EJEMPLOS USANDO CUDA (PARTE 2)

Francisco J. Hernández López

fcoj23@cimat.mx

SUMA DE LOS ELEMENTOS DE UN VECTOR

- Creamos el vector "A" en el CPU
- Inicializamos con cualquier valor el vector "A"
- Creamos el vector "d_A" en el GPU
- Copiamos el contenido de "A" del CPU al GPU
- Sumamos los elementos de "d_A" y el resultado lo guardamos en una variable en el GPU
- Copiamos el resultado del GPU al CPU
- Desplegamos la suma de los elementos del vector "A"

SUMA DE LOS ELEMENTOS DE UN VECTOR C1

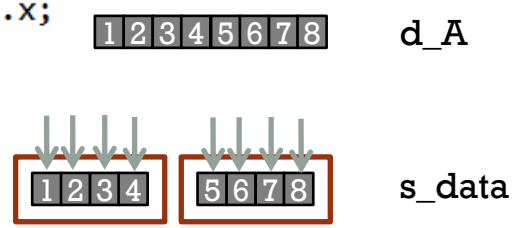
```
int main(int argc, char* argv[]) {
    double t_ini, t_fin;
    double time_generateData,time_cpu_seconds,time_gpu_seconds;
    double *A;
    double sum_total_cpu,sum_total_gpu;
    long int n=500000000;

    printf("De cuantos elementos son los arreglos\n");
    scanf("%ld", &n);

    t_ini = clock();
    A = generateRandomArray(n);
    t_fin = clock();
    time_generateData = (t_fin - t_ini) / CLOCKS_PER_SEC;

    t_ini = clock();
    sum_total_cpu = sumOfArraySeq(A, n);
    t_fin = clock();
    time_cpu_seconds = (t_fin - t_ini) / CLOCKS_PER_SEC;

    t_ini = clock();
    sum_total_gpu = sumOfArrayGPU(A,n);
    t_fin = clock();
    time_gpu_seconds = (t_fin - t_ini) / CLOCKS_PER_SEC;

    printf("La suma del arreglo en CPU es: %lf \n", sum_total_cpu);
    printf("La suma del arreglo en GPU es: %lf \n", sum_total_gpu);
    printf("Tiempo para generar datos: %lf segundos.\n", time_generateData);
    printf("Tiempo de procesamiento en CPU: %lf segundos.\n",time_cpu_seconds);
    printf("Tiempo de procesamiento en GPU: %lf segundos.\n", time_gpu_seconds);

    free(A);
}
```

SUMA DE LOS ELEMENTOS DE UN VECTOR C2

```
double sumOfArrayGPU(double *A, long int n){
    double *d_A;
    double *d_sum_total;
    double sum_total;

    //1. Crear memoria en la GPU
    cudaMalloc(&d_sum_total, sizeof(double));
    cudaMalloc(&d_A, n * sizeof(double));

    //Inicializamos en cero
    cudaMemset(d_sum_total, 0, sizeof(double));

    //2. Copiar memoria (CPU-->GPU)
    cudaMemcpy(d_A, A, n * sizeof(double), cudaMemcpyHostToDevice);

    //3. Ejecutar función Kernel
    sumOfArrayKernel <<<(n+TPB-1)/TPB,TPB >>> (d_sum_total,d_A,n);
    //sumOfArrayKernel_V2 << <(n + TPB - 1) / TPB, TPB >> > (d_sum_total, d_A, n);

    //4. Copiar memoria (GPU-->CPU)
    cudaMemcpy(&sum_total, d_sum_total, sizeof(double), cudaMemcpyDeviceToHost);

    cudaFree(d_sum_total);
    cudaFree(d_A);
    cudaDeviceReset();
    return(sum_total);
}
```

SUMA DE LOS ELEMENTOS DE UN VECTOR C3

```
#define TPB 1024
#define ATOMIC 1 // 0 para no usar el atomicAdd

__global__ void sumOfArrayKernel(double *d_sum_total, double *d_A, long int n) {
    const long int idx = threadIdx.x + blockDim.x * blockIdx.x;
    const int s_idx = threadIdx.x;
    __shared__ double s_data[TPB];

    s_data[s_idx] = (idx<n) ? d_A[idx] : 0.0;
    __syncthreads();

    if (s_idx==0) {
        double blockSum = 0.0;
        for (int j = 0; j < blockDim.x; j++) {
            blockSum += s_data[j];
        }
        //printf("Block_%d, blockSum = %lf\n", blockIdx.x, blockSum);
        if (ATOMIC) {
            atomicAdd(d_sum_total, blockSum);
        }
        else {
            *d_sum_total += blockSum; //Resultados no esperados
        }
    }
}
```

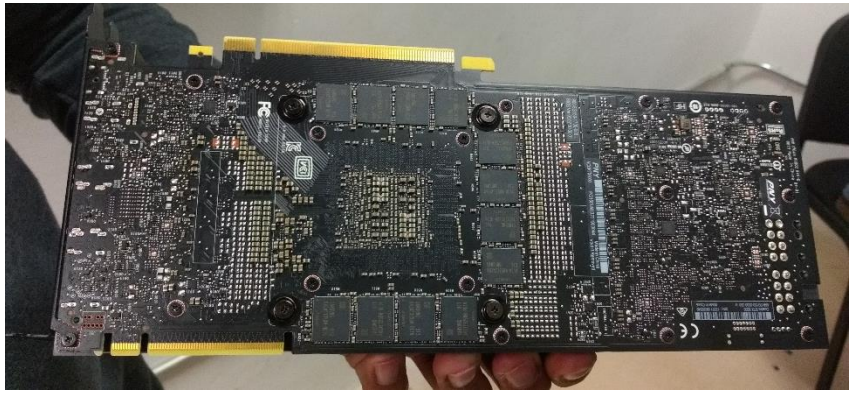


d_sum_total= 10 + 26 = 36

Para usar atomicAdd con tipos de datos double, necesitamos una GPU con capacidad >= 6.0

EXPERIMENTOS EN EL SERVIDOR DE CIMAT-MÉRIDA (SQUADRO8000)

Intel(R) Xeon(R) Gold 5222 CPU @ 3.80GHz
Core(s) per socket: 4
Socket(s): 2
RAM: 48 GB
Disco Duro: 1 TB
Ubuntu 18.04
1 Tarjeta NVIDIA Quadro RTX 8000



Gracias al Proyecto de Infraestructura 2019:
Laboratorio de Supercómputo del Bajío (Lab-SB)
https://supercomputobajio.cimat.mx/

TIEMPO DE PROCESAMIENTO

Compilación:

nvcc -arch=compute_75 kernel.cu sumatoria_main.cpp -o run_sumatoria

$ ./run_sumatoria
De cuantos elementos son los arreglos
1000000000
La suma del arreglo en CPU es: **50501072177.000000**
La suma del arreglo en GPU es: **50501072177.000000**
Tiempo para generar datos: 11.004083 segundos.
Tiempo de procesamiento en CPU: **2.367827** segundos.
Tiempo de procesamiento en GPU: **2.570765** segundos.

Nota: El tiempo en GPU incluye la creación de memoria en la GPU, las copias de memoria (CPU→GPU, GPU→CPU) y la ejecución de la función Kernel

TIEMPO DE PROCESAMIENTO DE LA FUNCIÓN KERNEL

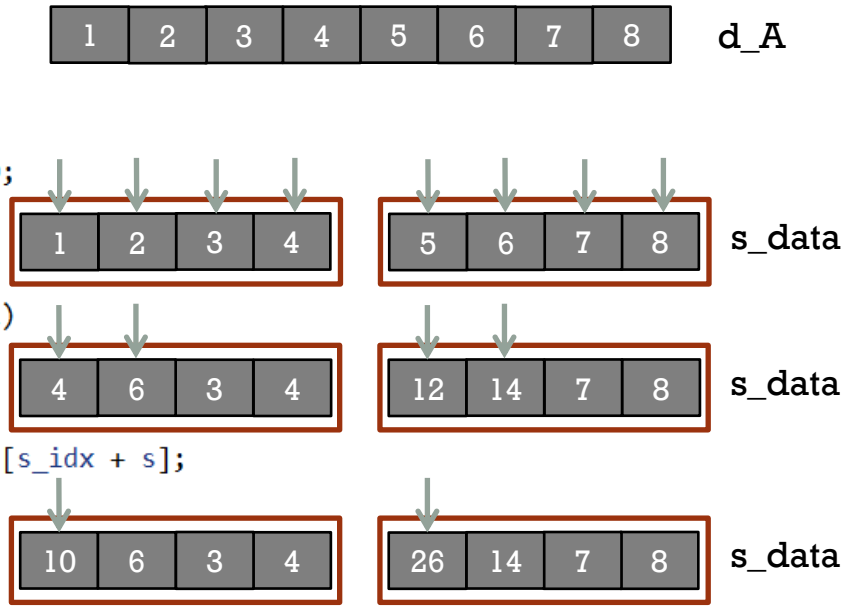
$ **nvprof** ./run_sumatoria_clock
De cuantos elementos son los arreglos
1000000000
==2579== NVPROF is profiling process 2579, command: ./run_sumatoria_clock
La suma del arreglo en CPU es: **50501072177.000000**
La suma del arreglo en GPU es: **50501072177.000000**
Tiempo para generar datos: 11.359986 segundos.
Tiempo de procesamiento en CPU: **2.364597** segundos.
Tiempo de procesamiento en GPU: **2.704036** segundos.
==2579== Profiling application: ./run_sumatoria_clock
==2579== Profiling result:

$$\boldsymbol{Speedup} = \frac{T_S}{T_P} = \frac{2.36s}{0.39s} \approx 6$$

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	82.73%	1.88169s	1	1.88169s	1.88169s	1.88169s	[CUDA memcpy HtoD]
	17.27%	392.67ms	1	392.67ms	392.67ms	**392.67ms**	**sumOfArrayKernel( )**
	0.00%	1.5360us	1	1.5360us	1.5360us	1.5360us	[CUDA memset]
	0.00%	1.3760us	1	1.3760us	1.3760us	1.3760us	[CUDA memcpy DtoH]

FUNCIÓN KERNEL V2

```
__global__ void sumOfArrayKernel_V2(double* d_sum_total, double* d_A, long int n) {
    const long int idx = threadIdx.x + blockDim.x * blockIdx.x;

    double blockSum = 0.0;
    const int s_idx = threadIdx.x;
    __shared__ double s_data[TPB];

    s_data[s_idx]= blockSum = (idx < n) ? d_A[idx] : 0.0;

    __syncthreads();

    for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
    {
        if (s_idx < s)
        {
            s_data[s_idx] = blockSum = blockSum + s_data[s_idx + s];
        }

        __syncthreads();
    }

    if (s_idx == 0) {
        if (ATOMIC) {
            atomicAdd(d_sum_total, blockSum);
        }
        else {
            *d_sum_total += blockSum; //Resultados no esperados
        }
    }
}
```



d_sum_total= 10 + 26 = 36

TIEMPO DE PROCESAMIENTO DE LA FUNCIÓN KERNEL V2

$ **nvprof** ./run_sumatoria_clock
De cuantos elementos son los arreglos
1000000000
==3642== NVPROF is profiling process 3642, command: ./run_sumatoria_clock
La suma del arreglo en CPU es: **50501072177.000000**
La suma del arreglo en GPU es: **50501072177.000000**
Tiempo para generar datos: 11.100356 segundos.
Tiempo de procesamiento en CPU: **2.366821** segundos.
Tiempo de procesamiento en GPU: **2.255754** segundos.
==3642== Profiling application: ./run_sumatoria_clock
==3642== Profiling result:

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	95.65%	1.74663s	1	1.74663s	1.74663s	1.74663s	[CUDA memcpy HtoD]
	4.35%	79.424ms	1	79.424ms	79.424ms	**79.424ms**	**sumOfArrayKernel_V2**()
	0.00%	1.6640us	1	1.6640us	1.6640us	1.6640us	[CUDA memcpy DtoH]
	0.00%	1.4720us	1	1.4720us	1.4720us	1.4720us	[CUDA memset]

$$\textbf{\textit{Speedup}} = \frac{T_S}{T_P} = \frac{2.37s}{0.079s} \approx 30$$

PRODUCTO PUNTO (MAIN.CPP)

```
/*
Ejemplo tomado de:
Storti, D., & Yurtoglu, M. (2015).
CUDA for engineers: an introduction to high-performance parallel computing.
Addison-Wesley Professional.
Chapter 6.

Ejemplo de Sumatorias en paralelo usando CUDA
Calculo del producto punto entre dos vectores

*Usando funciones Atómicas para realizar la sumatoria final de cada bloque de hilos
*/

#include "kernel.h"
#include <stdio.h>
#include <stdlib.h>
 #define N 1024      //Probar con 1025

int main() {
    int cpu_res = 0;
    int gpu_res = 0;
    int *a = (int*)malloc(N * sizeof(int));
    int *b = (int*)malloc(N * sizeof(int));

    //Initialize input arrays
    for (int i = 0; i < N; ++i) {
        a[i] = 2;
        b[i] = 1;
    }
```

```
    //Serial CPU
    for (int i = 0; i < N; ++i) {
        cpu_res += a[i] * b[i];
    }
    printf("cpu result = %d\n", cpu_res);

    //Parallel GPU
    dotLauncher(&gpu_res, a, b, N);
    printf("gpu result = %d\n", gpu_res);

    system("pause");
    free(a);
    free(b);
    return 0;
}
```

PRODUCTO PUNTO (KERNEL.H)

```
1  #ifndef KERNEL_H
2  #define KERNEL_H
3
4  void dotLauncher(int *res, const int *a, const int *b, int n);
5
6  #endif
```

PRODUCTO PUNTO (KERNEL.CU)

```cpp
#include "kernel.h"
#include <stdio.h>
#define TPB 64
#define ATOMIC 1 // 0 for non-atomic addition

__global__
void dotKernel(int *d_res, const int *d_a, const int *d_b, int n) {
    const int idx = threadIdx.x + blockDim.x * blockIdx.x;
    if (idx >= n) return;
    const int s_idx = threadIdx.x;

    __shared__ int s_prod[TPB];
    s_prod[s_idx] = d_a[idx] * d_b[idx];
    __syncthreads();//Sincroniza los hilos del mismo bloque

    if (s_idx == 0) {
        int blockSum = 0;
        for (int j = 0; j < blockDim.x; ++j) {
            blockSum += s_prod[j];
        }
        printf("Block_%d, blockSum = %d\n", blockIdx.x, blockSum);
        // Try each of two versions of adding to the accumulator
        if (ATOMIC) {
            atomicAdd(d_res, blockSum);
        }
        else {
            *d_res += blockSum; //Resultados no esperados
        }
    }
}
```

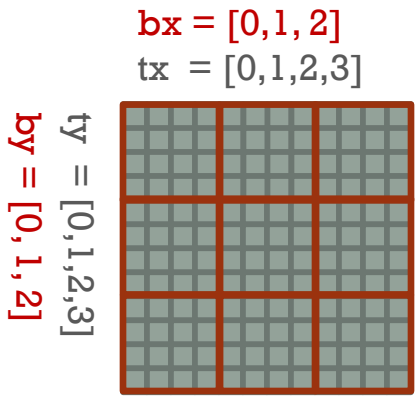
```cpp
void dotLauncher(int *res, const int *a, const int *b, int n) {
    int *d_res;
    int *d_a = 0;
    int *d_b = 0;

    cudaMalloc(&d_res, sizeof(int));
    cudaMalloc(&d_a, n * sizeof(int));
    cudaMalloc(&d_b, n * sizeof(int));

    cudaMemset(d_res, 0, sizeof(int));
    cudaMemcpy(d_a, a, n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, n * sizeof(int), cudaMemcpyHostToDevice);

    dotKernel << <(n + TPB - 1) / TPB, TPB >> >(d_res, d_a, d_b, n);
    //dotKernel_V2 << <(n + TPB - 1) / TPB, TPB >> >(d_res, d_a, d_b, n);
    cudaMemcpy(res, d_res, sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(d_res);
    cudaFree(d_a);
    cudaFree(d_b);
}
```

PRODUCTO PUNTO (KERNEL.CU) V2

```
32   __global__
33  void dotKernel_V2(int *d_res, const int *d_a, const int *d_b, int n) {
34      const int idx = threadIdx.x + blockDim.x * blockIdx.x;
35      if (idx >= n) return;
36      const int s_idx = threadIdx.x;
37
38      int blockSum = 0;
39      __shared__ int s_prod[TPB];
40      s_prod[s_idx] = blockSum=d_a[idx] * d_b[idx];
41      __syncthreads();
42
43      for (unsigned int s = blockDim.x / 2; s>0; s >>= 1)
44      {
45          if (s_idx < s)
46          {
47              s_prod[s_idx] = blockSum = blockSum + s_prod[s_idx + s];
48          }
49
50          __syncthreads();
51      }
52      if (s_idx == 0) {
53          printf("Block_%d, blockSum = %d\n", blockIdx.x, blockSum);
54          // Try each of two versions of adding to the accumulator
55          if (ATOMIC) {
56              atomicAdd(d_res, blockSum);
57          }
58          else {
59              *d_res += blockSum; //Resultados no esperados
60          }
61      }
62  }
```

MULTIPLICACIÓN DE MATRICES USANDO MEMORIA COMPARTIDA

```
//Multiplicacion de Matrices en Memoria Compartida (SM)
//Ver SDK (matrixMul), Each block must be contain BLOCK_SIZE*BLOCK_SIZE threads
__global__ void Multiplica_Matrices_SM(float *C,float *A,float *B,
                                       int nfil,int ncol)
{
    //Indices de Bloques
    int bx = blockIdx.x;
    int by = blockIdx.y;
    //Indices de Hilos
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Indice de la primer submatriz A procesada por el bloque
    int aBegin = ncol * BLOCK_SIZE * by;
    // Indice de la ultima submatriz A procesada por el bloque
    int aEnd   = aBegin + ncol - 1;
    // Tamaño de paso para iterar sobre las submatrices de A
    int aStep  = BLOCK_SIZE;
    // Indice de la primer submatriz B procesada por el bloque
    int bBegin = BLOCK_SIZE * bx;
    // Tamaño de paso para iterar sobre las submatrices de B
    int bStep  = BLOCK_SIZE * ncol;
    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float sum_sub = 0.0f;
```

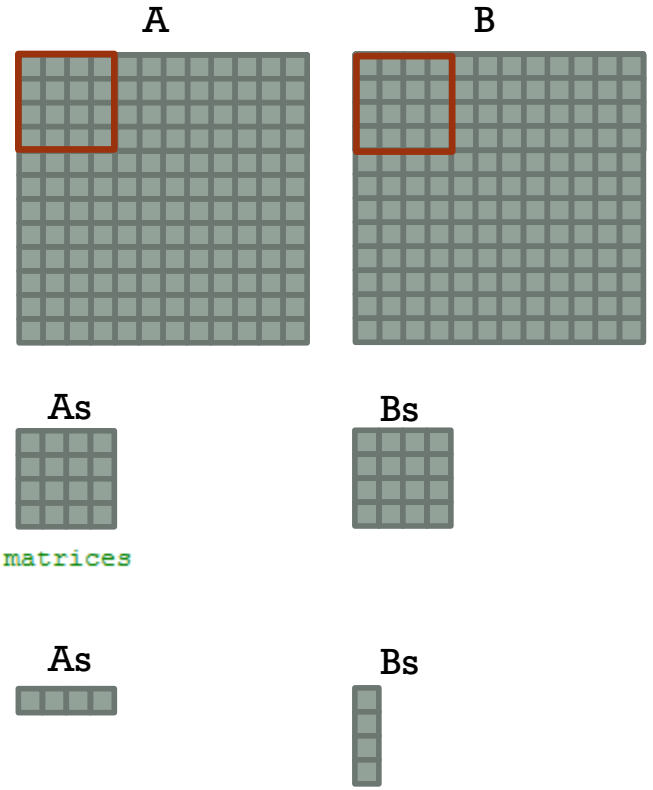


BLOCK_SIZE=4
aBegin=[0,48,96]
aEnd =[11,59,107]
aStep=4
bBegin=[0,4,8]
bStep=48

Ojo: esta variable (acumuladora) está en memoria local para cada hilo lanzado

MULTIPLICACIÓN DE MATRICES USANDO MEMORIA COMPARTIDA (C1)

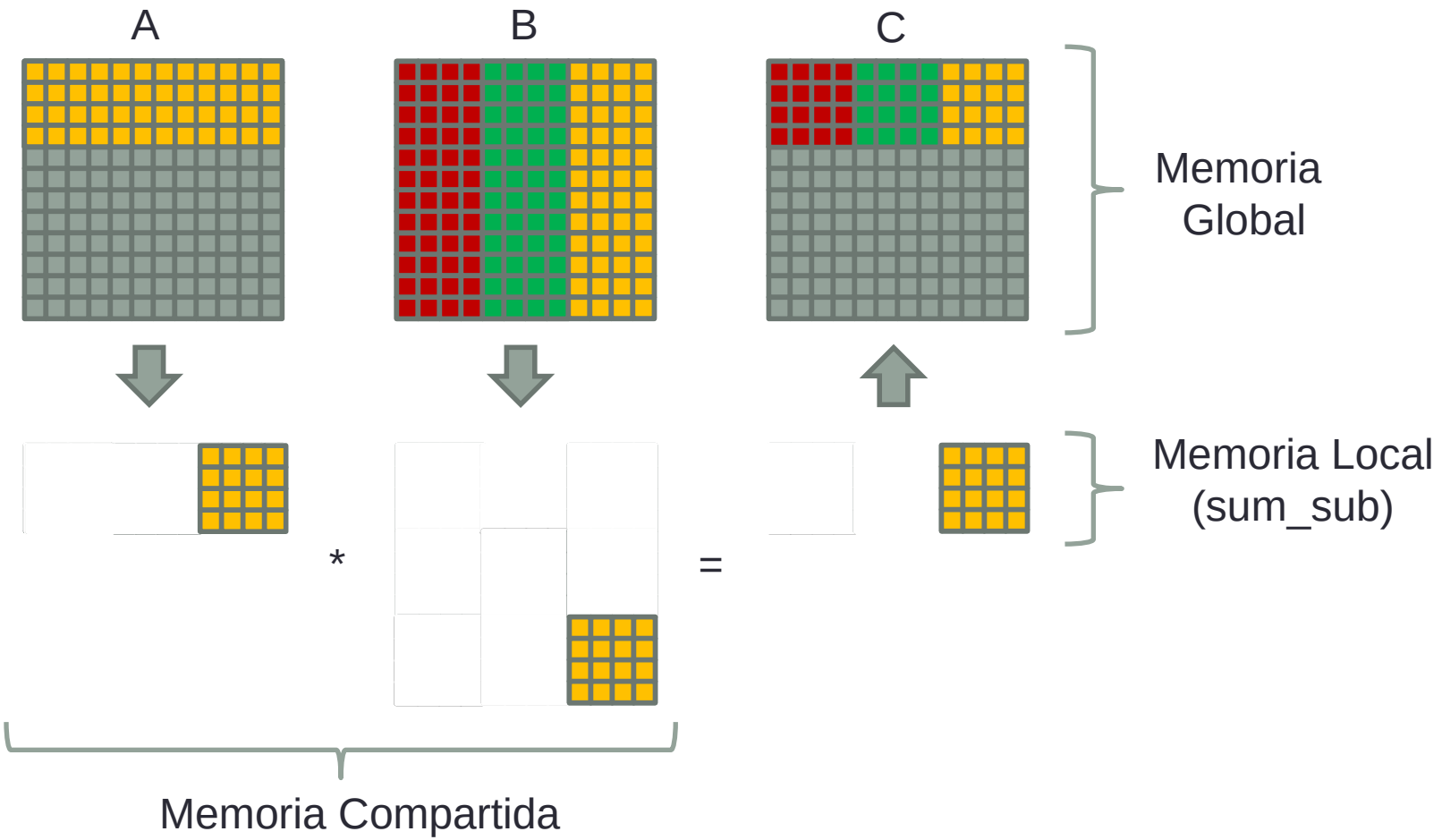
```
// Ciclo sobre todas las submatrices de A y B
for (int a = aBegin,b = bBegin;a <= aEnd;a += aStep, b += bStep) {
    //Memoria compartida para la submatriz A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    //Memoria compartida para la submatriz B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Almacenar las matrices desde la memoria global
    // a la memoria compartida; cada hilo almacena
    // un elemento de cada matriz
    As[ty][tx] = A[a + ncol * ty + tx];
    Bs[ty][tx] = B[b + ncol * ty + tx];
    // Sincronizamos los hilos para asegurar que se han cargado las matrices
    __syncthreads();

    // Multiplicamos las dos matrices
    #pragma unroll
    for (int k = 0; k < BLOCK_SIZE; k++)
        sum_sub += As[ty][k] * Bs[k][tx];
    // Sincronizamos para asegurar que el calculo anterior
    // se halla completado, antes de almacenar las nuevas submatrices
    __syncthreads();
}

// Guardamos el resultado en la memoria global
// Cada hilo guarda un elemento
int c = ncol * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + ncol * ty + tx] = sum_sub;
}
```



sum_sub va acumulando los productos puntos hasta llegar al último bloque.

Al final se le asigna el resultado de **sum_sub** a la correspondiente posición en la matriz resultante **C**.

MULTIPLICACIÓN DE MATRICES USANDO MEMORIA COMPARTIDA (C2)



GRACIAS POR SU ATENCIÓN

Francisco J. Hernández-López

fcoj23@cimat.mx

WebPage:

www.cimat.mx/~fcoj23