Optimised Matrix Multiplication

For 12th Gen Intel® Core™ i3-1215U × 8

By: Shubhajeet Das (24AI10013)

Introduction & Motivation

Matrix multiplication is a fundamental operation used in numerous fields such as scientific computing, graphics, and machine learning. However, the naive approach suffers from performance issues due to high computational complexity.

The repository aims to explore **optimization techniques** that reduce runtime and improve cache performance, addressing challenges in real-world applications.

Understanding Matrix Multiplication

Matrix multiplication involves taking two matrices, A (of size m*n) and B (of size n*p), and producing a resulting matrix C (of size m*p). Each element C[i][j] is computed as the dot product of the i-th row of A and the j-th column of B.

A naive matrix multiplication algorithm has a time complexity of **O(n^3)**, making it inefficient for large matrices.

Naive vs. Optimized Matrix Multiplication

The basic algorithm uses three nested loops to compute each element of the result matrix. It is straightforward but suffers from inefficient memory access patterns.

Example pseudocode:

```
for i in 0..n-1:
  for j in 0..n-1:
    for k in 0..n-1:
      C[i][j] += A[i][k] * B[k][j]
```

The optimized algorithm introduces techniques like **loop unrolling** and **cache blocking (tiling)** to improve performance.

It restructures the computation to maximize data reuse and reduce cache misses.

Optimization Technique 1: Loop Reordering/Blocking

Cache locality: Data access patterns significantly impact performance.

Loop reordering: Changing the order of loops (e.g., jik instead of ijk) can improve data locality.

Loop blocking (tiling): Dividing matrices into smaller blocks that fit into the cache.

Optimization Technique 2: Leveraging SIMD Instructions

SIMD (Single Instruction, Multiple Data) instructions allow parallel operations on multiple data elements.

Examples: AVX, SSE.

Code snippet (C++ using AVX intrinsics):

```
__m256 a, b, c;
// Load and multiply using AVX
c = _mm256_add_ps(c, _mm256_mul_ps(a, b));
```

```
Performance gains: Significant speedup by processing multiple
elements simultaneously.
```

Optimization Technique 3: Parallelization with Multithreading

Multithreading: Distributing the workload across multiple CPU cores.

Libraries: OpenMP, pthreads.

Code snippet (C++ with OpenMP):

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    // ... matrix multiplication code ...
}
```

```
Challenges: Synchronization, load balancing.
```

Requirements

- C++ compiler with AVX2 and OpenMP support (gcc/g++ recommended)
- Python 3.6+
- NumPy

Installation

- Clone this repository: git clone https://github.com/Shubhajeetgithub/optimized-matmul.git cd optimized-matmul
- Compile the C++ code: g++ -O3 -march=alderlake -mavx2 -ffast-math -fopenmp -shared -fPIC matmul.cpp -o matmul.so

Usage

```python
import numpy as np
from matmul_wrapper import matmul, benchmark
# Create example matrices
A = np.random.random((1000, 800)).astype(np.float32)
B = np.random.random((800, 1200)).astype(np.float32)
# Multiply using our optimized function (default implementation is 'avx')
C = matmul(A, B)
# Try different implementations
C_transposed = matmul(A, B, implementation='transposed')
C_tiled = matmul(A, B, implementation='tiled')
# Run benchmarks to find the fastest implementation
benchmark()
```

Thank you