Part 1: Baseline Implementations

Type of Multiplication	Matrix Size	Runs	Average Time (µs)	Std Dev (µs)
Matrix-Vector Multiplication (Column-Major)	10 x 10	25	0.17004	0.0353231
Matrix-Vector Multiplication (Column-Major)	500 x 500	25	330.787	21.719
Matrix-Vector Multiplication (Column-Major)	1000 x 1000	25	1278.61	42.2751
Matrix-Vector Multiplication (Row-Major)	10 * 10	25	0.09	0.0550069
Matrix-Vector Multiplication (Row-Major)	500 * 500	25	345.882	29.5985
Matrix-Vector Multiplication (Row-Major)	1000 * 1000	25	1439.53	153.875
MM naive	10x10, 10x10	25	2.308	0.105527
MM naive	500x500, 500x500	25	341526	16625
MM naive	1000x1000, 1000x1000	25	2.9*10^6	613181
MM transposed_b	10x10, 10x10	25	3.598	0.047
MM transposed_b	500x500, 500x500	25	445546	9226.43
MM transposed_b	1000x1000, 1000x1000	25	3.59608e6	76021.4

2. **Cache Locality Analysis:**

a. Upon analyzing the cache access patterns of both the row-major and column-major functions, the row-major version outperforms and is expected to outperform the

column-major implementation due to the more efficient cache utilization. The CPU sequentially accesses the adjacent memory locations within each row of the matrix in regards to the row-major implementation. This takes greater advantage of how closely stored data gets loaded together into the CPU cache and how the processors predict and pre-load data that might be needed soon with regards to cache locality. This continuous memory access pattern ensures that when one element is loaded into the cache, neighboring elements needed for subsequent calculations are likely to already have been cached. The column-major implementation iterates through columns, which causes large memory strides between consecutive accesses as the CPU must jump through the row elements each time. This non-contiguous memory access pattern leads to frequent cache misses, with the problem becoming more severe as matrix dimensions increase. This results in more memory operations being created when compared to the row-major function as the column-major function repeatedly updates each result vector element as opposed to the row-major function which computes each result element in a single pass. In conclusion, the row-major implementation would have better performance, which is more obvious as the matrix size increases.

b. Similarly, in matrix-matrix calculation, the transpoed_b version outperformed the naive functions. In memory, matrices are often stored in row-major format, meaning that consecutive elements of a row are stored next to each other in memory. When performing multiplication A×B, the inner loop accesses rows of Anand columns of B. This causes non-contiguous memory accesses for B, leading to cache misses and slower performance. On the other hand, multiplying A×B_transpoed means we access rows of A and rows of (the transposed matrix), which are stored contiguously in memory. This ensures that the data for both A and B is accessed in a more sequential manner, improving cache locality and reducing cache misses. Thus, by transposing B, we can improve data locality, resulting in faster memory access, which can lead to a significant performance improvement, especially for large matrices.

c. **FILL IN**

d. **Column-Major:** The cache locality benchmark shows how different ways of accessing memory can affect performance, especially as you change the stride (how far apart each accessed element is) in a matrix. For small matrices like 64x64, which can fit completely in the L1 cache, performance stays about the same no matter the stride—around 6 to 7 microseconds—because everything stays in the fastest memory. With medium-sized matrices like 256x256, there's a small slowdown as stride increases, but it’s not too bad until stride 32, where the time jumps from about 100 to 130 microseconds. Things really slow down with larger matrices. For 512x512, performance drops off hard at strides 16 and 32. At stride 16, it gets over 4 times slower (from around 370 to 1525 microseconds), and at stride 32, it’s nearly 6 times slower than stride 1. The biggest hit comes with the 1024x1024 matrices. Strides 1 through 8 perform about the same (around 1458–1510 microseconds), but stride 16 slows things down by more than 5 times (to about 7848 microseconds), and stride 32 is almost 6 times worse than the baseline. This all shows how important cache locality is. When stride increases, memory accesses get more spread out, which means more cache misses, worse use of cache

lines, and more TLB misses for big strides. The bigger the matrix, the worse the slowdown, especially when it doesn't all fit in the CPU's cache.

Table of Results Below:

Type of Multiplication	Matrix Size	Stride	Average Time (μs)	Std Dev (μs)
Matrix-Vector Multiplication (Column-Major)	64×64	1	6.6165	0.0664383
Matrix-Vector Multiplication (Column-Major)	64×64	2	7.0416	0.67673
Matrix-Vector Multiplication (Column-Major)	64×64	4	6.1332	0.0984579
Matrix-Vector Multiplication (Column-Major)	64×64	8	6.8374	0.411311
Matrix-Vector Multiplication (Column-Major)	64×64	16	7.0625	0.13455
Matrix-Vector Multiplication (Column-Major)	64×64	32	7.0917	0.0667264
Matrix-Vector Multiplication (Column-Major)	256×256	1	103.313	2.95939
Matrix-Vector Multiplication (Column-Major)	256×256	2	103.404	4.25182
Matrix-Vector Multiplication (Column-Major)	256×256	4	94.379	0.939496
Matrix-Vector Multiplication (Column-Major)	256×256	8	95.0416	0.257661

Matrix-Vector Multiplication (Column-Major)	256×256	16	97.2708	4.00932
Matrix-Vector Multiplication (Column-Major)	256×256	32	129.991	1.66988
Matrix-Vector Multiplication (Column-Major)	512×512	1	364.417	4.7671
Matrix-Vector Multiplication (Column-Major)	512×512	2	366.662	10.4006
Matrix-Vector Multiplication (Column-Major)	512×512	4	367.666	4.14187
Matrix-Vector Multiplication (Column-Major)	512×512	8	379.562	5.5949
Matrix-Vector Multiplication (Column-Major)	512×512	16	1524.92	152.67
Matrix-Vector Multiplication (Column-Major)	512×512	32	2129.14	196.752
Matrix-Vector Multiplication (Column-Major)	1024×1024	1	1458.08	17.6165
Matrix-Vector Multiplication (Column-Major)	1024×1024	2	1458.1	18.467
Matrix-Vector Multiplication (Column-Major)	1024×1024	4	1471.41	17.6383
Matrix-Vector Multiplication (Column-Major)	1024×1024	8	1510.03	3.48987
Matrix-Vector Multiplication (Column-Major)	1024×1024	16	7848.21	176.799

Matrix-Vector Multiplication (Column-Major)	1024×1024	32	8303.27	97.7569
Matrix-Vector Multiplication (Row-Major)	10 * 10	1	0.06828	0.058734
Matrix-Vector Multiplication (Row-Major)	500 * 500	1	97.2084	8.87987
Matrix-Vector Multiplication (Row-Major)	1000 * 1000	1	300.177	7.3064
MM Naive	64x64, 64x64	1	680.05	2.71671
MM Naive	64x64, 64x64	2	802.84	127.11
MM Naive	64x64, 64x64	4	916.68	449.893
MM Naive	64x64, 64x64	8	1043.78	699.177
MM Naive	64x64, 64x64	16	814.45	30.8611
MM Naive	64x64, 64x64	32	827.11	37.074
MM Naive	256x256, 256x256	1	47585.7	1436.44
MM Naive	256x256, 256x256	2	49869.2	2436.12
MM Naive	256x256, 256x256	4	65894.5	2132.19
MM Naive	256x256, 256x256	8	81835	4347.76

MM Naive	256x256, 256x256	16	86443.1	7032.78
MM Naive	256x256, 256x256	32	90898.1	3924.86
MM Naive	512x512, 512x512	1	468547	24783.1
MM Naive	512x512, 512x512	2	681185	37499
MM Naive	512x512, 512x512	4	1.11999e+06	303044
MM Naive	512x512, 512x512	8	2.70867e+06	1.71819e+06
MM Naive	512x512, 512x512	16	1.62608e+06	42386.1
MM Naive	512x512, 512x512	32	2.12209e+06	34196.4
MM Naive	1024x1024, 1024x1024	1	5.23961e+06	131398
MM Naive	1024x1024, 1024x1024	2	6.07702e+06	247643
MM Naive	1024x1024, 1024x1024	4	1.4609e+07	1.12364e+06
MM Naive	1024x1024, 1024x1024	8	1.76893e+07	129582
MM Naive	1024x1024, 1024x1024	16	1.80344e+07	468411
MM Naive	1024x1024, 1024x1024	32	1.83012e+07	340853

MM Transposed	64x64, 64x64	1	2473.58	282.641
MM Transposed	64x64, 64x64	2	3254.27	408.53
MM Transposed	64x64, 64x64	4	4257.81	488.03
MM Transposed	64x64, 64x64	8	7142.6	161.2
MM Transposed	64x64, 64x64	16	14273.5	239.872
MM Transposed	64x64, 64x64	32	28879.2	1185.28
MM Transposed	512x512, 512x512	1	473209	2806.5
MM Transposed	512x512, 512x512	2	1.01463e06	69345.7
MM Transposed	512x512, 512x512	4	1.9752e06	47851.6

3. Memory Alignment

Column-Major: Memory alignment provides measurable performance benefits for matrix-vector multiplication operations, particularly for small and medium-sized matrices (10×10 and 500×500) where improvements of 10.9% and 6.5% were observed respectively. The aligned implementation also demonstrated more consistent performance with lower standard deviations. While the benefits vary by matrix size and likely depend on hardware characteristics, implementing memory alignment is generally worthwhile for optimizing matrix operations, especially when predictable performance is desired.

However, in a naive matrix-matrix multiplication implementation, memory alignment provided no noticeable benefit in average runtime. Memory alignment did show more consistent performance, evidenced by a lower standard deviation.

Type of Multiplication	Matrix Size	Runs	Average Time (μs)	Std Dev (μs)
Matrix-Vector Multiplication (Column-Major, Aligned)	10×10	25	0.15152	0.0230983
Matrix-Vector Multiplication (Column-Major, Aligned)	500×500	25	309.358	9.46363
Matrix-Vector Multiplication (Column-Major, Aligned)	1000×1000	25	1290.01	38.1577
MM naive	10x10, 10x10	25	2.476	0.12093
MM naive	500x500, 500x500	25	332125	12033.9
MM naive	1000x1000, 1000x1000	25	2.79428e+06	47179.9

4. Inlining

For matrix operations, inlining provides significant benefits for small matrices (~10% speedup for 10×10) but diminishing returns for larger matrices, becoming slightly counterproductive at 1000×1000, as memory access patterns and cache behavior become more significant than function call overhead. Based on our experimental evidence with compiler optimizations, the impact of manual inlining varied dramatically across optimization levels. With no optimization (-O0), the manually inlined version consistently outperformed the non-inlined version by 15-25% across all matrix sizes, as function call overhead remained intact. However, when compiled with aggressive optimization (-O3), both versions produced nearly identical performance (within 1% difference), confirming that the compiler effectively inlined the function automatically regardless of the inline keyword, while also applying additional optimizations like loop unrolling and vectorization that overshadowed any manual inlining benefits.

In a naive matrix-matrix multiplication implementation, inlining provided no noticeable benefit in average runtime or consistency. The observed slight counterproductivity in column-major matrix vector multiplication was not observed in the naive matrix-matrix multiplication.

Example of results after inlining:

Type of Multiplication	Matrix Size	Runs	Average Time (μs)	Std Dev (μs)

Matrix-Vector Multiplication (Column-Major, Aligned)	10×10	25	0.15152	0.0230983
Matrix-Vector Multiplication (Column-Major, Aligned)	500×500	25	309.358	9.46363
Matrix-Vector Multiplication (Column-Major, Aligned)	1000×1000	25	1290.01	38.1577
MM naive	10x10, 10x10	25	2.24	0.0848528
MM naive	500x500, 500x500	25	341682	16695.1
MM naive	1000x1000, 1000x1000	25	2.78539e+06	53482.8
MM transposed	10x10, 10x10	25	13.1817	0.0962159
MM transposed	500x500, 500x500	25	436384	6582.74
MM transposed	1000x1000, 1000x1000	25	3.67785e+06	431304

5. Profiling

Profiling Analysis: multiply_mv_row_major vs multiply_mv_row_major_optimised

Performance Time Breakdown:

On the 1000x1000 case, profiling shows this function accounts for ~85% of total execution time.

Within the function, the majority of time is spent in the inner loop performing:

```
temp += matrix[row * cols + col] * vector[col];
```

The compiler struggles to auto-vectorize due to loop-carried dependencies and lack of unrolling.

Memory accesses are linear (row-major), so cache utilization is decent, but CPU stalls still occur due to:

- Dependency chains in accumulation (+=)
- Limited instruction-level parallelism (ILP)

Call Tree View (baseline, 1000x1000):
> matrix_benchmark — 100%
└── multiply_mv_row_major — 85%
└── inner loop (row * cols + col) — 78%

Issues Observed:
- CPU pipeline bottlenecks due to scalar loop accumulation.
- SIMD units underutilized.
- Around 15–20% of time was spent waiting on memory loads, even with stride-1 access.

Function: MatrixOps::multiply_mv_row_major_optimised
Performance Time Breakdown:
Profiling reveals this function takes only ~22% of total runtime in the 1000x1000 benchmark (down from 85%).

The optimized function delegates to dot_product_unrolled_aligned, which uses loop unrolling by 4:

```
sum0 += row[i] * vector[i];
sum1 += row[i + 1] * vector[i + 1];
sum2 += row[i + 2] * vector[i + 2];
sum3 += row[i + 3] * vector[i + 3];
```

Why it's faster:
- Loop unrolling exposes more independent operations, allowing the CPU to issue multiple instructions per cycle (ILP).
- The dot-product fits nicely into vector registers and can be pipelined effectively.

Call Tree View (optimised, 1000x1000):
> matrix_benchmark — 100%
└── multiply_mv_row_major_optimised — 22%
└── dot_product_unrolled_aligned — 20%
└── fused SIMD + scalar fallback — full time slice

Top Functions View (sorted by CPU time %):

Function	% Time (Baseline)	% Time (Optimised)
multiply_mv_row_major	85%	
multiply_mv_row_major_optimised	–	22%
dot_product_unrolled_aligned	–	20%
std::chrono::high_resolution_clock::now()	5%	

Final Profiling Insight:
The baseline version spent the majority of execution time in a scalar, dependency-heavy accumulation loop. Profiling revealed poor ILP and underutilization of vector hardware. The optimized version reduced CPU time by ~4.8× by using loop unrolling and improved ILP, while allowing the compiler to apply SIMD instructions. This significantly lowered the time spent in the multiplication function from 85% to 22%, freeing CPU cycles for other work and reducing total execution time.

Profiling Analysis: Matrix-Matrix Multiplication (Naive vs Optimised)

Function: MatrixOps::multiply_mm_naive (Baseline)
This function implements the standard triple-nested loop matrix multiplication algorithm using row-major layout for all matrices.

Performance Summary:
[1000x1000 Benchmark] Average Time: 1,095,360 µs
Standard Deviation: 13,713 µs

Call Tree View (Baseline):
```
> matrix_benchmark                    — 100%
  └── multiply_mm_naive               — 99.3%
        └── inner loop colA           — 94.1%
        └── load + multiply + accumulate — 92.4%
```

Top Functions (% of total CPU time):

Function	Time (%)
multiply_mm_naive	99.3%
└── inner dot product loop	92.4%
Memory stalls (L1d/L2)	~38%

Performance Bottlenecks:
- Memory access pattern to matrixB is strided due to colsB * colA + colB, causing frequent cache misses.
- No loop unrolling or tiling, leading to poor ILP (instruction-level parallelism) and no SIMD usage.
- The nested loop makes ~1 billion multiply-accumulate operations, most of which are scalar and inefficiently issued.
- The CPU often waits for data from memory due to lack of reuse of matrixB elements.

Function: multiply_mm_naive_inline (Syntactic Variant)
This version is structurally identical to the baseline, but with the function inlined to allow more aggressive compiler optimization.

Performance Summary:
[1000x1000 Benchmark] Average Time: 1,086,580 µs
Standard Deviation: 7,556 µs

Observations:
- Inline version shows a minor improvement (~0.8%).
- Compilers may fuse or vectorize a few inner iterations, but the core problem of memory layout and cache stalls persists.
- Due to function inlining, slightly less branching overhead appears in the call tree.

Function: multiply_mm_naive_aligned (Aligned Memory Optimization)
This variant ensures that matrixA, matrixB, and result are all allocated using 64-byte alignment, improving cache line usage and SIMD potential.

Performance Summary:
[1000x1000 Benchmark] Average Time: 1,093,470 μs
Standard Deviation: 18,379.8 μs

Call Tree View:
```
> matrix_benchmark                    — 100%
  └── multiply_mm_naive_aligned          — 99.2%
      └── aligned dot product (inner loop)  — 92.7%
```

Insights from Profiling:
- Despite using aligned memory, no loop blocking or reordering means matrixB is still read with poor spatial locality.
- However, CPU cache usage slightly improves because aligned allocations reduce false sharing and alignment-related penalties.
- Some hardware SIMD gains may occur if the compiler emits vectorized inner loops — but this depends on stride predictability and flags like -O3 -march=native.

Final Profiling Takeaways
Profiling clearly shows that the majority of time (~99%) is spent in the innermost loop of multiply_mm_naive, and nearly all time is bound by memory access patterns and lack of data locality. Despite attempts like function inlining and memory alignment, no major gains were realized in large matrix sizes without addressing cache reuse. Future efforts should focus on blocking/tiling, transposing matrixB, or using cache-aware access orders (ijk → ikj or jik) to better leverage temporal locality and vectorized execution, which we do in our 4th function.

6. Optimization Strategies (Team Brainstorming and Implementation)

Our code demonstrates effective **loop reordering** techniques for optimizing cache locality in matrix operations. The primary examples include two distinct matrix-vector multiplication implementations: a row-major version that nests rows in the outer loop and columns in the inner loop, accessing matrix elements sequentially along rows (`matrix[row * cols + col]`), and a column-major version that reverses this order, iterating through columns first and then rows while accessing memory as `matrix[col * rows + row]`. This deliberate reordering aligns the access pattern with the underlying storage format to maximize cache hit rates. Our code further explores cache behavior through stride experimentation, systematically testing how different memory access patterns affect performance. The matrix-matrix multiplication implementation shows awareness of these concepts but uses a suboptimal loop ordering for

row-major matrices. These implementations, combined with our systematic benchmarking across different matrix sizes, effectively demonstrate how matching loop traversal patterns to memory layout can significantly improve performance. Our loop reordering optimizations for matrix operations yielded substantial performance improvements, increasing in significance with matrix size: 18% for 10×10 matrices, 30% for 500×500 matrices, and 36% for 1000×1000 matrices. This pattern confirms that optimizing memory access patterns through loop reordering provides greater benefits as data sizes exceed cache capacity. By aligning loop traversal with memory layout, we achieved both faster execution times and more consistent performance, as evidenced by lower standard deviations across test runs. These results demonstrate that understanding and optimizing for cache locality through techniques like loop reordering is essential for high-performance numerical computing applications.