

EJEMPLOS USANDO CUDA

Francisco J. Hernández López

fcoj23@cimat.mx



HELLO WORLD

```
#include <stdio.h>

// printf() is only supported
// for devices of compute capability 2.0 and higher

__global__ void helloCUDA(float e){
    printf("Hello, I am thread %d of block %d with value e=%f\n", threadIdx.x, blockIdx.x, e);
}

int main(int argc, char **argv){

    helloCUDA<<<3, 4>>>(2.71828f);

    cudaDeviceReset();//is called to reinitialize the device.
    system("pause");
    return(0);
}
```

```
Hello, I am thread 0 of block 1 with value e=2.718280
Hello, I am thread 1 of block 1 with value e=2.718280
Hello, I am thread 2 of block 1 with value e=2.718280
Hello, I am thread 3 of block 1 with value e=2.718280
Hello, I am thread 0 of block 0 with value e=2.718280
Hello, I am thread 1 of block 0 with value e=2.718280
Hello, I am thread 2 of block 0 with value e=2.718280
Hello, I am thread 3 of block 0 with value e=2.718280
Hello, I am thread 0 of block 2 with value e=2.718280
Hello, I am thread 1 of block 2 with value e=2.718280
Hello, I am thread 2 of block 2 with value e=2.718280
Hello, I am thread 3 of block 2 with value e=2.718280
Presione una tecla para continuar . . . _
```

COMPILANDO HELLO WORLD

- Windows:

- Agregamos al path del sistema:

- c:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\x86_amd64

- nvcc** CUDA_HelloWorld.cu -o CUDA_HelloWorld.exe

- Linux:

- nvcc** -I /usr/local/cuda/include/ -I

- ~/NVIDIA_GPU_Computing_SDK/C/common/inc/ -L

- /usr/local/cuda/lib64/ -lcuda -lcudart CUDA_HelloWorld.cu -o CUDA_HelloWorld

Printf(): dentro de un kernel, solo es soportado por GPUs con capacidad ≥ 2.0 . Entonces no hay que olvidar ponerle **-arch=compute_20** al compilador nvcc.

CONOCIENDO LOS ÍNDICES DE LOS HILOS

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cuda_runtime.h>
4 #include <device_launch_parameters.h>
```

```
5
6 __global__ void idThreads_kernel(int *block_dev,
7     int *threadLocal_dev,
8     int *warp_dev,
9     int *threadGlobal_dev) { ... }
```

} Función Kernel

```
20
21 #define TAM 128
22 // #define TAM 140
```

```
23
24 int main(void){
25     int num_blocks = 4;
26     int num_threads = 32;
27     //int num_threads = 35;
28
29     /*Arreglos estáticos en el host*/
30     int block_host[TAM];
31     int threadLocal_host[TAM];
32     int warp_host[TAM];
33     int threadGlobal_host[TAM];
```

Se propone un malla unidimensional



CONOCIENDO LOS ÍNDICES DE LOS HILOS

```
35  /*Arreglos dinámicos en el device*/
36  int *block_dev;
37  int *threadLocal_dev;
38  int *warp_dev;
39  int *threadGlobal_dev;
40  /*Crear memoria dinámica en el device*/
41  size_t TAM_Bytes_int=TAM*sizeof(int);
42  cudaMalloc((void*)&block_dev, TAM_Bytes_int);
43  cudaMalloc((void*)&threadLocal_dev, TAM_Bytes_int);
44  cudaMalloc((void*)&warp_dev, TAM_Bytes_int);
45  cudaMalloc((void*)&threadGlobal_dev, TAM_Bytes_int);
46
47  /*Inicializar con -1 en el device*/
48  cudaMemset(threadGlobal_dev,-1,TAM_Bytes_int);
49  cudaMemset(threadLocal_dev, -1, TAM_Bytes_int);
50  cudaMemset(warp_dev, -1, TAM_Bytes_int);
51  cudaMemset(block_dev, -1, TAM_Bytes_int);
52
53  /*Ejecutar Kernel*/
54  idThreads_kernel<<<num_blocks, num_threads>>>(block_dev, threadLocal_dev, warp_dev,threadGlobal_dev);
55
56  /*Copiar memoria desde el device al host*/
57  cudaMemcpy(block_host, block_dev, TAM_Bytes_int, cudaMemcpyDeviceToHost);
58  cudaMemcpy(threadLocal_host, threadLocal_dev, TAM_Bytes_int, cudaMemcpyDeviceToHost);
59  cudaMemcpy(warp_host, warp_dev, TAM_Bytes_int, cudaMemcpyDeviceToHost);
60  cudaMemcpy(threadGlobal_host, threadGlobal_dev, TAM_Bytes_int, cudaMemcpyDeviceToHost);
61
62  /*Liberar memoria*/
63  cudaFree(block_dev);
64  cudaFree(threadLocal_dev);
65  cudaFree(warp_dev);
66  cudaFree(threadGlobal_dev);
```

Se ejecuta la malla

Malla de bloques de hilos

32 hilos

32 hilos

32 hilos

32 hilos

SUMA DE VECTORES

- Suma de vectores ($c = a + b$)
 - Creamos los vectores “a_h”, “b_h” y “c_h” en el host.
 - Inicializamos con cualquier valor los vectores “a_h” y “b_h”
 - Creamos los vectores “a_d”, “b_d” y “c_d” en el device
 - Copiamos el contenido de los vectores a y b del host al device
 - Sumamos a y b; el resultado lo guardamos en c; en el device
 - Copiamos el resultado del device al host
 - Desplegamos la suma de los vectores a y b

SUMA DE VECTORES (C1)

```
// Código principal que se ejecuta en el Host
int main(void){
    float *a_h, *b_h, *c_h; //Punteros a arreglos en el Host
    float *a_d, *b_d, *c_d; //Punteros a arreglos en el Device
    const int N = 24; //Número de elementos en los arreglos (probar 1000000)

    size_t size=N * sizeof(float);

    a_h = (float *)malloc(size); // Pedimos memoria en el Host
    b_h = (float *)malloc(size);
    c_h = (float *)malloc(size);//También se puede con cudaMallocHost

    //Inicializamos los arreglos a,b en el Host
    for (int i=0; i<N; i++){
        a_h[i] = (float)i;
        b_h[i] = (float)(i+1);
    }

    printf("\nArreglo a:\n");
    for (int i=0; i<N; i++) printf("%f ", a_h[i]);
    printf("\n\nArreglo b:\n");
    for (int i=0; i<N; i++) printf("%f ", b_h[i]);

    cudaMalloc((void **) &a_d, size); // Pedimos memoria en el Device
    cudaMalloc((void **) &b_d, size);
    cudaMalloc((void **) &c_d, size);

    //Pasamos los arreglos a y b del Host al Device
    cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
```

SUMA DE VECTORES (C2)

```
//Realizamos el cálculo en el Device
int block_size =8;
int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);

Suma_vectores <<< n_blocks, block_size >>> (c_d,a_d,b_d,N);

//Pasamos el resultado del Device al Host
cudaMemcpy(c_h, c_d, size,cudaMemcpyDeviceToHost);

//Resultado
printf("\n\nArreglo c:\n");
for (int i=0; i<N; i++) printf("%f ", c_h[i]);

_getche();

// Liberamos la memoria del Host
free(a_h);
free(b_h);
free(c_h);

// Liberamos la memoria del Device
cudaFree(a_d);
cudaFree(b_d);
cudaFree(c_d);
return(0);
}
```

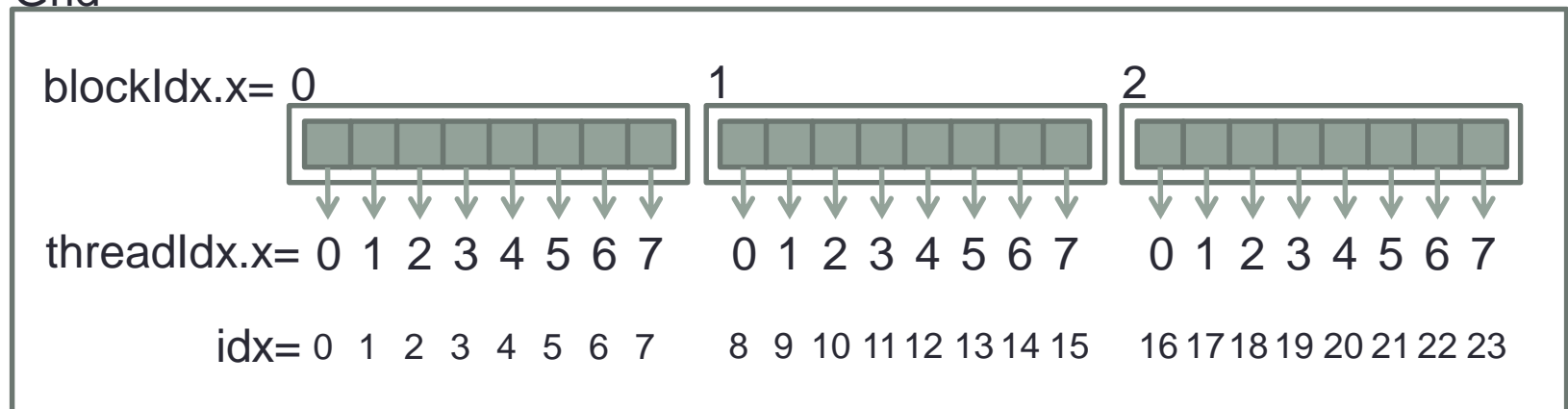
```
// Función Kernel que se ejecuta en el Device.
__global__ void Suma_vectores(float *c,float *a,float *b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx<N){
        c[idx] = a[idx] + b[idx];
    }
}
```


SUMA DE VECTORES (C3)

$idx = blockIdx.x * blockDim.x + threadIdx.x$

$N=24$ y $blockDim.x=8$

Grid



MULTIPLICACIÓN DE MATRICES

```
// Código principal que se ejecuta en el Host
int main(void) {
    float *A_h,*B_h,*C_h; //Punteros a matrices en el Host
    float *A_d,*B_d,*C_d; //Punteros a matrices en el Device
    int nfil = 12; //Número de filas
    int ncol = 12; //Número de columnas
    int N=nfil*ncol; //Número de elementos de la matriz

    //GPU Time
    cudaEvent_t start, stop;
    float time;

    size_t size=N * sizeof(float);

    A_h = (float *)malloc(size); // Pedimos memoria en el Host
    B_h = (float *)malloc(size);
    C_h = (float *)malloc(size);//También se puede con cudaMallocHost

    //Inicializamos las matrices a,b en el Host
    for (int i=0; i<nfil; i++){
        for(int j=0;j<ncol;j++){
            A_h[i*ncol+j] = 1.0f;
            B_h[i*ncol+j] = 2.0f;
        }
    }

    cudaMalloc((void **) &A_d,size); // Pedimos memoria en el Device
    cudaMalloc((void **) &B_d,size);
    cudaMalloc((void **) &C_d,size);

    //Pasamos las matrices a y b del Host al Device
    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);
}
```

MULTIPLICACIÓN DE MATRICES (C1)

```
//Realizamos el cálculo en el Device
dim3 block_size(BLOCK_SIZE,BLOCK_SIZE);
dim3 n_blocks(div_up(ncol,block_size.x),div_up(nfil,block_size.y)) ;
Multiplica_Matrices_GM<<< n_blocks, block_size >>> (C_d,A_d,B_d,nfil,ncol);

//Pasamos el resultado del Device al Host
cudaMemcpy(C_h, C_d, size,cudaMemcpyDeviceToHost);

//Resultado
printf("\n\nMatriz c:\n");
for (int i=0; i<10; i++){
    for(int j=0;j<10;j++){
        printf("%.2f ", C_h[i*ncol+j]);
    }
    printf("\n");
}

// Liberamos la memoria del Host
free(A_h);
free(B_h);
free(C_h);

// Liberamos la memoria del Device
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
return(0);
```

```
//Multiplicacion de Matrices en Memoria Global (GM)
__global__ void Multiplica_Matrices_GM(float *C,float *A,float *B,
                                       int nfil,int ncol)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int index=idy*ncol+idx;
    if (idy<nfil && idx<ncol){
        float sum=0.0f;
        for(int k=0;k<ncol;k++){
            sum+=A[idy*ncol+k]*B[k*ncol+idx];
        }
        C[index] = sum;
    }
}
```

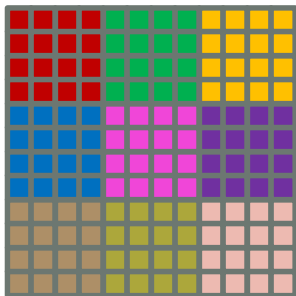
MULTIPLICACIÓN DE MATRICES (C2)

$idx = blockIdx.x * blockDim.x + threadIdx.x$

$idy = blockIdx.y * blockDim.y + threadIdx.y$

$nfil=12, ncol=12, BLOCK_SIZE=4$

Grid



$blockIdx.x=\{0,1,2\}$

$blockIdx.y=\{0,1,2\}$

$threadIdx.x=\{0,1,2,3\}$

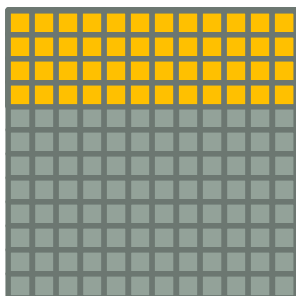
$threadIdx.y=\{0,1,2,3\}$

$idx=\{0,1,2,\dots,11\}$

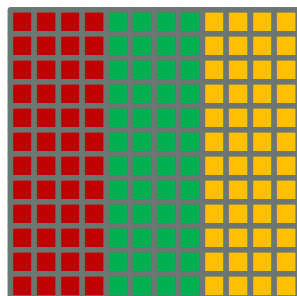
$idy=\{0,1,2,\dots,11\}$

```
//Multiplicacion de Matrices en Memoria Global (GM)
__global__ void Multiplica_Matrices_GM(float *C,float *A,float *B,
int nfil,int ncol)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int index=idy*ncol+idx;
    if (idy<nfil && idx<ncol){
        float sum=0.0f;
        for(int k=0;k<ncol;k++){
            sum+=A[idy*ncol+k]*B[k*ncol+idx];
        }
        C[index] = sum;
    }
}
```

A



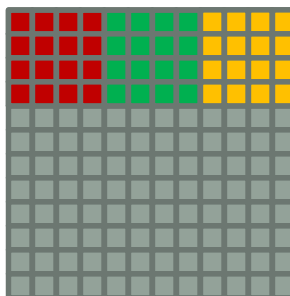
B



*

=

C



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;
```

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;
}
```

MULTIPLICACIÓN DE MATRICES USANDO MEMORIA COMPARTIDA (C2)

