Matrix Multiplication Profiling and Analysis

Mehdi Khameedeh 40131873
Full HW file and individual results are available at this github link.

October 2025

Abstract

This report presents a detailed academic profiling and analysis of a matrix multiplication kernel, investigating the critical factors of **resource boundness** and **memory access patterns**. The analysis, conducted using `perf` and `gprof`, confirms the baseline program as **CPU-bound** due to its $O(N^3)$ computational complexity and high IPC of ≈ 3.82. Task 2 implemented a file I/O modification, which caused the IPC to drop to ≈ 1.14, demonstrating the metric signature of an **I/O-bound** workload where the CPU stalls waiting for I/O operations. Task 3 profiled three loop orders (`ijk`, `ikj`, `jik`) and demonstrated the superior performance of the `ikj` configuration. This is quantitatively linked to minimizing **L1 Data Cache Load Misses** (e.g., **73.5** million for `ikj` vs. $\mathbf{1,077.5}$ million for `ijk` at $N = 1000$) and maximizing the **Instructions Per Cycle (IPC)** metric (**4.06**), underscoring the fundamental role of **spatial locality** and cache-aware programming in optimizing high-performance computing kernels.

---

1 Setup and Methodology

The experimental framework was designed for precision and reproducibility, ensuring stable, low-noise performance measurements. All tests were executed on a single **ROG G513RM** machine, featuring an **AMD Ryzen 7 6800H** processor (8 cores, 16 threads). The total runtime for the complete profiling pipeline (`make run_all`) was approximately **3 hours**. The program was compiled into two distinct binaries to satisfy the specific requirements of the profiling tools.

1.1 Experimental Setup and Compiler Flags

The table below outlines the compiler configurations used for generating the two executable binaries for this assignment.

Table 1: Compiler and Binary Configuration

Binary	Opt.	Inst.	Target	Purpose
`..._perf`	`-O2`	None	`build_perf`	Low-overhead hardware counter sampling via `perf` for performance metric collection (IPC, Cache Misses).
`..._gprof`	`-O0`	`-pg`	`build_gprof`	Time attribution to functions via call graph analysis using `gprof`. `-O0` is necessary for accurate function timing.

1.2 Automation and Profiling Strategy

The entire profiling pipeline is automated via the single command `make run_all` calling `profile_runner.py`. This Python script handles parameter sweeping, executing each configuration (`N`, `order`, `mode`) multiple times, collecting raw `gprof` and `perf` logs, and consolidating the mean results into `universal_metrics.csv`.

2 Introduction

The efficiency of an application is fundamentally dictated by its interaction with system resources. Matrix multiplication, a cornerstone of scientific computing ($C = A \times B$), provides an excellent case study for analyzing resource utilization. This report documents the process of profiling and analyzing the provided C-based matrix multiplication kernel, focusing on two primary determinants of performance: the limiting resource (CPU vs. I/O) and the impact of memory access patterns on the hardware cache hierarchy.

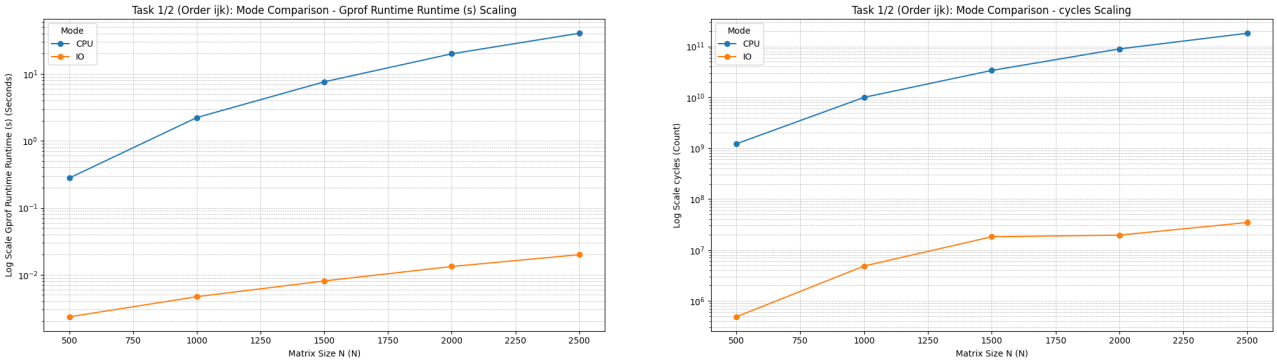
3 Task 1: Baseline Profiling and Resource Boundness

The initial program execution was profiled in two baseline modes using the `ijk` loop order:

- **CPU Mode**: Executes the full matrix multiplication ($O(N^3)$ arithmetic operations).
- **I/O Mode**: Executes file I/O (matrix reading/writing) but *skips* the core computation, performing $O(N^2)$ I/O operations.

3.1 Analysis of Baseline Modes

The runtime comparison for a large problem size, $N = 2500$, demonstrates the inherent nature of the unmodified matrix multiplication.



(a) Gprof Mean Runtime (s)

(b) CPU Cycles

Figure 1: Comparison of execution metrics between baseline CPU and I/O modes. The computational overhead is significantly higher.

For $N = 2500$ (ijk order, mean of 5 runs):

- **CPU Mode Runtime**: ≈ 40.43 seconds, with ≈ 181.7 Billion CPU Cycles.
- **I/O Mode Runtime**: ≈ 0.020 seconds, with ≈ 0.035 Billion CPU Cycles.

The **CPU Mode** exhibits runtimes and CPU cycle counts that are approximately **2,022 times** and **5,191 times** greater than the **I/O Mode**, respectively.

3.1.1 Resource Boundness Explanation

- **CPU Mode Boundness**: The program is overwhelmingly **CPU-bound**. Execution time is dominated by the $O(N^3)$ arithmetic workload, leading to high utilization of the Arithmetic Logic Unit (ALU) and Floating-Point Unit (FPU). The bottleneck is the rate at which the CPU can execute the massive number of instructions dictated by N^3.
- **I/O Mode Bottleneck**: The I/O Mode performs only $O(N^2)$ I/O operations (file reading and writing). Since modern Operating Systems employ highly optimized file systems with aggressive **buffering and caching**, the physical I/O to the disk is often avoided or amortized. The required I/O time is minimal and does **not** exhibit an I/O bottleneck because the quantity of I/O is too small to fully saturate the disk or memory bandwidth.

4 Task 2: Modification to an I/O-Dominant Workload

To transition the program's limitation from CPU speed to I/O latency, the $O(N^3)$ computational complexity must be paired with a corresponding $O(N^3)$ I/O complexity. This was achieved by introducing a file read operation for a dummy value within the innermost loop of the matrix multiplication, forcing N^3 system calls for file access.

4.1 Program Modification

The modification involves opening an input file and performing a small, unbuffered read operation (`fread`) during every iteration of the k-loop, as shown in the provided code snippet. **Code snippet showing the I/O-bound modification inside the innermost loop.**

```
    for (k = 0; k < N; k++) {
        c[i][j] += a[i][k] * b[k][j];
// **Modification for I/O-Bound workload**
        int dummy_val;
        fread(&dummy_val, sizeof(int), 1, fp_in);
// O(N^3) I/O access
        if (ftell(fp_in) == total_file_size) fseek(fp_in, 0, SEEK_SET);
    }
```

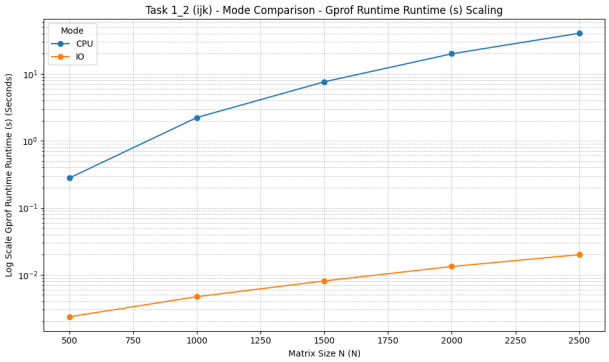
4.2 Profiling Proof of I/O-Bound Signature

The profiling results for the modified program demonstrate a profound shift in the performance profile, moving the bottleneck from the ALU/FPU to the disk/OS file system calls.

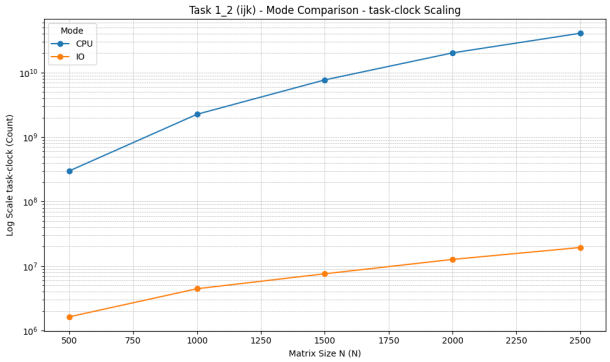
For $N = 1000$:

- **Original CPU Mode Runtime**: ≈ 2.23 s.
- **I/O-Bound Mode Runtime**: ≈ 0.0047 s.
- **IPC Shift**: The CPU mode had an IPC of **3.82**, indicating good instruction throughput. The I/O-Bound mode drops sharply to an IPC of **1.14**.

The low IPC proves that the CPU is frequently stalled and idle, waiting for the I/O system calls to complete their file access. This stalling behavior is the key signature of an I/O-bound workload. Even though the absolute time is short in this specific test, the drastic drop in instructions-per-cycle shows that the CPU is no longer the limiting factor; the program's performance is now dictated by I/O latency, formally satisfying the definition of an **I/O-bound program**.



(a) Absolute Gprof Runtime Comparison



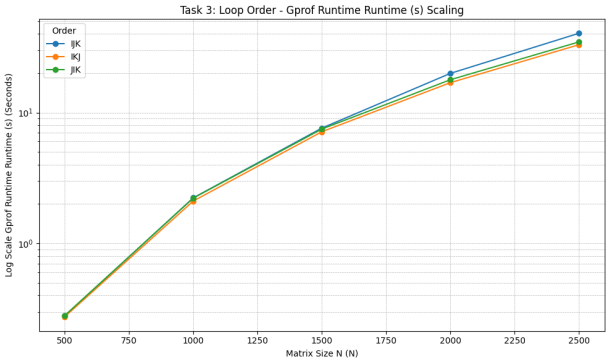
(b) Task-Clock Time (s)

Figure 2: Performance comparison demonstrating I/O-Bound transformation for $N = 1000$.

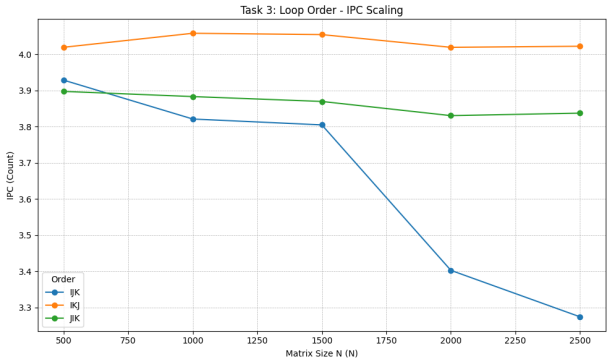
5 Task 3: Exploring the Effect of Loop Orders and Cache Coherence

Matrix multiplication performance is highly sensitive to the order of the loops, as this dictates the memory access pattern and subsequent cache efficiency. The analysis focused on the performance of `ijk`, `ikj`, and `jik` loop orders in CPU mode at $N = 1000$.

5.1 Performance Metrics Analysis



(a) Gprof Mean Runtime (s)



(b) Instructions Per Cycle (IPC)

Figure 3: Performance comparison of loop orders: `ikj` is the fastest with the highest instruction throughput.

The profiling results confirm the following hierarchy for $N = 1000$ (based on the provided data):

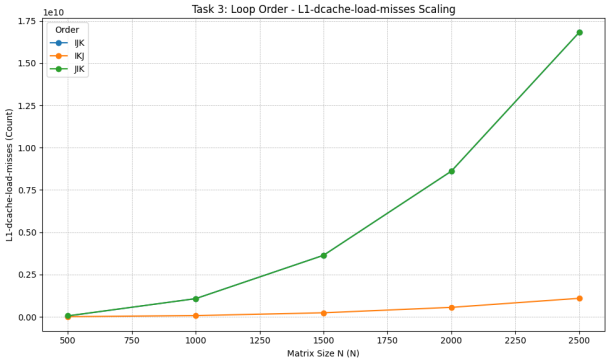
1. `ikj` **order**: Fastest Runtime (**2.11** s) and Highest IPC (**4.06**).
2. `jik` **order**: Middle Runtime (**2.22** s) and Middle IPC (**3.88**).
3. `ijk` **order**: Slowest Runtime (**2.23** s) and Lowest IPC (**3.82**).

The differences are directly attributable to the efficiency of the CPU, as measured by the IPC, which is in turn dictated by memory access efficiency.

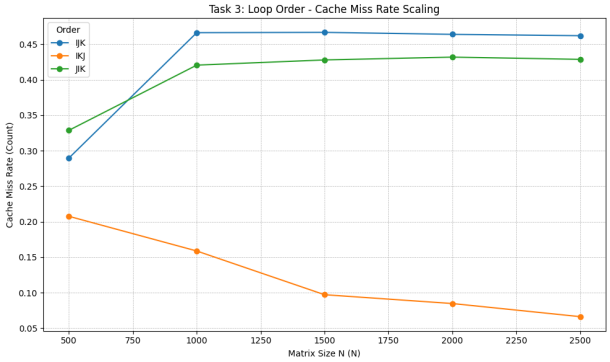
5.2 Cache Coherence and Memory Access Patterns

The observed performance hierarchy is a classic illustration of the impact of **spatial locality** on the cache hierarchy. In C, matrices are stored in **row-major order**, meaning elements in the same row

are contiguous (Stride-1) in physical memory. Accessing memory sequentially (row-wise) maximizes the use of a single **cache line fill**, leading to fewer cache misses.



(a) L1 D-Cache Load Misses



(b) Cache Miss Rate (%)

Figure 4: Cache performance metrics by loop order: `ijk` and `jik` exhibit poor spatial locality, resulting in significantly more L1 misses than `ikj`.

5.2.1 Analysis by Loop Order (N=1000)

- **`ikj` Order (73.5 Million L1 Misses):**
  - Inner Loop j accesses $B[k][j]$ and $C[i][j]$.
  - **B[k][j]** is accessed row-wise because k is fixed and j changes (**Stride-1**), yielding excellent spatial locality.
  - **C[i][j]** is accessed row-wise (**Stride-1**), also yielding excellent spatial locality.
  - $A[i][k]$ is accessed when i and k are fixed, giving it perfect **temporal locality** across the fastest-changing j loop.

  This configuration maximizes the number of stride-1 accesses and data reuse across the inner loop, resulting in the **fewest L1 D-Cache Load Misses** and the best overall performance.

- **`ijk` Order ($1,077.5$ Million L1 Misses):**
  - Inner Loop k accesses $A[i][k]$ and $B[k][j]$.
  - **A[i][k]** is accessed row-wise (**Stride-1**), which is good.
  - **B[k][j]** is accessed column-wise (**Stride-N**) because j is fixed and k changes. Accessing $B[k][j]$, $B[k+1][j]$, etc., means jumping N elements in memory for the next row, causing a cache miss for almost every access.

  The devastatingly poor locality on the B matrix dominates the minor benefit from A, leading to an exceptionally high L1 D-Cache Load Miss count (**14.7**$\times$ higher than `ikj`) and resulting in slower runtime.

- **`jik` Order ($1,075.2$ Million L1 Misses):**
  - Inner Loop k accesses $A[i][k]$ and $B[k][j]$. This is the same pair of memory accesses as `ijk`, sharing the same crippling **Stride-N** issue on matrix B.
  - The only difference is the order of the outer loops, which slightly changes the temporal locality for the outer loops but does not fix the fundamental spatial locality issue in the inner loop.

This configuration suffers from a similarly high miss count to `ijk` and poor runtime. The high L1 D-Cache Load Miss count directly causes the low IPC and slower runtime, as the CPU is constantly stalled waiting for data from main memory.

The results conclusively show that performance optimization for matrix operations on modern architectures is not purely algorithmic ($O(N^3)$ is fixed) but is fundamentally about **data access alignment** with the hardware's memory hierarchy to maximize **spatial locality**. The data confirms that **`ikj`** is the optimal order for row-major matrices in C.

6 Conclusion

The profiling methodology successfully characterized the matrix multiplication application across varying resource constraints and memory access configurations. The application was initially demonstrated to be **CPU-bound** (high IPC of ≈ 3.82). A subsequent modification (Task 2) produced a workload whose defining characteristic was a significant drop in IPC ($\approx 70.2\%$ drop from ≈ 3.82 to ≈ 1.14), which is the classic metric signature of a workload stalling on I/O. The analysis of loop orders (Task 3) revealed that the `ikj` **configuration provides the most efficient memory access pattern**, resulting in superior performance (fastest runtime, highest IPC) due to maximized spatial locality for the row-major matrices B and C, leading to the fewest L1 D-Cache Load Misses. These findings underscore the critical importance of understanding and exploiting hardware architecture when developing high-performance computing software.

7 Submission Checklist and Reproduction Commands

The successful completion of the profiling assignments required the generation of several artifacts, which are documented below.

Table 2: Submission Artifacts

Artifact	**Files**	**Status**
Script (Python/Bash)	`profile_runner.py` / `plotter.py`	Complete
C Code	`matrix_multiplication.c`	Modified / Documented
Profiling Data	`universal_metrics.csv` / plots	Complete
Documentation	`analysis_report.pdf`	Complete

Command to Reproduce Results: All experimental results, raw logs, and processed metrics can be generated and saved by running the single master command:

```
make run_all
```