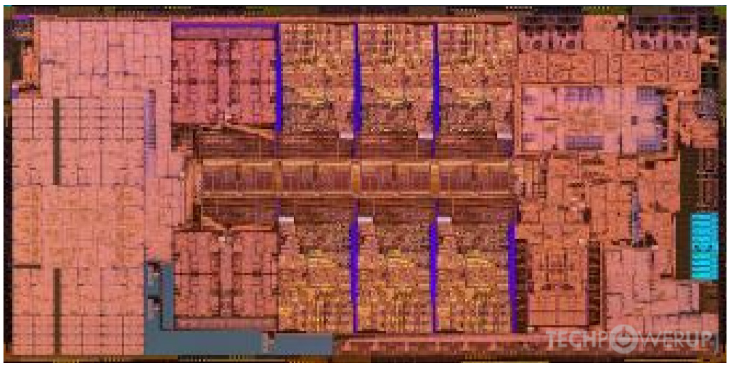
Challenge.pdf Report

Vector and Matrix Multiplication Optimization



Prepared by:
Berk Ali Demir

Department of Computer Engineering
Politecnico di Torino

July 12, 2025

Contents

1 Abstract

The purpose of this report is to analyze the performance of vector dot product and matrix multiplication in C++ using single-threaded, multithreaded (pthreads), and SIMD (AVX) approaches. Tests were conducted on an Intel Core i9-12900H under two core configurations: all cores (6P+8E) and P-cores only (6P). Results compare execution times, revealing the efficiency gains from multithreading and AVX optimizations on modern hybrid CPUs.

2 Introduction

The dot product of two vectors and the multiplication of two matrices are fundamental operations in linear algebra, widely used in scientific computing and numerical simulations. In this experiment, we evaluate the performance of both:

- The dot product of two vectors, each containing 10,000 single-precision floating-point elements.
- The multiplication of two square matrices of size $N \times N$.

For each operation, we compare three implementations in C++:

1. Single-threaded (plain C++).
2. Multi-threaded using pthreads.
3. SIMD (AVX) combined with multithreading.

All tests were run on an Intel Core i9-12900H under two different core configurations (all cores vs. only P-cores), and execution times are analyzed to quantify the benefits of parallelism and vectorization.

3 Mathematical Background

Let $\vec{a}, \vec{b} \in \mathbb{R}^n$. The dot product is defined as:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i \cdot b_i$$

$$\begin{tikzcd} \vec{a} = & 1 \arrow[d, "\times"] & 2 \arrow[d, "\times"] & 3 \arrow[d, "\times"] \\ \vec{b} = & 4 & 5 & 6 \end{tikzcd}$$

Figure 1: Visual representation of the dot product $\vec{a} \cdot \vec{b} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$. Each element of vector $\vec{a}$ is multiplied with the corresponding element of vector $\vec{b}$, and the resulting products are summed to produce a single scalar. This operation is fundamental in linear algebra and widely used in numerical computing, machine learning, and performance benchmarking.

4 Vector Generator

A custom C++ utility was developed to generate vectors populated with uniformly distributed random floating-point values in the range [1, 5]. The generated data was persisted in text format (.txt files) to facilitate reproducible testing across different implementations.

5 Dot Product – Plain C++

Below is a basic implementation of the dot product using a single loop.

```
float dot_product(const float* a, const float* b, int N) {
        float sum = 0.0f;
        for (int i = 0; i < N; ++i) {
                sum += a[i] * b[i];
        }
        return sum;
}
```

Listing 1: Plain dot product implementation

6 Dot Product Computation (Function Framework)

Each thread was assigned a portion of the input vectors. A ‘ThreadData‘ structure was used to store the start and end indices.

```
void* ComputeDotProduct(void* args) {
ThreadData* data = (ThreadData*) args;
data->result = 0.0f;
for (int i = data->start; i < data->end; ++i){
data->result += data->a[i] * data->b[i];}
pthread_exit(NULL);}
```

Listing 2: Thread worker example

7 Thread Management

Think of the `pthread_create()` call as running a kitchen, and each thread as a chef. We have multiple chefs cooking the same dish in parallel: one chef takes care of half the meal, while another chef (or a group of chefs) handles the rest. The parameters correspond to different parts of the cooking process:

- `&threads[i]`: The Chef – Actual thread, or "chef," that will be summoned to perform the task. Each chef works on a specific portion of the overall dish (problem).
- Second parameter (`NULL` or `&attr`): Chef Preferences – Defines how the chef operates. It can include special working conditions like preferred tools, personal workspace (stack size), or restrictions such as which station (CPU core) they can work at.
- `ComputeDotProduct`: The Recipe – Set of instructions the chef follows to prepare their assigned dish. Every chef executes the same recipe independently on their portion of ingredients.
- `(void*)&threadData[i]`: The Ingredients –pecific ingredients and instructions the chef needs to do their job. This includes which slice of the problem they're assigned (start and end indices), the raw materials (input arrays), and where to place the finished result.

In cooking terms:

"Call chef threads[i], provide their working conditions, hand them the ComputeDotProduct recipe, and give them their td[i] ingredients to cook their part of the meal."

```
    pthread_t threads[2];
    ThreadData data[2];

    int mid = N/2;

    data[0] = {0,mid,&a,&b,0};
    pthread_create(&threads[0],NULL, ComputeDotProduct, &data[0]);

    data[1] = {mid,N,&a,&b,0};
    pthread_create(&threads[1],NULL, ComputeDotProduct, (void*)&data[1]);
                             }
```

Listing 3: Thread management

8 Dot Product – Threading with SSE (SIMD)

In this version, we combine multithreading with SIMD instructions (specifically, SSE – Streaming SIMD Extensions) to accelerate the dot product computation. Each thread handles a chunk of the input vectors and applies vectorized operations using 128-bit registers that can process four `float` values simultaneously.

This approach aims to exploit both task-level parallelism (with threads) and data-level parallelism (via SSE), leading to significantly better performance on modern CPUs.

The process can be described as:

- The input vectors are divided evenly among threads.
- Each thread computes its own partial dot product using `_mm_loadu_ps` and `_mm_mul_ps` SSE instructions to process 4 elements at a time.
- Remaining elements (if the size isn't divisible by 4) are handled with a scalar loop.
- After all threads finish, their results are accumulated into a final scalar.

```
    void* ComputeDotProduct(void* args){
            ThreadData* data = (ThreadData*)args;

        int start = data->start;
        int end = data->end;

        __m128 sum = _mm_setzero_ps();
        int i = start;

        for (; i <= end - 4; i += 4){
                __m128 vector_a = _mm_loadu_ps(&a[i]);
                __m128 vector_b = _mm_loadu_ps(&b[i]);
                __m128 prod = _mm_mul_ps(vector_a, vector_b);
                sum = _mm_add_ps(sum, prod);}

                float temp[4];
                _mm_storeu_ps(temp, sum);
                float local_sum = temp[0] + temp[1] + temp[2] + temp[3];

                for (; i < end; i++){
                local_sum += a[i] * b[i];}

                results[data->id] = local_sum;
                pthread_exit(NULL);
                }
```

Listing 4: SSE + Threaded dot product

Implementation leverages `SSE` intrinsics such as `_mm_loadu_ps`, `_mm_mul_ps`, and `_mm_add_ps`, which operate on four floating-point values in parallel. While threading divides the workload, SSE further accelerates computation inside each thread. However, it's important to note that alignment and cache behavior may affect performance, and tuning the number of threads in relation to physical cores (especially P-cores vs. E-cores) plays a critical role in real-world efficiency.

9 Matrix Multiplication – Mathematical Background

Let $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ be two matrices. Their product $C = A \cdot B$ is defined as a new matrix $C \in \mathbb{R}^{m \times p}$, where each element c_{ij} is computed as the dot product of the i-th row of A and the j-th column of B:

$$c_{ij} = \textstyle\sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

In other words, to compute the entry at row i and column j in the result matrix C, we take the i-th row of A and the j-th column of B, multiply corresponding elements, and sum them.

Matrix multiplication is a fundamental operation in linear algebra with broad applications in numerical computing, computer graphics, machine learning, and scientific simulations.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 \cdot 7 + 2 \cdot 9 + 3 \cdot 11 & 1 \cdot 8 + 2 \cdot 10 + 3 \cdot 12 \\ 4 \cdot 7 + 5 \cdot 9 + 6 \cdot 11 & 4 \cdot 8 + 5 \cdot 10 + 6 \cdot 12 \end{bmatrix}$$

$$\begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

Figure 2: Example of matrix multiplication: a 2×3 matrix multiplied by a 3×2 matrix produces a 2×2 result. The final result matrix is enlarged here for emphasis.

10 Matrix Generator

Before performing matrix multiplication, we first generate two random matrices and store them in text files.

The following C++ utility function creates a matrix of specified size with floating-point values uniformly distributed in a given range. Each row of the matrix is written to a new line, with values separated by spaces.

This utility is helpful for creating large input datasets that can be used consistently across different implementations (single-threaded, multithreaded, and SIMD).

- `Generate_Matrix_File(filename, rows, columns, min, max)` generates a matrix and writes it to a file.
- `matrix_a.txt` and `matrix_b.txt` are both 1000×1000 matrices with random float values between 0.0 and 5.0.

11 Matrix Multiplication – Plain C++

We begin with a straightforward implementation of matrix multiplication using three nested loops, corresponding to the mathematical definition:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} \cdot B_{kj}$$

This implementation has cubic time complexity $\mathcal{O}(n^3)$ and serves as a baseline for comparison with parallel and SIMD-accelerated methods.

The matrix elements are stored in one-dimensional arrays and accessed using row-major indexing. Matrices A and B are read from previously generated files, and the resulting product C is written back to disk.

- `MultiplyMatrix(...)` performs the multiplication using three nested loops.

Execution time is measured using `std::chrono`, which provides millisecond-level precision for benchmarking. This default implementation does not use any threading or SIMD optimization.

```
void MultiplyMatrix(float* A, float* B, float* C, int N) {
        for (int i = 0; i < N; ++i) {
                for (int j = 0; j < N; ++j) {
                 float sum = 0.0f;
                        for (int k = 0; k < N; ++k) {
                          sum += A[i * N + k] * B[k * N + j];}
                        C[i * N + j] = sum;
          }
        }
  }
```

Listing 5: Classic $O(n^3)$ matrix multiplication in plain C++

Note on memory layout: In this implementation, we deliberately use flat `float*` arrays to store matrices instead of 2D `std::vector<std::vector<float>>` containers. The primary reason for this design choice is performance:

- Flat arrays stored in row-major order ensure continuous memory access, which improves cache locality and enables faster sequential reads.
- Accessing 2D vectors involves an extra layer of indirection and memory fragmentation, which leads to non-contiguous memory and inefficient traversal.
- Using flat arrays also simplifies compatibility with low-level optimizations like SIMD instructions and allows easy control over memory layout during operations like matrix transposition.

For these reasons, a flat memory model is preferred in high-performance computing applications where memory access patterns significantly affect execution time.

12 Matrix Multiplication – Multithreaded (Pthreads)

To improve performance, we parallelized the matrix multiplication using POSIX threads. Instead of computing the entire output matrix sequentially, the workload is divided row-wise across multiple threads. Each thread is responsible for computing a contiguous block of rows of the result matrix.

This approach leverages task-level parallelism on multi-core CPUs, significantly reducing execution time on systems with sufficient cores and memory bandwidth.

- The matrix is split row-wise across `NUM_THREADS`.
- Each thread independently computes its assigned chunk using the same $O(n^3)$ algorithm.
- Synchronization is ensured using `pthread_join()` after all threads finish.

The code below demonstrates this multithreaded matrix multiplication:

```
void* MultiplyPart(void* arg){
        ThreadData* data = (ThreadData*)arg;
         for (int i = data->start_row; i < data->end_row; ++i) {
          for (int j = 0; j < N; ++j) {
                float sum = 0.0f;
                 for (int k = 0; k < N; ++k) {
                        sum += A[i * N + k] * B[k * N + j];}
                                C[i * N + j] = sum;
                                }
                        }
                pthread_exit(nullptr);
                return NULL;
                }
```

Listing 6: Parallel matrix multiplication using POSIX threads

This implementation avoids shared data conflicts since each thread writes to a distinct section of matrix C. However, due to memory bandwidth constraints and cache contention, speedup is not always linear with the number of threads.

13 Matrix Multiplication – Multithreaded with AVX

To maximize computational performance, we implemented a hybrid approach that combines multithreading (via `pthreads`) with SIMD vectorization (using AVX intrinsics). The goal is to exploit both thread-level and data-level parallelism on modern CPUs.

Each thread is assigned a contiguous set of rows from the result matrix C, similar to the previous multithreaded implementation. However, instead of computing each dot product scalar-wise, we use AVX intrinsics to compute eight multiplications in parallel.

Key optimizations:

- **Transpose of matrix** B: Since AVX loads memory contiguously in row-major order, we transpose matrix B into B^T so that its columns become rows, ensuring cache-friendly access when traversing.
- **AVX intrinsic usage**: The dot product between row i of A and column j of B is computed with `_mm256_mul_ps` and `_mm256_add_ps`, which operate on 256-bit vectors (8 `float` values at once).
- **Thread-level decomposition**: The matrix is partitioned by rows, where each thread processes a different segment of the output matrix. Threads write independently to different parts of C, avoiding write conflicts.

This combined method significantly reduces runtime for large matrices, especially on CPUs with high core count and wide vector registers.

```
for (int i = start; i < end; ++i) {
 for (int j = 0; j < N; ++j) {
        __m256 sum = _mm256_setzero_ps();
        int k = 0;

   for (; k + 7 < N; k += 8) {
         __m256 a = _mm256_loadu_ps(&A[i * N + k]);
         __m256 b = _mm256_loadu_ps(&B_T[j * N + k]);
     sum = _mm256_add_ps(sum, _mm256_mul_ps(a, b));
                                }

        float temp[8];
        _mm256_storeu_ps(temp, sum);
        float total = temp[0] + temp[1] + ... + temp[7];

        for (; k < N; ++k) {
        total += A[i * N + k] * B_T[j * N + k];}

        C[i * N + j] = total;
 }
}
```

Listing 7: AVX + pthreads matrix multiplication (inner loop)

Although AVX greatly accelerates the inner loop computation, the benefits depend on data alignment, memory bandwidth, and cache behavior. Transposing matrix B is essential to ensure that SIMD instructions can be efficiently executed without incurring memory access penalties.

14 Performance Comparison

To evaluate the effectiveness of different optimization strategies, we conducted a series of experiments on both vector and matrix operations under various hardware configurations. Results include single-threaded (naive) versions, multithreaded implementations, and SIMD-accelerated (SSE/AVX) approaches.

Vector Dot Product Results

Configuration	**Time (ms)**
Naive (single thread, all cores enabled)	0.0102
2 Threads (no SSE)	69.9456
2 Threads + SSE	59.0385
Naive (single thread, P-cores only)	0.0057
2 Threads (P-cores only, no SSE)	35.963
2 Threads + SSE (P-cores only)	36.8955

Matrix Multiplication Results (1000×1000)

Configuration	**Time (ms)**
Naive (single thread, E+P cores)	1643.18
4 Threads	1594.72
4 Threads + AVX	1595.12
Naive (P-cores only)	1657.07
4 Threads (P-cores only)	4362.00
4 Threads + AVX (P-cores only)	4306.77

Matrix Multiplication – Larger Dataset

Configuration	**Time (ms)**
4 Threads (E+P cores)	6617.05
4 Threads + AVX (E+P cores)	25772.7

15 Results and Observations

- Default (no thread): 1600 ms
- Threading (4 threads): 6000 ms (overhead due to thread creation and memory contention)
- AVX + Threading (20 threads): initially 25000 ms, later improved to 4000 ms after core affinity tuning
- Best performance: **Single-threaded AVX offload with E-cores disabled**

16 System Specs

- Laptop: ASUS ROG Zephyrus M16
- CPU: Intel Core i9-12900H (6P+8E cores, 20 threads total)
- OS: Windows 11
- Compilation: `g++ -O2 -mavx -pthread main.cpp -o program`

Conclusion

We conclude that while multithreading can help scale workloads, optimal AVX usage requires careful core management. Disabling efficiency cores improved AVX threading significantly.