



INSTITUTO POLITÉCNICO NACIONAL  
ESCUELA SUPERIOR DE CÓMPUTO



**Unidad de Aprendizaje: Analysis and Design of Parallel Algorithms.**

**“Reporte Práctica 2 - CUDA.”**

Gabriela Moreno González.

**Profa.** Sandra Luz Morales Guitrón.

3CV9.

## Programa 1. Hello World en CUDA.

Primeramente se realizó el código para hacer un HolaMundo muy similar al Hola Mundo hecho en CUDA en la práctica 1, pero este no te saludaba únicamente de una sola parte, sino que también te permitía ver varias veces el mensaje. El código completo del programa es el siguiente:

```
1  #include <stdio.h>
2
3  __global__ void helloCUDA(float e)
4  {
5      printf("Hello, I am thread %d of block %d with value e = %f\n", threadIdx.x, blockIdx.x, e);
6  }
7
8  int main(int argc, char **argv)
9  {
10     helloCUDA<<<3, 4>>>(2.71828f);
11
12     cudaDeviceReset();
13     system("pause");
14     return(0);
15 }
```

Así, corriendo el programa obtenemos lo siguiente:

```
Símbolo del sistema
Microsoft Windows [Versión 6.3.9600]
(c) 2013 Microsoft Corporation. Todos los derechos reservados.

C:\Users\alumno>cd Desktop
C:\Users\alumno\Desktop>cd "Práctica 2 - CUDA"
C:\Users\alumno\Desktop\Práctica 2 - CUDA>nvcc Hello_World.cu -o Hello_World.exe
Hello_World.cu
Creando biblioteca Hello_World.lib y objeto Hello_World.exp

C:\Users\alumno\Desktop\Práctica 2 - CUDA>Hello_World.exe
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
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 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
Presione una tecla para continuar . . .

C:\Users\alumno\Desktop\Práctica 2 - CUDA>_
```

## Programa 2. Suma de vectores.

Como lo hicimos en MPI, también en CUDA podemos sumar vectores usando prácticamente el mismo proceso pero transformando nuestro código al lenguaje CUDA, por lo que el código que nos permite sumar vectores es el siguiente:

```
Suma_vectores.cu x
1  #include <stdio.h>
2
3  __global__ void Suma_vectores(float *c, float *a, float *b, int N)
4  {
5      int idx = blockIdx.x * blockDim.x + threadIdx.x;
6
7      if(idx < N)
8      {
9          c[idx] = a[idx] + b[idx];
10     }
11 }
12
13 int main(void)
14 {
15     float *a_h, *b_h, *c_h;
16     float *a_d, *b_d, *c_d;
17     const int N = 24;
18
19     size_t size = N * sizeof(float);
20
21     a_h = (float *)malloc(size);
22     b_h = (float *)malloc(size);
23     c_h = (float *)malloc(size);
24
25     for(int i=0; i<N; i++)
26     {
27         a_h[i] = (float)i;
28         b_h[i] = (float)(i + 1);
29     }
30
31     printf("\nArreglo a:\n");
32
33     for(int i=0; i<N; i++)
34     {
```

```

35     printf("%f ", a_h[i]);
36 }
37
38 printf("\nArreglo b:\n");
39
40 for(int i=0; i<N; i++)
41 {
42     printf("%f ", b_h[i]);
43 }
44
45 cudaMalloc((void **) &a_d,size);
46 cudaMalloc((void **) &b_d,size);
47 cudaMalloc((void **) &c_d,size);
48
49 cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
50 cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
51
52 int block_size = 8;
53 int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
54
55 Suma_vectores <<< n_blocks, block_size >>> (c_d, a_d, b_d, N);
56
57 cudaMemcpy(c_h, c_d, size, cudaMemcpyDeviceToHost);
58
59 printf("\n\nArreglo c:\n");
60
61 for(int i=0; i<N; i++)
62 {
63     printf("%f ", c_h[i]);
64 }
65
66 printf("\n");
67 system("pause");
68
69 free(a_h);
70 free(b_h);
71 free(c_h);
72
73 cudaFree(a_d);
74 cudaFree(b_d);
75 cudaFree(c_d);
76
77 return(0);
78 }

```

Y corriendo el programa obtenemos la siguiente salida:

```
Símbolo del sistema
C:\Users\alumno\Desktop\Práctica 2 - CUDA>nvcc Suma_vectores.cu -o Suma_vectores.exe
Suma_vectores.cu
Creando biblioteca Suma_vectores.lib y objeto Suma_vectores.exp
C:\Users\alumno\Desktop\Práctica 2 - CUDA>Suma_vectores.exe
Arreglo a:
0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
9.000000 10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000
17.000000 18.000000 19.000000 20.000000 21.000000 22.000000 23.000000
Arreglo b:
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000 17.000000
18.000000 19.000000 20.000000 21.000000 22.000000 23.000000 24.000000
Arreglo c:
1.000000 3.000000 5.000000 7.000000 9.000000 11.000000 13.000000 15.000000 17.000000
19.000000 21.000000 23.000000 25.000000 27.000000 29.000000 31.000000 33.000000
35.000000 37.000000 39.000000 41.000000 43.000000 45.000000 47.000000
Presione una tecla para continuar . . .
C:\Users\alumno\Desktop\Práctica 2 - CUDA>
```

### Programa 3. Multiplicación de matrices.

El 3er y último programa de la práctica indica que hagamos una multiplicación de matrices, similar de nuevo al hecho en MPI, pero esta vez convirtiéndolo en CUDA, por lo que nuestro código es el siguiente:

```
Multiplica_matrices.cu x
1 #include <stdio.h>
2
3 __global__ void Multiplca_Matrices_GM(float *C,float *A,float *B,int nfil,int ncol)
4 {
5     int idx = blockIdx.x * blockDim.x + threadIdx.x;
6     int idy = blockIdx.y * blockDim.y + threadIdx.y;
7     int index = idy*ncol+idx;
8
9     if(idy<nfil && idx<ncol)
10    {
11        float sum=0.0f;
12
13        for(int k=0;k<ncol;k++)
14        {
15            sum+=A[idy*ncol+k]*B[k*ncol+idx];
16        }
17        C[index] = sum;
18    }
19 }
20
21 int div_up(int a, int b)
22 {
23     if (a % b) /* does a divide b leaving a remainder? */
24         return a / b + 1; /* add in additional block */
25     else
26         return a / b; /* divides cleanly */
27 }
28
29 int main(void)
30 {
31     float *A_h,*B_h,*C_h;
32     float *A_d,*B_d,*C_d;
33     int nfil = 12;
34     int ncol = 12;
```

```

35     int BLOCK_SIZE = 4;
36     int N=nfil*ncol;
37
38     size_t size=N * sizeof(float);
39
40     A_h = (float *)malloc(size);
41     B_h = (float *)malloc(size);
42     C_h = (float *)malloc(size);
43
44     for(int i=0; i<nfil; i++)
45     {
46         for(int j=0;j<ncol;j++)
47         {
48             A_h[i*ncol+j] = 1.0f;
49             B_h[i*ncol+j] = 2.0f;
50         }
51     }
52
53     cudaMalloc((void **) &A_d, size);
54     cudaMalloc((void **) &B_d, size);
55     cudaMalloc((void **) &C_d, size);
56
57     cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
58     cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);
59
60     dim3 block_size(BLOCK_SIZE,BLOCK_SIZE);
61     dim3 n_blocks(div_up(ncol,block_size.x),div_up(nfil,block_size.y));
62
63     Multiplca_Matrices_GM<<< n_blocks, block_size >>> (C_d,A_d,B_d,nfil,ncol);
64
65     cudaMemcpy(C_h,C_d,size,cudaMemcpyDeviceToHost);
66
67     printf("\n\nMatriz c:\n");
68

```

```

69     for(int i=0; i<10; i++)
70     {
71         for(int j=0; j<10; j++)
72         {
73             printf("%.2f ", C_h[i*ncol+j]);
74         }
75         printf("\n");
76     }
77
78     free(A_h);
79     free(B_h);
80     free(C_h);
81
82     cudaFree(A_d);
83     cudaFree(B_d);
84     cudaFree(C_d);
85
86     return(0);
87 }

```

Ahora si, corriendo nuestro programa obtenemos lo siguiente:

```
Símbolo del sistema
C:\Users\alumno\Desktop\Práctica 2 - CUDA>nvcc Suma_vectores.cu -o Suma_vectores.exe
Suma_vectores.cu
  Creando biblioteca Suma_vectores.lib y objeto Suma_vectores.exp
C:\Users\alumno\Desktop\Práctica 2 - CUDA>Suma_vectores.exe
Arreglo a:
0.000000 1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
9.000000 10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000
17.000000 18.000000 19.000000 20.000000 21.000000 22.000000 23.000000
Arreglo b:
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000 9.000000
10.000000 11.000000 12.000000 13.000000 14.000000 15.000000 16.000000 17.000000
18.000000 19.000000 20.000000 21.000000 22.000000 23.000000 24.000000
Arreglo c:
1.000000 3.000000 5.000000 7.000000 9.000000 11.000000 13.000000 15.000000 17.000000
19.000000 21.000000 23.000000 25.000000 27.000000 29.000000 31.000000 33.000000
35.000000 37.000000 39.000000 41.000000 43.000000 45.000000 47.000000
Presione una tecla para continuar . . .
C:\Users\alumno\Desktop\Práctica 2 - CUDA>
```

## Conclusiones:

Las diferencias entre MPI y CUDA son prácticamente de la manera en la cual se escriben las instrucciones, además de que MPI solo trabaja en el procesador actual y CUDA nos permite emplear más de uno utilizando su GPU NVIDIA, sin embargo una vez entendiendo esto, es sencillo abstraer los problemas.