

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN
COMPUTACIÓN PARALELA Y DISTRIBUIDA



Laboratorio 1

Presentado por:

Merisabel Ruelas Quenaya

Docente :

Alvaro Mamani Aliaga



Laboratorio 01

1. Repositorio

<https://github.com/MrsblR/CP/tree/main/L01>

2. Ejercicio 1

Implementar y comparar los 2-bucles anidados FOR presentados en el cap. 2 del libro, pag 22.

```
#include <chrono>
#include <iostream>
#include <sys/time.h>
#include <ctime>
#include <cstdlib>

using namespace std;

const int MAX = 500;
double A[MAX][MAX], x[MAX], y[MAX];

void init() {
    for(int i = 0; i < MAX ; i++) {
        y[i] = 0;
        x[i] = 1 + rand() % 9; // Genera n meros entre 1 y 9
        for(int j = 0 ; j < MAX ; j++)
            A[i][j] = 1 + rand() % 100; // Genera n meros entre 1 y 100
    }
}

double calcTime(clock_t t0, clock_t t1) {
    return static_cast<double>(t1 - t0) / CLOCKS_PER_SEC;
}

void nestedLoop1() {
    clock_t t0, t1;
    init();
    t0 = clock();
    for (int i = 0; i < MAX; i++)
        for (int j = 0; j < MAX; j++)
            y[i] += A[i][j] * x[j];
    t1 = clock();
    cout << "Tiempo de Ejecuci n del Bucle Anidado 1: " << calcTime(t0,
t1) << " segundos" << endl;
}

void nestedLoop2() {
    clock_t t0, t1;
    init();
```

```
t0 = clock();
for (int j = 0; j < MAX; j++)
    for (int i = 0; i < MAX; i++)
        y[i] += A[i][j] * x[j];
t1 = clock();
cout << "Tiempo de Ejecuci n del Bucle Anidado 2: " << calcTime(t0,
t1) << " segundos" << endl;
}

int main() {
    srand(time(0)); // Semilla para n meros aleatorios
    nestedLoop1();
    nestedLoop2();
    return 0;
}
```

2.1. Comparativa

- Bucle Anidado 1 (por filas): 0.001205 segundos
- Bucle Anidado 2 (por columnas): 0.002199 segundos

El Bucle Anidado 1 es claramente más rápido, tomando aproximadamente la mitad del tiempo que el Bucle Anidado 2.

Ambos bucles muestran cierta variabilidad en sus tiempos de ejecución, lo cual es normal.

El Bucle Anidado 1, que recorre la matriz por filas, es significativamente más eficiente que el Bucle Anidado 2, que la recorre por columnas.

Recorrer matrices por filas es más rápido que por columnas en C++ porque:

- Las matrices se almacenan en *row-major order* (por filas).
- Al recorrer por filas, accedemos a posiciones contiguas en memoria.
- Esto mejora la *localidad espacial* y el uso de la caché del procesador.
- Al recorrer por columnas, los accesos no son contiguos, lo que produce más fallos de caché.

3. Ejercicio 2

Implementar en C/C++ la multiplicación de matrices clásica, la versión de tres bucles anidados y evaluar su desempeño considerando diferentes tamaños de matriz.

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

using namespace std;
using namespace std::chrono;

void classicMatrixMultiplication(vector<vector<int>>& A, vector<vector<
int>>& B, vector<vector<int>>& C, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    vector<int> sizes = {128, 256, 512, 1024};

    for (int N : sizes) {
        vector<vector<int>> A(N, vector<int>(N, 1));
        vector<vector<int>> B(N, vector<int>(N, 2));
        vector<vector<int>> C(N, vector<int>(N, 0));

        cout << "Multiplicacion clasica para tamaño de matriz " << N <<
"x" << N << ":" << endl;

        auto start = high_resolution_clock::now();
        classicMatrixMultiplication(A, B, C, N);
        auto stop = high_resolution_clock::now();

        auto duration = duration_cast<milliseconds>(stop - start);
        cout << "Duracion: " << duration.count() << " ms" << endl <<
endl;
    }

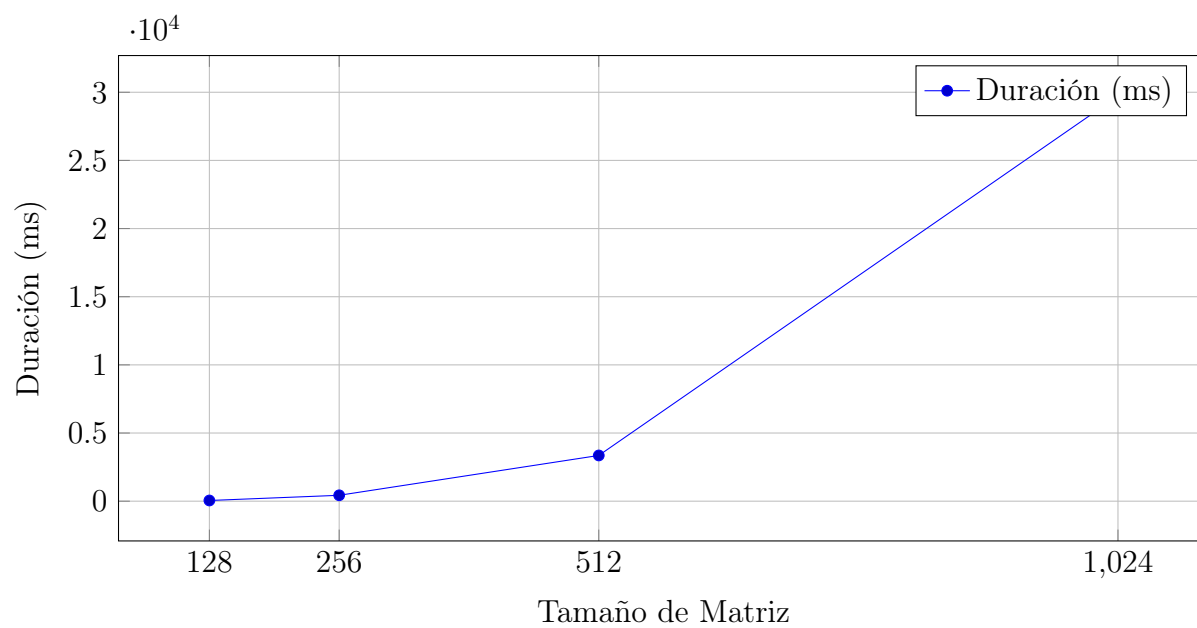
    return 0;
}
```

Resultados de la Multiplicación Clásica de Matrices

Tabla

Tamaño de Matriz	Duración (ms)
128x128	51
256x256	432
512x512	3355
1024x1024	29725

Gráfico



4. Ejercicio 3

Implementar la versión por bloques (investigar en internet), seis bucles anidados, evaluar su desempeño y compararlo con la multiplicación de matrices clásica.

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

using namespace std;
using namespace std::chrono;

void classicMatrixMultiplication(vector<vector<int>>& A, vector<vector<
int>>& B, vector<vector<int>>& C, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

void blockMatrixMultiplication(vector<vector<int>>& A, vector<vector<int>
>& B, vector<vector<int>>& C, int N, int blockSize) {
    for (int ii = 0; ii < N; ii += blockSize) {
        for (int jj = 0; jj < N; jj += blockSize) {
            for (int kk = 0; kk < N; kk += blockSize) {
                // Multiplicar los bloques
                for (int i = ii; i < min(ii + blockSize, N); i++) {
                    for (int j = jj; j < min(jj + blockSize, N); j++) {
                        for (int k = kk; k < min(kk + blockSize, N); k
++)) {
                            C[i][j] += A[i][k] * B[k][j];
                        }
                    }
                }
            }
        }
    }
}

int main() {
    vector<int> sizes = {128, 256, 512, 1024};

    int blockSize = 64;

    for (int N : sizes) {
        cout << "Probando multiplicaci n para tamano de matriz " << N
<< "x" << N << ":" << endl;

        vector<vector<int>> A(N, vector<int>(N, 1));
        vector<vector<int>> B(N, vector<int>(N, 2));
        vector<vector<int>> C(N, vector<int>(N, 0));
```

```

        auto start = high_resolution_clock::now();
        classicMatrixMultiplication(A, B, C, N);
        auto stop = high_resolution_clock::now();
        auto durationClassic = duration_cast<milliseconds>(stop - start)
;
        cout << "Duracion (clasica): " << durationClassic.count() << "
ms" << endl;

        fill(C.begin(), C.end(), vector<int>(N, 0));

        start = high_resolution_clock::now();
        blockMatrixMultiplication(A, B, C, N, blockSize);
        stop = high_resolution_clock::now();
        auto durationBlock = duration_cast<milliseconds>(stop - start);
        cout << "Duracion (por bloques): " << durationBlock.count() << "
ms" << endl << endl;
    }

    return 0;
}

```

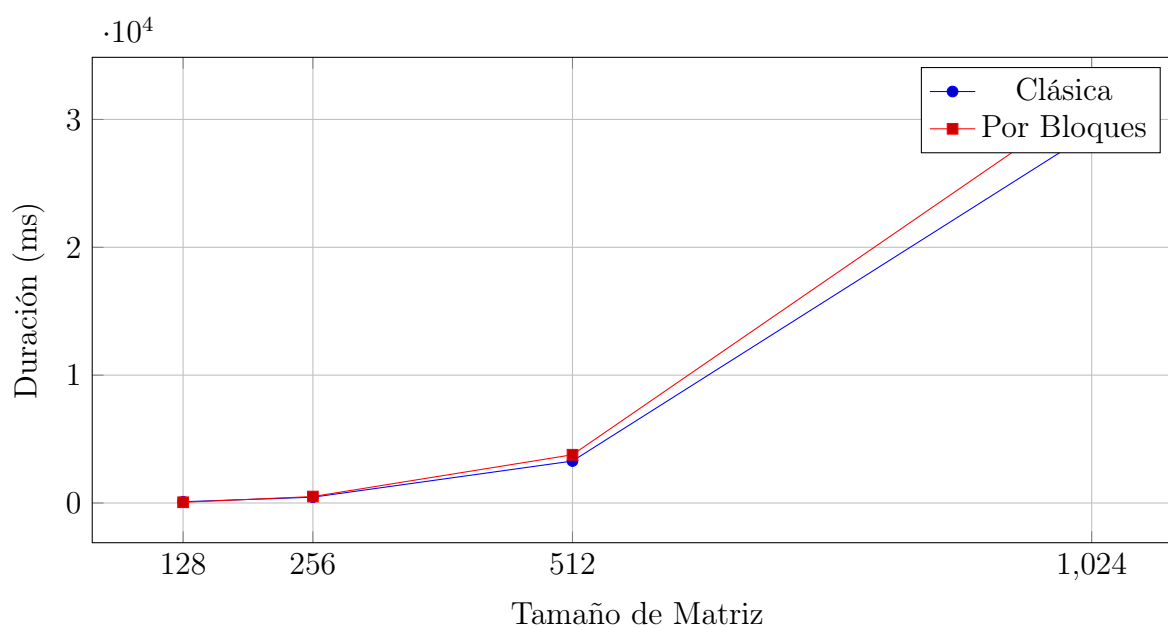
Comparación de Multiplicación Clásica vs. Multiplicación por Bloques

Tabla

Tamaño de Matriz	Duración Clásica (ms)	Duración por Bloques (ms)
128x128	97	57
256x256	453	504
512x512	3281	3774
1024x1024	28986	31695

Cuadro 1: Comparación de duración entre la multiplicación clásica y la multiplicación por bloques.

Gráfico

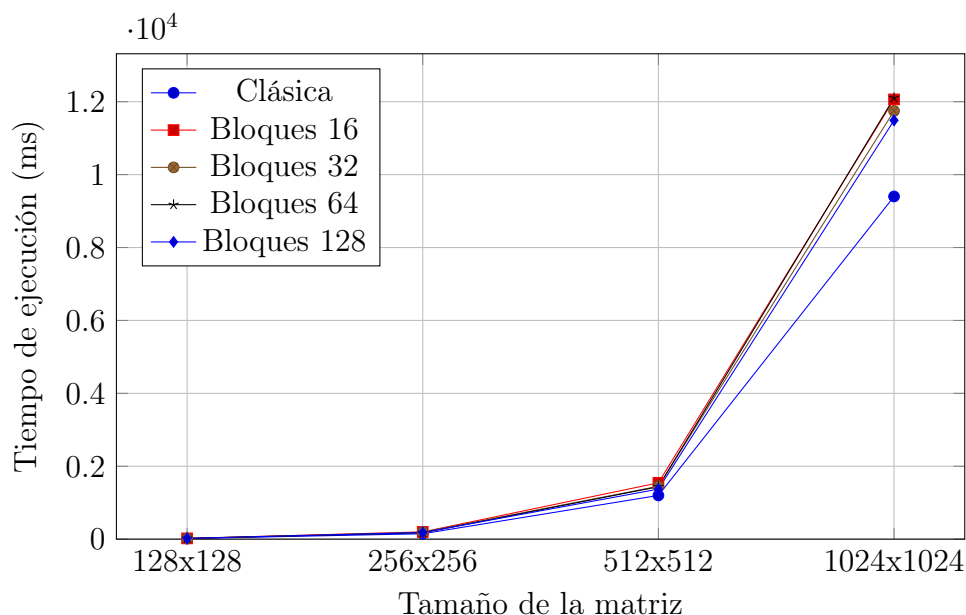


Comparación de Multiplicación Clásica vs. Multiplicación por Bloques (Considerando distintos tamaños de Bloque)

Tamaño Matriz	Clásica(ms)	Bq.16(ms)	Bq.32(ms)	Bq.64(ms)	Bq.128(ms)
128x128	17	21	21	21	20
256x256	148	199	191	173	181
512x512	1199	1545	1450	1437	1373
1024x1024	9401	12063	11747	12105	11488

Cuadro 2: Tiempos de ejecución para multiplicación de matrices clásica y por bloques.

4.1. Gráfico Comparativo



5. Ejercicio 4

Ejecutar ambos algoritmos paso a paso, y analizar el movimiento de datos entre la memoria principal y la memoria cache. Hacer una evaluación de acuerdo a la complejidad algorítmica.

5.1. Multiplicación Clásica

- Accesos a Memoria: Accede a elementos de 'A' y 'B' repetidamente.
- Caché: Acceso secuencial puede causar fallos de caché.
- Complejidad Temporal: $O(N^3)$. Tres bucles anidados, cada uno recorriendo N elementos.
- Complejidad Espacial: $O(N^2)$. Se necesitan matrices 'A', 'B' y 'C', cada una con $N \times N$ elementos.

5.2. Multiplicación por Bloques

- Accesos a Memoria: Menos accesos a memoria principal, mejora el uso de la caché. Cada bloque se multiplica dentro de la caché, lo que reduce el número de accesos a memoria principal.

- Caché: Mejor localización temporal y espacial dentro de bloques. La localización espacial es mejorada porque los datos accesibles en un bloque se mantienen en caché durante más tiempo, reduciendo fallos de caché.
- Complejidad Temporal: $O(N^3)$.
- Complejidad Espacial: $O(N^2)$.

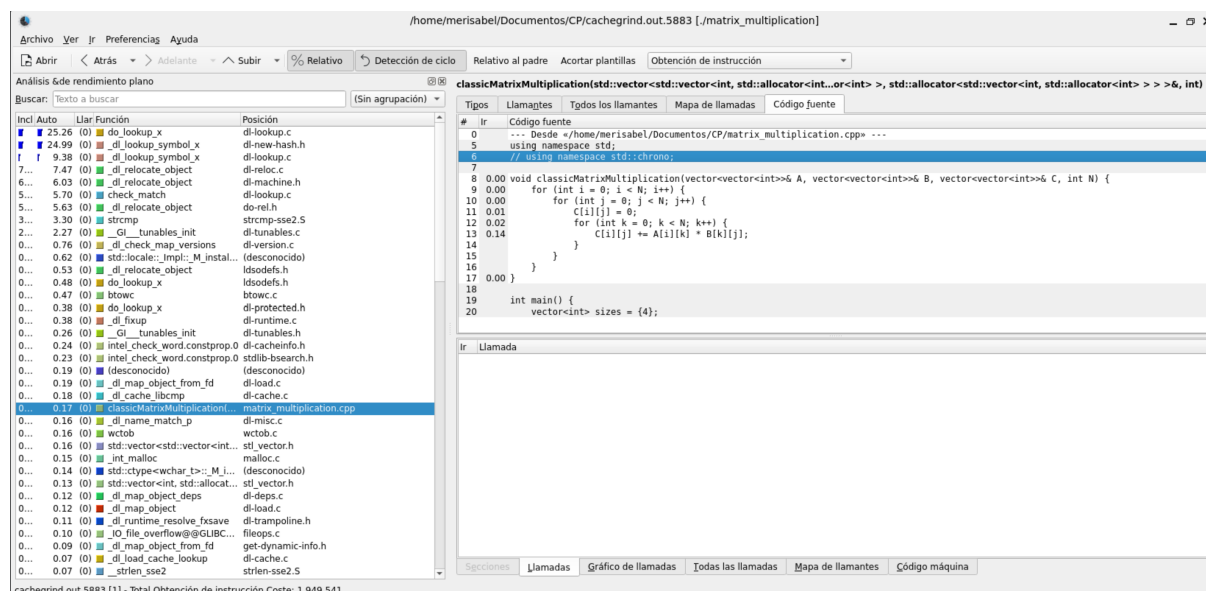
6. Ejercicio 5

Ejecutar ambos algoritmos utilizando las herramientas valgrind y kcachegrind para obtener una evaluación mas precisa de su desempeño .

6.1. Multiplicación Clásica

```
merisabel@merisabel-VirtualBox:~/Documentos/CP$ valgrind --tool=kcachegrind ./matrix_multiplication
==5621== Cachegrind, a high-precision tracing profiler
==5621== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==5621== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==5621== Command: ./matrix_multiplication
==5621==
Multiplicacion clasica para tamaño de matriz 4x4:
Duracion: 1 ms

==5621==
==5621== I refs:      1,953,796
```



The screenshot shows the Valgrind Kcachegrind interface. The left pane displays the call graph with the following functions and their relative costs:

Incl	Auto	Ular	Función	Posición
25.26	(0)	do_lookup_x	di-lookup.c	
24.99	(0)	_di_lookup_symbol_x	di-new-hash.h	
9.38	(0)	_di_lookup_symbol_x	di-lookup.c	
7.47	(0)	_di_relocate_object	di-reloc.c	
6.03	(0)	_di_relocate_object	di-machine.h	
5.70	(0)	check_match	di-lookup.c	
5.63	(0)	_di_relocate_object	di-reloc.h	
3.30	(0)	strncpy	strncpy-ase2.S	
2.27	(0)	_di_tunables_init	di-tunables.c	
0.76	(0)	_di_map_versions	di-version.c	
0.62	(0)	std::locale::impl::M_instal...	(desconocido)	
0.53	(0)	_di_relocate_object	di-reloc.h	
0.48	(0)	do_lookup_x	di-lookup.c	
0.47	(0)	btowc	btowc.c	
0.38	(0)	do_lookup_x	di-protected.h	
0.38	(0)	_di_fixup	di-runtime.c	
0.26	(0)	_di_tunables_init	di-tunables.h	
0.24	(0)	intel_check_word_constprop.0	di-cacheinfo.h	
0.23	(0)	intel_check_word_constprop.0	di-cacheinfo.h	
0.19	(0)	(desconocido)	(desconocido)	
0.19	(0)	_di_map_object_from_fd	di-load.c	
0.18	(0)	_di_cache_libcnp	di-cache.c	
0.17	(0)	classMatrixMultiplication...	matrix_multiplication.cpp	
0.16	(0)	_di_name_match_p	di-misc.c	
0.16	(0)	wctob	wctob.c	
0.16	(0)	std::vector<std::vector<int>...	std_vector.h	
0.15	(0)	int_malloc	malloc.c	
0.14	(0)	std::ctype<wchar_t>::M_...	(desconocido)	
0.13	(0)	std::vector<int, std::allocat...	std_vector.h	
0.12	(0)	_di_map_object_deps	di-deps.c	
0.12	(0)	_di_map_object	di-load.c	
0.11	(0)	_di_runtime_resolve_fxsave	di-trampoline.h	
0.10	(0)	_io_file_overflow@GLIBC...	fileops.c	
0.09	(0)	_di_map_object_from_fd	get-dynamic-info.h	
0.07	(0)	_di_load_cache_lookup	di-cache.c	
0.07	(0)	_di_cache_sse2	di-cache-ase2.S	

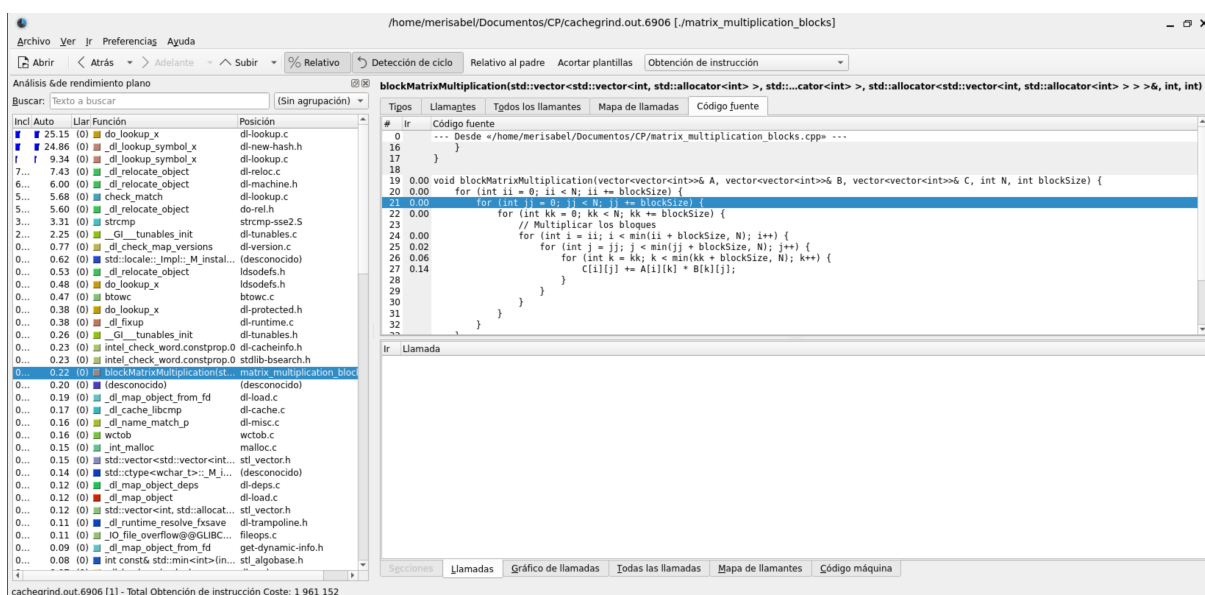
The right pane shows the source code of the `classMatrixMultiplication` function:

```
0  --- Desde /home/merisabel/Documentos/CP/matrix_multiplication.cpp ---
5  using namespace std;
6  // using namespace std::chrono;
7
8  0.00 void classMatrixMultiplication(vector<vector<int>>& A, vector<vector<int>>& B, vector<vector<int>>& C, int N) {
9  0.00     for (int i = 0; i < N; i++) {
10 0.00         for (int j = 0; j < N; j++) {
11 0.01             C[i][j] = 0;
12 0.02             for (int k = 0; k < N; k++) {
13 0.14                 C[i][j] += A[i][k] * B[k][j];
14             }
15         }
16     }
17 }
18
19 int main() {
20     vector<int> sizes = {4};
```

6.2. Multiplicación por Bloques

```
merisabel@merisabel-VirtualBox:~/Documentos/CP$ valgrind --tool=cachegrind ./matrix_multiplication_blocks
==6906== Cachegrind, a high-precision tracing profiler
==6906== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==6906== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==6906== Command: ./matrix_multiplication_blocks
==6906==
Probando multiplicación para tamaño de matriz 4x4:
Duración (clásica): 0 ms
Duración (por bloques): 2 ms

==6906==
==6906== I refs:      1,961,152
```



6.3. Resultados

6.3.1. Modelo de optimización de memoria

- Factores que afectan el rendimiento:
 - Tiempo de operaciones aritméticas.
 - Tiempo de acceso a la memoria.
- Dos niveles de memoria:
 - Memoria rápida (caché).
 - Memoria lenta (RAM).
- Conceptos clave:
 - **m**: Número de movimientos de datos desde la memoria lenta a la rápida.
 - **f**: Número de operaciones aritméticas.

- **q**: Intensidad computacional (cuántas operaciones se realizan por cada acceso a la memoria lenta), dado por:

$$q = \frac{f}{m}$$

Un valor elevado de q indica que se realizan muchas operaciones aritméticas antes de necesitar acceder nuevamente a la memoria lenta, lo que es beneficioso, ya que reduce la dependencia de los tiempos de acceso a la memoria.

6.3.2. Multiplicación Clásica

- Objetivo: Calcular $C = A \times B$ para matrices A y B de tamaño $n \times n$.
- Requerimientos:
 - Total de operaciones aritméticas: $2n^3$.
 - Accesos a memoria: Altos, con un valor de **q** alrededor de 2.

6.3.3. Multiplicación de matrices bloqueada

- Mejora que se basa en el principio de **localidad**:
 - Dividir matrices A , B y C en bloques pequeños de tamaño $b \times b$.
- Funcionamiento:
 1. Multiplicar bloques de matrices en lugar de matrices completas.
 2. Cargar bloques en la caché y realizar operaciones.
 3. Reutilizar bloques en la memoria rápida.

6.3.4. Análisis simplificado de la multiplicación bloqueada

- Reducción del número de accesos a la memoria lenta:

$$m = (2N + 2) \times n^2$$

- Cálculo de intensidad computacional:

$$q \approx \frac{n}{b}$$

- Importancia de elegir un tamaño de bloque adecuado:
 - Mejora la eficiencia del algoritmo.

6.3.5. Condiciones para el tamaño de los bloques

- Limitaciones de tamaño:
 - Todos los bloques deben caber en la memoria rápida.
 - Condición a cumplir:

$$3b^2 \leq M_{\text{fast}}$$