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.

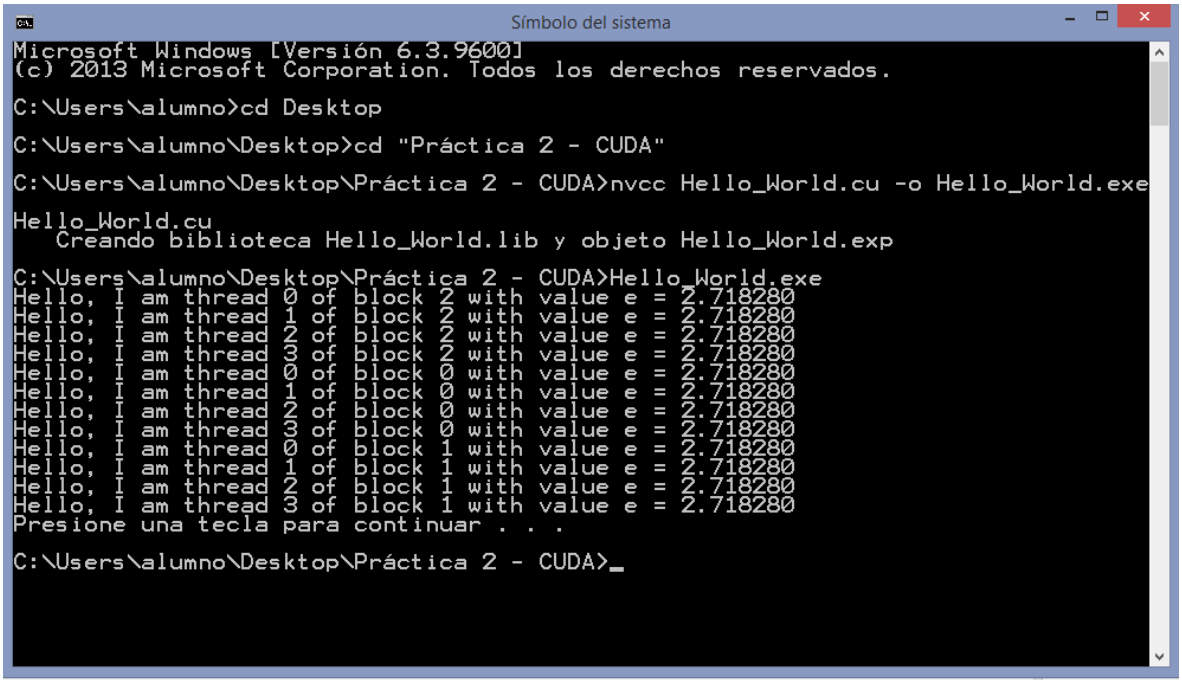
Programa 1. Hello World en CUDA.

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 unicamente de una sola parte, sino que también te permitía ver varias veces el mensaje. El código completo del programa es el siguiente:

Hello_World.cu

```
#include <stdio.h>

__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();
    system("pause");
    return(0);
}
```

Así, corriendo el programa obtenemos lo siguiente:



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:

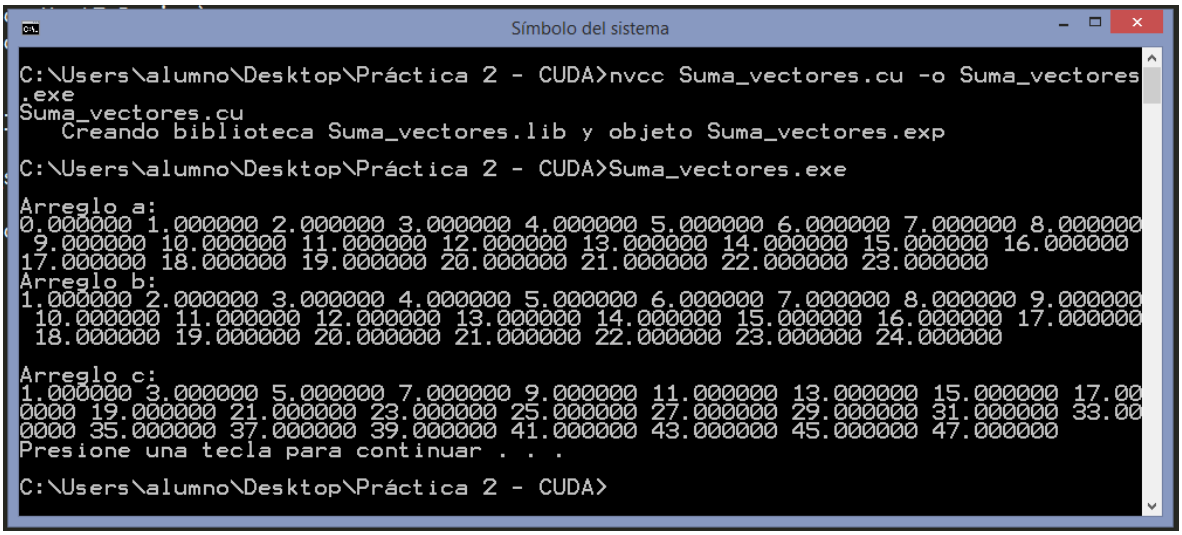
Suma_vectores.cu

```
#include <stdio.h>

__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];
    }
}

int main(void)
{
    float *a_h, *b_h, *c_h;
    float *a_d, *b_d, *c_d;
    const int N = 24;

    size_t size = N * sizeof(float);

    a_h = (float *)malloc(size);
    b_h = (float *)malloc(size);
    c_h = (float *)malloc(size);

    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++)
    {
```

```
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:



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

```
#include <stdio.h>

__global__ void Multiplca_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;
    }
}

int div_up(int a, int b)
{
    if (a % b)  /* does a divide b leaving a remainder? */
        return a / b + 1; /* add in additional block */
    else
        return a / b; /* divides cleanly */
}

int main(void)
{
    float *A_h,*B_h,*C_h;
    float *A_d,*B_d,*C_d;
    int nfil = 12;
    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:

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.