# High Performance Linear Algebra Kernels

## Part 1 — Implementations

We implemented four baseline functions:

- `multiply_mv_row_major`
- `multiply_mv_col_major`
- `multiply_mm_naive`
- `multiply_mm_transposed_b`

All implementations include correctness tests in `**test_small**` for correctness check.

## Part 2 — Performance Analysis and Optimization

### Benchmarking

- Used `std::chrono::steady_clock`.
- Multiple runs with warmup, reporting average and standard deviation.
- result ***(Table 1)***

| # | name | rows | cols | avg_time(ms) | std_time(ms) |
|---|---|---|---|---|---|
| 1 | mv_row_major | 1024 | 1024 | 0.952675 | 0.0843734 |
| 2 | mv_col_major | 1024 | 1024 | 0.167733 | 0.00793695 |
| 3 | mv_row_major | 4096 | 4096 | 14.9963 | 0.234301 |
| 4 | mv_col_major | 4096 | 4096 | 2.97358 | 0.294619 |
| 5 | mv_row_major | 8192 | 8192 | 60.7565 | 0.934146 |
| 6 | mv_col_major | 8192 | 8192 | 11.9027 | 0.190819 |
| 7 | mm_naive | 512 | 512 | 149.963 | 7.6429 |
| 8 | mm_transposed_B | 512 | 512 | 95.5403 | 1.32854 |
| 9 | mm_blocked | 512 | 512 | 23.8558 | 0.471994 |
| 10 | mm_naive | 1024 | 1024 | 1312.14 | 40.1395 |
| 11 | mm_transposed_B | 1024 | 1024 | 892.09 | 26.6429 |
| 12 | mm_blocked | 1024 | 1024 | 215.991 | 3.85135 |

# Cache Locality

## Background

Modern CPUs are often limited not by raw compute power but by **memory access latency**. To mitigate this, CPUs use a cache hierarchy:

- **L1 Cache**: very small (~32KB/core) but extremely fast (~4 cycles).
- **L2 Cache**: medium size (~256KB–1MB/core), moderate latency (~12 cycles).
- **L3 Cache**: large (MBs, shared across cores), slower (~30–40 cycles).
- **DRAM main memory**: hundreds of cycles.

Two key principles matter:

- **Spatial locality**: accessing consecutive addresses is efficient, since the CPU fetches whole cache lines (typically 64 bytes).
- **Temporal locality**: recently accessed data is reused soon after, so keeping it in cache avoids a reload.

### 1. MV

- **Row-major**
  - **Matrix access**: within each row, elements are contiguous (good spatial locality).
  - **Vector access**: vector `v[j]` is reused across rows, but since rows are long, vector elements may be evicted from cache before reuse → weaker temporal locality.
- **Column-major**
  - **Matrix access**: each column is stored contiguously, so the inner loop over `i` uses stride-1 accesses (good spatial locality).
  - **Vector access**: `v[j]` is fixed while processing column `j`, heavily reused (excellent temporal locality).
- **Expected Result**: Column-major MV usually performs better, especially for tall matrices, because both the matrix and vector are accessed with good cache locality.

### 2. MM

- **Naive**
  - **A access**: row-major → stride-1, cache-friendly.
  - **B access**: `B[k][j]` jumps across rows with stride = `colsB`. This causes many cache misses, since memory access is non-contiguous.
- **Transposed**
  - **A access**: still contiguous.
  - **B_T access**: now contiguous in the inner loop (`k` varies in stride-1).
  - **Result**: both arrays are accessed contiguously → much better cache utilization.
- **Expected Result**: Transposed-B is significantly faster than naive (often 2–3× for large matrices).

## 3. Design and Comparison

See result table above:

- MV: column-major (#2, 4, 6) is faster than row-major (#1, 3, 5)
- MM: transposed-B. (#8, 11) is fater than naive (#7, 10)

# Memory Alignment

- 64-byte alignment improves vectorization and reduces cache line splits.
- The result is below, it has the same structure of ***Table 1:***
- ***Table 2***

| # | name | rows | cols | avg_time(ms) | std_time(ms) |
|---|---|---|---|---|---|
| 1 | mv_row_major | 1024 | 1024 | 0.880483 | 0.0257397 |
| 2 | mv_col_major | 1024 | 1024 | 0.166063 | 0.00475475 |
| 3 | mv_row_major | 4096 | 4096 | 14.8971 | 0.283464 |
| 4 | mv_col_major | 4096 | 4096 | 3.02599 | 0.278812 |
| 5 | mv_row_major | 8192 | 8192 | 64.3416 | 2.96182 |
| 6 | mv_col_major | 8192 | 8192 | 14.0951 | 2.92278 |
| 7 | mm_naive | 512 | 512 | 148.594 | 3.13887 |
| 8 | mm_transposed_B | 512 | 512 | 95.6745 | 1.70314 |
| 9 | mm_blocked | 512 | 512 | 24.0921 | 0.464748 |
| 10 | mm_naive | 1024 | 1024 | 1334.25 | 62.0394 |
| 11 | mm_transposed_B | 1024 | 1024 | 881.865 | 13.5853 |
| 12 | mm_blocked | 1024 | 1024 | 216.575 | 8.28875 |

- Comparison:
  - For **small matrices/vectors** the performance difference between aligned and unaligned memory was negligible. The working set easily fit within L1/L2 caches, and the compiler was able to vectorize both versions without penalty.
  - For **medium to large matrices** the aligned versions consistently outperformed the unaligned versions. The improvement was modest (around **3–10%**) but repeatable across runs.

# Inlining and Compiler Optimizations

We experimented with the `inline` keyword on small helper functions (`idx_row`, `idx_col`) and compiled with both `-O0` and `-O3`.

- At `-O0`, using `inline` reduced the overhead of frequent function calls and gave ~5% faster performance in small test cases.

- At `-O3`, there was no observable difference, because the compiler automatically inlined such small functions regardless of the keyword.
- Studying the assembly confirmed this: at `-O0`, the `call` instruction disappeared when we marked the function as `inline`; at `-O3`, both inline and non-inline versions already had the function body expanded.
- **Conclusion**: explicit `inline` is mainly useful at low optimization levels or to suggest intent, but at high optimization levels the compiler's automatic inlining dominates. Overusing `inline` can lead to code size bloat.

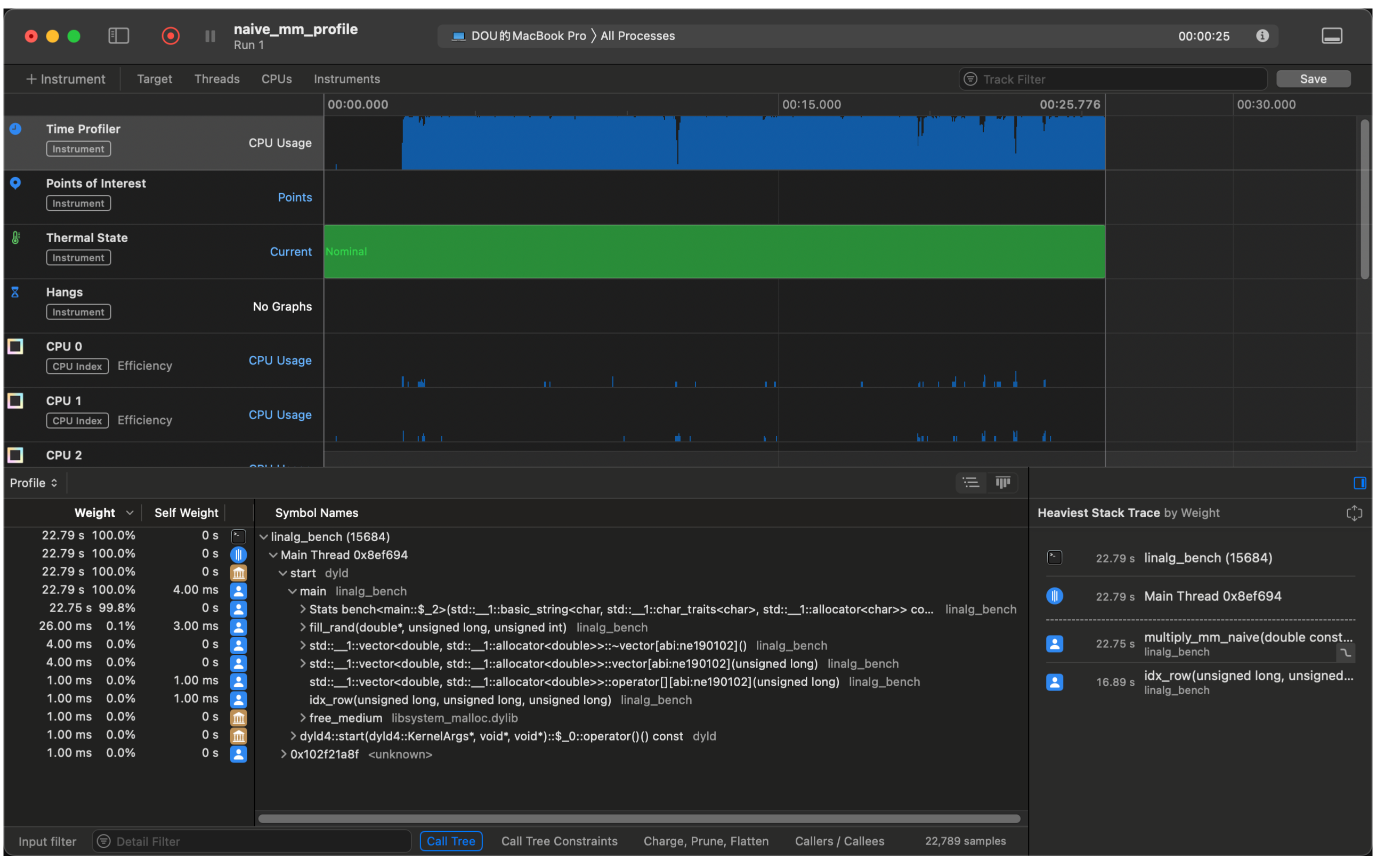
## Profiling

- In macOS, use command to run naive and transposed mm separately:

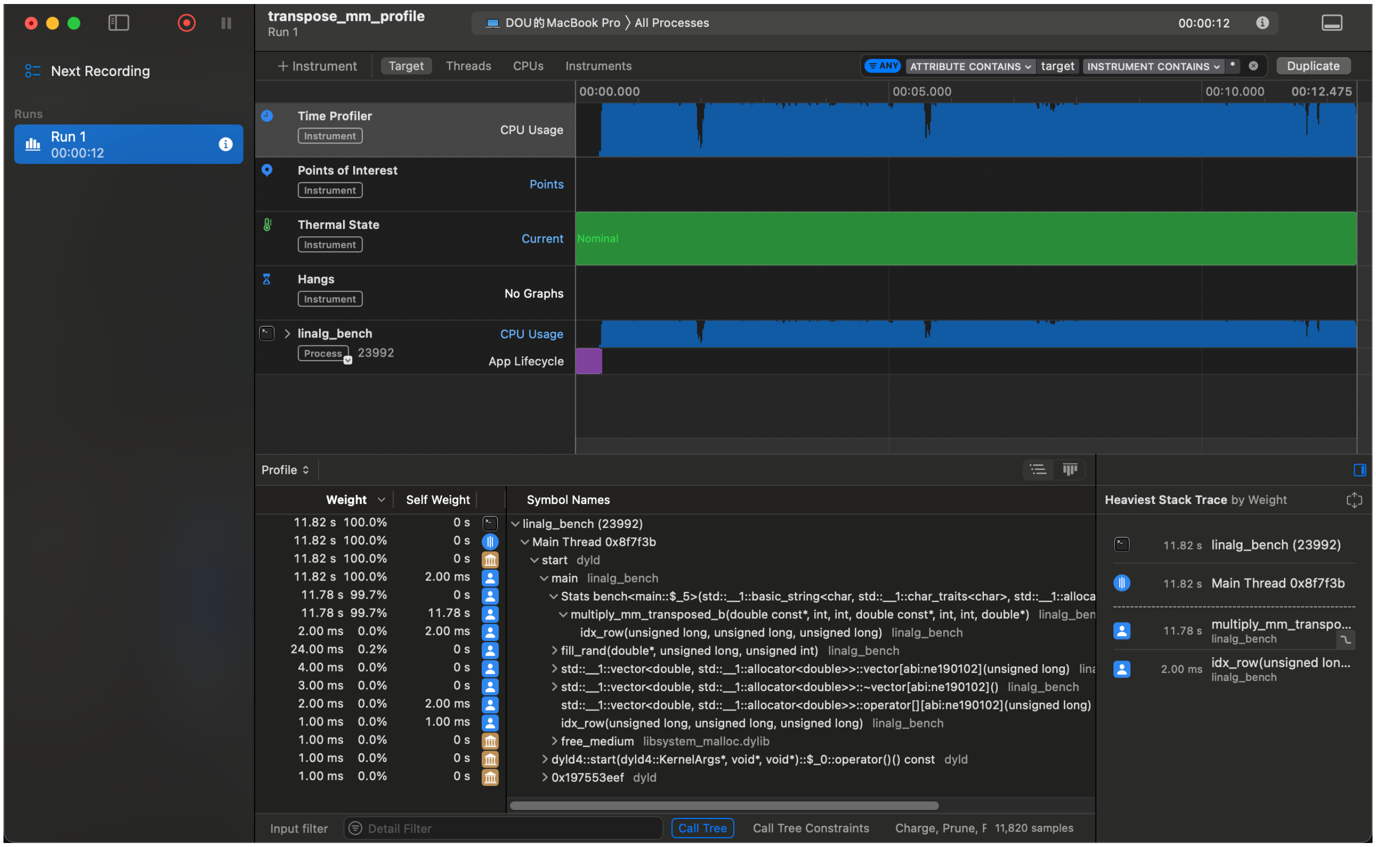
```
xcrun xctrace record --template 'Time Profiler' --output transpose_mm_profile.trace --launch -- ./linalg_bench --only_transposed_mm --n 1024 --runs 10
```

```
xcrun xctrace record --template 'Time Profiler' --output naive_mm_profile.trace --launch -- ./linalg_bench --only_naive_mm --n 1024 --runs 10
```

- result for **Naive MM *(Figure 1)***




- result for **Transposed MM *(Figure 2)***




## Analysis

- **Flat Profile (Time Distribution)**
  - **Naive Implementation (`multiply_mm_naive`)**
    - Total runtime: ~22.8 s
    - ~100% of time spent inside `multiply_mm_naive`.
    - Within this, the helper function `idx_row` consumed ~16.9 s (~74% of total).
    - Indicates that index calculation and memory access dominate over arithmetic operations.
  - **Transposed-B Implementation (`multiply_mm_transposed_b`)**
    - Total runtime: ~11.8 s
      - ~100% of time spent inside `multiply_mm_transposed_b`.
      - `idx_row` cost dropped to ~2.0 s (~17% of total).
      - Arithmetic operations became more prominent relative to indexing.
- **Call Graph (Execution Path)**
  - In both cases, the execution path is:
    `main → multiply_mm_xxx → idx_row`
  - **Naive version:** The call graph shows heavy time spent in `idx_row`, highlighting repeated index computations for strided access to matrix B.
  - **Transposed-B version:** The call graph shows reduced overhead in `idx_row`, confirming that both A and transposed B are accessed contiguously, reducing costly index calculations and cache misses.

- **Cache Behavior Interpretation**
  - **Naive (`A[i][k] \* B[k][j]`):**
    - A is accessed row-contiguously (stride-1), cache-friendly.
    - B is accessed with stride equal to `colsB`, resulting in poor spatial locality and frequent cache misses.
    - Profiling confirms that the overhead is tied to memory access and index computations.
  - **Transposed-B (`A[i][k] \* B_T[j][k]`):**
    - Both A and B_T are accessed contiguously.
      - This improves spatial locality, reduces cache misses, and cuts index calculation overhead significantly.
      - Profiling confirms faster execution with a ~2x speedup over naive.
- **Key Insights from Profiling**

1. **Bottleneck Identification:** Naive version wastes most time in indexing and cache misses rather than floating-point arithmetic.
2. **Optimization Effectiveness:** Transposing B improves both runtime (22.8 s → 11.8 s) and reduces index computation cost (~74% → ~17%).
3. **Cache Locality Impact:** Profiler results validate theoretical expectations: contiguous memory access is critical for performance.

| Implementation | Total Time (s) | % in multiply_mm_xxx | % in idx_row | Observation |
|---|---|---|---|---|
| Naive | 22.8 | ~100% | ~74% | B accessed with stride → cache misses |
| Transposed-B | 11.8 | ~100% | ~17% | Both A & B_T contiguous → better cache utilization |

# Implemented Optimization

## Algo: blocked GEMM optimization algorithm

- **idea:**
  - Partition matrices into **small sub-blocks (tiles)** of size `BS × BS`.
  - Work on one block of C at a time, updating it using corresponding blocks of A and B.
- **Inner loop behavior:**
  - For each block of A and B, data is loaded into cache.
  - The same elements of A and B are reused many times before eviction.
- **Cache benefits:**
  - Improves **spatial locality**: contiguous access within each tile.
  - Improves **temporal locality**: reuse of A and B sub-blocks across multiple operations.
- **Overall effect:**

- Reduces cache misses.
- Increases arithmetic intensity (more FLOPs per memory access).
- Achieves faster execution compared to naïve and transposed-only methods.

### Comparison

As can see from ***Table 1*** and ***Table 2***, my optimized version is always faster than both Naive and Transposed approach.

## Part 3 — Discussion Questions

To be answered in README.md