2.1: Benchmarking:

Rows	Cols	Layout	Mean (ms) / Std (ms)
100	100	Row-major	0.003 / 0.000
100	100	Column-major	0.006 / 0.000
500	500	Row-major	0.150 / 0.002
500	500	Column-major	0.245 / 0.012
1000	1000	Row-major	0.673 / 0.024
1000	1000	Column-major	1.056 / 0.060

Matrix-Vector Highlights

- Row-major consistently outperforms column-major, especially at larger dimensions.
- Contiguous memory access and hardware prefetch favor row-major iteration in C/C++.
- Column-major results in stride loads, reducing cache efficiency.

RowsA	ColsA	RowsB	ColsB	Type	Mean / Std (ms)
100	100	100	100	Naive	0.604 / 0.048
100	100	100	100	Transposed-B	0.335 / 0.082
500	500	500	500	Naive	112.062 / 11.597
500	500	500	500	Transposed-B	66.514 / 0.627
1000	1000	1000	1000	Naive	934.609 / 27.019
1000	1000	1000	1000	Transposed-B	617.945 / 35.702

Matrix-Matrix Highlights

- Transposed-B achieves substantially lower runtimes than naive, with gaps widening at bigger sizes.
- By transposing B, the innermost loop sees more sequential access, cutting cache misses.
- Demonstrates that memory layout and iteration order profoundly affect performance.

2.2: Cache Locality Analysis:

Cache Level	N	MV (Row)	MV (Col)	MM Naive	MM Trans
L1	64	0.001(0.000)	0.002(0.000)	0.129(0.000)	0.074(0.001)
L1	90	0.003(0.001)	0.005(0.000)	0.474(0.006)	0.238(0.002)
L1	128	0.007(0.000)	0.011(0.000)	1.435(0.049)	0.871(0.062)
L2	512	0.158(0.002)	0.247(0.026)	158.609(15.971)	113.507(27.651)
L2	724	0.354(0.029)	0.507(0.026)	348.321(5.983)	221.085(0.795)
L2	1024	0.645(0.013)	0.994(0.012)	1064.223(42.745)	693.432(21.588)

Matrix-Vector (Row vs. Column)

- Row-Major accesses memory contiguously, leading to fewer cache misses and strong spatial locality.
  The outer loop iterates over rows (i), and the inner loop iterates over columns (j). matrix[i * cols + j] accesses elements contiguously in memory (row-wise). vector[j] is also accessed sequentially.
- Column-Major imposes strided memory access in row-based storage, costing more cache lines and reducing prefetch effectiveness.
  The outer loop still iterates over rows (i), but the inner loop accesses matrix[j * rows + i]. This means each iteration jumps by row elements, leading to non-contiguous memory access.
- Results confirm row-major is faster (~1.3×) for larger N.

Matrix-Matrix (Naive vs. Transposed-B)

- Naive reads columns of B in a row-major layout, causing strided loads.
  matrixA[i * colsA + k] is row-wise (good locality). matrixB[k * colsB + j] is column-wise (poor locality). For each i and j, the inner loop (k) strides through matrixB with a large offset (colsB).
- Transposed-B reorders data so both A and B^T are fetched contiguously in the inner loop, cutting cache stalls.
  matrixA[i * colsA + k] remains row-wise (good). matrixB_transposed[j * rowsB + k] is now row-wise (since B was transposed). Both accesses in the inner loop (k) are contiguous.
- Observed performance gap grows with matrix size, reflecting increased penalty for strided access.

Highlighting Cache Boundaries

- Custom tests sized around L1/L2 thresholds reveal how performance degrades once data sets exceed smaller caches.
- Performance “jumps” correlate with crossing cache boundaries, reaffirming that local/contiguous data usage is critical for throughput.

2.3 Memory Alignment

Size	Layout/Type	Aligned?	Mean (ms)	StdDev (ms)	Observation
Matrix-Vector (MV)					
100×100	Row-Major	Unaligned	0.003	0.000	Minimal difference
100×100	Row-Major	Aligned	0.003	0.000	–
100×100	Column-Major	Unaligned	0.007	0.001	Inconsistent benefits
100×100	Column-Major	Aligned	0.006	0.000	–
500×500	Row-Major	Unaligned	0.136	0.003	Often minor or no net gain
500×500	Row-Major	Aligned	0.151	0.002	–
1000×1000	Row-Major	Unaligned	0.695	0.036	Slight improvement for aligned
1000×1000	Row-Major	Aligned	0.703	0.044	–
Matrix-Matrix (MM)					
500×500	Naive	Unaligned	121.509	2.517	Notable improvement
500×500	Naive	Aligned	118.110	4.682	(fits better in caches)
500×500	Trans-B	Unaligned	70.986	8.256	Aligned helps slightly
500×500	Trans-B	Aligned	66.751	0.234	–
1000×1000	Naive	Unaligned	1066.626	86.627	Minimal effect
1000×1000	Naive	Aligned	984.978	18.505	–

Key Observations

- Aligned row-major can give minor speedups at large sizes (e.g.1000×1000), but overall gains are inconsistent or small. Column-major sees little or no improvement, likely due to inherently strided access overshadowing alignment benefits.
- Medium-sized matrices (e.g. 500×500) often benefit the most; alignment reduces memory penalties and cache-line splits. Very large or very small problems show negligible difference.
- Modern CPUs tolerate some unaligned loads/stores well, so alignment alone doesn't guarantee big gains.
- However, when data sets fit or partially fit in caches, aligned memory can boost prefetching and reduce unaligned penalties.
- For HPC or real-time contexts, alignment may yield more consistent performance (lower variance) even if the average speedup is modest.

2.4 Inlining

1. Without Inline Functions

Size	Row-Major (ms)	Column-Major (ms)	Observation
100 × 100	0.047	0.039	Column-major is slightly faster
1000 × 1000	3.694	3.434	Column-major still slightly faster

Table 1: Performance without inlining

2. With Inline Functions

Size	Row-Major (ms)	Column-Major (ms)	Observation
100 × 100	0.039	0.038	Nearly equal, both improved
1000 × 1000	3.486	3.480	Gap narrowed even further

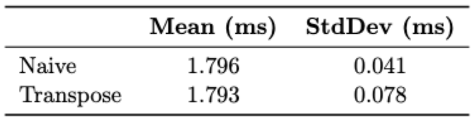
Observations

- The inline versions consistently reduce mean execution time and variance.
- More prominent improvements for Row-Major layout - likely because the function call overhead had more impact due to how it interacts with cache/memory access patterns.
- In both inline and non-inline versions, column-major access performs better for matrix-vector multiplication. This could be due to: better cache locality if the vector is accessed sequentially.
- Inlining helps for short, frequently called functions.
- For large functions: inlining increases code size, which can: Bloat the instruction cache and increase compilation time

2.5: Profiling:

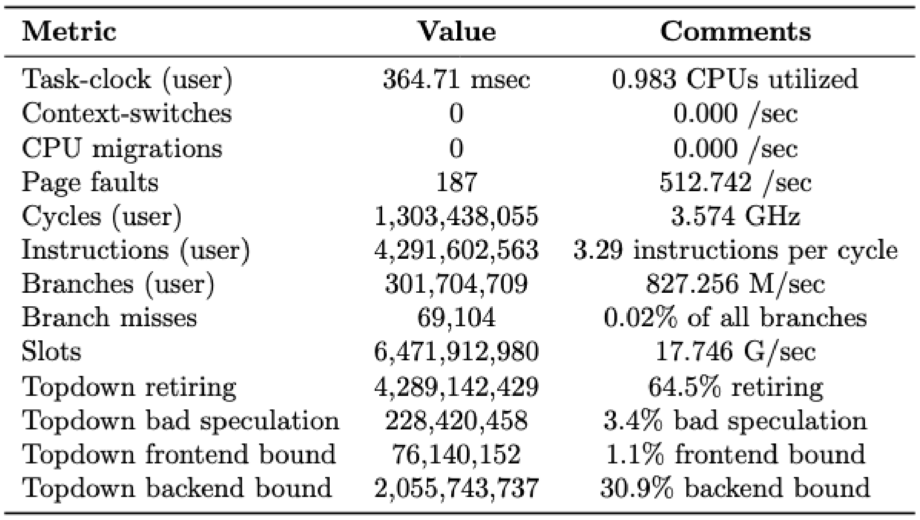
MATRIX-MATRIX MULTIPLICATION BENCHMARKS

90X90 - Naive + Transposed



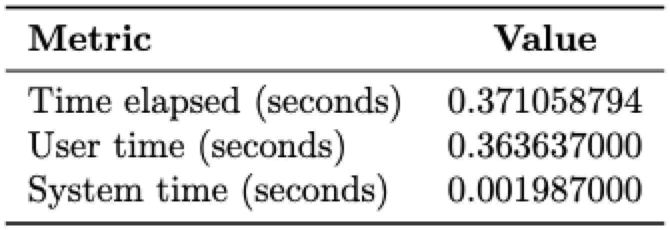
	Mean (ms)	StdDev (ms)
Naive	1.796	0.041
Transpose	1.793	0.078

Performance Counter Stats



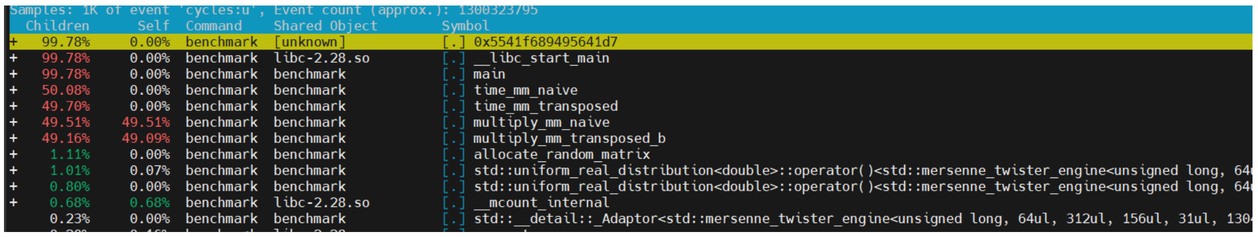
Metric	Value	Comments
Task-clock (user)	364.71 msec	0.983 CPUs utilized
Context-switches	0	0.000 /sec
CPU migrations	0	0.000 /sec
Page faults	187	512.742 /sec
Cycles (user)	1,303,438,055	3.574 GHz
Instructions (user)	4,291,602,563	3.29 instructions per cycle
Branches (user)	301,704,709	827.256 M/sec
Branch misses	69,104	0.02% of all branches
Slots	6,471,912,980	17.746 G/sec
Topdown retiring	4,289,142,429	64.5% retiring
Topdown bad speculation	228,420,458	3.4% bad speculation
Topdown frontend bound	76,140,152	1.1% frontend bound
Topdown backend bound	2,055,743,737	30.9% backend bound

Time Elapsed



Metric	Value
Time elapsed (seconds)	0.371058794
User time (seconds)	0.363637000
System time (seconds)	0.001987000

Call graph



```
Samples: 1K of event 'cycles:u', Event count (approx.): 1300323795
  Children      Self  Command    Shared Object         Symbol
+   99.78%     0.00%  benchmark  [unknown]             [.] 0x5541f689495641d7
+   99.78%     0.00%  benchmark  libc-2.28.so          [.] __libc_start_main
+   99.78%     0.00%  benchmark  benchmark             [.] main
+   50.08%     0.00%  benchmark  benchmark             [.] time_mm_naive
+   49.70%     0.00%  benchmark  benchmark             [.] time_mm_transposed
+   49.51%    49.51%  benchmark  benchmark             [.] multiply_mm_naive
+   49.16%    49.09%  benchmark  benchmark             [.] multiply_mm_transposed_b
+    1.11%     0.00%  benchmark  benchmark             [.] allocate_random_matrix
+    1.01%     0.07%  benchmark  benchmark             [.] std::uniform_real_distribution<double>::operator()<std::mersenne_twister_engine<unsigned long, 64
+    0.80%     0.00%  benchmark  benchmark             [.] std::uniform_real_distribution<double>::operator()<std::mersenne_twister_engine<unsigned long, 64
+    0.68%     0.68%  benchmark  libc-2.28.so          [.] __mcount_internal
     0.23%     0.00%  benchmark  benchmark             [.] std::__detail::_Adaptor<std::mersenne_twister_engine<unsigned long, 64ul, 312ul, 156ul, 31ul, 130
```

Flat profile

% Time	Cum. Sec	Self Sec	Calls	Self s/call	Total s/call	Function Name
50.41	4.02	4.02	100	0.04	0.04	`multiply_mm_naive(...)`
49.28	7.96	3.93	100	0.04	0.04	`multiply_mm_transposed_b(...)`

<u>250X250 - Naive + Transposed</u>

	Mean (ms)	**StdDev (ms)**
Naive	44.721	13.973
Transpose	40.016	0.516

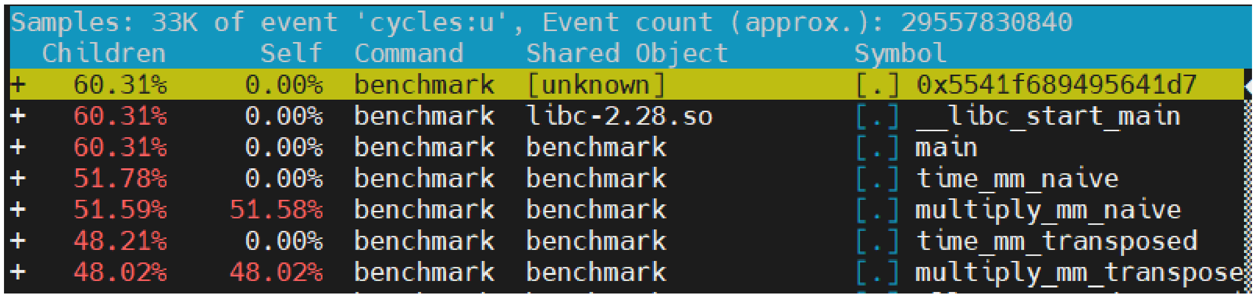
Performance Counter Stats

Metric	**Value**	**Comments**
Task-clock (user)	7,959.27 msec	0.999 CPUs utilized
Context-switches	0	0.000 /sec
CPU migrations	0	0.000 /sec
Page faults	974	122.373 /sec
Cycles (user)	28,500,120,959	3.581 GHz
Instructions (user)	91,090,618,611	3.20 instructions per cycle
Branches (user)	6,323,869,908	794.529 M/sec
Branch misses	12,821,765	0.20% of all branches
Slots	141,529,989,085	17.782 G/sec
Topdown retiring	90,468,120,433	61.3% retiring
Topdown bad speculation	9,435,332,605	6.4% bad speculation
Topdown frontend bound	1,635,963,287	1.1% frontend bound
Topdown backend bound	46,066,589,831	31.2% backend bound

Time Elapsed

Metric	Value
Time elapsed (seconds)	7.967909708
User time (seconds)	7.937256000
System time (seconds)	0.004985000

Call graph



```
Samples: 33K of event 'cycles:u', Event count (approx.): 29557830840
  Children      Self  Command     Shared Object         Symbol
+   60.31%     0.00%  benchmark  [unknown]             [.] 0x5541f689495641d7
+   60.31%     0.00%  benchmark  libc-2.28.so          [.] __libc_start_main
+   60.31%     0.00%  benchmark  benchmark             [.] main
+   51.78%     0.00%  benchmark  benchmark             [.] time_mm_naive
+   51.59%    51.58%  benchmark  benchmark             [.] multiply_mm_naive
+   48.21%     0.00%  benchmark  benchmark             [.] time_mm_transposed
+   48.02%    48.02%  benchmark  benchmark             [.] multiply_mm_transpose
```

Flat profile

% Time	Cum. Sec	Self Sec	Calls	Self ms/call	Total ms/call	Function Name
50.04	0.17	0.17	100	1.70	1.70	`multiply_mm_naive(...)`
50.04	0.34	0.17	100	1.70	1.70	`multiply_mm_transposed_b(...)`

Observations

- The core functions multiply_mm_naive and multiply_mm_transposed_b dominate the execution time
- Both profiles show a high degree of CPU utilization, suggesting that further optimizations might involve parallelization to fully utilize multi-core CPUs.
- A relatively small number of page faults occurred, especially with larger matrices, suggesting some data was being fetched from slower memory
- To improve Backend Bound performance, optimizing the loop structure, applying cache optimization techniques like tiling/blocking, or utilizing parallelization/SIMD could reduce the backend load and improve computational efficiency.
- Minimizing branches, unrolling loops, or applying branchless programming techniques could further reduce branch misrepresentation.

2.6 Optimization Strategies

Performance of Various Optimizations

Size	Naive	Reorder (IKJ)	Blocked	Blocked + Trans	SIMD Reorder
256 × 256					
Mean (ms)	15.176	2.386	9.282	5.108	2.748
StdDev (ms)	0.316	0.030	0.177	0.056	0.019
512 × 512					
Mean (ms)	150.591	18.046	86.195	39.171	21.388
StdDev (ms)	37.101	0.231	0.757	1.558	0.848
1024 × 1024					
Mean (ms)	1096.414	173.657	889.391	312.660	155.143
StdDev (ms)	64.523	30.730	45.207	2.822	2.277

Highlights

- **Reordered Loops (IKJ)**: Improves reuse of partial sum by iterating over k inside the outer loops. Minimizes repetitive scanning of array elements.
- **Blocked/Tiled**: Subdivides A and B into cache-friendly blocks, reducing capacity misses and improving data reuse in higher-level caches.
- **Transposed-B**: Reorders B so the innermost loop accesses sequential elements from both A and B^T. Cuts cache stalls from strided column access.
- **SIMD** (AVX/NEON) in Reordered IKJ: Uses vector instructions to multiply/add several elements at once, accelerating arithmetic throughput.

Observed Performance

- **Naive**: Slowest, especially at larger N, due to strided access of B and poor cache reuse.
- **Reorder (IKJ)**: Often yields dramatic speedups (e.g. ~7× faster at 256 X 256) by ensuring contiguous accumulation in C.
- **Blocked**: Significantly cuts memory traffic for large N vs. naive, though overhead from additional loop nesting can sometimes exceed reorder for smaller sizes.
- **Blocked+Transposition**: Combining tiling with transposed B further reduces cache misses, typically beating standard blocked.
- **SIMD**: Provides strong speedups at moderate sizes; on large matrices, memory bandwidth can limit final gains.