Universidad Nacional de San Antonio Abad del Cusco

Departamento Académico de Informática

ALGORITMOS PARALELOS Y DISTRIBUIDOS

Práctica Nº 7

COMPUTACION HETEROGENEA

1. OBJETIVO.

- Conocer el modelo de programación CUDA.
- Implementar y ejecutar programas paralelos utilizando la GPU.

2. INTRODUCCION.

2.1. Computación Heterogénea.

Podemos definir la computación sobre tarjetas gráficas (en inglés GPU computing) como el uso de una tarjeta gráfica (GPU - Graphics Processing Unit) para realizar cálculos científicos de propósito general. El modelo de computación sobre tarjetas gráficas consiste en usar conjuntamente una CPU (Central Processing Unit) y una GPU de manera que formen un modelo de computación heterogéneo (Figura 1). Siguiendo este modelo, la parte secuencial de una aplicación se ejecutaría sobre la CPU (comúnmente denominada host) y la parte más costosa del cálculo se ejecutaría sobre la GPU (que se denomina device). Desde el punto de vista del usuario, la aplicación simplemente se va a ejecutar más rápido porque está utilizando las altas prestaciones de la GPU para incrementar el rendimiento.

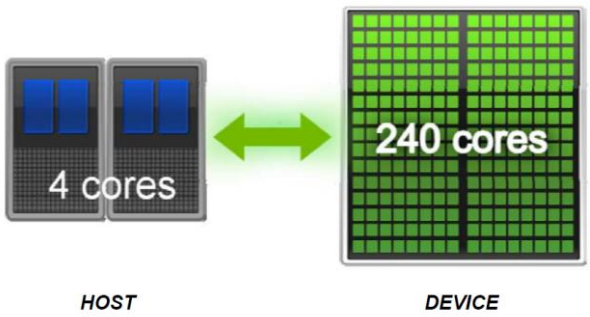


Fig.1 Modelo de computación en sistemas heterogéneos

2.2. Arquitectura CUDA.

En la Figura 2 se muestra la arquitectura de una tarjeta gráfica compatible con CUDA.

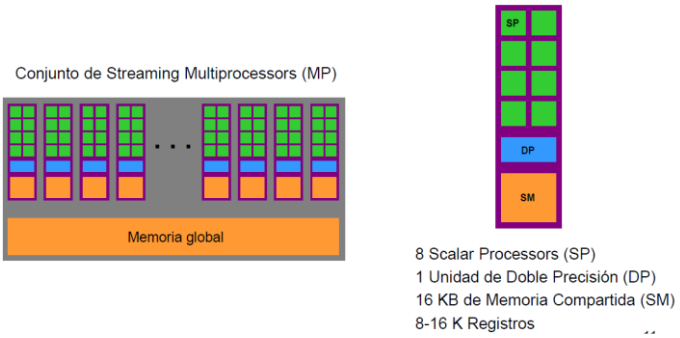


Fig. 2. Arquitectura CUDA

En ella se puede observar la presencia de unas unidades de ejecución denominadas Streaming Multiprocessors (SM), 8 unidades en el ejemplo de la figura, que están interconectadas entre sí por una zona de memoria común. Cada SM está compuesto a su vez por unos núcleos de cómputo llamados "núcleos CUDA" o Streaming Processors (SP), que son los encargados de ejecutar las instrucciones y que en nuestro ejemplo vemos que hay 8 núcleos por cada SM, lo que hace un total de 64 núcleos de procesamiento. Este diseño hardware permite la programación sencilla de los núcleos de la GPU utilizando un lenguaje de alto nivel como puede ser el lenguaje C para CUDA. De este modo, el programador simplemente escribe un programa secuencial dentro del cual se llama a lo que se conoce como kernel, que puede ser una simple función o un programa completo. Este kernel se ejecuta de forma paralela dentro de la GPU como un conjunto de hilos (threads) y que el programador organiza dentro de una jerarquía en la que pueden agruparse en bloques (blocks), y que a su vez se pueden distribuir formando una malla (grid), tal como se muestra en la Figura 3.

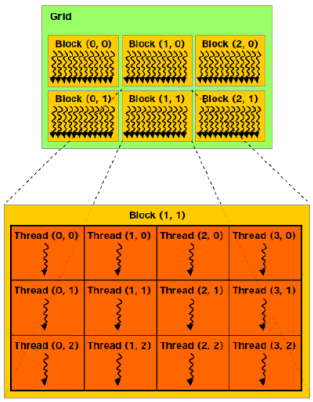


Fig. 3 Jerarquía de hilos

Por conveniencia los bloques y las mallas pueden tener una, dos o tres dimensiones. Existen multitud de situaciones en las que los datos con los que se trabaja poseen de forma natural una estructura de malla, pero en general, descomponer los datos en una jerarquía de hilos no es una tarea fácil. Así pues, un bloque de hilos es un conjunto de hilos concurrentes que pueden cooperar entre ellos a través de mecanismos de sincronización y compartir accesos a un espacio de memoria exclusivo de cada bloque. Y una malla es un conjunto de bloques que pueden ser ejecutados independientemente y que por lo tanto pueden ser lanzados en paralelo en los Streaming Multiprocessors (SM).

Cuando se invoca un kernel, el programador especifica el número de hilos por bloque y el número de bloques que conforman la malla. Una vez en la GPU, a cada hilo se le asigna un único número de identificación dentro de su bloque, y cada bloque recibe un identificador dentro de la malla. Esto permite que cada hilo decida sobre qué datos tiene que trabajar, lo que simplifica enormemente el direccionamiento de memoria cuando se trabaja con datos multidimensionales, como es el caso del procesado de imágenes o la resolución de ecuaciones diferenciales en dos y tres dimensiones. Otro aspecto a destacar en la arquitectura CUDA es la presencia de una unidad de distribución de trabajo que se encarga de distribuir los bloques entre los SM disponibles. Los hilos dentro de cada bloque se ejecutan concurrentemente y cuando un bloque termina, la unidad de distribución lanza nuevos bloques sobre los SM libres. Los SM mapean cada hilo sobre un núcleo SP, y cada hilo se ejecuta de manera independiente con su propio contador de programa y registros de estado. Dado que cada hilo tiene asignados sus propios registros, no existe penalización por los cambio de contexto, pero en cambio sí existe un límite en el número máximo de hilos activos debido a que cada SM tiene un número determinado de registros.

Una característica particular de la arquitectura CUDA es la agrupación de los hilos en grupos de 32. Un grupo de 32 hilos recibe el nombre de warp, y se puede considerar como la unidad de ejecución en paralelo, ya que todos los hilos de un mismo warp se ejecutan físicamente en paralelo y por lo tanto comienzan en la misma instrucción (aunque después son libres de bifurcarse y ejecutarse independientemente). Así, cuando se selecciona un bloque para su ejecución dentro de un SM, el bloque se divide en warps, se selecciona uno que esté listo para ejecutarse y se emite la siguiente instrucción a todos los hilos que forman el warp. Dado que todos ellos ejecutan la misma instrucción al unísono, la máxima eficiencia se consigue cuando todos los hilos coinciden en su ruta de ejecución (sin bifurcaciones). Aunque el programador puede ignorar este comportamiento, conviene tenerlo en cuenta si se pretende optimizar alguna aplicación.

En cuanto a la memoria, durante su ejecución los hilos pueden acceder a los datos desde diferentes espacios dentro de una jerarquía de memoria (Figura 4). Así, cada hilo tiene una zona privada de **memoria local** y cada bloque tiene una zona de **memoria compartida** visible por todos los hilos del mismo bloque, con un elevado ancho de banda y baja latencia (similar a una cache de nivel 1). Finalmente, todos los hilos tienen acceso a un mismo espacio de memoria global (del orden de MiB o GiB) ubicada en un chip externo de memoria DRAM. Dado que esta memoria posee una latencia muy elevada, es una buena práctica copiar los datos que van a ser accedidos frecuentemente a la zona de memoria compartida.

El modelo de programación CUDA asume que tanto el host como el device mantienen sus propios espacios separados de memoria. La única zona de memoria accesible desde el host es la memoria global. Además, tanto la reserva o liberación de memoria global como la transferencia de datos entre el host y el device debe hacerse de forma explícita desde el host mediante llamadas a funciones específicas de CUDA.

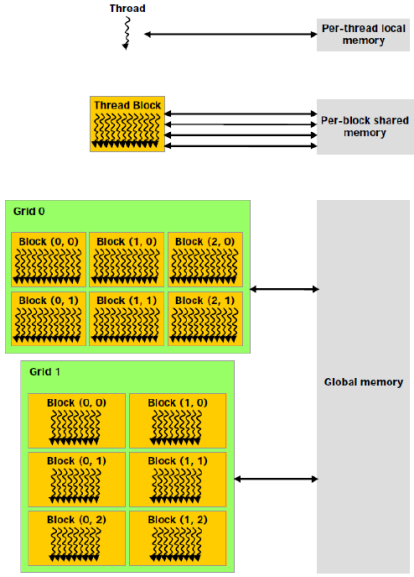


Fig. 4. Jerarquía de memoria

3. DESARROLLO DE LA PRÁCTICA

3.1. Ejercicio 1.

En el siguiente ejemplo se define un hilo por bloque, el hilo imprime el numero del bloque y el identificador del hilo.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

__global__ void helloworld()
{ printf("Hola Mundo..! Soy el hilo con id bloque: {%d %d}, id de hilo{ %d %d %d }\n",
          blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y, threadIdx.z);
}
int main()
{
    helloworld <<< 1, 1 >>> (); //--se define un hilo por bloque
    return 0;

}
```

3.2. Ejercicio 2.

En el siguiente ejemplo se define un bloque de hilos tridimensional y un grid bidimensional.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

__global__ void helloworld()
{ printf("Hola Mundo..! Soy el hilo con id bloque: {%d %d}, id de hilo{ %d %d %d }\n",
              blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y, threadIdx.z);
}
int main()
{
        dim3 threads(1, 2, 4); //--nro de hilos por bloque
        dim3 grid(2, 1); //--nro de bloques por grid
        helloworld <<< grid, threads >>> (); //--se define un hilo por bloque
        return 0;

 }
```

3.3. Ejercicio 3.

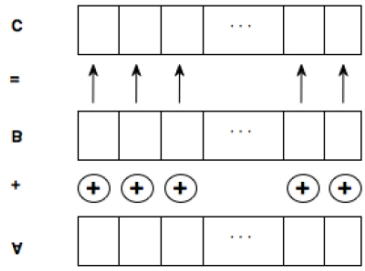
El siguiente ejercicio suma dos números enteros .

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <stdlib.h>

// declaracion de funciones
// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void suma_GPU(int a, int b, int* c)
{
        *c = a + b;
}
// HOST: funcion llamada y ejecutada desde el host
__host__ int suma_CPU(int a, int b)
{
        return (a + b);
}

int main()
{
        // declaraciones
        int n1 = 1, n2 = 2, c = 0;
        int* hst_c;
        int m1 = 10, m2 = 20;
        int* dev_c;
        // reserva en el host
        hst_c = (int*)malloc(sizeof(int));
        // reserva en el device
        cudaMalloc((void**)&dev_c, sizeof(int));
        // llamada a la funcion suma_CPU
        c = suma_CPU(n1, n2);
        // resultados CPU
        printf("CPU:\n");
        printf("%2d + %2d = %2d \n", n1, n2, c);
        // llamada a la funcion suma_GPU
        suma_GPU << <1, 1 >> > (m1, m2, dev_c);
        // recogida de datos desde el device
        cudaMemcpy(hst_c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
        // resultados GPU
        printf("GPU:\n");
        printf("%2d + %2d = %2d \n", m1, m2, *hst_c);
        // salida
        printf("\npulsa INTRO para finalizar...");
        fflush(stdin);
        char tecla = getchar();

        return 0;
}
```

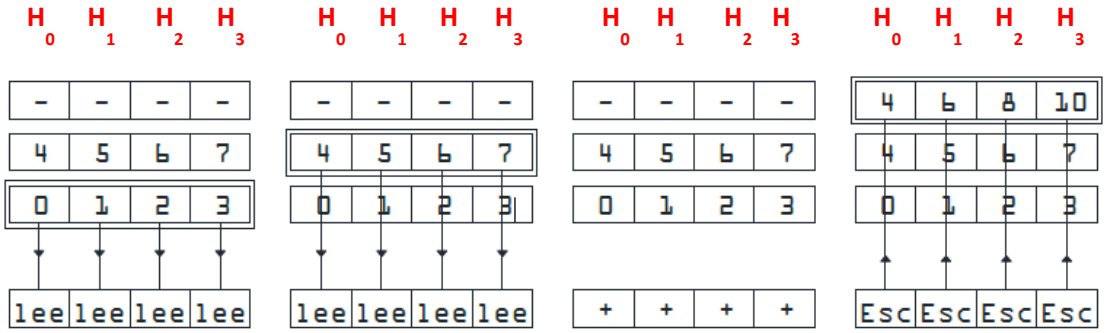
3.4. Ejercicio 4. Suma de dos vectores.

El programa secuencial para la suma de dos vectores es el siguiente:

```
#define N 10
void Suma_vec( int *a, int *b, int *c ) {
  int i = 0; // Los vectores van desde 0..N-1
  for (i=0; i < N; i++)
c[i] = a[i] + b[i];
}
int main( void ) {
  int a[N], b[N], c[N];
inicializa(a);    //Inicializa los vectores de entrada a y b
  inicializa(b);
  Suma_vec( a, b, c );
  mostrar(a,b,c);  // muestra el resultado
  return 0;
}
```



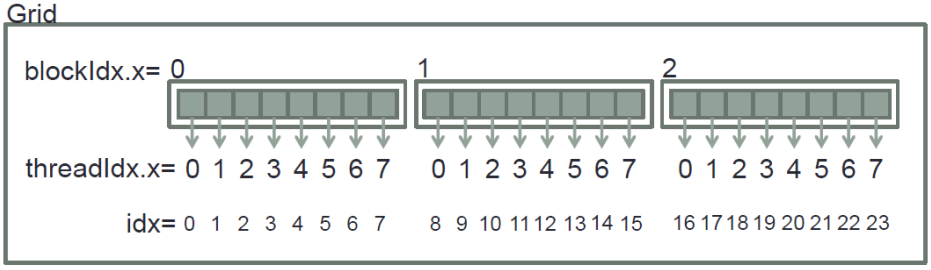
El diseño del programa paralelo es el siguiente:



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

N=24 y blockDim.x= 8

Grid




```
#include <stdio.h>
#include <stdlib.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#define N 16 // tamano de los vectores
#define BLOCK 5 // tamano del bloque
// declaracion de funciones
// GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
__global__ void suma(float* a, float* b, float* c)
{
	int myID = threadIdx.x + blockDim.x * blockIdx.x;
	// Solo trabajan N hilos
	if (myID < N)
	{
		c[myID] = a[myID] + b[myID];
	}
}
// MAIN: rutina principal ejecutada en el host
int main(int argc, char** argv)
{
	// declaraciones
	float* vector1, * vector2, * resultado;
	float* dev_vector1, * dev_vector2, * dev_resultado;
```

```
        // reserva en el host
        vector1 = (float*)malloc(N * sizeof(float));
        vector2 = (float*)malloc(N * sizeof(float));
        resultado = (float*)malloc(N * sizeof(float));
        // reserva en el device
        cudaMalloc((void**)&dev_vector1, N * sizeof(float));
        cudaMalloc((void**)&dev_vector2, N * sizeof(float));
        cudaMalloc((void**)&dev_resultado, N * sizeof(float));
        // inicializacion de vectores
        for (int i = 0; i < N; i++)
        {
                vector1[i] = (float)rand() / RAND_MAX;
                vector2[i] = (float)rand() / RAND_MAX;
        }
        // copia de datos hacia el device
        cudaMemcpy(dev_vector1, vector1, N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_vector2, vector2, N * sizeof(float), cudaMemcpyHostToDevice);
        // lanzamiento del kernel
        // calculamos el numero de bloques necesario para un tamaño de bloque fijo
        int nBloques = N / BLOCK;
        if (N % BLOCK != 0)
        {
                nBloques = nBloques + 1;
        }
        int hilosB = BLOCK;
        printf("Vector de %d elementos\n", N);
        printf("Lanzamiento con %d bloques (%d hilos)\n", nBloques, nBloques * hilosB);
        suma << < nBloques, hilosB >> > (dev_vector1, dev_vector2, dev_resultado);
        // recogida de datos desde el device
        cudaMemcpy(resultado, dev_resultado, N * sizeof(float), cudaMemcpyDeviceToHost);
        // impresion de resultados
        printf("> vector1:\n");
        for (int i = 0; i < N; i++)
        {
                printf("%.2f ", vector1[i]);
        }
        printf("\n");
        printf("> vector2:\n");
        for (int i = 0; i < N; i++)
        {
                printf("%.2f ", vector2[i]);
        }
        printf("\n");
        printf("> SUMA:\n");
        for (int i = 0; i < N; i++)
        {
                printf("%.2f ", resultado[i]);
        }
        printf("\n");
        // liberamos memoria en el device
        cudaFree(dev_vector1);
        cudaFree(dev_vector2);
        cudaFree(dev_resultado);
        // salida
        printf("\npulsa INTRO para finalizar...");
        fflush(stdin);
        char tecla = getchar();
        return 0;

}
```

1. TAREA

1.1. Escribir un programa CUDA para suma dos matrices.

1.2. Escribir un programa CUDA para multiplicar una matriz por un vector.

2. BIBLIOGRAFIA.

- César Represa Pérez. (2016) Introducción a la Programación en CUDA. Universidad de Burgos.
- Rob Farber. (2011). CUDA Application Design and Development. Ed. Elsevier

- Jason Sanders. (2011) CUDA By Example. Addison Wesley.
- John Cheng, Max Grossman, Ty McKercher (2014) Professional CUDA Programming. Ed. Wrox.