Parallel & Distributed Computing

Assignment 2

Anas Bin Rashid
22I-0907
CS-6A

Question 01

SIMD Optimization for Matrix Transposition and Element-wise Multiplication

Introduction

This report presents an in-depth analysis of optimizing matrix transposition with element-wise multiplication using Single Instruction, Multiple Data (SIMD) operations via AVX intrinsics. The optimization was carried out on Ubuntu running in a Virtual Machine (VM), and the performance improvements achieved with SIMD were measured against scalar implementations.

The task involves computing the transpose of a matrix A and then performing element-wise multiplication with matrix B, storing the result in matrix C:

$$C=A^T \times B$$

Three implementations were tested:

1. **Scalar 2D Implementation**: Using nested loops for row-major access.
2. **Scalar 1D Implementation**: Using a 1D array to simulate a 2D structure for better memory access patterns.
3. **SIMD Implementation**: Using **AVX instructions** to speed up both transposition and multiplication.

Dynamic memory allocation was used to handle large matrices and avoid stack overflow issues.

Problem Statement

Given two square matrices A and B of size N×N, we need to:

1. **Compute the transpose** of matrix A, storing it in A_T.
2. **Perform element-wise multiplication** between A_T and B, storing the result in C.
3. **Compare execution times** of scalar and SIMD implementations for different values of N.

This problem is computationally expensive for large N due to poor cache locality in transposition and the large number of floating-point operations. Using SIMD optimizations can significantly improve performance.

Thought Process and Approach

Initial Considerations

When approaching this problem, my **primary concerns** were:

1. **Handling large matrix sizes efficiently**:
   - Statically allocated large matrices cause stack overflow, so dynamic memory allocation was necessary.
2. **Optimizing memory access patterns**:
   - Transposing a matrix breaks row-major ordering, leading to cache inefficiency.
3. **Leveraging SIMD for parallel processing**:
   - SIMD processes multiple elements simultaneously, reducing loop iterations and improving throughput.

Using Dynamic Memory Allocation

Using large static arrays, such as float A[1024][1024], exceeds the stack memory limit (usually 8MB on Linux).
Instead, I used heap allocation (malloc) to store large matrices in contiguous memory, preventing stack overflows and improving cache efficiency.

Implementing Different Approaches

To validate performance gains, I implemented three versions of the operation:

1. Scalar 2D Implementation (Baseline Approach)

- Uses a nested loop to transpose the matrix and perform element-wise multiplication.
- Works well but poor cache locality slows down performance.
- 2D array indexing (A[i][j]) results in frequent cache misses.

2. Scalar 1D Implementation (Optimized Scalar)

- Uses a single contiguous block of memory for better cache utilization.
- The matrix is stored as a 1D array (A[i*N + j] instead of A[i][j]).
- Results in fewer cache misses, improving performance slightly.

3. SIMD Implementation (Optimized with AVX)

- Uses AVX intrinsics to process 8 floating-point values at a time (__m256).
- Transposition and multiplication are fully vectorized using _mm256_loadu_ps() and _mm256_storeu_ps().
- Unrolled loops further reduce overhead.

Execution Results

```
N = 256
Scalar 2D time: 0.000418 seconds
Scalar 1D time: 0.000545 seconds
SIMD time: 0.000105 seconds
```

Figure 1 N = 256

```
N = 512
Scalar 2D time: 0.002112 seconds
Scalar 1D time: 0.002324 seconds
SIMD time: 0.001127 seconds
```

Figure 2 N = 512

```
N = 1024
Scalar 2D time: 0.011155 seconds
Scalar 1D time: 0.011656 seconds
SIMD time: 0.003719 seconds
```

Figure 3 N = 1024

```
N = 2048
Scalar 2D time: 0.080564 seconds
Scalar 1D time: 0.085311 seconds
SIMD time: 0.014642 seconds
```

Figure 4 N = 2048

Performance Analysis

The following table shows execution times (in seconds) for different values of N, comparing Scalar 2D, Scalar 1D, and SIMD implementations:

Matrix Size (N × N)	Scalar 2D Time (s)	Scalar 1D Time (s)	SIMD Time (s)	Speedup (Scalar 2D / SIMD)
256 × 256	0.000517	0.000622	0.000172	3.00
512 × 512	0.002585	0.002046	0.000987	2.61
1024 × 1024	0.012183	0.014570	0.003914	3.11
2048 × 2048	0.057914	0.069366	0.014464	4.00

Observation

SIMD consistently outperforms scalar implementations, with speedups ranging from 3.00x to 4.00x.

Challenges and Solutions

Stack Overflow for Large Matrices

- Issue: Storing large matrices statically on the stack led to segmentation faults.
- Solution: Used heap allocation (malloc) instead of stack allocation.

SIMD Transposition Incorrect Output

- Issue: Early versions of the SIMD transposition stored incorrect values.
- Solution: Fixed memory alignment issues and ensured correct 1D indexing (A_T[j * N + i]).