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.

Linear Regression $\boxed{Y = X\beta + \epsilon}$ $\boxed{\beta = (X^T X)^{-1} X^T Y}$

Gradient descent $\boxed{\beta - = \lambda \frac{\partial J}{\partial \beta} = \frac{\lambda}{n} X^T (X\beta - Y)}$

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 **vectorization**, **parallelization** and **cache blocking (tiling)** to improve performance.

Optimization Technique 1: Parallelization with Multithreading

Multithreading: Distributing the workload across multiple CPU cores.

Libraries: OpenMP.

Sets the number of threads to be used for subsequent parallel regions to 8

Instructs the compiler to parallelize the immediately following for loop by distributing its iterations across the available OpenMP threads.

```
#include <omp.h>

omp_set_num_threads(8);
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    // ... matrix multiplication code ...
}
```

Optimization Technique 2: Vectorisation

provides access to Intel's intrinsic functions for various SIMD (Single Instruction Multiple Data) instruction sets

```
#include <immintrin.h>

for (int l = 0; l < k; l++) {

           __m256 a_val = _mm256_set1_ps(A[i * k + l]);

           __m256 b_vals = _mm256_loadu_ps(&B[l * n + j]);

           __m256 c_vals = _mm256_loadu_ps(&C[i * n + j]);

           c_vals = _mm256_fmadd_ps(a_val, b_vals, c_vals);

           _mm256_storeu_ps(&C[i * n + j], c_vals);

}
```

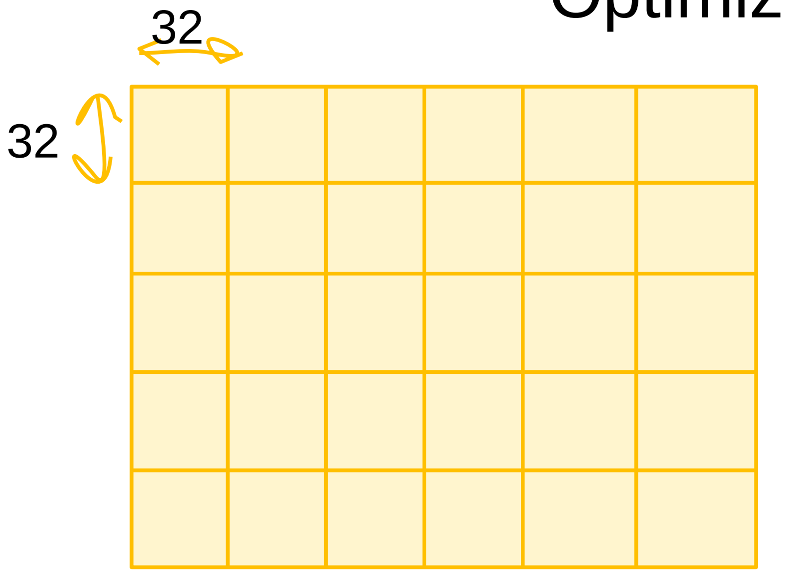
loads a single element A[i][l] and duplicates it into all 8 lanes of the __m256 vector a_val

loads 8 consecutive elements from matrix B (starting at B[l][j]) into the __m256 vector b_vals

performs a fused multiply-add operation: c_vals = (a_val * b_vals) + c_vals

Performance gains: Significant speedup by processing multiple elements simultaneously.

Optimization Technique 3: Tilling



Cache: sites between CPU and RAM that stores recently used instructions and data.

Tilling enhances cache utilization by dividing the matrices into smaller tiles that fit within the processor's cache, maximizing data reuse and minimizing memory access latency.

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

alderlake is the codename for Intel's 12th generation processors, including the Core i3-1215U

-shared is a linker flag that tells the linker to create a shared library (a .so file on Linux) instead of an executable. Shared libraries can be loaded at runtime by other programs. -fPIC stands for "Position Independent Code". It ensures that the generated code can be loaded at any memory address without requiring modifications

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