CUDA

INTRODUCCIÓN, HILOS, BLOQUES Y GRIDS

GPU(*Graphic Processing Units*).

CUDA(*Compute Unified Device Architecture*). Interfaz de programación.

Al comienzo de las GPUs fueron many-cores con docenas o cientos de unidades de procesamiento simple → especializados en el cálculo de objetos gráficos (imágenes, video) → Progresivamente han estado más flexibles y programables.

- En el 2008, NVIDIA introdujo G80, la primera GPU suministrada con una nueva interfaz de programación y ejecución llamada CUDA.
- Con CUDA se pueden escribir programas usando la CPU para la parte secuencial del programa y la GPU para la parte paralela corriendo en múltiples hilos.

Ejemplo de aplicaciones

- Seismic Database
  http://www.headwave.com 66x to 100x
- Mobile Phone Antenna Simulation
  http://www.acceleware.com 45x
- Molecular Dynamics
  http://www.ks.uiuc.edu/Research/vmd 21x to 100x
- Neuron Simulation
  http://www.evolvedmachines.com 100x
- MRI processing
  http://bic-test.beckman.uiuc.edu 245x to 415x
- Atmospheric Cloud Simulation
  - http://www.cs.clemson.edu/~jesteel/clouds.html 50x

Fuente: www.nvidia.com/object/IO_43499.html

Actividad a realizar en clase

- Realizar una búsqueda de las tarjetas gráficas que soporta CUDA

Definiciones en la arquitectura CUDA

- **Thread:** Ejecución de un kernel con un index dado. Cada thread usa su index para acceder a los elementos en un arreglo. La colección de todos los hilos procesan cooperativamente el conjunto de datos.
- Los hilos se ejecutan en un Stream Processor.
- Cada MP es específicamente dividido en un número de stream processors (SPs). Cada SP maneja uno o más hilos en el bloque.

- **Bloque:** Grupo de threads. Se pueden coordinador usando la función:

_syncthreads()

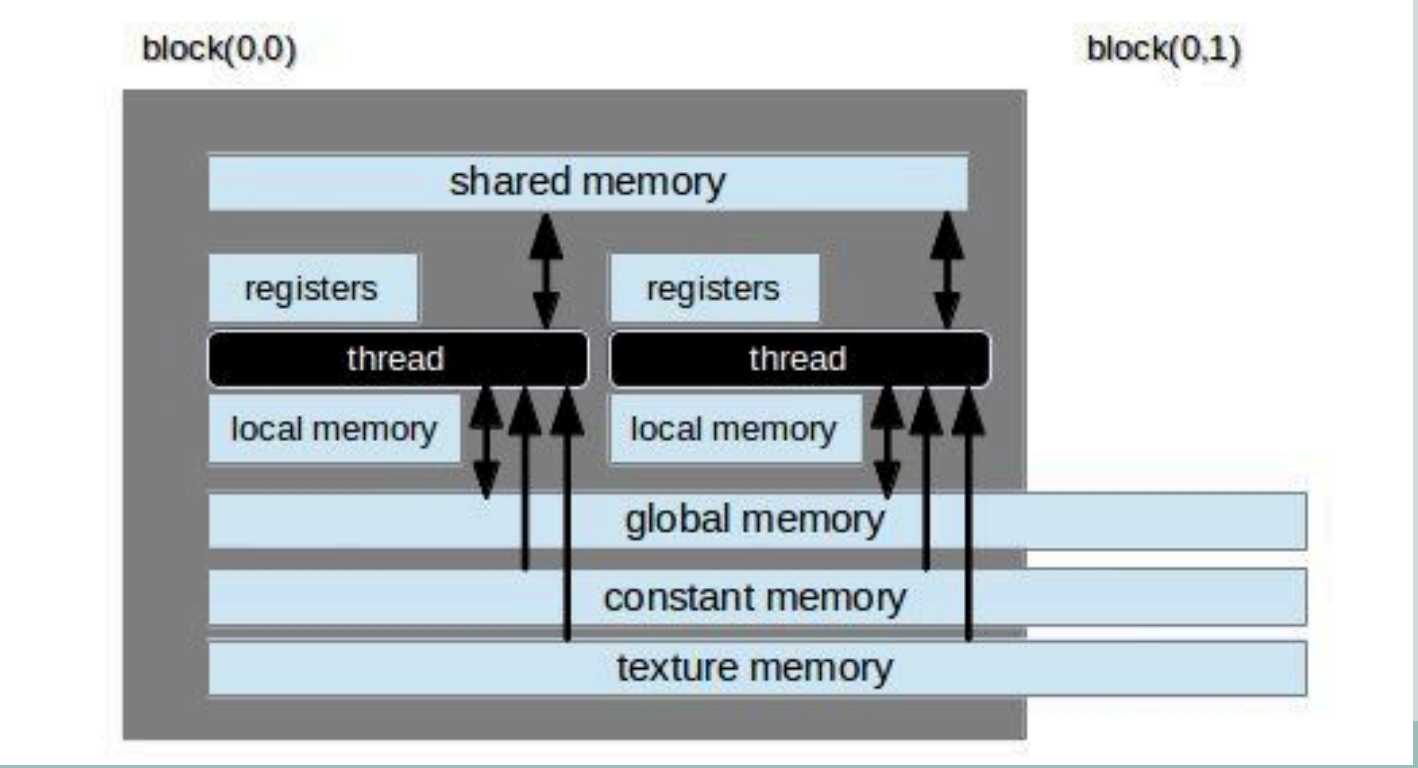
que hace un stop de hilos en un cierto punto en el kernel, hasta que todos los hilos en ese bloque se encuentren en el mismo punto.

- Los bloques se ejecutan en los multiprocesadores.
- El chip GPU está organizado en una colección de multiprocesadores (MPs), cada multiprocesador es responsable de manejar uno o más bloques en un grid.
- Los bloques nunca se dividen más allá de los múltiplos de MPs.

- **Grid:** Grupo de bloques. No hay sincronización entre bloques.
- El grid completo es manejado por un simple chip GPU.

Arquitectura CUDA.
Hilos, bloques y grids

- Los hilos están organizados **en bloques**.
- Un conjunto de bloques forman un **grid**.



- Una memoria global: está disponible a todos los streaming multiprocessors así como para la CPU en la máquina del host. Es de lectura y escritura. También llamado *device memory*.
- La memoria compartida: está disponible a los grupos de hilos corriendo en el mismo bloque. Es de lectura y escritura.
- Memoria constante y memoria de textura: está disponible a todos los hilos globalmente en modo solo lectura. Actúa como una cache
- Cada hilo tiene su propio conjunto de registros y su memoria local.

¿Qué es un kernel?.

Es una función llamable desde el host y ejecutado en el *device* CUDA, simultáneamente por muchos hilos en paralelo.

¿Cómo hacer que el host llame al kernel?

- Esto involucra específicamente el nombre del kernel junto con una configuración de ejecución (número de hilos paralelos en un grupo y el número de grupos a usar cuando se corre el kernel para el *device* CUDA).
  - ¿cómo sincronizar los kernels y código del host?

Sistemas Operativos soportados por CUDA

- Linux (Red Hat, OpenSuse,Ubuntu, Fedora)
- Windows (W XP, W Vista, W 7,) → Visual Studio
- Mac OS (Al menos la versión 10.5.7).

Compilador de CUDA

- nvcc es el compilador que convierte el código fuente en aplicaciones funcionales de CUDA.
- Los archivos CUDA tienen la extensión .cu

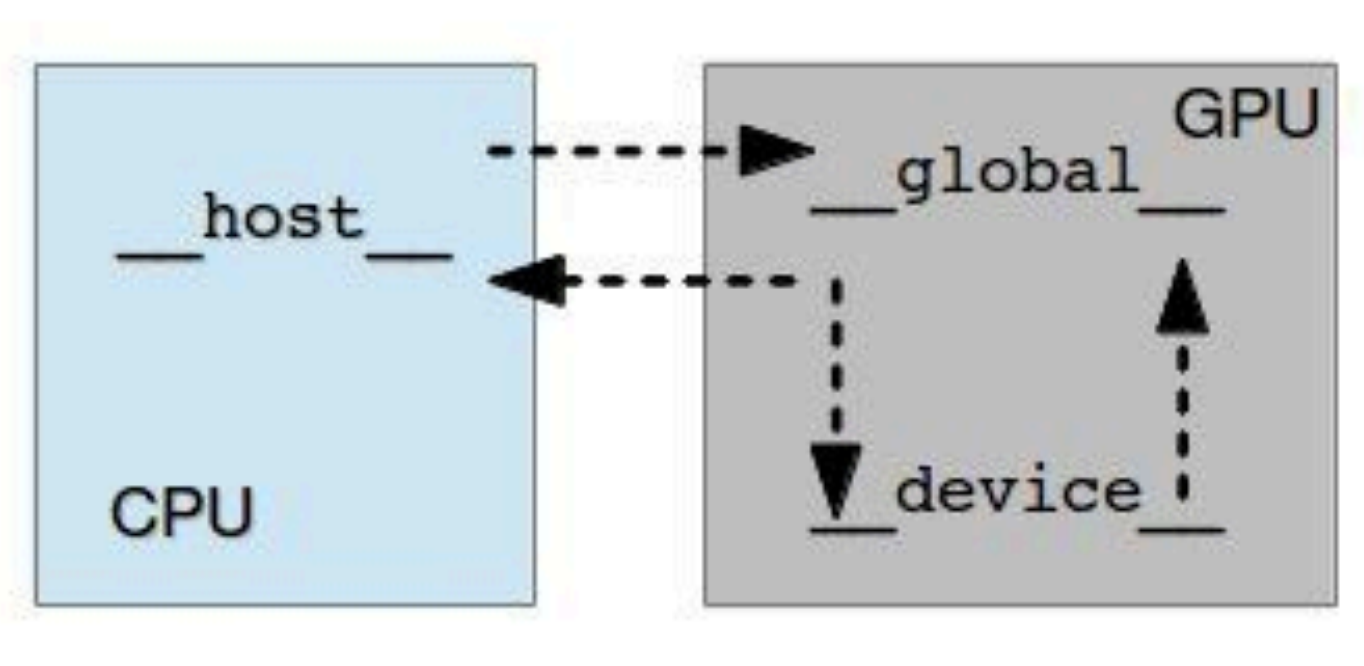
¿Como compilar?

$nvcc –o programaexe programafuente.cu

- El compilador nvcc produce un programa binario que corre en CPU para la parte secuencial y en GPU para la parte paralela.

Programación básica de CUDA

- Los programas de CUDA consisten esencialmente de codigo puro de C para la ejecución en CPU y del código extendido C para la ejecución en GPU.
- Se definen tres tipos de funciones:
  - __host__: corre únicamente en la CPU (__host__ es opcional la extensión).
  - __global__ : corre en la GPU, llamado por la CPU. También llamado **kernel**
  - __device__ : corre en la GPU, llamado por la GPU


GPU
__global__
__host__
__device__
CPU

- La llamada de una función global esta organizada de un conjunto de hilos y bloques que serán activados.
- Un kernel se define como:

  kernel<<<blocs, threads>>>(arguments)

- Los parámetros: bloques e hilos se definen antes de llamar al kernel.

Estructura interna de un kernel

- Dentro del kernel, se puede obtener los parámetros de la organización a través de las variables automáticas asociadas con cada uno de los hilos.
- Estas variables son:

  threadIdx, blockIdx, blockDim, gridDim.

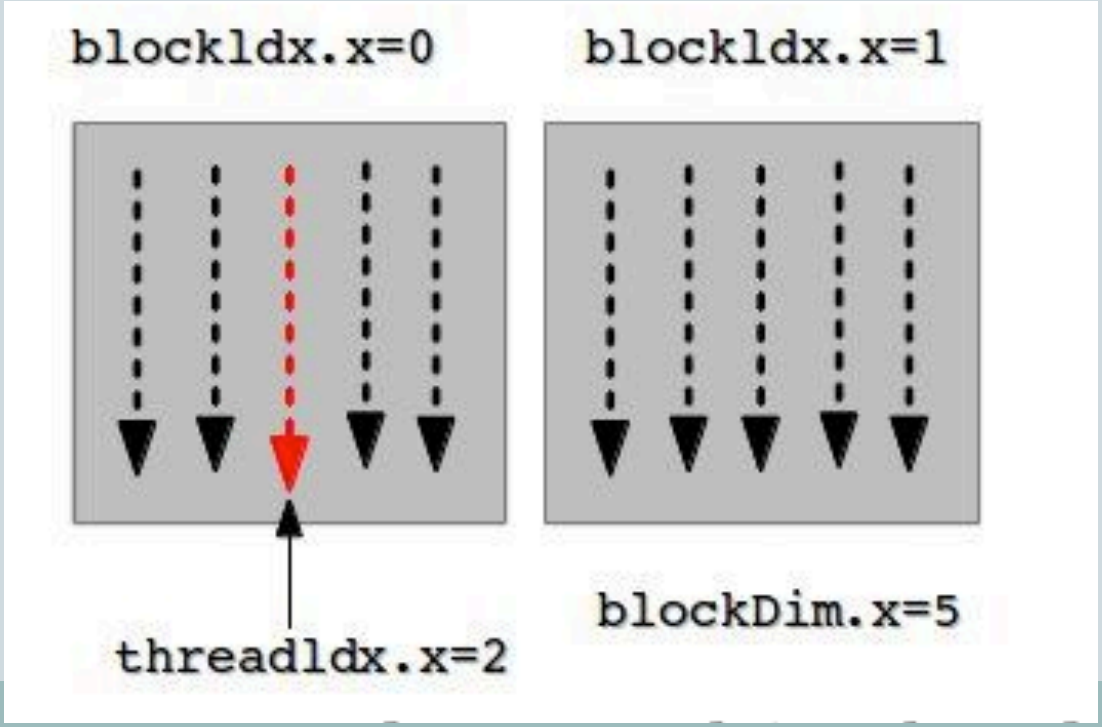
- Para una simple organización unidimensional, los índices se consideran como x :

  threadIdx.x, blockIdx.x, blockDim.x, gridDim.x

- Ejemplo:

  kernel<<<2,5>>>(arguments);

  Corre sobre 2 bloque de 5 hilos.



- En algunos problemas, se necesita explotar la organización de los datos en arreglos multidimensionales.
- Para asignar una organización, usamos el tipo de estructura **dim3.** Una estructura del tipo dim3 contiene 3 elementos, cada uno identificado por una dimensión (x,y y z).
- Por ejemplo:

dim3 threads=(4,4,2);

Define la organización de los hilos en bloque en una matriz tri-dimensional de 4*4*2.

- Se usa la misma técnica para definir la estructura de un **grid**
- Por ejemplo:

dim3 blocks=(3,3,1);

Define la organización de los bloques en una matriz grid de 3*3.

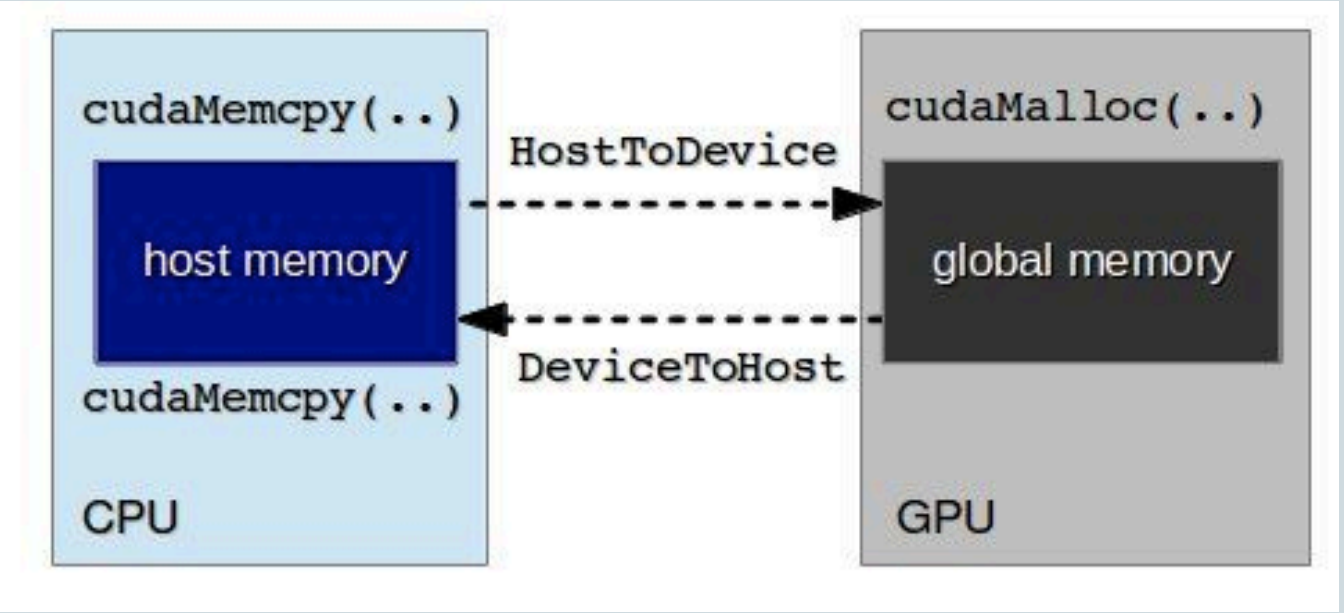
Una llamada del kernel incluye los parámetros de los hilos y bloques:

kernel<<<nb,nt>>>(arguments);

- Ejemplo de 2 bloques:

  aV<<<2,vsize/2>>>(CV1,CV2,Cres);

- Organización interna del kernel:



- Las funciones cudaMalloc() y cudaMemcpy() se utilizan principalmente en la preparación de los datos en la memoria global de la GPU.
- Las constantes:

cudaMemcpyHostToDevice y
cudaMemcpyDeviceToHost

indican la dirección de la transferencia de datos.

Resumiendo la programación en CUDA

Un programa simple de CUDA tiene el siguiente flujo:

- La inicialización en el host de los datos.
- Copiar los datos de la memoria del host a la memoria de la GPU (*device*).
- Realizar las operaciones sobre los datos en la GPU.
- Copiar los datos de la memoria de la GPU a la memoria del host.

Ejemplo

```
// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}

// main routine that executes on the host
int main(void)
{
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 12;  // Number of elements in arrays
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);        // Allocate array on host
  cudaMalloc((void **) &a_d, size);   // Allocate array on device
```

```
// Initialize host array and copy it to CUDA device
 for (int i=0; i<N; i++) a_h[i] = (float)i;
cudaMemcpy(a_d, a_h, size,
cudaMemcpyHostToDevice);
 // Do calculation on device:
 int block_size = 4;
 int n_blocks = N/block_size;
```

```
 square_array <<< n_blocks, block_size >>> (a_d, N);
 // Retrieve result from device and store it in host array
      cudaMemcpy(a_h, a_d, sizeof(float)*N,
        cudaMemcpyDeviceToHost);

 // Print results
 for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
 // Cleanup
 free(a_h);
cudaFree(a_d);
}
```

Ejemplos de grids, bloques e hilos.

```
const int N = 16;
const int blocksize = 16;
dim3 dimBlock( blocksize, 1 );
dim3 dimGrid( 1, 1 );
hello<<<dimGrid, dimBlock>>>(ad, bd);
```

Si quiero tener 512 hilos por bloque y 1024 bloques, como se define?

```
int n_threads_per_block = ?;  // 512 threads per block
  int n_blocks = ?;  // 1024 blocks
   square_array <<< n_blocks, n_threads_per_block >>> (a_d, N);
```

Tarea:

- Leer Capitulo 6. Diseño de programas paralelos:

https://computing.llnl.gov/tutorials/parallel_comp/

- ¿Qué es un warp?
- ¿Qué es la coalescencia?