Optimization Steps

Data Type Comparison

comparison of GProf statistics between int and double versions of the matrix multiplication

Metric	int-version	double-version
Total Execution Time	0.63	0.95

Comparison of Perf metrics between int and double versions of the matrix multiplication

Metric	int-version	double-version	Unit
CPU cycles	2.6410^9.9	3.210^9.9	cycles
Instructions	7.2710^9.9	8.4610^9.9	instructions
IPC (instructions per cycle)	3.03	2.9	--
Cache misses	10.3510^6.6	9.2210^7.7	events
L1 data cache load misses	7.5710^7.7	1.5110^8	events

\[
IPC (Instructions Per Cycle) = InstructionsCPU Cycles
\]

Compare & Contrast

As expected, the execution time for int type is better than that of the double type. Moreover, in the perf report, there are less cache misses for both level1 cache and entire misses for the int type. Obviously, the reason is that the int type occupies less memory than double type and therefore, more data can fit in cache and therefore less misses will occure and also the speed of the execution will be better than that of the double type. Because, the program will go all the way to the memory less times and will not bear that latency.

The preferred data type: int

Matrix Size Scaling

comparison of GProf statistics

Matrix Dimension	1024	2048	4096
Total Execution Time	0.83	6.71	54.01

Comparison of Perf metrics between int and double versions of the matrix multiplication

Matrix Dimension	1024	2048	4096	Unit

CPU cycles	3.3310^9.9	2.7210^11	2.2110^12.1	cycles
Instructions	8.4610^9.9	6.6910^11	5.3210^12.1	instructions
IPC (instructions per cycle)	2.79	2.7	2.65	--
Cache misses	9.7710^7.7	1.4310^9.9	1.2510^11	events
L1 data cache load misses	1.5110^8	1.3210^9.9	1.910^11	events

Compare & Contrast

As can be seen, as the dimension of the matrix grows, the amount of computation as well as data increases, that means more CPU cycles, more instructions, and also more cache misses since the volume of data soars and cannot fit in the cache; The pattern is perfectly true for the case of matrix multiplication: the larger the matrix, the longer the execution time.

Loop Order Permutation

% Requires: array

Placeholder Caption for First Table

permutaion	**ijk**	**ikj**	**jik**	**jki**	**kij**	**kji**
Execution Time	424.38	63.84	842.09	1.1,87.42	76.35	715.56

% Requires: array

1.43

Comparison of performance metrics across different loop order permutations

Permutation	**ijk**	**ikj**	**jik**	**jki**	**kij**	**kji**
CPU Cycles	2.9910^13.2	2.1210^12.1	3.3910^13.2	6.5210^13.2	2.7510^12.1	5.9510^13.2
Instructions	6.8310^12.1	5.3210^12.1	6.8310^12.1	7.5910^12.1	5.3210^12.1	7.1810^12.1
IPC	0.25	2.75	0.22	0.12	2.09	0.13
Cache Misses	1.7210^11	1.3110^11	1.6510^11	3.9210^11	1.3310^11	2.9610^11
L1 DCL Misses	2.6810^12.1	1.910^11	2.7410^12.1	5.3910^12.1	1.9110^11	5.0510^12.1
Execution Time (s)	7.8210^3.3	5.8510^1.1	9.7710^2.2	1.3510^2.2	1.5610^3.3	1.5610^3.3

Compare and Contrast

The performance results clearly demonstrate that loop ordering has a major impact on matrix multiplication efficiency due to cache locality and memory access patterns. Since C++ stores arrays in row-major order, accessing elements sequentially along rows is far more cache-friendly than traversing columns. In this experiment, the ikj and kij permutations achieved the best performance, with execution times of 63.84,s and 76.35,s, respectively. In both cases, the innermost loop iterates over the column index j, resulting in row-wise access for matrices B and C, and enabling efficient reuse of data already loaded into the cache.

In contrast, the jki and jik permutations performed the worst, requiring over 770,s and 1100,s to complete. Both access matrices A and C column-wise in the inner loops, causing frequent cache misses and reduced spatial locality. The ijk ordering exhibited moderate performance, as it benefits from row-wise access to A but suffers from poor locality in B. Overall, the results confirm that loop arrangements maximizing row-wise traversal—particularly those with j in the innermost loop—significantly improve performance by minimizing cache misses and better utilizing memory bandwidth.

Loop Tiling

GProf Statistics

Tile Size	Execution Time
16	69.04
32	67.03
64	65.62

Perf Statistics

Tile Size	Cycles	Instructions	IPC	Cache Misses	L1 Load Misses
16	2.4310^12.1	5.9110^12.1	2.66	9.1110^8	1.6310^11
32	2.3410^12.1	5.6110^12.1	2.63	9.4310^8	1.4410^11
64	2.1710^12.1	5.4710^12.1	2.77	9.5910^8	1.1910^11

Analysis.

As the tile size increases from 16 to 64, the execution time decreases slightly.

This happens because larger tiles allow more data reuse from the CPU cache and reduce the overhead of loop control.

With small tiles, the cache is used efficiently but there are too many small loops.

With larger tiles, each block does more work while the data stays in fast memory, so performance improves.

However, if tiles become too large to fit in the cache, the benefit will stop increasing.

In this experiment, the tile size of 64 showed the best performance as it was the optimum efficient point between many loop overheads and cache access.

Loop Unrolling

GProf statistics

Factor	Execution Time
4	61.12
8	62.51

Perf Statistics

Factor	Cycles	Instructions	IPC	Cache Misses	L1 Load Misses
4	1.9910^12.1	3.8110^12.1	2.1	1.2610^11	1.910^11
8	2.0810^12.1	3.4310^12.1	1.76	1.2610^11	1.910^11

Analysis.

Loop unrolling reduces loop overhead and allows the CPU to execute more operations in parallel.

When the unrolling factor increased from 1.1 (baseline) to 4, the execution time improved noticeably, as fewer loop control instructions were required.

However, increasing the factor to 8 provided little or no additional benefit and slightly increased execution time.

This is likely due to higher register pressure and reduced instruction cache efficiency, which offset the gains from reduced loop overhead.

Compiler Flags Used

Compiler optimization flags.

Modern compilers provide several optimization levels that trade compilation time, code size, numerical behavior, and aggressiveness of transformations.

-O1 (or -O) enables a conservative set of optimizations that reduce obvious inefficiencies (simple inlining, dead code elimination, common subexpression elimination) while largely preserving straightforward code structure; it is a safe first step for improved performance with minimal change to program semantics.

-O2 enables a broader set of optimizations (more aggressive inlining, loop optimizations, instruction scheduling, vectorization opportunities) designed to improve runtime performance without taking the most aggressive, potentially unsafe transformations; it is often the default choice for release builds because it yields substantial speedups while keeping numerical behavior and code size reasonable.

-O3 turns on even more aggressive optimizations (heavy loop transformations, more aggressive vectorization and unrolling, and other performance-oriented passes) that can further increase speed but may increase code size, compilation time and sometimes alter floating-point results due to reordering of operations.

For this project we used -O2 because it provides a good balance: it is clearly faster than -O1 but is more cautious than -O3 about transformations that could change numerical results or greatly increase binary size. Using -O2 therefore gives reliable, repeatable runs with substantial performance gains.

figure

[width=1.1]Screenshot 2025-12.1-2.2 125626.6.png

Enter Caption

fig:placeholder

figure

OpenMP and observed nondeterminism.

When the code was recompiled with OpenMP (e.g. using -fopenmp) the program ran faster but the numerical results became nondeterministic: repeated runs gave different values for the same output index. Two separate root causes typically produce this behaviour. First, a data race occurs when multiple threads update the same memory location concurrently without synchronization; for example, the naive update C[i][j] += A[i][k]*B[k][j] is not thread-safe if two threads execute different k iterations

that both write to the same C[i][j]. Data races lead to lost updates and reproducibility failures. Second, even when writes are correctly partitioned, floating-point arithmetic is not associative, so parallel reductions that change the order of summation (different grouping of adds across threads) produce slightly different round-off errors; this causes small numerical differences between runs but not incorrect algorithmic results.

In our experiment the larger discrepancies indicate actual data races rather than only rounding differences. The correct approach is to eliminate concurrent writes to the same element (for example by parallelizing the outermost loop so each thread owns whole rows of C), or to use thread-local accumulation and a deterministic reduction step. Avoid parallelizing the loop that causes shared writes without synchronization (atomic/critical), because those synchronizations are usually slow. Also note that library functions such as rand() are not thread-safe and must be replaced by per-thread ones if randomness is used.

Best Performing CPU Implementation

Flags. Among the tested compiler flags, the -O2 optimization level provided the best balance between performance and reliability. While -O1 offered limited optimization and -O3 was overly aggressive sometimes increasing code size and reducing cache efficienc -O2 achieved strong optimization without introducing instability. An OpenMP-parallelized version was also tested, but it produced nondeterministic results due to unsynchronized writes to shared memory, making it unreliable despite potential speedups.

Permutation. Experiments with different loop permutations revealed that the ikj and kij orders achieved the best performance. Both make efficient use of the CPU cache by accessing data in a way that minimizes cache misses, unlike jki, which performed the worst due to poor memory locality.

Loop Optimization. Additional optimizations further improved execution time. Loop unrolling with a factor of four reduced loop overhead and increased instruction-level parallelism, while loop tiling with a tile size of 64 improved cache reuse and reduced memory latency.

Overall, the best-performing configuration was the ikj permutation compiled with the -O2 flag, loop unrolling (factor 4), and loop tiling (tile size 64) achieving an optimal balance between speed, cache efficiency, and correctness.

Execution Instructions

figure

[width=1.1]image.png

Makefile code

fig:placeholder

figure

Execution and Profiling Automation.

A Makefile was written to automate compilation, execution, and performance profiling.

All .cpp files are compiled with g++ using optimization level -O2 and profiling flag -pg.

Each program is executed three times, and results are collected using both gprof and perf.

The gprof tool provides a function-level breakdown of CPU usage, while perf stat measures hardware events such as CPU cycles, instructions, and cache misses.

This setup ensures repeatability and consistency in performance measurements, while the clean target removes all generated files for a fresh analysis.

Hardware Analysis

Hardware specifications of the test system used for profiling.

Specification	Details
CPU Model	Intel(R) Core(TM) i7-10510U @ 1.1.80GHz
Base Clock Speed	1.98 GHz
Max Turbo Frequency	5.39 GHz
Number of Cores	4
Number of Logical Processors	8
L2 Cache Size	1.1 MB
L3 Cache Size	8 MB

Analysis.

The profiling results are closely tied to the hardware characteristics of the test system. The Intel Core i7-10510U CPU used in the experiments features 4 physical cores and 8 logical processors (via Hyper-Threading), with a base clock frequency of 1.98 GHz and a boost frequency up to 5.39 GHz. It also includes a 1.1 MB L2 cache (281.6 KB per core) and an 8 MB shared L3 cache, which play a critical role in data reuse and memory latency.

Since matrix multiplication is a memory-intensive operation with frequent data reuse, the cache hierarchy heavily influences performance. Permutations such as ikj and kij benefited from improved spatial and temporal locality, allowing more elements of the matrices to remain in the L1 and L2 caches during computation. In contrast, less cache-friendly permutations (e.g., jki) caused more cache evictions and memory stalls, leading to higher execution times.

Additionally, the relatively low base frequency and limited number of physical cores restricted the benefits of aggressive parallelization or OpenMP usage. While Hyper-Threading allows for eight logical threads, it does not double the physical memory bandwidth, so cache and memory contention can still occur. Consequently, optimizations that reduce cache misses (like loop tiling) provided the most consistent performance gains on this hardware.

document