HPC Memory management

- Write a CUDA program to print: « my cuda course »
- Use 3 block, and 5 threads per block.
- Compile
- Execute

[9]
0s

```
%%writefile cuda_print.cu
#include <stdio.h>
#include <cuda_runtime.h>

// CUDA kernel function
__global__ void printMessage() {
    int globalThreadId = blockIdx.x * blockDim.x + threadIdx.x;

    printf("Block %d, Thread %d (Global Thread %d): my cuda course\n",
           blockIdx.x, threadIdx.x, globalThreadId);
}

int main() {
    int numBlocks = 3;
    int threadsPerBlock = 5;

    printf("=== CUDA Program Starting ===\n");
    printf("Configuration: %d blocks, %d threads per block\n", numBlocks, threadsPerBlock);
    printf("Total threads: %d\n\n", numBlocks * threadsPerBlock);

    printMessage<<<numBlocks, threadsPerBlock>>>();

    cudaDeviceSynchronize();

    printf("\n=== CUDA Program Completed ===\n");
    return 0;
}
```

```
Overwriting cuda_print.cu
```

```
Overwriting cuda_print.cu
```

[12] 1s

```
!nvcc -arch=native cuda_print.cu -o cuda_print
```

[13] 0s

```
!./cuda_print
```

```
=== CUDA Program Starting ===
Configuration: 3 blocks, 5 threads per block
Total threads: 15

Block 2, Thread 0 (Global Thread 10): my cuda course
Block 2, Thread 1 (Global Thread 11): my cuda course
Block 2, Thread 2 (Global Thread 12): my cuda course
Block 2, Thread 3 (Global Thread 13): my cuda course
Block 2, Thread 4 (Global Thread 14): my cuda course
Block 0, Thread 0 (Global Thread 0): my cuda course
Block 0, Thread 1 (Global Thread 1): my cuda course
Block 0, Thread 2 (Global Thread 2): my cuda course
Block 0, Thread 3 (Global Thread 3): my cuda course
Block 0, Thread 4 (Global Thread 4): my cuda course
Block 1, Thread 0 (Global Thread 5): my cuda course
Block 1, Thread 1 (Global Thread 6): my cuda course
Block 1, Thread 2 (Global Thread 7): my cuda course
Block 1, Thread 3 (Global Thread 8): my cuda course
Block 1, Thread 4 (Global Thread 9): my cuda course

=== CUDA Program Completed ===
```

globalID = threadIdx.x + blockIdx.x × blockDim.x

bloc 0

threadIdx.x	globalID
0	0 + 0×5 = **0**
1	1 + 0×5 = **1**
2	2 + 0×5 = **2**
3	3 + 0×5 = **3**
4	4 + 0×5 = **4**

bloc 1

threadIdx.x	globalID
0	0 + 1×5 = **5**
1	1 + 1×5 = **6**

threadIdx.x	globalID
2	2 + 1×5 = **7**
3	3 + 1×5 = **8**
4	4 + 1×5 = **9**

bloc 2

threadIdx.x	globalID
0	0 + 2×5 = **10**
1	1 + 2×5 = **11**
2	2 + 2×5 = **12**
3	3 + 2×5 = **13**
4	4 + 2×5 = **14**

```
Overwriting square_array.cu
```

[29] 1s

```
!nvcc square_array.cu -o square_array
```

[31] 1s

```
!nvcc -arch=sm_75 square_array.cu -o square_array
!./square_array
```

```
Results: A[i] = A[i] * A[i]
A[0] = 1.000000
A[1] = 4.000000
A[2] = 9.000000
A[3] = 16.000000
A[4] = 25.000000
A[5] = 36.000000
A[6] = 49.000000
A[7] = 64.000000
A[8] = 81.000000
A[9] = 100.000000
```

Write

a CUDA program to solve: A[i]=A[i]*A[i]

```
%%writefile square_array.cu

// square_array.cu
#include <stdio.h>
#include <cuda_runtime.h>

#define CUDA_CHECK(err) \
    if (err != cudaSuccess) { \
        printf("CUDA ERROR: %s\n", cudaGetErrorString(err)); \
        return 1; \
    }

// CUDA kernel: A[i] = A[i] * A[i]
__global__ void squareKernel(float *A, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        A[i] = A[i] * A[i];
    }
}

int main() {
    int N = 10;

    float *h_A, *d_A;

    h_A = (float*)malloc(N * sizeof(float));

    for (int i = 0; i < N; i++)
        h_A[i] = (float)(i + 1);

    CUDA_CHECK(cudaMalloc(&d_A, N * sizeof(float)));
    CUDA_CHECK(cudaMemcpy(d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevic
e));

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
```

```
    squareKernel<<<blocksPerGrid, threadsPerBlock>>>(d_A, N);

    CUDA_CHECK(cudaDeviceSynchronize());
    CUDA_CHECK(cudaGetLastError());

    CUDA_CHECK(cudaMemcpy(h_A, d_A, N * sizeof(float), cudaMemcpyDeviceToHo
st));

    // 🔥 Print results (this part was missing in your output)
    printf("Results: A[i] = A[i] * A[i]\n");
    for (int i = 0; i < N; i++)
        printf("A[%d] = %f\n", i, h_A[i]);

    cudaFree(d_A);
    free(h_A);

    return 0;
}
```

2D 4x4 matrix multiplication with tiles

```
%%writefile matrix_mul.cu
#include <stdio.h>

#define TILE_SIZE 2
#define N 4

__global__ void matrixMulTiled(float *A, float *B, float *C, int width) {
    __shared__ float sharedA[TILE_SIZE][TILE_SIZE];
    __shared__ float sharedB[TILE_SIZE][TILE_SIZE];

    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;
```

```
    float sum = 0.0f;

    for (int tile = 0; tile < width / TILE_SIZE; tile++) {
        sharedA[threadIdx.y][threadIdx.x] =
            A[row * width + (tile * TILE_SIZE + threadIdx.x)];

        sharedB[threadIdx.y][threadIdx.x] =
            B[(tile * TILE_SIZE + threadIdx.y) * width + col];

        __syncthreads();

        for (int k = 0; k < TILE_SIZE; k++) {
            sum += sharedA[threadIdx.y][k] * sharedB[k][threadIdx.x];
        }

        __syncthreads();
    }

    if (row < width && col < width)
        C[row * width + col] = sum;
}

int main() {
    const int size = N * N * sizeof(float);
    float h_A[N*N], h_B[N*N], h_C[N*N];

    for (int i = 0; i < N*N; i++) {
        h_A[i] = 1.0f;
        h_B[i] = 1.0f;
        h_C[i] = 0.0f;
    }

    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
    dim3 dimBlock(TILE_SIZE, TILE_SIZE);
    dim3 dimGrid(N / TILE_SIZE, N / TILE_SIZE);

    matrixMulTiled<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);
    cudaDeviceSynchronize();

    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        printf("CUDA error: %s\n", cudaGetErrorString(err));
        return 1;
    }

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    printf("\nResult Matrix C (4×4):\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf("%4.1f ", h_C[i*N + j]);
        printf("\n");
    }

    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    return 0;
}
```

[4]

```
# D'abord, vérifier quel GPU vous avez
!nvidia-smi --query-gpu=compute_cap --format=csv
```

```
compute_cap
7.5
```

[5]

```
# Pour la plupart des GPU Colab (T4, P100, V100):
!nvcc -arch=sm_75 matrix_mul.cu -o matrix_mul

# OU pour des GPU plus anciens (K80):
# !nvcc -arch=sm_37 matrix_mul.cu -o matrix_mul

# OU pour des GPU plus récents (A100):
# !nvcc -arch=sm_80 matrix_mul.cu -o matrix_mul

# OU laissez CUDA détecter automatiquement:
# !nvcc -arch=native matrix_mul.cu -o matrix_mul

!./matrix_mul
```

```
Result Matrix C (4x4):
 4.0  4.0  4.0  4.0
 4.0  4.0  4.0  4.0
 4.0  4.0  4.0  4.0
 4.0  4.0  4.0  4.0
```

for Global memory

```
# Compiler (ajustez -arch selon votre GPU)
!nvcc -arch=sm_75 -O3 matrix_comparison.cu -o matrix_comparison

# Exécuter
!./matrix_comparison
```

```
==============================================
Matrix Multiplication Performance Comparison
==============================================
Matrix Size: 1024 x 1024
Tile Size: 16 x 16

Running GLOBAL MEMORY version...
Global Memory Time: 9.317 ms

Running SHARED MEMORY version...
Shared Memory Time: 5.848 ms

==============================================
Results Verification: PASSED
==============================================
Performance Summary:
----------------------------------------------
Global Memory:  9.317 ms
Shared Memory:  5.848 ms
----------------------------------------------
Speedup:        1.59x
Time Saved:     3.469 ms (37.2%)
==============================================
```

Performance globale

- **Global Memory**: 9.317 ms
- **Shared Memory**: 5.848 ms
- **Speedup**: **1.59x** (59% plus rapide)
- **Temps économisé**: 3.469 ms (37.2%)

```
%%writefile matrix_comparison.cu
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <math.h>

#define TILE_SIZE 16
#define N 1024

__global__ void matrixMulGlobal(float *A, float *B, float *C, int width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    if (row < width && col < width) {
        float sum = 0.0f;
        for (int k = 0; k < width; k++) {
            sum += A[row * width + k] * B[k * width + col];
        }
        C[row * width + col] = sum;
    }
}

__global__ void matrixMulShared(float *A, float *B, float *C, int width) {
    __shared__ float sharedA[TILE_SIZE][TILE_SIZE];
    __shared__ float sharedB[TILE_SIZE][TILE_SIZE];

    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    float sum = 0.0f;

    for (int tile = 0; tile < (width + TILE_SIZE - 1) / TILE_SIZE; tile++) {

        if (row < width && (tile * TILE_SIZE + threadIdx.x) < width)
            sharedA[threadIdx.y][threadIdx.x] = A[row * width + (tile * TILE_SIZE + thread
Idx.x)];
        else
            sharedA[threadIdx.y][threadIdx.x] = 0.0f;

        if ((tile * TILE_SIZE + threadIdx.y) < width && col < width)
            sharedB[threadIdx.y][threadIdx.x] = B[(tile * TILE_SIZE + threadIdx.y) * width
+ col];
        else
            sharedB[threadIdx.y][threadIdx.x] = 0.0f;

        __syncthreads();

        for (int k = 0; k < TILE_SIZE; k++) {
            sum += sharedA[threadIdx.y][k] * sharedB[k][threadIdx.x];
        }

        __syncthreads();
```

```
    }

    if (row < width && col < width)
        C[row * width + col] = sum;
}

void initMatrix(float *mat, int size) {
    for (int i = 0; i < size; i++) {
        mat[i] = (float)(rand() % 10) / 10.0f;
    }
}

bool verifyResults(float *C1, float *C2, int size) {
    for (int i = 0; i < size; i++) {
        if (fabs(C1[i] - C2[i]) > 1e-3) {
            return false;
        }
    }
    return true;
}

int main() {

    printf("==================================================\n");
    printf("Matrix Multiplication Performance Comparison\n");
    printf("==================================================\n");
    printf("Matrix Size: %d x %d\n", N, N);
    printf("Tile Size: %d x %d\n\n", TILE_SIZE, TILE_SIZE);

    const int size = N * N * sizeof(float);

    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C_global = (float*)malloc(size);
    float *h_C_shared = (float*)malloc(size);

    initMatrix(h_A, N * N);
    initMatrix(h_B, N * N);
```

```
float *d_A, *d_B, *d_C;
cudaMalloc(&d_A, size);
cudaMalloc(&d_B, size);
cudaMalloc(&d_C, size);

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

dim3 dimBlock(TILE_SIZE, TILE_SIZE);
dim3 dimGrid((N + TILE_SIZE - 1) / TILE_SIZE, (N + TILE_SIZE - 1) / TILE_SIZE);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
float milliseconds = 0;

printf("Running GLOBAL MEMORY version...\n");

cudaEventRecord(start);
matrixMulGlobal<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);
cudaEventRecord(stop);

cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);

cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
   printf("CUDA Error (Global): %s\n", cudaGetErrorString(err));
   return 1;
}

cudaMemcpy(h_C_global, d_C, size, cudaMemcpyDeviceToHost);

float global_time = milliseconds;
printf("Global Memory Time: %.3f ms\n\n", global_time);

printf("Running SHARED MEMORY version...\n");

cudaEventRecord(start);
```

```
matrixMulShared<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);
cudaEventRecord(stop);

cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);

err = cudaGetLastError();
if (err != cudaSuccess) {
    printf("CUDA Error (Shared): %s\n", cudaGetErrorString(err));
    return 1;
}

cudaMemcpy(h_C_shared, d_C, size, cudaMemcpyDeviceToHost);

float shared_time = milliseconds;
printf("Shared Memory Time: %.3f ms\n\n", shared_time);

printf("============================================\n");
printf("Results Verification: ");
if (verifyResults(h_C_global, h_C_shared, N * N)) {
    printf("PASSED\n");
} else {
    printf("FAILED\n");
}

printf("============================================\n");
printf("Performance Summary:\n");
printf("----------------------------------------\n");
printf("Global Memory:  %.3f ms\n", global_time);
printf("Shared Memory:  %.3f ms\n", shared_time);
printf("----------------------------------------\n");
printf("Speedup:        %.2fx\n", global_time / shared_time);
printf("Time Saved:     %.3f ms (%.1f%%)\n",
       global_time - shared_time,
       100.0 * (global_time - shared_time) / global_time);
printf("============================================\n");

cudaFree(d_A);
cudaFree(d_B);
```

```
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C_global);
    free(h_C_shared);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}
```

Rewrite the code to multiply a 2D 8×8 matrices using (4×4) tiles.

```
%%writefile matrix_8×8.cu
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <math.h>

#define TILE_SIZE 4
#define N 8

__global__ void matrixMulGlobal(float *A, float *B, float *C, int width) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < width && col < width) {
        float sum = 0.0f;
        for (int k = 0; k < width; k++) {
            sum += A[row * width + k] * B[k * width + col];
        }
        C[row * width + col] = sum;
    }
}
```

```
__global__ void matrixMulShared(float *A, float *B, float *C, int width) {
    __shared__ float sharedA[TILE_SIZE][TILE_SIZE];
    __shared__ float sharedB[TILE_SIZE][TILE_SIZE];

    int row = blockIdx.y * TILE_SIZE + threadIdx.y;
    int col = blockIdx.x * TILE_SIZE + threadIdx.x;

    float sum = 0.0f;

    int numTiles = width / TILE_SIZE;

    for (int tile = 0; tile < numTiles; tile++) {

        sharedA[threadIdx.y][threadIdx.x] = A[row * width + (tile * TILE_SIZE + threadId
x.x)];
        sharedB[threadIdx.y][threadIdx.x] = B[(tile * TILE_SIZE + threadIdx.y) * width +
col];

        __syncthreads();

        for (int k = 0; k < TILE_SIZE; k++) {
            sum += sharedA[threadIdx.y][k] * sharedB[k][threadIdx.x];
        }

        __syncthreads();
    }

    C[row * width + col] = sum;
}

void initMatrix(float *mat, int size, int seed) {
    srand(seed);
    for (int i = 0; i < size; i++) {
        mat[i] = (float)(rand() % 10);
    }
}

void printMatrix(const char *name, float *mat, int width) {
    printf("%s:\n", name);
```

```
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < width; j++) {
            printf("%6.1f ", mat[i * width + j]);
        }
        printf("\n");
    }
    printf("\n");
}

bool verifyResults(float *C1, float *C2, int size) {
    for (int i = 0; i < size; i++) {
        if (fabs(C1[i] - C2[i]) > 0.01f) {
            return false;
        }
    }
    return true;
}

int main() {

    printf("Matrix Multiplication: 8×8 with 4×4 Tiles\n");
    printf("==========================================\n\n");

    const int size = N * N * sizeof(float);

    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C_global = (float*)malloc(size);
    float *h_C_shared = (float*)malloc(size);

    initMatrix(h_A, N * N, 42);
    initMatrix(h_B, N * N, 123);

    printMatrix("Matrix A (8×8)", h_A, N);
    printMatrix("Matrix B (8×8)", h_B, N);

    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
```

```
cudaMalloc(&d_C, size);

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

dim3 dimBlock(TILE_SIZE, TILE_SIZE);
dim3 dimGrid(N / TILE_SIZE, N / TILE_SIZE);

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
float milliseconds = 0;

printf("Running GLOBAL MEMORY version...\n");
cudaEventRecord(start);
matrixMulGlobal<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);

cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    printf("CUDA Error: %s\n", cudaGetErrorString(err));
    return 1;
}

cudaMemcpy(h_C_global, d_C, size, cudaMemcpyDeviceToHost);
float global_time = milliseconds;
printf("Time: %.6f ms\n\n", global_time);

printf("Running SHARED MEMORY version...\n");
cudaEventRecord(start);
matrixMulShared<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);

err = cudaGetLastError();
if (err != cudaSuccess) {
```

```
    printf("CUDA Error: %s\n", cudaGetErrorString(err));
    return 1;
}

cudaMemcpy(h_C_shared, d_C, size, cudaMemcpyDeviceToHost);
float shared_time = milliseconds;
printf("Time: %.6f ms\n\n", shared_time);

printMatrix("Result Matrix C (Global Memory)", h_C_global, N);
printMatrix("Result Matrix C (Shared Memory)", h_C_shared, N);

printf("========================================\n");
printf("Verification: ");
if (verifyResults(h_C_global, h_C_shared, N * N)) {
    printf("PASSED - Results match!\n");
} else {
    printf("FAILED - Results differ!\n");
}

printf("========================================\n");
printf("Performance:\n");
printf("  Global Memory:  %.6f ms\n", global_time);
printf("  Shared Memory:  %.6f ms\n", shared_time);
if (shared_time > 0) {
    printf("  Speedup:        %.2fx\n", global_time / shared_time);
}
printf("========================================\n");

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
free(h_A);
free(h_B);
free(h_C_global);
free(h_C_shared);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

```
    return 0;
}
```

Matrix Multiplication: 8x8 with 4x4 Tiles

Matrix A (8×8):
6.0 0.0 1.0 1.0 2.0 8.0 1.0 0.0
5.0 3.0 4.0 3.0 7.0 4.0 6.0 2.0
2.0 8.0 8.0 9.0 7.0 2.0 9.0 3.0
7.0 9.0 9.0 4.0 1.0 6.0 3.0 7.0
6.0 4.0 0.0 0.0 4.0 1.0 1.0 0.0
5.0 7.0 5.0 4.0 1.0 1.0 7.0 3.0
0.0 7.0 4.0 9.0 0.0 3.0 4.0 9.0
5.0 5.0 3.0 8.0 1.0 9.0 5.0 0.0

Matrix B (8×8):
3.0 3.0 3.0 0.0 9.0 1.0 5.0 2.0
6.0 1.0 4.0 8.0 1.0 1.0 6.0 1.0
9.0 7.0 6.0 3.0 6.0 5.0 9.0 1.0
6.0 8.0 6.0 8.0 5.0 2.0 5.0 8.0
5.0 8.0 1.0 7.0 9.0 6.0 9.0 6.0
0.0 5.0 4.0 1.0 7.0 2.0 4.0 8.0
1.0 0.0 3.0 9.0 8.0 3.0 1.0 4.0
3.0 7.0 4.0 1.0 9.0 9.0 9.0 7.0

Running GLOBAL MEMORY version...
Time: 0.124896 ms

Running SHARED MEMORY version...
Time: 0.029056 ms

Result Matrix C (Global Memory):
44.0 89.0 67.0 42.0 147.0 44.0 95.0 101.0
134.0 160.0 118.0 169.0 244.0 120.0 197.0 153.0
233.0 229.0 194.0 295.0 295.0 168.0 282.0 207.0
209.0 212.0 197.0 178.0 284.0 159.0 289.0 179.0
63.0 59.0 45.0 70.0 109.0 39.0 95.0 52.0
147.0 123.0 135.0 177.0 201.0 101.0 179.0 117.0
163.0 185.0 166.0 188.0 210.0 144.0 220.0 186.0
130.0 158.0 153.0 174.0 220.0 80.0 172.0 180.0

Result Matrix C (Shared Memory):
44.0 89.0 67.0 42.0 147.0 44.0 95.0 101.0
134.0 160.0 118.0 169.0 244.0 120.0 197.0 153.0
233.0 229.0 194.0 295.0 295.0 168.0 282.0 207.0
209.0 212.0 197.0 178.0 284.0 159.0 289.0 179.0
63.0 59.0 45.0 70.0 109.0 39.0 95.0 52.0
147.0 123.0 135.0 177.0 201.0 101.0 179.0 117.0
163.0 185.0 166.0 188.0 210.0 144.0 220.0 186.0
130.0 158.0 153.0 174.0 220.0 80.0 172.0 180.0

================================
Verification: PASSED - Results match!

Performance:
Global Memory: 0.124896 ms
Shared Memory: 0.029056 ms
Speedup: 4.30x

========================

3️⃣ Pourquoi les tiles “n’existent” que pour shared memory

- Le concept de **tile** est lié à la **réutilisation locale de données dans un bloc**.
- Dans la mémoire globale, chaque thread lit directement depuis la mémoire GPU, donc **on ne peut pas stocker temporairement un tile pour réutilisation**.
- Le tiling **n’a de sens que si tu as un espace rapide partagé pour tous les threads d’un bloc** → c’est exactement le rôle de `__shared__` memory.

Résumé simple :

- Global memory = accès lent → pas de tiling possible.
- Shared memory = accès rapide + partagé entre threads → tiling possible et efficace.

- Le “tile” = un petit bloc de données stocké **dans shared memory**, pour réduire le nombre d’accès à global memory.

1. Change the tile size to 2×2. How might this differ from the previous scenario?

```
1 !./tile_comparison
2
```

```
Global Memory: 0.161949 ms
Shared 4x4 Tiles: 0.029262 ms
Shared 2x2 Tiles: 0.024548 ms
```

a) Global Memory

- Le kernel qui utilise **la mémoire globale uniquement** est **beaucoup plus lent** (~0.16 ms)
- Cela confirme que **accéder directement à la mémoire globale est coûteux**, surtout pour des multiplications matricielles.

b) Shared 4×4 Tiles

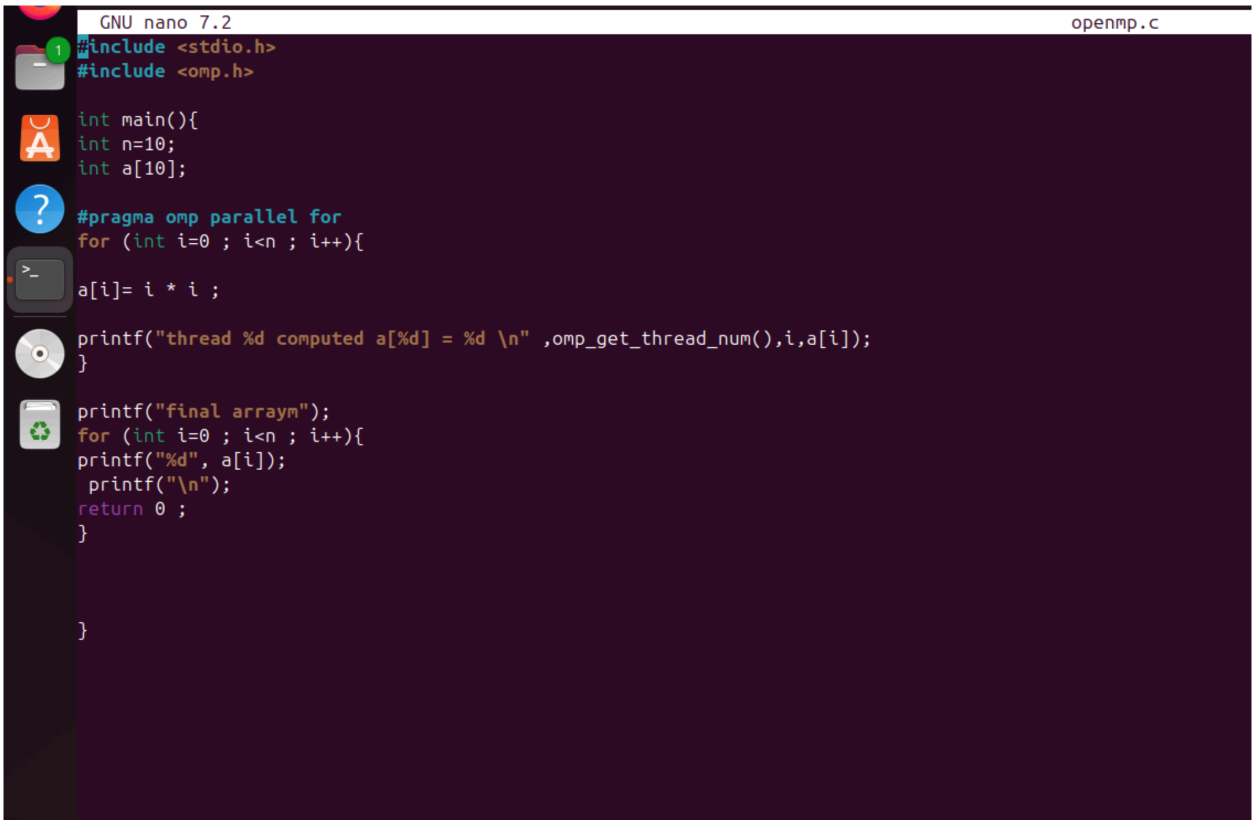
- Le kernel avec des **tiles 4×4** est **plus rapide** (~0.029 ms)
- Pourquoi ?
    - Chaque tile 4×4 est chargé une seule fois depuis la mémoire globale, puis réutilisé **4 fois** par les threads du bloc
    - Moins d’accès mémoire globale → moins de latence

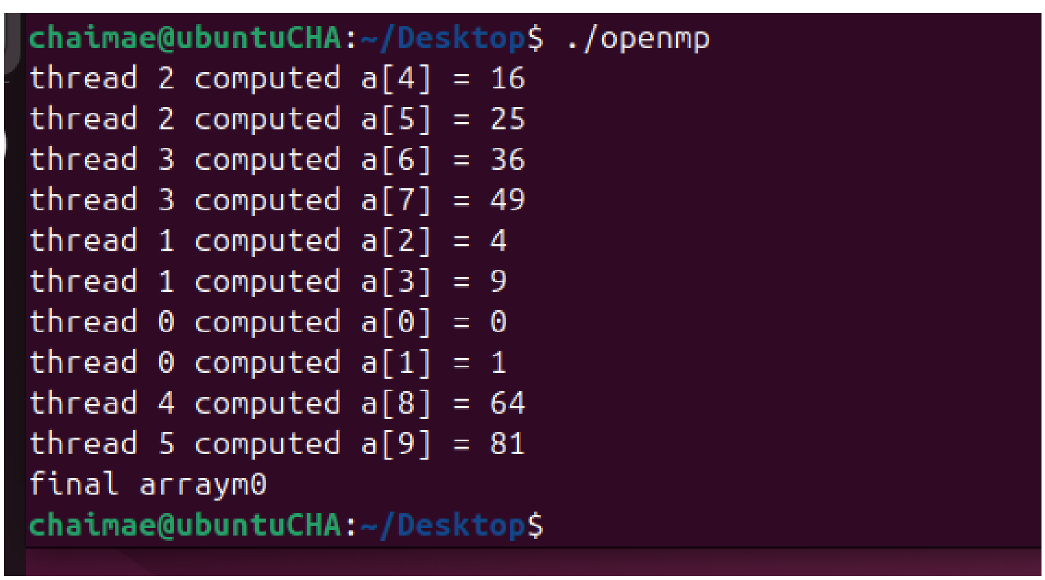
c) Shared 2×2 Tiles

- Le kernel avec des **tiles 2×2** est encore un peu plus rapide (~0.024 ms)
- Pourquoi ?

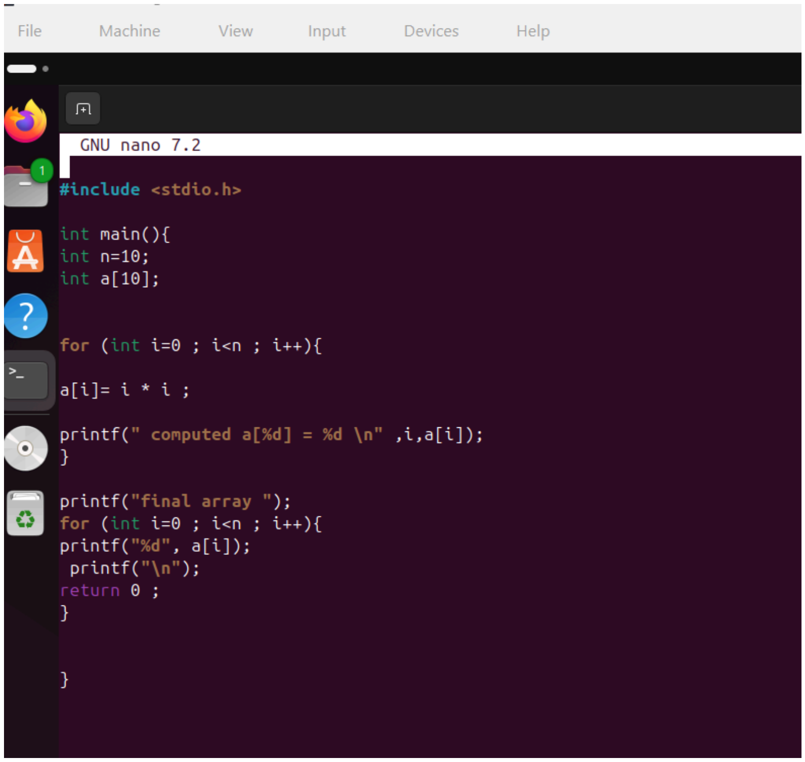
- Avec une matrice très petite (8×8), les différences sont faibles et l'overhead de synchronisation et de threads est minime
- Le GPU peut mieux gérer la petite taille de bloc, donc l'exécution est légèrement plus rapide que 4×4 pour cette taille spécifique

- openmp

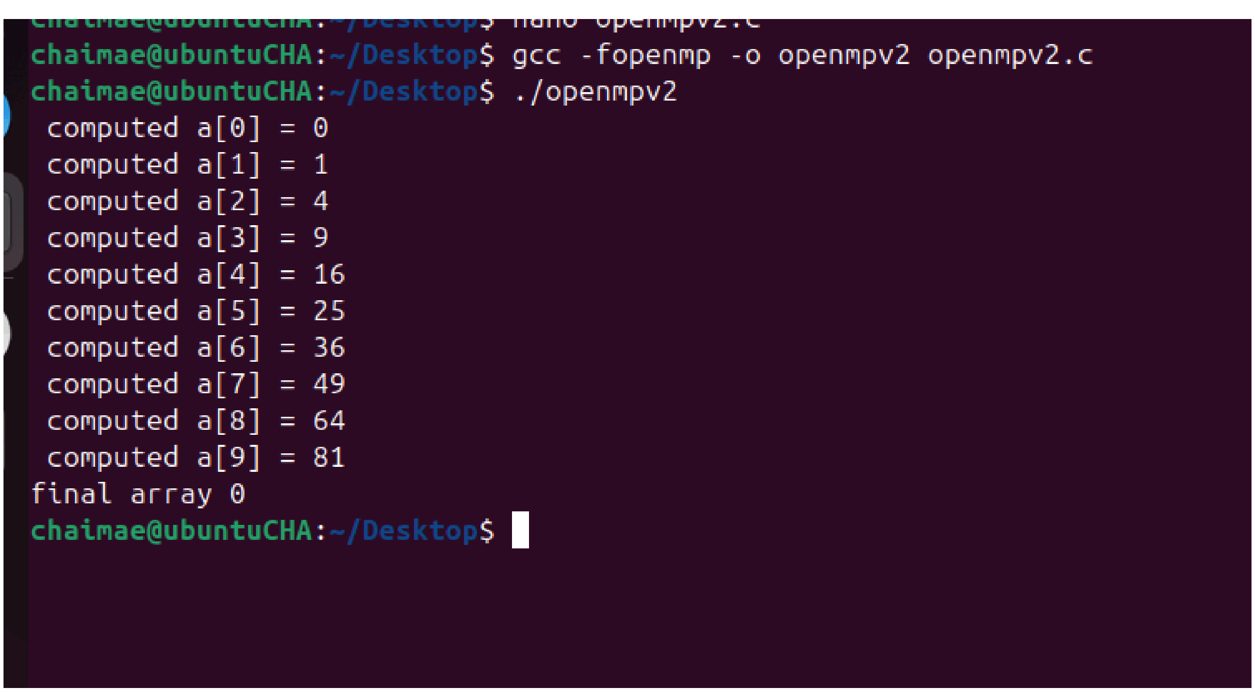




sequentielle un seul thread



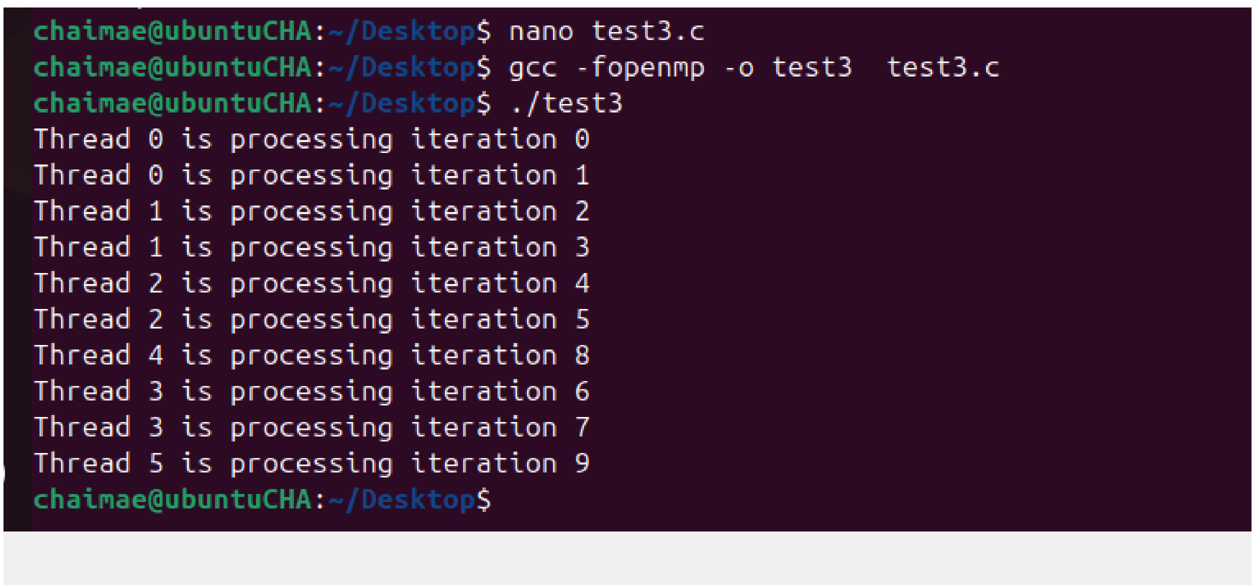
```
File    Machine    View    Input    Devices    Help
GNU nano 7.2

#include <stdio.h>

int main(){
int n=10;
int a[10];


for (int i=0 ; i<n ; i++){

a[i]= i * i ;

printf(" computed a[%d] = %d \n" ,i,a[i]);
}

printf("final array ");
for (int i=0 ; i<n ; i++){
printf("%d", a[i]);
 printf("\n");
return 0 ;
}


}
```



```
chaimae@ubuntuCHA:~/Desktop$ gcc -fopenmp -o openmpv2 openmpv2.c
chaimae@ubuntuCHA:~/Desktop$ ./openmpv2
 computed a[0] = 0
 computed a[1] = 1
 computed a[2] = 4
 computed a[3] = 9
 computed a[4] = 16
 computed a[5] = 25
 computed a[6] = 36
 computed a[7] = 49
 computed a[8] = 64
 computed a[9] = 81
final array 0
chaimae@ubuntuCHA:~/Desktop$
```

compiler directives directive name


chaimae@ubuntuCH
GNU nano 7.2
test
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel for
for (int i=0;i<10;i++) {
printf("Thread %d is processing iteration %d\n", omp_get_thread_num(),i);
}
return 0; }


chaimae@ubuntuCHA:~/Desktop$ nano test3.c
chaimae@ubuntuCHA:~/Desktop$ gcc -fopenmp -o test3 test3.c
chaimae@ubuntuCHA:~/Desktop$ ./test3
Thread 0 is processing iteration 0
Thread 0 is processing iteration 1
Thread 1 is processing iteration 2
Thread 1 is processing iteration 3
Thread 2 is processing iteration 4
Thread 2 is processing iteration 5
Thread 4 is processing iteration 8
Thread 3 is processing iteration 6
Thread 3 is processing iteration 7
Thread 5 is processing iteration 9
chaimae@ubuntuCHA:~/Desktop$

ex 2

```
  GNU nano 7.2                                                                   test4.c
#include <stdio.h>
#include <omp.h>

int main() {
  #pragma omp parallel sections
  {
    #pragma omp section
    printf("Section 1 executed by thread %d\n", omp_get_thread_num());

    #pragma omp section
    printf("Section 2 executed by thread %d\n", omp_get_thread_num());
  }
  return 0;
}
```

```
chaimae@ubuntuCHA:~/Desktop$ nano test4.c
chaimae@ubuntuCHA:~/Desktop$ gcc -fopenmp -o test4  test4.c
chaimae@ubuntuCHA:~/Desktop$ ./test4
Section 1 executed by thread 4
Section 2 executed by thread 3
chaimae@ubuntuCHA:~/Desktop$
```

sections:

allows each section to run independently on a thread.

ex 3

```
  GNU nano 7.2
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel
{
#pragma omp single
printf("This is executed by thread %d\n", omp_get_thread_num());
}
return 0; }
```

```
chaimae@ubuntuCHA:~/Desktop$ ./directive
This is executed by thread 4
chaimae@ubuntuCHA:~/Desktop$
```

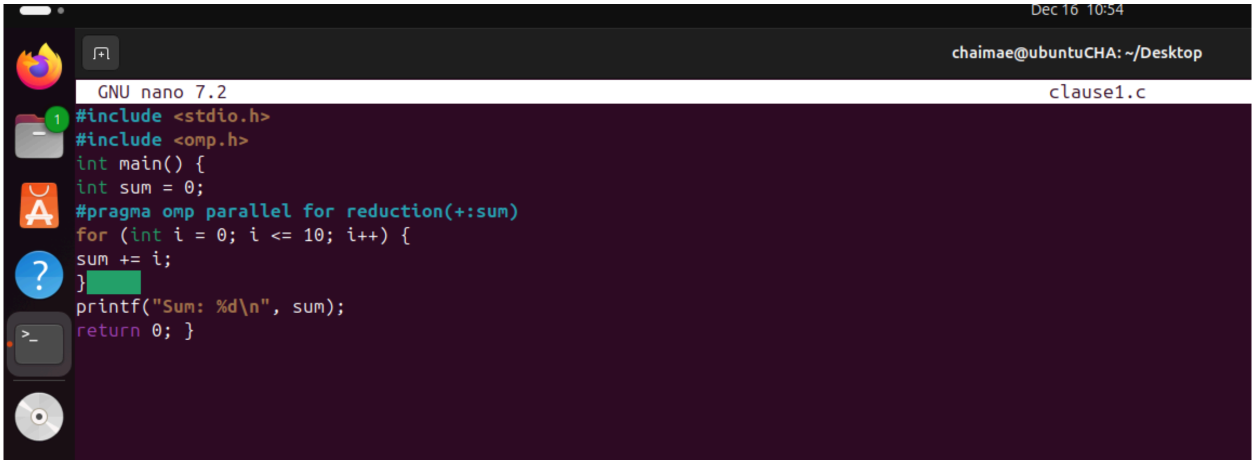
single:
Specifies that a block of code should be executed by only one thread.

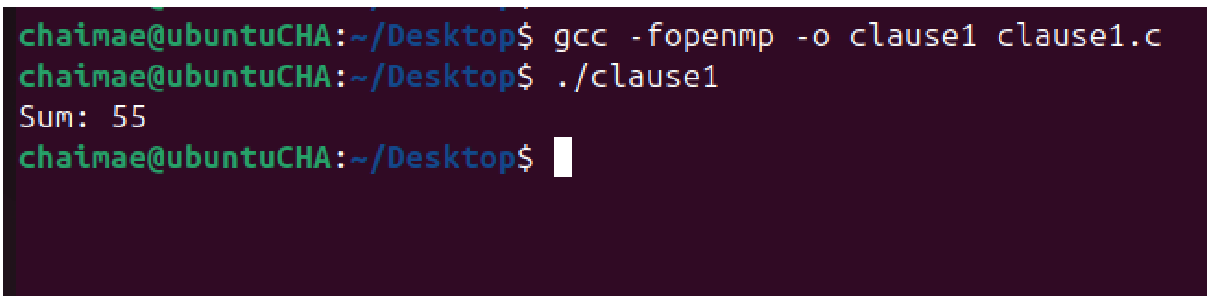
barrier

```
GNU nano 7.2                                                    barrier.c
#include <stdio.h>
#include <omp.h>
int main() {
#pragma omp parallel
{
printf("Before barrier, thread %d\n", omp_get_thread_num());
#pragma omp barrier
printf("After barrier, thread %d\n", omp_get_thread_num());
}  return 0;  }
```

```
chaimae@ubuntuCHA:~/Desktop$ nano barrier.c
chaimae@ubuntuCHA:~/Desktop$ gcc -fopenmp -o barrier barrier.c
chaimae@ubuntuCHA:~/Desktop$ ./barrier
Before barrier, thread 3
Before barrier, thread 2
Before barrier, thread 5
Before barrier, thread 0
Before barrier, thread 1
Before barrier, thread 4
After barrier, thread 1
After barrier, thread 5
After barrier, thread 4
After barrier, thread 2
After barrier, thread 0
After barrier, thread 3
chaimae@ubuntuCHA:~/Desktop$ nano barrier.c
chaimae@ubuntuCHA:~/Desktop$ SS
```

clause





=================================

🌟 OpenMP – Runtime Library Routines (explication simple)

🔷 Qu’est-ce que les *Runtime Library Routines* ?

Ce sont des **fonctions OpenMP déjà prêtes** qui permettent :

- de **contrôler les threads**
- de **connaître l’état de l’exécution**
- de **modifier le comportement du programme pendant l’exécution**

👉 Elles sont utilisées **à l’intérieur du code C/C++**, pas avec `#pragma`.

Gestion des threads (Thread Management)

1️⃣ `omp_get_num_threads()`

👉 **Donne le nombre de threads actifs**

- Fonctionne **dans une région parallèle**
- Si tu es hors d’une région parallèle → retourne `1`

📌 Exemple :

```
#pragma omp parallel
{
printf("Threads = %d\n", omp_get_num_threads());
}
```

🧠 Idée simple :

> “Combien de threads travaillent maintenant ?”

2️⃣ `omp_get_thread_num()`

👉 **Donne l’ID du thread courant**

- Chaque thread a un numéro unique
- Les IDs vont de `0` à `N-1`

📌 Exemple :

```
#pragma omp parallel
{
printf("I am thread %d\n", omp_get_thread_num());
}
```

🧠 Idée simple :

> “Qui suis-je parmi les threads ?”

3️⃣ `omp_set_num_threads(int num_threads)`

👉 **Fixe le nombre de threads à utiliser**

- S’applique aux prochaines régions parallèles
- Le système peut utiliser moins de threads si nécessaire

📌 Exemple :

```
omp_set_num_threads(4);

#pragma omp parallel
{
printf("Thread %d\n", omp_get_thread_num());
}
```

🧠 Idée simple :

> “Je veux 4 threads pour ce travail”

4️⃣ `omp_set_nested(int nested)`

👉 **Active ou désactive le parallélisme imbriqué**

- `1` → activé
- `0` → désactivé (par défaut)

📌 Exemple :

```
omp_set_nested(1);
```

🧠 Idée simple :

> “Autoriser des régions parallèles à l’intérieur d’autres régions parallèles”

5️⃣ `omp_get_nested()`

👉 **Vérifie si le parallélisme imbriqué est activé**

📌 Exemple :

```
if (omp_get_nested())
printf("Nested parallelism enabled\n");
```

🧠 Idée simple :

> “Est-ce que le parallélisme imbriqué est actif ?”

6️⃣ `omp_get_thread_limit()`

👉 **Donne le nombre maximum de threads autorisés**

- Limite globale pour tout le programme

📌 Exemple :

```
printf("Max threads allowed = %d\n", omp_get_thread_limit());
```

🧠 Idée simple :

> “Quel est le maximum de threads que je peux utiliser ?”

🧠 Résumé ultra-simple

Fonction	Sert à quoi ?
`omp_get_num_threads()`	Combien de threads travaillent
`omp_get_thread_num()`	ID du thread courant
`omp_set_num_threads(n)`	Fixer le nombre de threads
`omp_set_nested(1/0)`	Activer / désactiver le parallélisme imbriqué
`omp_get_nested()`	Vérifier le parallélisme imbriqué
`omp_get_thread_limit()`	Nombre max de threads

> Runtime library routines in OpenMP allow the programmer to query and control the execution environment, especially thread creation, scheduling, and parallel execution behavior.

```
  GNU nano 7.2                                                    omp_num_threads_test.c
#include <omp.h>
#include <stdio.h>
int main() {
omp_set_num_threads(4);
#pragma omp parallel
{
printf("Number of threads: %d\n", omp_get_num_threads());
printf("This is thread %d\n", omp_get_thread_num());
}
return 0;
}
```

```
chaimae@ubuntuCHA:~/Desktop$ nano omp_num_threads_test.c
chaimae@ubuntuCHA:~/Desktop$ gcc -fopenmp -o omp_num_threads_test omp_num_threads_test.c
chaimae@ubuntuCHA:~/Desktop$ ./omp_num_threads_test
Number of threads: 4
This is thread 1
Number of threads: 4
This is thread 0
Number of threads: 4
This is thread 3
Number of threads: 4
This is thread 2
chaimae@ubuntuCHA:~/Desktop$
```

Le **scheduling** détermine **comment les itérations d'une boucle parallèle sont réparties entre les threads**.

C'est important pour **optimiser les performances**.

Fonctions principales

- `omp_set_schedule(kind, chunk_size)` : Définit la politique de scheduling et la taille des blocs (chunks).
- `omp_get_schedule(&kind, &chunk_size)` : Récupère la politique et la taille actuelle.

Politiques de scheduling

Politique	Description
`static`	Les itérations sont réparties en blocs égaux dès le départ.
`dynamic`	Les itérations sont données aux threads au fur et à mesure qu'ils terminent.
`guided`	Comme `dynamic`, mais les blocs diminuent progressivement.
`auto`	Le compilateur/OS choisit la meilleure stratégie.

C’est tout ce qu’il faut retenir pour révision rapide :

- **set** → choisir la stratégie
- **get** → savoir la stratégie utilisée
- **politiques** → static, dynamic, guided, auto
