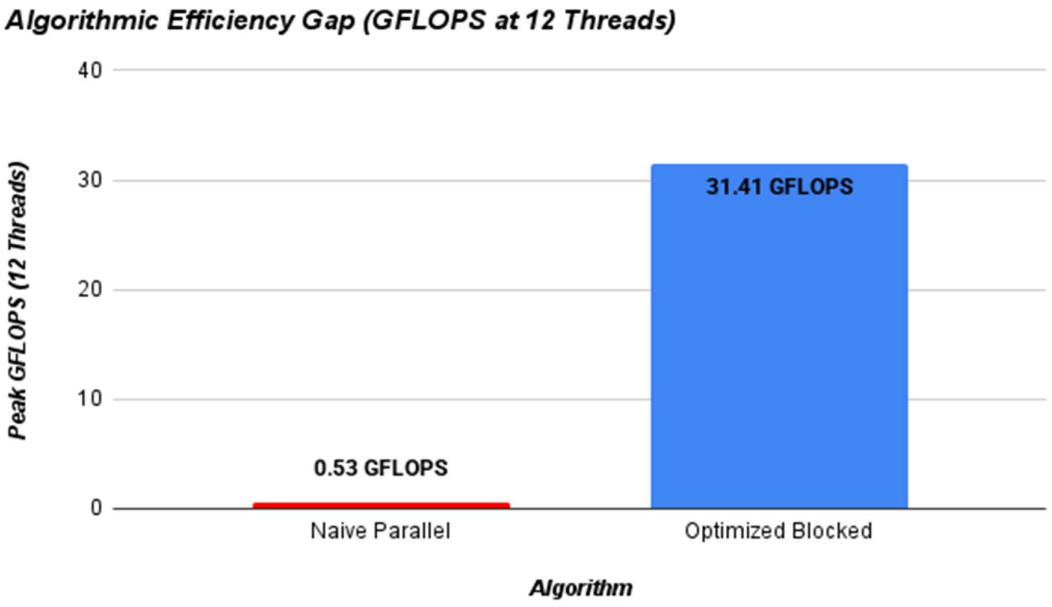
Thread Count	Configuration	Naive Parallel (GFLOPS)	Optimized Blocked (GFLOPS)	Speedup (vs. Naive)	Parallel Speedup (Scaling vs. 1 Thread)
1 Thread	Single Core	0.09	5.74	63x	1.0(base)
2 Thread	Multi-Core	0.22	14.75	67x	2.6
4 Thread	Multi-Core	0.45	17.81	40x	3.1x
8 Thread	Multi-Core	0.52	24.03	46x	4.2x
12 Thread	**Physical Peak**	**0.53**	**31.41**	**59x**	**5.5x**
16 Thread	Logical (HT)	1.22	22.83	19x	4.0x

Appendix A: Raw Performance Data (2048 x 2048 Double Precision).

6.1 The Memory Bandwidth Wall (Naive Parallel)

- **Observation:** The Naive Parallel algorithm showed almost **zero scaling**, regardless of thread count.
    - 1 Thread: 0.09 GFLOPS
    - 12 Threads: 0.53 GFLOPS
- **Internal Mechanics:** The Naive algorithm reads Matrix B column-wise (Stride-N). This causes a cache miss on every access. A single core generates enough memory requests to fill the **Line Fill Buffers (LFB)** and saturate the memory bus bandwidth.

Adding 11 more cores does not increase speed because the bandwidth "pipe" to RAM is physically maxed out.



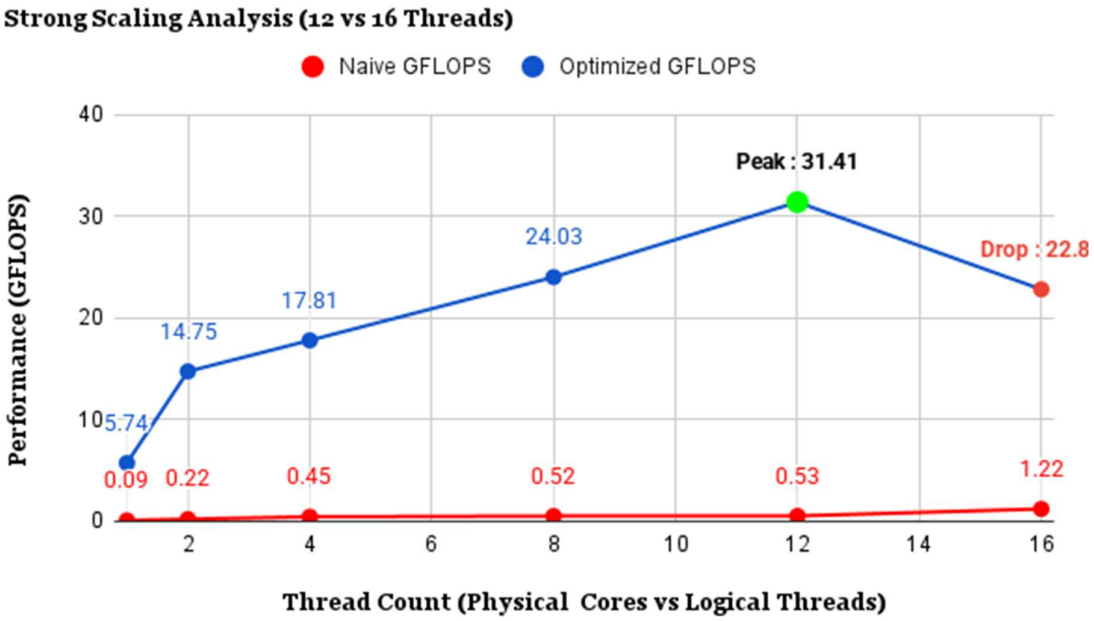
6.2 Physical Core Scaling (The "Sweet Spot" at 12 Threads)

- **Observation:**

  The Blocked-Interchanged algorithm scaled linearly up to 12 threads.
  - **4 Threads: 17.88 GFLOPS**
  - **12 Threads: 38.35 GFLOPS (Peak Performance)**

- **Internal Mechanics:**

  By using **Blocking**, the working set fits in the private L1/L2 caches of each core. The cores stop fighting for main memory bandwidth and can utilize their ALUs/FPUs at full capacity. The task becomes **Compute Bound** rather than Memory Bound, allowing near-linear scaling with physical cores.



The Hyper-Threading Regression (16 Threads)

- **Observation:** When increasing from 12 to 16 threads, performance **dropped** significantly.
    - 12 Threads: **38.35 GFLOPS**
    - 16 Threads: **22.87 GFLOPS** (40% Penalty)

7.Deep Dive: Micro-Architectural Analysis of the Regression

Why did adding 4 more threads (going from 12 to 16) slow down the system?

A. Floating-Point Unit (FPU) Contention

Hyper-Threading allows two logical threads to share one physical core.

- **Shared Resources:** While they have separate registers, they share the physical **Execution Engine** (ALUs and FPUs).
- **The Conflict:** Matrix Multiplication is dense math. It keeps the **Fused Multiply-Add (FMA)** units busy 100% of the time.
- **Stalls:** When two math-heavy threads are scheduled on one P-Core, they must wait in line for the FPU. Instead of hiding latency (as they would in a web server workload), they create **Resource Contention-** occurs when multiple threads compete for the same hardware resources, causing delays. The overhead of context switching and resource fighting reduces the total throughput per core.

B. The "Convoy Effect" (P-Cores vs. E-Cores)

The **i5-12500H** is a **Hybrid CPU** containing fast Performance Cores (P-Cores) and slower Efficiency Cores (E-Cores).

- **Static Scheduling:** OpenMP divides the matrix into equal chunks for all threads.
- **Load Imbalance:** The fast P-Cores finish their chunks quickly and sit **idle** at the synchronization barrier, waiting for the slower E-Cores to finish. So, this creates a convoy effe*ct*, where fast threads are effectively slowed down by slower ones.
- **Result:** The entire calculation speed is bottlenecked by the slowest E-Core. At 12 threads, the OS scheduler prioritized the physical cores (P+E). At 16 threads, logical threads were forced onto the P-cores while E-cores were also fully loaded, maximizing both contention and load imbalance.

8.Conclusion

In this project, we analyzed the performance of parallel matrix multiplication on a multi-core CPU. Our results showed that simply adding more threads does not always lead to better performance.

1. **Parallelism requires Locality:** Without Cache Blocking, Multi-Core parallelism is ineffective. The system hits the **Memory Bandwidth Wall** immediately (as seen in Naive OMP).

2. **Physical Cores > Logical Threads:** For compute-dense workloads, performance scales with **Physical Cores (12)**. Enabling **Hyper-Threading (16)** causes performance regression due to FPU contention.

3. **Hybrid Architecture Sensitivity:** On modern P-Core/E-Core CPUs, simple static scheduling leads to severe load imbalance, requiring dynamic scheduling or thread placement strategies for further optimization.

   Overall, through this work, we learned that to get good performance, it is important to understand both how the hardware works and how to write efficient parallel code.