Universidad Nacional de San Agustin

Escuela Profesional de Ciencia de la Computación

LABORATORIO

Curso:
Computación Paralela y Distribuida

Docente:
Alvaro Henry Mamani Aliaga

Alumno:
Miguel Angel Deza Cuela

Puedes encontrar mi repositorio en GitHub en el siguiente enlace:
Mi Repositorio en GitHub.

September 24, 2023

1 Introducción

Este informe analiza el rendimiento de la multiplicación de matrices implementada en el código proporcionado. Se comparan dos enfoques: la multiplicación de matrices clásica y la multiplicación de matrices por bloques. El objetivo es evaluar el impacto del tamaño de las matrices y el tamaño del bloque en el rendimiento de la multiplicación de matrices.

2 Código Fuente

A continuación, se muestra el código fuente utilizado en el análisis:

```
#include <iostream>
#include <chrono>
#include <vector>
#include <cstdlib>

using namespace std;
using namespace chrono;

// Funci n para multiplicar matrices cl sicamente (tres bucles anidados)
void multiplyMatricesClassic(int** A, int** B, int** C, int rowsA, int colsA, int colsB) {
    for (int i = 0; i < rowsA; ++i) {
        for (int j = 0; j < colsB; ++j) {
            C[i][j] = 0;
            for (int k = 0; k < colsA; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// Funci n para multiplicar matrices por bloques (seis bucles anidados)
void multiplyMatricesBlocked(int** A, int** B, int** C, int rowsA, int colsA, int colsB, int blockSize) {
    for (int i = 0; i < rowsA; i += blockSize) {
        for (int j = 0; j < colsB; j += blockSize) {
            for (int k = 0; k < colsA; k += blockSize) {
                for (int ii = i; ii < min(i + blockSize, rowsA); ++ii) {
                    for (int jj = j; jj < min(j + blockSize, colsB); ++jj) {
                        for (int kk = k; kk < min(k + blockSize, colsA); ++kk) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    int sizes[] = {100, 200, 500, 1000};  // Tama os de las matrices
    int blockSize = 32;  // Tama o del bloque para la multiplicaci n por bloques

    for (int size : sizes) {
        // Crear matrices A, B y C como matrices din micas de punteros a punteros
        int** A = new int*[size];
        int** B = new int*[size];
```

```
        int** C = new int*[size];
        for (int i = 0; i < size; ++i) {
            A[i] = new int[size];
            B[i] = new int[size];
            C[i] = new int[size];
        }

        // Inicializar matrices A y B con valores aleatorios
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                A[i][j] = rand() % 100;  // N meros aleatorios entre 0 y 99
                B[i][j] = rand() % 100;
            }
        }

        // Medir el tiempo de ejecuci n para la multiplicaci n  cl sica
        auto start_time_classic = high_resolution_clock::now();
        multiplyMatricesClassic(A, B, C, size, size, size);
        auto end_time_classic = high_resolution_clock::now();
        auto duration_classic = duration_cast<milliseconds>(end_time_classic -
            start_time_classic);

        // Medir el tiempo de ejecuci n para la multiplicaci n por bloques
        auto start_time_blocked = high_resolution_clock::now();
        multiplyMatricesBlocked(A, B, C, size, size, size, blockSize);
        auto end_time_blocked = high_resolution_clock::now();
        auto duration_blocked = duration_cast<milliseconds>(end_time_blocked -
            start_time_blocked);

        cout << "Tama o de la matriz: " << size << "x" << size << endl;
        cout << "Tiempo de ejecuci n (Cl sico): " << duration_classic.count()
            << " ms" << endl;
        cout << "Tiempo de ejecuci n (Bloques, bloque de tama o " << blockSize
             << "): " << duration_blocked.count() << " ms" << endl;
        cout << "-----------------------------------------------------" << endl;

        // Liberar memoria de las matrices
        for (int i = 0; i < size; ++i) {
            delete[] A[i];
            delete[] B[i];
            delete[] C[i];
        }
        delete[] A;
        delete[] B;
        delete[] C;
    }

    return 0;
}
```

Listing 1: Código fuente de multiplicación de matrices

3 Metodología

Se realizaron pruebas con matrices de diferentes tamaños (100x100, 200x200, 500x500 y 1000x1000) y se midió el tiempo de ejecución de la multiplicación de matrices clásica y por bloques. Se utilizó la biblioteca de C++ `chrono` para medir el tiempo de ejecución.

4 Resultados

A continuación, se presentan los resultados obtenidos para cada tamaño de matriz y tamaño de bloque:

Tamaño de la Matriz	Tiempo (Clásico) (ms)	Tiempo (Bloques, Tamaño del Bloque 32) (ms)
100x100	16	18
200x200	133	145
500x500	2071	2137
1000x1000	20308	17799

Table 1: Resultados de tiempo de ejecución

5 Análisis

Los resultados muestran que la multiplicación de matrices por bloques tiende a ser más eficiente que la multiplicación de matrices clásica a medida que aumenta el tamaño de las matrices. Esto se debe a la optimización de la memoria caché al dividir las matrices en bloques más pequeños.

5.1 Análisis del Movimiento de Datos

Para comprender mejor por qué la multiplicación de matrices por bloques es más eficiente, analicemos el movimiento de datos entre la memoria principal y la memoria caché en ambos enfoques:

5.1.1 Multiplicación de Matrices Clásica

En la multiplicación de matrices clásica, se accede a los elementos de las matrices A, B y C de manera secuencial en bucles anidados. Esto puede provocar un alto número de fallos de caché, ya que los datos pueden no estar disponibles en la memoria caché cuando se necesitan debido a la falta de localidad espacial.

5.1.2 Multiplicación de Matrices por Bloques

En la multiplicación de matrices por bloques, se dividen las matrices en bloques más pequeños y se procesan de manera iterativa. Esto mejora la localidad espacial, ya que los datos en un bloque son más propensos a estar en la memoria caché cuando se accede a ellos. El bucle más interno que opera en los bloques tiene un menor número de iteraciones, lo que reduce aún más los fallos de caché.

5.2 Complejidad Algorítmica

Ambos enfoques tienen una complejidad algorítmica de $O(N^3)$, donde N es el tamaño de la matriz. Sin embargo, la multiplicación de matrices por bloques tiende a tener un mejor rendimiento en la práctica debido a una mejor gestión de la memoria caché.

6 Resultados de Cache Misses

A continuación, se presentan los resultados obtenidos del archivo "callgrind.out.5541", que contiene información sobre los "cache misses" durante la ejecución de los algoritmos:

```
Descarga completa de resultados de "callgrind.out.5541"...
(Aquí se muestra una parte de los resultados debido a su extensión.)

Total de Cache Misses: 18612404250
```

Los resultados indican que durante la ejecución de los algoritmos, se produjeron un total de 18,612,404,250 "cache misses". Estos fallos de caché pueden afectar significativamente el rendimiento de la aplicación y son una métrica importante para evaluar la eficiencia de los algoritmos en términos de acceso a la memoria caché.

7 Análisis

El alto número de "cache misses" observados durante la ejecución de los algoritmos sugiere que se debe prestar atención a la gestión de la memoria caché en la implementación. Los "cache misses" ocurren cuando los datos necesarios no se encuentran en la memoria caché y deben recuperarse desde la memoria principal, lo que implica un mayor tiempo de acceso.

Para mejorar el rendimiento y reducir los "cache misses", se pueden considerar las siguientes estrategias:

- **Optimización de la localidad espacial**: Reorganizar el acceso a los datos en la memoria de manera que los datos utilizados juntos estén cerca en memoria para aprovechar la localidad espacial.
- **Optimización del tamaño del bloque**: Evaluar y ajustar el tamaño del bloque utilizado en la multiplicación de matrices por bloques para maximizar la eficiencia de la memoria caché.
- **Técnicas de pre-carga (prefetching)**: Utilizar técnicas de pre-carga de datos para anticipar las necesidades de datos futuros y traerlos a la memoria caché de manera anticipada.
- **Optimización de bucles**: Revisar la estructura de bucles en el código para minimizar los accesos innecesarios a la memoria y maximizar la reutilización de datos en caché.

8 Conclusiones

Basado en los resultados, se puede concluir que la multiplicación de matrices por bloques es una técnica eficiente para matrices grandes. Sin embargo, el tamaño del bloque también puede influir en el rendimiento, por lo que es importante ajustarlo adecuadamente según el problema y la arquitectura del sistema.

Los resultados de "cache misses" obtenidos a través del análisis de Valgrind y KCacheGrind proporcionan una valiosa información sobre la eficiencia de la gestión de la memoria caché en los algoritmos de multiplicación de matrices. Para mejorar el rendimiento de los algoritmos y reducir los "cache misses", se deben aplicar estrategias de optimización de la gestión de la memoria y la localidad espacial.

El análisis de "cache misses" es esencial para identificar áreas críticas de optimización en el código y garantizar un rendimiento eficiente en aplicaciones que realizan operaciones intensivas en matrices.