Parallel and Block-Optimized Matrix Multiplication using OpenMP

Bhaskar

November 12, 2024

Abstract

This report discusses parallel matrix multiplication using OpenMP. Two approaches were implemented: a straightforward parallelization of matrix multiplication and an optimized block-based matrix multiplication. The performance of both methods was evaluated by comparing execution times for large matrices of size 2048x2048. The results demonstrated a significant improvement in performance with the block-optimized approach, achieving a time reduction from 78.56 seconds to 42.25 seconds. These findings highlight the effectiveness of parallel processing and cache optimization in computationally intensive tasks.

1 Introduction

Matrix multiplication is a fundamental operation in numerous scientific, engineering, and data-intensive applications. However, multiplying large matrices is computationally expensive, requiring optimizations to execute within reasonable time frames. This report explores two optimization strategies for matrix multiplication: parallelization with OpenMP and block-based cache optimization.

Matrix multiplication has a time complexity of $O(N^3)$ for two matrices of size $N \times N$. For applications in machine learning, physics simulations, and large-scale data analysis, these computations are essential but resource-intensive. Optimizations like parallel processing and cache management are necessary to make these computations feasible.

2 Background and Motivation

The large computational demand for matrix multiplication has made it a target for optimization in high-performance computing. Traditional matrix multiplication requires each element to be multiplied and accumulated, leading to high processing time and memory access requirements. This report focuses on OpenMP-based parallelism, aiming to harness multiple CPU cores to perform simultaneous computations. Additionally, a block-based matrix multiplication method is employed to improve memory access efficiency and cache usage.

The high-level goals for this project include:

- Reducing computation time using parallel processing.
- Improving memory access patterns to leverage CPU cache effectively.
- Comparing the performance of a basic parallel method and an optimized block-based method.

3 Project Objectives

The objectives are divided into two tasks:

- **Task 1:** Implement a straightforward parallel matrix multiplication using OpenMP and evaluate its performance.
- **Task 2:** Implement a block-based matrix multiplication approach using OpenMP to improve cache efficiency.

4 Methodology

4.1 Overview of OpenMP

OpenMP is an API for parallel programming in C, C++, and Fortran, allowing developers to add parallelism to their applications with minimal code changes. It uses a fork-join model where a master thread forks a team of threads to execute code concurrently, effectively utilizing multiple CPU cores. This project employs OpenMP's parallel for loops for efficient workload distribution.

4.2 Task 1: Parallel Matrix Multiplication

In Task 1, matrix multiplication was implemented by creating threads for each row in the result matrix. The program divides rows across threads, reducing computation time by utilizing all available CPU cores.

4.2.1 Algorithm for Task 1

For two matrices A and B, the element $C[i][j]$ in the result matrix C is calculated as:

$$C[i][j] = \sum_{k=0}^{N-1} A[i][k] \cdot B[k][j]$$

Each thread performs these calculations independently for assigned rows, using OpenMP directives.

4.2.2 Code Implementation for Task 1

Listing 1: Task 1: Parallel Matrix Multiplication Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#define N 2048

double A[N][N];
double B[N][N];
double C[N][N];

void initializeMatrix() {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
            C[i][j] = 0.0;
        }
    }
}

void matrixMultiply() {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    initializeMatrix();
    double start = omp_get_wtime();
    matrixMultiply();
    double end = omp_get_wtime();
    printf("Time taken: %f seconds\n", end - start);
    return 0;
}
```

4.3 Task 2: Block-Optimized Matrix Multiplication

In Task 2, a block-based approach was used to improve cache utilization. By dividing matrices into smaller blocks, each block can fit within the CPU cache, minimizing cache misses and reducing memory access latency. The block size used in this project is 64x64.

4.3.1 Algorithm for Task 2

For matrix sizes that do not divide evenly by the block size, padding or dynamic adjustment may be needed. In this implementation, blocks are processed iteratively, calculating partial results for submatrices of C.

4.3.2 Code Implementation for Task 2

Listing 2: Task 2: Block-Optimized Matrix Multiplication Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N 2048
#define BLOCK_SIZE 64

double A[N][N];
double B[N][N];
double C[N][N];

void initializeMatrix() {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
            C[i][j] = 0.0;
        }
    }
}

void blockMatrixMultiply() {
    #pragma omp parallel for
    for (int i = 0; i < N; i += BLOCK_SIZE) {
        for (int j = 0; j < N; j += BLOCK_SIZE) {
            for (int k = 0; k < N; k += BLOCK_SIZE) {
                for (int ii = i; ii < i + BLOCK_SIZE; ii++) {
                    for (int jj = j; jj < j + BLOCK_SIZE; jj++) {
                        for (int kk = k; kk < k + BLOCK_SIZE; kk++) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    initializeMatrix();
    double start = omp_get_wtime();
    blockMatrixMultiply();
    double end = omp_get_wtime();
    printf("Time taken: %f seconds\n", end - start);
    return 0;
}
```

5 Results and Performance Analysis

5.1 Task 1 Results

For Task 1, the execution time for parallel matrix multiplication was recorded as 78.56 seconds. This reflects a significant improvement compared to sequential execution, thanks to OpenMP's parallelism.

5.2 Task 2 Results

For Task 2, the execution time using block-optimized matrix multiplication was 42.25 seconds, approximately 46% faster than Task 1. This demonstrates the impact of cache optimization in reducing memory access latency.

6 Discussion

The observed performance gains in Task 2 are due to reduced cache misses. By accessing smaller matrix blocks, the algorithm takes advantage of spatial locality, a key optimization principle. Future improvements could involve experimenting with various block sizes or even hybrid models that combine parallelism with GPU acceleration.

7 Conclusion

This project demonstrates that both parallelization and block-based optimization are effective methods for improving matrix multiplication performance. Task 2 achieved significant speed gains by minimizing cache misses through block optimization.

8 Future Work

Future directions include exploring alternative matrix sizes, experimenting with GPU-based acceleration, and further optimizations to handle matrices of non-standard sizes effectively.