Practica CUDA Nº 2

M. en C. Sandra Luz Morales Güitrón.

Realice los siguientes ejercicios, realice su reporte y no olvide las conclusiones.

1.- 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);
}
```

2.- Suma de Vectores.

```
// 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);
```

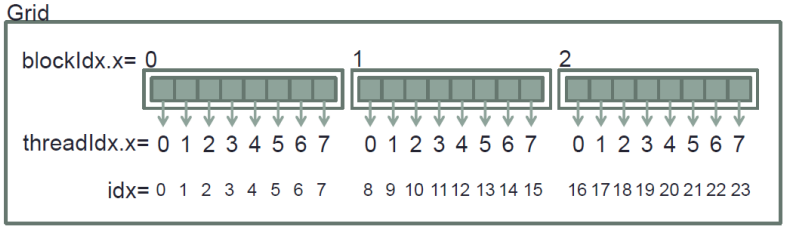
```
    //Realizamos el cálculo en el Device
    int block_size =8;
    int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);

    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];
  }
}
```

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

N=24 y blockDim.x= 8

Grid

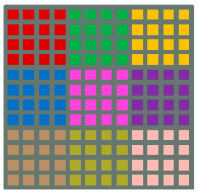


3.- Multiplicación de Matrices.

idx = blockIdx.x * blockDim.x + threadIdx.x
idy = blockIdx.y * blockDim.y + threadIdx.y

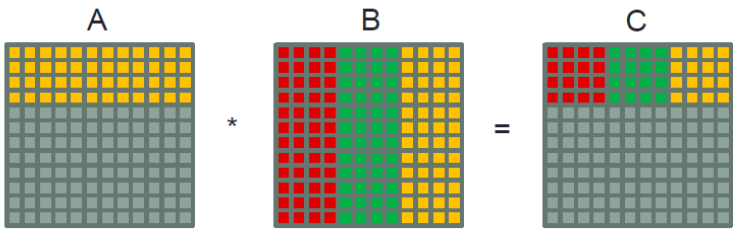
nfil=12, ncol=12, BLOCK_SIZE=4

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,…,11}
idy={0,1,2,…,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;
    }
}
```



```
// 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);

  //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;
    }
}
```