Multiplicación de Matrices y Análisis de Rendimiento

Autor
Massiel Oviedo Sivincha
Tutores
Alvaro Henry Mamani Aliaga

Universidad Católica San Pablo
Computación Paralela y Distribuida Arequipa, Perú
Agosto del 2023

1. Introducción

En este análisis, nos centraremos en la multiplicación de matrices cuadradas utilizando dos enfoques diferentes: la multiplicación simple y la multiplicación por bloques. Además, vamos a explorar cómo varía el rendimiento de estos enfoques cuando se aplican a matrices de diferentes tamaños. Para nuestro análisis, utilizaremos un entorno de desarrollo en WSL (Windows Subsystem for Linux) con Ubuntu. El código puede ser encontrado en el siguiente link Multiply-Matrix.git.

2. Multiplicación de Matrices Clásica

Durante la multiplicación de matrices, se accede a los elementos de cada matriz. Este proceso implica leer los valores de las matrices A-B y los valores resultantes se almacenan en la memoria asignada para la matriz resultado. La memoria se actualiza con los nuevos valores calculados.
Durante la multiplicación de matrices, el procesador intentará acceder a la memoria caché para buscar los datos necesarios. Si los datos de las matrices están en la caché, el acceso será más rápido. Si no están en la caché, se traerán desde la memoria principal. El rendimiento de la multiplicación de matrices puede verse afectado por el movimiento de datos entre la memoria principal y la memoria caché. Si los datos de las matrices no se ajustan completamente en la caché, puede haber más transferencias entre la memoria principal y la caché, lo que puede ralentizar la operación.

```
#define MAX 4

void simple_multiply(int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX
    ]) {
  for (int i = 0; i < MAX; i ++) {
    for (int j = 0; j < MAX; j ++) {
      for (int k = 0; k < MAX; k ++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

Listing 1: Simple Multiplication Algorithm C++

2.1. Evaluación de Rendimiento

En este enfoque, multiplicamos directamente los elementos de las matrices de entrada sin ningún tipo de subdivisión. Este método es el más simple pero puede no ser el más eficiente para matrices grandes.

2.1.1. Tiempo de Ejecución

Para el analisis de tiempo de ejecucion de la multiplicacion simple se toma en cuenta los siguientes datos; la columna Size indica el tamaño de las matrices cuadradas que se están multiplicando. En este caso, se tienen matrices de tamaños 200x200, 400x400, 560x560 y 800x800. Este dato es esencial para comprender cómo afecta el rendimiento a medida que aumenta el tamaño de las matrices.

MAX	Excute Time (s)
200	1,2772
400	9,2151
560	20,1692
800	139,1370

Tabla 1: Tiempo de compilación en segundos.

2.1.2. Análisis de Complejidad

El primer bucle for itera sobre la variable i desde 0 hasta MAX - 1, lo que significa que se ejecutará MAX veces.
El segundo bucle for itera sobre la variable j desde 0 hasta MAX - 1, nuevamente ejecutándose MAX veces.
El tercer bucle for itera sobre la variable k desde 0 hasta MAX - 1, ejecutándose MAX veces.
Dentro de los bucles anidados, se realiza una operación constante: la multiplicación de dos elementos de las matrices A y B y la acumulación en la matriz C. Esto es una operación de tiempo constante O(1).
Por lo tanto, la complejidad de esta función Simple Multiply es (MAX^3), lo que significa que el tiempo de ejecución aumenta cúbicamente en función del tamaño de entrada

MAX.

3. Multiplicación por Bloques de Matrices

La multiplicación de matrices por bloques es una técnica de optimización que busca utilizar la memoria y la caché de manera más eficiente, reduciendo la latencia de acceso a la memoria principal y mejorando el rendimiento en comparación con enfoques más simples. Durante la ejecución de este algoritmo, se producen patrones de acceso a la memoria que están diseñados para aprovechar al máximo la jerarquía de caché de la CPU y minimizar las transferencias de datos entre la memoria principal y el caché.

```
#define MAX 4
#define BLOCK_SIZE 2

void block_multiply(int A[MAX][MAX], int B[MAX][MAX], int C[MAX][MAX])
    {
  //READ INTO FAST MEMORY
  for (int i = 0; i < MAX; i += BLOCK_SIZE) {
    for (int j = 0; j < MAX; j += BLOCK_SIZE) {
      for (int k = 0; k < MAX; k += BLOCK_SIZE) {
        // BLOCK MULTIPLICATION
        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];
            }
          }
        }
      }
    }
  }
}
```

3.1. Evaluación de Rendimiento

En este enfoque, dividimos las matrices en bloques más pequeños y realizamos la multiplicación en bloques. Luego, combinamos los resultados de los bloques para obtener la matriz de salida. La eficiencia de este enfoque puede variar según el tamaño de los bloques utilizados. En nuestro análisis, consideraremos tamaños de bloque del 12.5 %, 25 %, 50 %, y 100 % del tamaño total de las matrices.

3.1.1. Tiempo de Ejecución

Para el analisis , se observa cómo los tiempos de ejecución varían en las columnas correspondientes a los diferentes porcentajes de tamaño de bloque (12.50 %, 25 %, 50 %, 100 %). Esto permitirá identificar cuál de los tamaños de bloque que proporciona un mejor rendimiento para matrices de diferentes tamaños.

MAX	12,50 %	25 %	50 %	100 %
200	1,33829	1,33008	1,6137	1,3806
400	12,1687	12,3978	12,9702	13,2677
560	26,3410	24,0193	27,9917	30,9951
800	86,2412	89,4897	95,0549	95,1320

Tabla 2: Tiempo de compilación en segundos.

3.1.2. Análisis de Complejidad

En el primer bucle externo for se ejecutará un total de MAX / BLOCKSIZE veces. Dado que BLOCKSIZE se establece en 2, este bucle se ejecutará 2 veces para una matriz de tamaño 4x4. Por lo tanto, la complejidad de este bucle es $O(MAX/BLOCKSIZE)$.
El segundo bucle externo for es similar al primero y también se ejecutará un total de MAX / BLOCKSIZE veces. Su complejidad también es $O(MAX/BLOCKSIZE)$.
El tercer bucle externo for es nuevamente similar a los dos primeros y se ejecutará MAX / BLOCKSIZE veces. Su complejidad es $O(MAX/BLOCKSIZE)$.

Dentro de estos bucles externos, tenemos tres bucles anidados más:

El primer bucle interno for se ejecutará BLOCKSIZE veces. Dado que BLOCKSIZE es 2 en este caso, la complejidad de este bucle es $O(BLOCKSIZE)$, que es constante. El segundo bucle interno for también se ejecutará BLOCKSIZE veces, lo que significa que su complejidad es $O(BLOCKSIZE)$.
El tercer bucle más interno for también se ejecutará BLOCKSIZE veces, por lo que su complejidad es $O(BLOCKSIZE)$.

Dentro del bucle más interno, se realiza una multiplicación y una suma en la matriz C, lo que tiene una complejidad constante O(1).

En resumen, la complejidad de Block Multiply se puede expresar como:
$O((MAX/BLOCKSIZE)^3 * BLOCKSIZE^2)$

4. Evaluación del Movimiento de Datos y Uso de la Memoria Caché

4.1. Configuracón Hardware

Se utilizo un entorno de desarrollo en WSL (Windows Subsystem for Linux) con Ubuntu para llevar a cabo las pruebas de rendimiento de la multiplicación de matrices. Por ello se proporcionará detalles sobre la configuración de hardware en la que se llevaron a cabo las pruebas de rendimiento, el cual cuenta con cache size: 8192 KB y tiene las siguientes especificaciones:

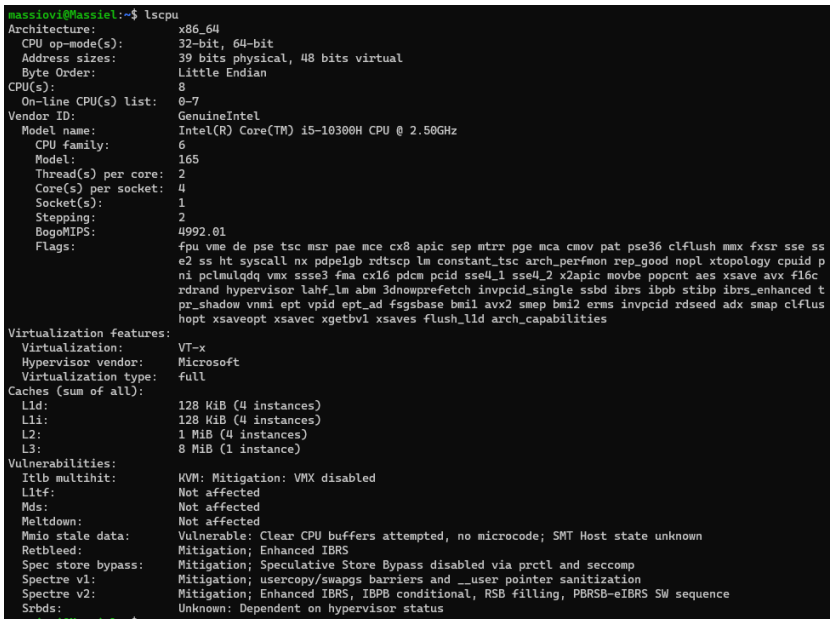

Figura 1: Detalles de Configuración.

Los detalles de la caché son especialmente relevantes para el análisis del rendimiento y el uso de la memoria caché durante la ejecución de los programas. La información proporcionada es específica para esta ejecución y puede variar según la configuración del sistema anfitrión.

4.2. Análisis con Valgrind y KCacheGrind

Explicación detallada de cómo se analiza el movimiento de datos entre la memoria principal y la memoria caché. Resultados de la evaluación de cache misses y su impacto en el rendimiento. Conclusiones basadas en el análisis de datos. Resultados obtenidos de la herramienta KCacheGrind y su interpretación:

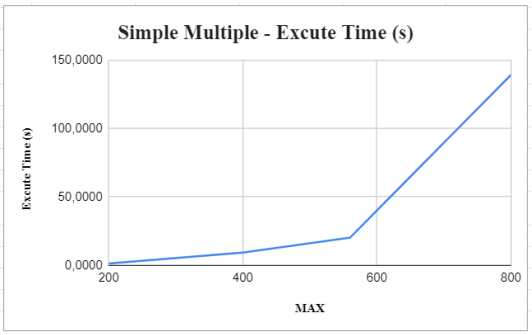

Figura 2: Simple Time Execution Graph.

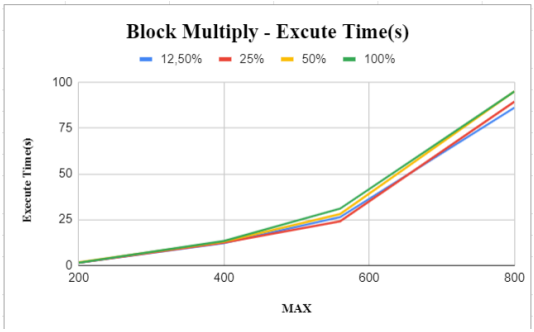

Figura 3: Block Time Execution Graph.

Simple Multiply	
Size	**L1 Miss Sum**
200	95,07000
400	98,8300
560	99,9500
800	99,9700

Figura 4: Simple L1 Sum.

Block Multiply - L1 Miss Sum				
MAX/Block Size %	**12,50%**	**25%**	**50%**	**100%**
200	67,61	59,27	95,61	95,07
400	86,21	98,94	98,84	98,83
560	88,56	99,36	99,32	99,95
800	99,62	99,97	99,60	99,97

Figura 5: Block L1 Sum.

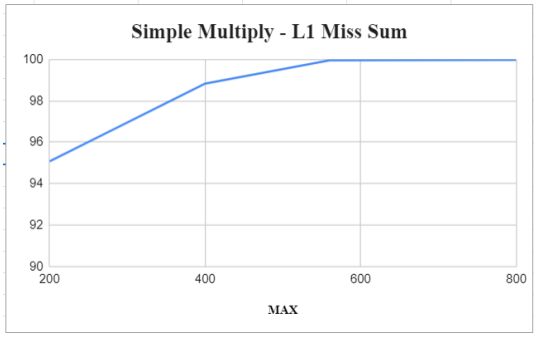

Figura 6: Simple L1 Sum Graph.

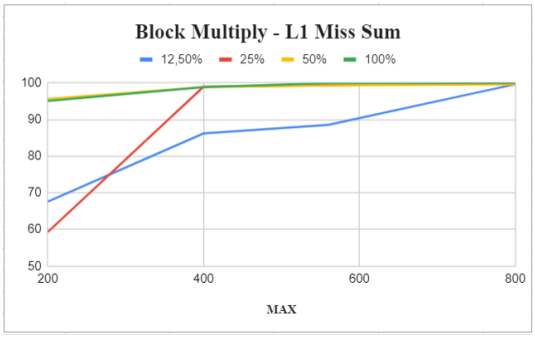

Figura 7: Block L1 Sum Graph.

Simple Multiply	
Size	**Last-Level Sum**
200	0,03000
400	0,0100
560	0,0100
800	0,0000

Figura 8: Simple LL Sum.

Block Multiply - Last-Level Sum				
MAX/Block Size %	**12,50%**	**25%**	**50%**	**100%**
200	0,03	0,03	0,03	0,04
400	0,01	0,01	0,02	0,02
560	0,01	0,01	0,01	0,01
800	0,00	0,01	0,00	0,01

Figura 9: Block LL Sum.

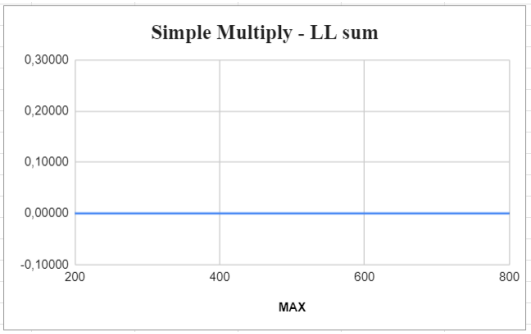

Figura 10: Simple LL Sum Graph.

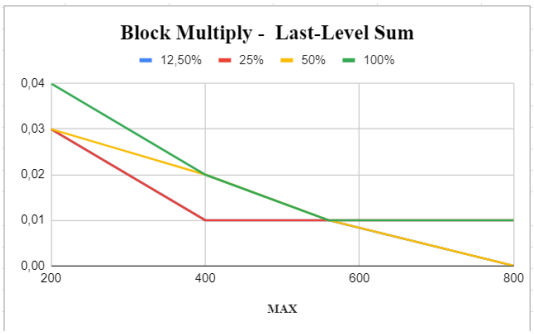

Figura 11: Block LL Sum Graph.

Simple Multiply	
Size	**L1 Instr. Fetch Miss**
200	0,26000
400	0,2600
560	0,2100
800	0,2600
Simple Multiply	
Size	**L1 Data Read Miss**
200	97,33000
400	99,6600
560	99,9900
800	100,0000
Simple Multiply	
Size	**L1 Data Write Miss**
200	0,00000
400	0,0000
560	0,0000
800	0,0000

Figura 12: Feactures of Simple L1 Sum .

Block Multiply - L1 Instr. Fetch Miss				
MAX/Block Size %	**12,50%**	**25%**	**50%**	**100%**
200	0,30	0,30	0,30	0,35
400	0,30	0,30	0,34	0,34
560	0,26	0,26	0,26	0,26
800	0,30	0,34	0,34	0,34
Block Multiply - L1 Data Read Miss				
MAX/Block Size %	**12,50%**	**25%**	**50%**	**100%**
200	79,79	73,35	97,63	97,33
400	95,57	99,69	99,66	99,66
560	97,64	99,88	99,87	99,99
800	99,96	100,00	99,96	100,00
Block Multiply - L1 Data Write Miss				
MAX/Block Size %	**12,50%**	**25%**	**50%**	**100%**
200	0,00	0,00	0,00	0,00
400	0,00	0,00	0,00	0,00
560	0,00	0,00	0,00	0,00
800	0,00	0,00	0,00	0,00

Figura 13: Feactures of Block L1 Sum.

5. Conclusión

- La multiplicación por bloques tiende a ser más eficiente que la multiplicación simple para matrices grandes, ya que puede reducir la latencia de memoria y aprovechar la caché del sistema.
- La elección del tamaño de bloque es crítica. Un tamaño de bloque incorrecto puede llevar a un peor rendimiento.
- Los resultados pueden variar según la arquitectura de la CPU y otros factores del sistema.
- En aplicaciones de alto rendimiento que involucran multiplicaciones de matrices, la optimización de caché y la elección del algoritmo adecuado son esenciales para lograr un rendimiento óptimo.

En resumen, el análisis de la multiplicación de matrices simple frente a la multiplicación por bloques muestra que la segunda opción puede ser más eficiente para matrices

grandes, especialmente cuando se elige cuidadosamente el tamaño de bloque. Sin embargo, la optimización de caché y el rendimiento pueden variar según la configuración del hardware y deben ajustarse en función de las necesidades específicas de la aplicación.