Sesión 5: Programación en CUDA

Pedro Oscar Pérez Murueta, MTI

Multiprocesadores

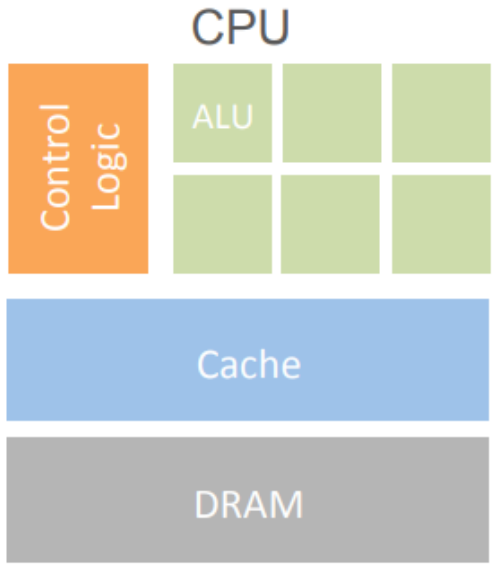
Tecnológico de Monterrey

Cómputo heterogéneo

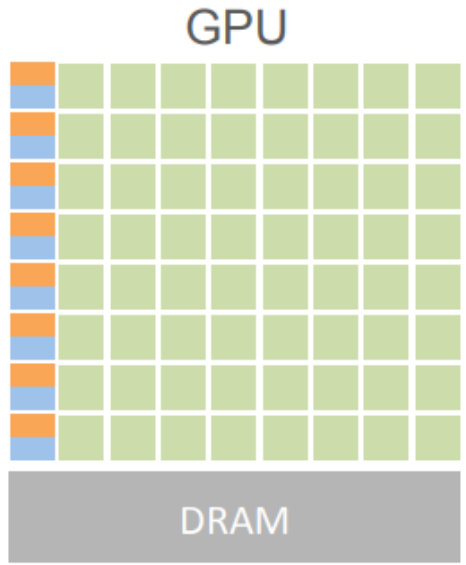
- CPU:
  - Optimizado para la ejecución rápida de un hijo.
  - Los núcleos del CPU está diseñados para ejecutar 1 o 2 hilos simultáneamente.
  - Grandes caches permiten ocultar los tiempos de acceso a la DRAM.
  - Núcleos optimizados para acceso de caché de baja latencia.
  - Lógica de control compleja con el fin de emplear la ejecución fuera de orden.

- GPU:
  - Optimizado para tener alto rendimiento en multihilo.
  - Núcleos diseñados para ejecutar muchos hilos paralelos al mismo tiempo.
  - Núcleos optimizados para datos en paralelo.
  - Los chips utilizan multihilo intensivo para tolerar los tiempos de acceso a la DRAM.

Cómputo heterogéneo (cont.)



Less than 20 cores
1-2 threads per core
Latency is hidden by large cache



512 cores
10s to 100s of threads per core
Latency is hidden by fast context switching

GPUs don't run without CPUs

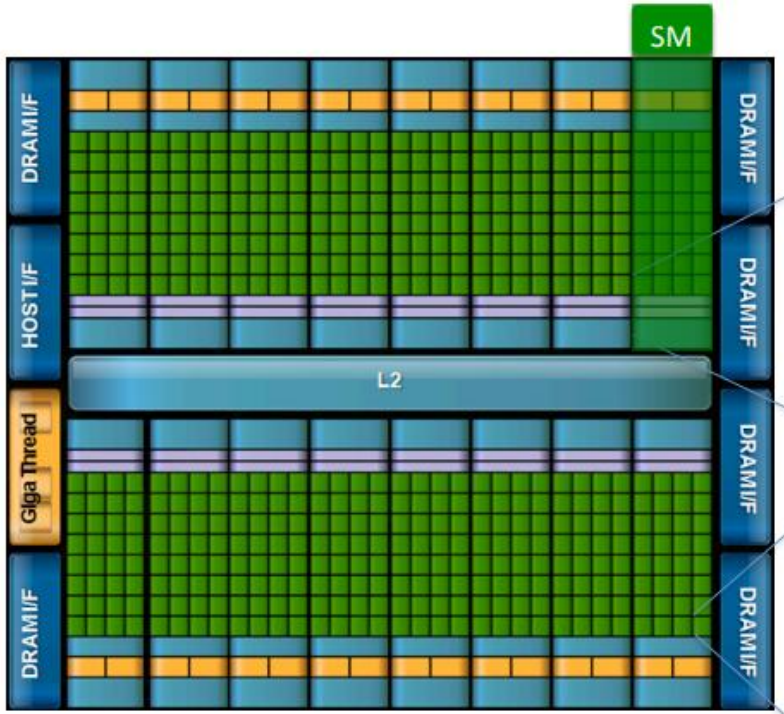
Hardware

- Actualmente existen 5 generaciones de tarjetas que podemos utilizar:
  - Kepler (compatibilidad 3.X): Liberadas en 2010, HPC, tarjetas K40s and K80s.
  - Maxwell (compatibilidad 5.X): Liberadas en 2014, videojuegos.
  - Pascal (compatibilidad 6.X): Liberadas en 2016, tarjetas de videojuegos y para HPC.
  - Volta (compatibilidad 7.X): Liberadas en 2018, HPC.
  - Turing (compatibilidad 7.X): Liberadas en 2018, videojuegos.

Hardware (cont.)

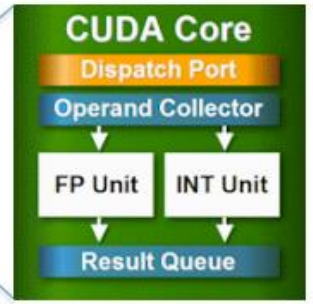
NVIDIA FERMI

16 Stream Multiprocessors (SM)
512 CUDA cores (32/SM)
IEEE 754-2008 floating point (DP and SP)
6 GB GDDR5 DRAM (Global Memory)
ECC Memory support
Two DMA interface



Reconfigurable L1 Cache and Shared Memory
48 KB / 16 KB

L2 Cache 768 KB



Load/Store address width 64 bits. Can calculate addresses of 16 threads per clock.

Hardware (cont.)

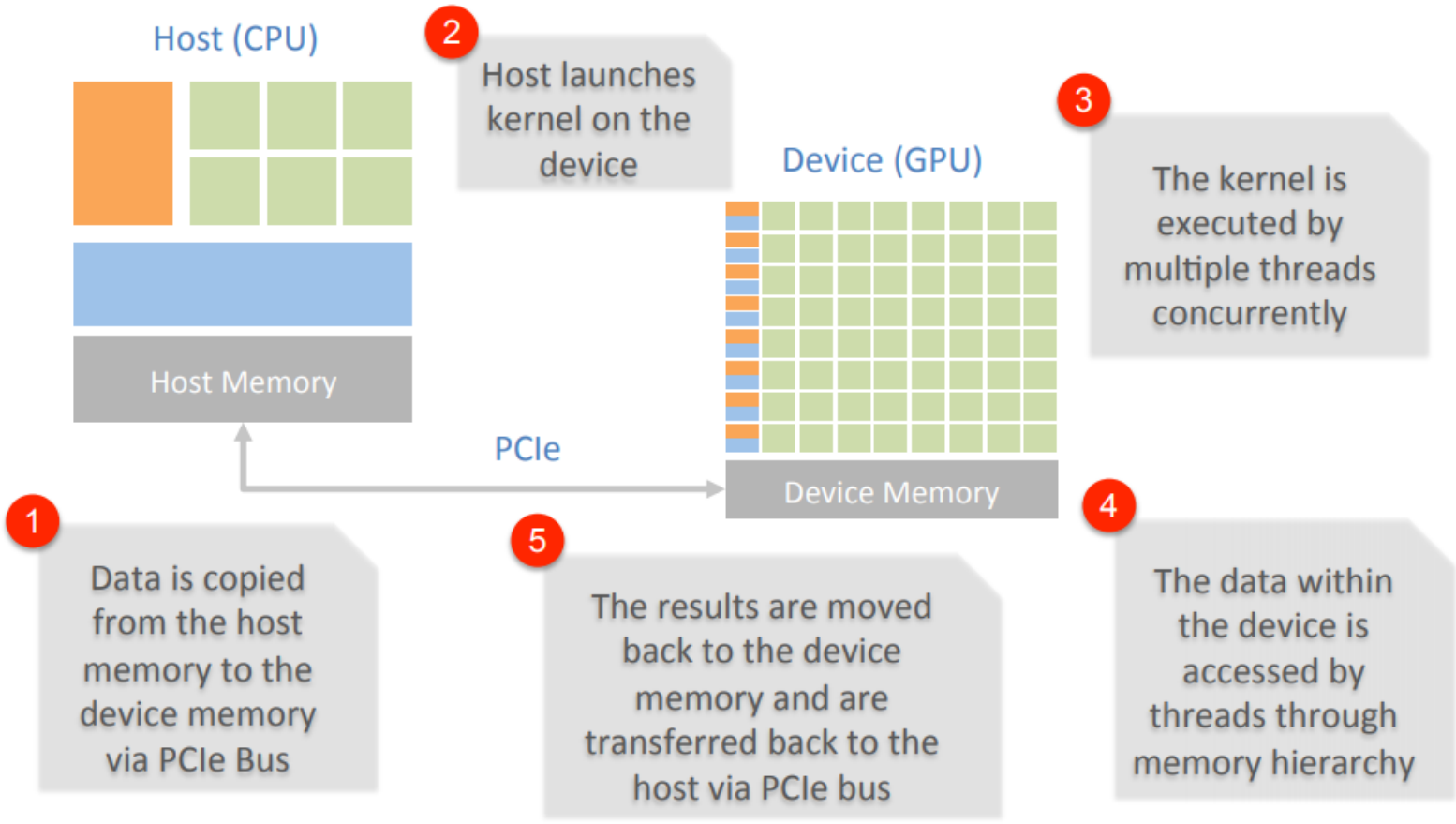
- Generación Pascal:
  - GTX 1060 : 1280 cores, 6 Gb.
  - GTX 1070: 1920 cores, 8 Gb.
  - **GTX 1080: 2560 cores, 8 Gb.**
  - GTX 1080 Ti: 2584 cores, 11 Gb.

- Cada "Streaming Processor" (SM) tiene:
  - 120 cores y registros de 64Kb.
  - 96 Kb de memoria compatida.
  - 48 Kb de cache L1.
  - 8-16 Kb de cache para constantes.
  - Hasta 2K hilos por SM.

Hardware (cont.)

- La característica clave es que todos los núcleos en un SM son núcleos SIMT (Simple Instruction Multiple Threads):
  - Grupos de 32 núcloes ejecutan las mismas instrucciones **simultáneamente**, pero con datos diferentes. Conocidos como ***warp***.
  - Especializada en computación vectorial (tipo CRAY).
  - Especializadas en el procesamiento de gráficos y cómputo científico.
- Muchos hilos activos son la clave del alto rendimiento:
  - No hay cambio de contexto; cada hilo tiene sus propios registros, aunque esto limita el número de hilos activos.
  - La ejecución se alterne entre ***warps*** activos y ***warps*** temporalmente inactivos (los que están esperando por datos).

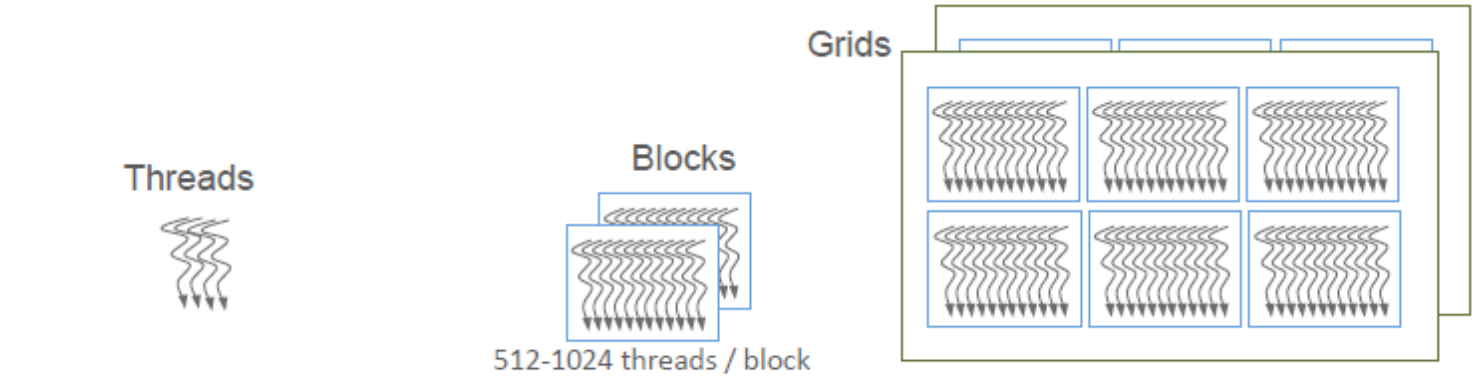
Ejecución de un programa usando tecnología CUDA.



Modelo de programación CUDA

- CUDA Kernels:
  - Se le llama así a la porción paralela de la aplicación.
  - Toda la GPU (o parte) utiliza el mismo kernel.
  - Los kernels son capaces de crear, de manera eficiente, miles de hilos CUDA.

Modelo de programación CUDA (cont.)



Kernel is executed by threads processed by CUDA Core

Maximum 8 blocks per SM
32 parallel threads are executed at the same time in a *WARP*

One grid per kernel with multiple concurrent kernels

Modelo de programación CUDA (cont.)

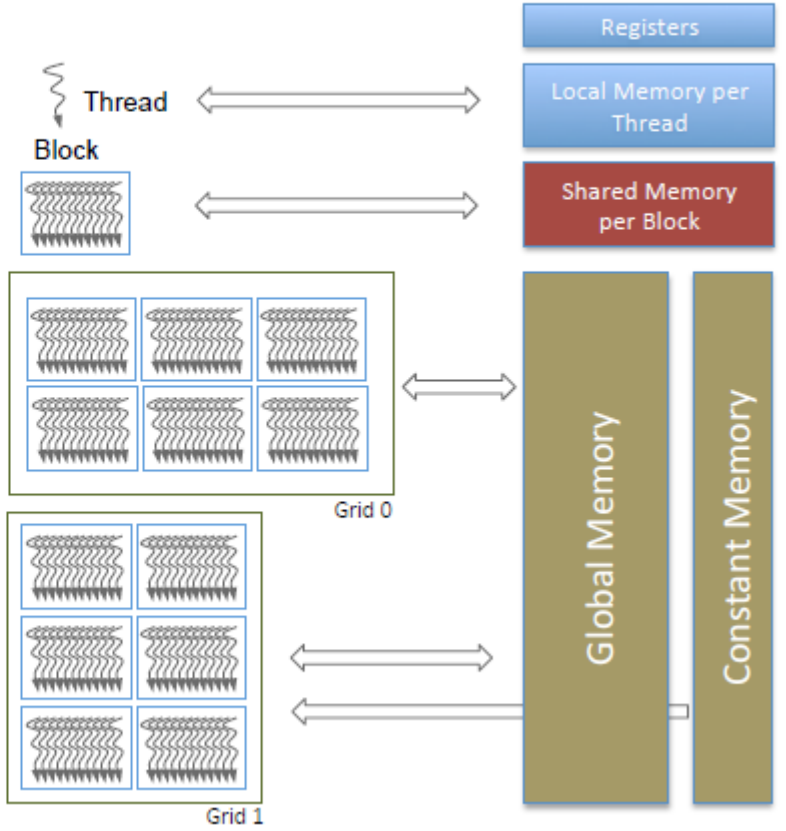
Memory Hierarchy

Private memory
Visible only to the thread

Shared memory
Visible to all the threads in a block

Global memory
Visible to all the threads
Visible to host
Accessible to multiple kernels
Data is stored in row major order

Constant memory (Read Only)
Visible to all the threads in a block



Manejando blocks

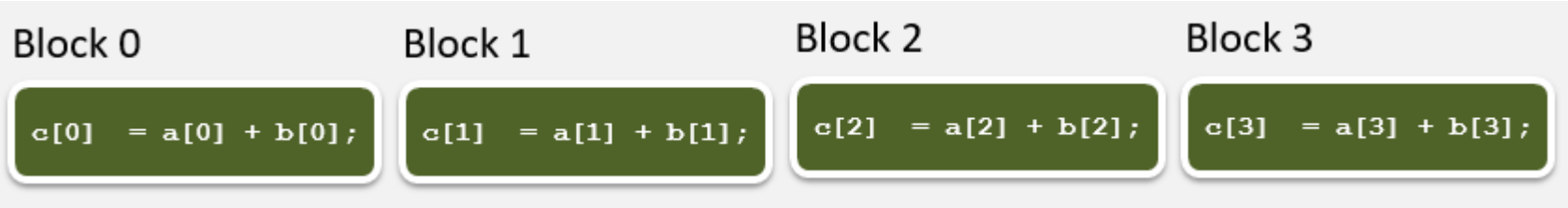
```
add<<< N, 1 >>>();
```

Cada invocación de **`add()`** se conoce como un block

- El conjunto de bloques se conoce como grid
- Cada invocación puede referirse a su índice de bloque usando **`blockIdx.x`**

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Al usar **`blockIdx.x`** para indexar la matriz, cada bloque maneja un índice diferente.



Manejando hilos

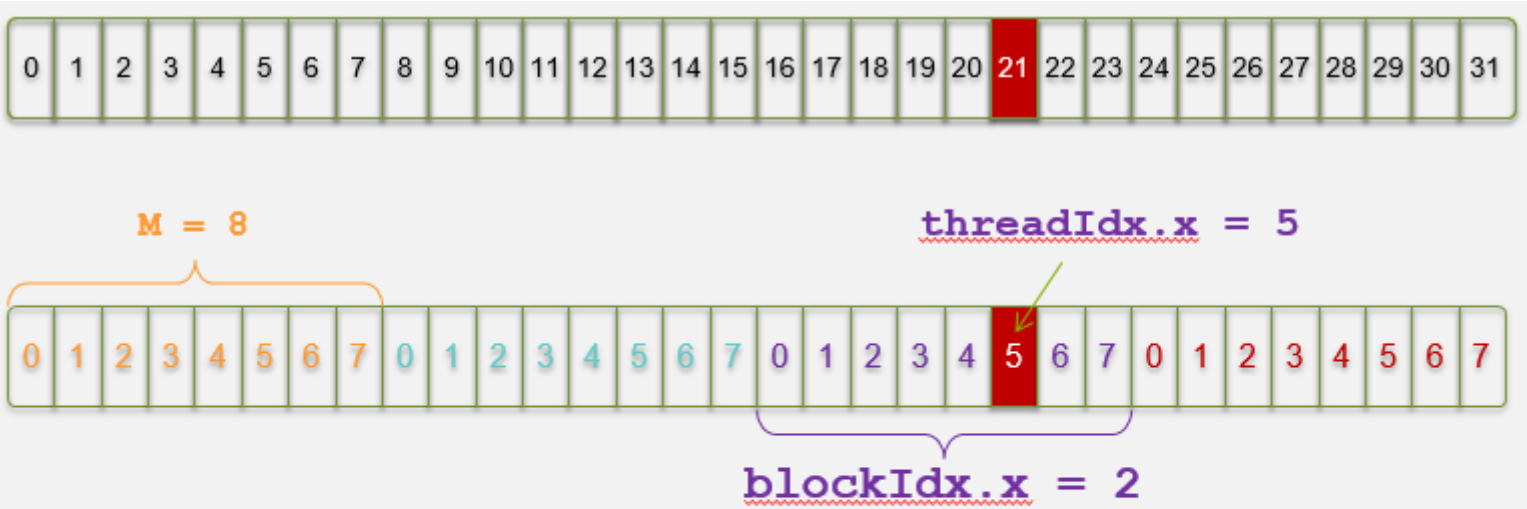
```
add<<< 1, N >>>();

__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

Combinando bloques e hilos


```
int index = threadIdx.x + blockIdx.x * M;
```

Combinando bloques e hilos




```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +     2      * 8;
          = 21;
```

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```