# T0. INTRODUCCION PROGRAMACION PARALELA

¿Qué es?

Es una forma de cómputo en donde se ejecutan varias instrucciones simultáneamente o en donde varios procesadores trabajan juntos para la resolución de alguna tarea. Con ello se consigue más capacidad de cómputo dividiendo y modificando el problema.

Formas de paralelismo

**Segmentación**: divide una unidad funcional en etapas independientes, intercalando registros para almacenar datos intermedios pasando los datos de una etapa a otra en tiempos de reloj.

**Réplica**: aumenta el número de operaciones por unidad de tiempo, varias unidades ejecutan una operación completa, así se tienen tantas instrucciones como unidades funcionales.

Hay paralelismo interno que queda oculto en la arquitectura del ordenador aumentando la velocidad sin disminuir prestaciones (segmentación funciones) y el explícito que si es visible al usuario.

Tipos de procesadores

**Procesador vectorial**: trabajan sobre datos homogéneos, organización de memoria en módulos, mejor rendimiento en instrucciones vectoriales y muchos datos para llenar.

**Procesador escalar**: mejor rendimiento con instrucciones vectoriales y réplica/segmentación, cuatro etapas con duración de submúltiplo de ciclo de reloj.

**Procesador segmentado**: cuatro etapas, después de la etapa inicial se ejecutan instrucciones por ciclo de reloj, mejor con segmentación.

Las 4 etapas son *búsqueda, decodificación, ejecución, almacenamiento.*

**Procesador supersegmentado**: cada etapa se divide en subetapas y se lanzan sin completar ciclos de reloj (1/2 o 1/4).

**Procesador superescalar**: varias instrucciones de forma simultánea.

**Multithreading**: varias procesos ligeros a la vez compartiendo procesador y recursos.

**Procesadores VLIW**: instrucciones más largas, varias operaciones por instrucción.

Clasificación de Hwang-Briggs

Establece una aproximación a las clases de computadores paralelos fijando 3 configuraciones que son; *computadores pipeline, computadores matriciales y sistemas multiprocesador*.

Pudiendo distinguir:

- Paralelismo temporal: varias operaciones en el mismo instante en una unidad funcional.
- Paralelismo espacial: síncronos o asíncronos.

# TEMA 1. PROGRAMACION DE PROCESOS MASIVAMENTE PARALELOS

GPU vs CPU

| GPU | CPU |
|---|---|
| + Cores, - Cache, - Control | - Cores, + Cache, + Control |
| Hardware control más simple | Hardware de control más complejo |
| Hardware para cálculo | Más rendimiento |
| Más eficiente en energía | Más flexibilidad |
| Programación más restrictiva | Cara en términos de poder |
| Gran ancho de banda (Flops*) | Bajo ancho de banda |
| Alta latencia, muchos datos más lento | Baja latencia, pocos datos muy rápido |
| Gran cantidad de hilos, ligeros | Cantidad limitada de hilos |
| Conseguir una eficiencia de 80% en CPU malo, conseguir un 20% en GPU bueno | |

*Flops -> Operaciones flotantes por segundo.

Componentes GPU: GPU, memoria video, RAMDAC, disipador, ventilador y alimentación.

Compilador CUDA

Compilador basado en LLVM.

Para la compilación se utiliza driver NVCC, que invoca todas las herramientas de compilación necesarias (cudacc, g++, cl…) y reescribe los kernels CUDA para aprovechar el paralelismo.

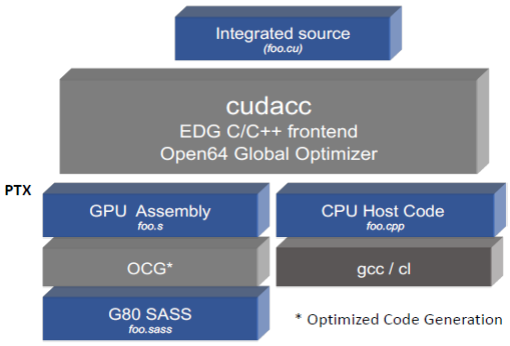
NVCC separa el código que se ejecuta en la CPU o en la GPU y produce:

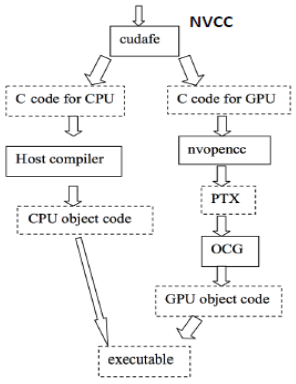
- Código en C: es para la CPU, se compila por separado con gcc, cl…
- Código PTX: código abierto que se interpreta en tiempo de ejecución.

Para un .exe en CUDA necesitamos dos librerías: CUDA runtime (cudart) y CUDA core (cuda).

Mediante nvcc -deviceemu se puede ejecutar un ejecutable desde el host, en donde cada hilo es un hilo del host. Se puede hacer depuración, acceder a un dato o llamar a una función de device a host o viceversa y detectar situaciones de deadlock.

Puede haber resultados diferentes ya que, mediante emulación los hilos se ejecutan secuencialmente y no simultáneamente, puede haber errores de punteros más fácilmente y además los resultados de cálculos en coma flotante será ligeramente diferentes.



## Taxonomía de Flynn

Esta taxonomía se basa en el número de instrucciones concurrentes y de flujo de datos:

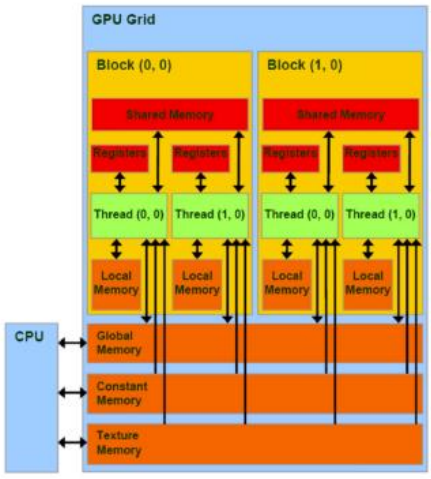
- SISD: Single Instruction Single DataStream (arquitectura x86)
- SIMD: Single Instruction Multiple DataStream (arquitecturas vectoriales)
- MISMD: Multiple Instruction Single DataStream (sistemas con tolerancia a fallos)
- MIMD: Multiple Instruction Multiple DataStream (arquitecturas distribuidas, multicore)

Las GPUs en la actualidad serían como un subtipo de las SIMD ya que desde un único programa se puede operar y acceder a flujos de datos de forma concurrente gracias a los hilos.

## Jerarquía de memoria en CUDA

Ordenadas de mayor a menor latencia, sabiendo que en la mayoría de casos cuanta más velocidad menos capacidad tendrán:

- Global Memory (aplicación): accesible en R/W por todos los hilos y la CPU, haciendo que sea posible la comunicación GPU-CPU. El patrón de acceso a memoria puede afectar al rendimiento. OFF-CHIP
- Local Memory (hilo): es parte de la memoria global, es privada para cada hilo (memoria virtual). Es usada por el compilador para gestionar o guardar variables. OFF-CHIP.
- Texture Memory: controlada por el programador. Similar a la CM y solo R para GPU. Está cacheada, pero es OFF-CHIP.
- Constant Memory: es parte de la memoria global y de solo lectura, puede ser cargada en la cache de la SM para acelerar las transferencias. CPU tiene R/W pero los hilos de GPU solo R. OFF-CHIP
- Shared Memory (SM bloque): es limitada (16KB), accesible en R/W por los hilos de un mismo SM. Comunica entre los hilos de un mismo bloque teniendo cada bloque un espacio de SM. Su tiempo de vida es igual que el de un bloque. ON-CHIP.
- Registros: mismo núcleo para todos los hilos, accesible en R/W de forma privada. Cada hilo tiene su propio conjunto de registros. Están gestionados por el compilador. ON-CHIP.




| Memoria | Localización | Cache | Acceso | Ámbito | Existencia |
|---|---|---|---|---|---|
| Registros | On-chip | N/A | R/W | Un thread | Thread |
| Compartida | On-chip | N/A | R/W | Threads dentro de un mismo bloque | Bloque |
| Global | Off-chip | No | R/W | Todos los threads + Host | Aplicación |
| Constantes | Off-chip | Sí | R | Todos los threads + Host | Aplicación |
| Texturas | Off-chip | Sí | R | Todos los threads + Host | Aplicación |

Hilos en CUDA

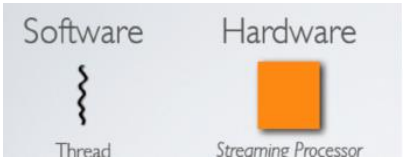
SP (Streaming Processor), SM (Streaming Multiprocessor).

SPMD (Single Program Multiple Data).

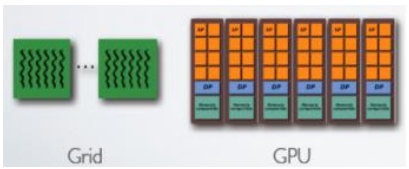
Cada hilo en CUDA se ejecuta en un SP.

Los bloques de hilos se ejecutan en los SM y estos bloques no pueden ir de un SM a otro, por otro lado, la ejecución concurrente está limitada por el número de registros y capacidad de la memoria compartida.

Los kernels se ejecutan como grids de bloques habiendo un único kernel concurrente.








Un kernel ejecuta un array de hilos, teniendo todos los hilos el mismo código (SPMD), pero cada hilo tiene un ID diferente para acceder a memoria y poder controlar el comportamiento. Este array se divide en diferentes bloques, en donde los hilos de un bloque colaboran con memoria compartida, sincronización y operaciones, no pudiendo colaborar con los hilos de otro bloque.

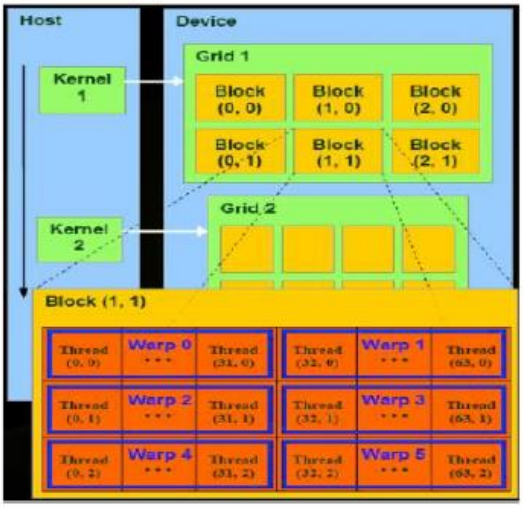
Que los hilos se ejecuten de esta manera permite que sea más fácil procesar datos que sean multidimensionales (matrices, procesamiento de imágenes…).

funcionKernel <<< tamañoGrid(bloques), tamañoBloque(hilos) >>>

Las llamadas a los kernels son asíncronas teniendo que utilizar herramientas de sincronización __syncthreads().

Funcionamiento y comportamiento de los hilos

Los hilos agrupados ya por bloques se asignan a un SM, teniendo estos SM un limite de bloques que pueden ejecutar. El SM mantiene el id hilo/bloque además de manejar y planificar su ejecución.




El hardware puede asignar bloques a cualquiera de los SM en cualquier momento.

A la hora de la ejecución de los hilos estos se agrupan en WARPS, los cuales son unidades de planificación en los SM.

Se planifica de la siguiente forma; los WARPS que están listos para ejecutarse pasan a ser elegible para su ejecución, pasando como a una “cola de espera” en donde se seleccionan para ejecutarse en función de una política de planificación priorizada, ejecutando todos los hilos de un WARP la misma instrucción.

## Tiempos de acceso

### Instrucciones

- Aritméticas -- 4 ciclos de reloj (suma, producto, MAD sobre coma flotante).

  (suma, operaciones nivel bit operandos enteros, max/min, conversión).

- Logarítmicas o cálculo de inversa – 16 ciclos de reloj.

- Multiplicación datos 32 bits – 16 ciclos de reloj.

Buena especialización en operaciones en coma flotante.

### Memoria

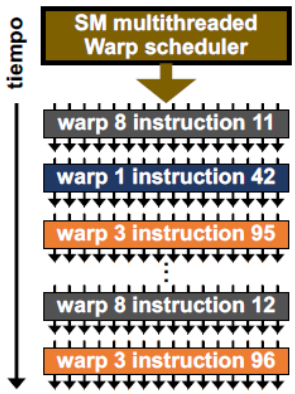
- Accesos – sobrecoste de 4 ciclos de reloj.

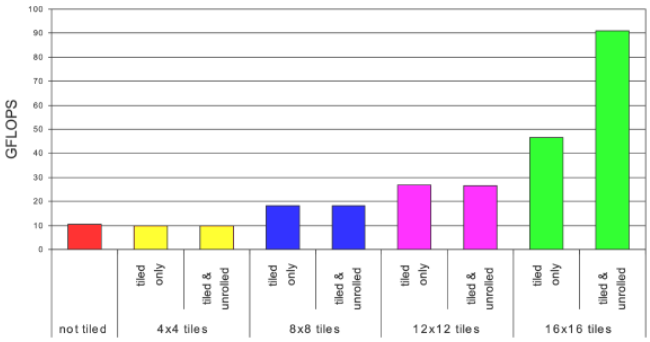
- Memoria global – 400-600 ciclos de reloj.

- Sincronización – sobrecoste de 4 ciclos de reloj + tiempo de espera.

Para optimizar el rendimiento se usa el algoritmo de baldosa, los cuales reducen los accesos a memorias mediante un factor llamado TILE_WIDTH agrupa los hilos de un bloque en teselas, haciendo que se aproveche al máximo el ancho de banda ya que no se necesitan hacer tantos accesos a memoria.

Los WARPS hacen que el tiempo de acceso a instrucción de un conjunto de hilos sea menor, ya que en vez de nHilos accesos se hace solo 1.




## Compute capabilities

NVIDIA utiliza un formato para poder especificar las características de las diferentes tarjetas CUDA, a esto se le llama compute capabilities. Esto incluye dos números, el primero implica los cambios de generación y la segunda su revisión.

La primera fue compute capabilitiy 1.0 y las nuevas son compute capabilities 6.0 (Pascal) y la última versión 7.0 (Volta) llegando a la 7.5.

## Consideraciones sobre hilos

Se lanza la cuadrícula en la SPA, los bloques se distribuyen a todos los SM que lanzan los WARPS de hilos, después los SM planifican y ejecutan los WARPS y una vez completados los recursos se liberan y SPA distribuye más bloques.

| | |
|---|---|
| SPA | *Streaming Processor Array*: variable la serie GeForce 8 –la GeForce 8800 tiene 8- |
| TPC | *Texture Processor Cluster* (2 SM + TEX) |
| SM | *Streaming Multiprocessor* (8 SP)<br>Núcleo multi-hilo<br>Unidad de procesamiento para bloques de hilos en CUDA |
| SP | *Streaming Processor*<br>ULA escalar para un único hilo en CUDA |

Un hilo puede:

- R/W registros por hilo.
- R/W memorial local por hilo.
- R/W memoria compartida por bloque.
- R/W memoria global por grid.
- R memoria constante por grid.
- R memoria texturas por grid.

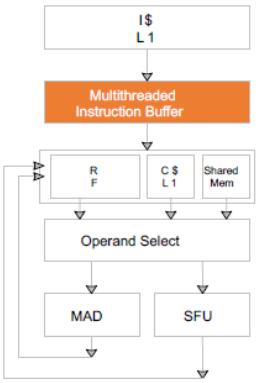
## Consideraciones hardware

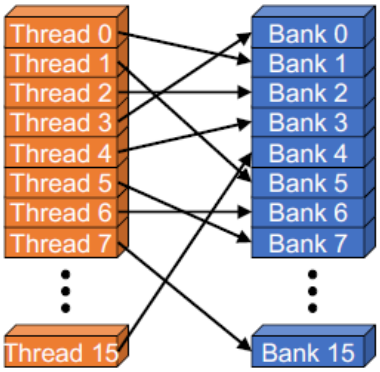
Para la planificación de WARPS se hace mediante el buffer de instrucciones el cual coge un instrucción WARP por ciclo de la cache de instrucciones a cualquier buffer, después ejecuta un ready to go WARP del buffer implementando la marcación de operandos (scoreboarding) y ejecuta un WARP basándose en Round-robin/edad y entonces el SM envía la misma instrucción al WARP seleccionado.

Los registros se reparten de forma dinámica y no automática (lo elige el programador) a lo largo de los bloques de un SM, si se asignan a un bloque no son accesibles por hilos de otro bloque.

**Coalescencia**: es la fusión de dos bloques libres adyacentes de memoria, para así hacer que todos los hilos de medio WARP accedan a direcciones contiguas.

En un máquina paralela muchos hilos acceden a memoria con lo que se **divide en bancos** para así poder servir una dirección de memoria por ciclo. Si dos hilos acceden al mismo banco habrá conflicto. Esto hace que varios hilos puedan acceder simultáneamente a la memoria. Podrán acceder tantos hilos a la vez como bancos haya. Si todos los hilos de medio WARP acceden a la misma posición no hay conflicto (broadcast).



Técnicas de optimización

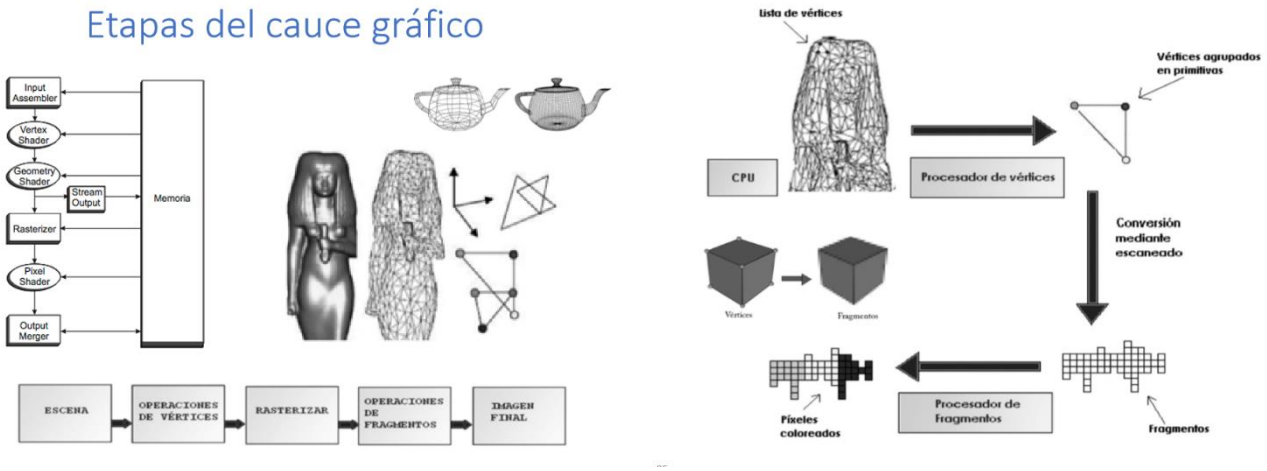
**Unrolling**: lo que hace es sustituir los bucles, que tienen condicionales y saltos por asignaciones de variables.

Cauce gráfico

**Culling**: comprueba que vértices de los triángulos están dentro o fuera del frustum, si todos los vértices están fuera se descartan, por el contrario, si hay vértices del triángulo dentro y fuera se pasa al clipping. Los triángulos se eliminan mediante back-face culling

**Clipping**: descarta parte de los triángulos que están fuera del frustum y los que están dentro y fuera se dividen.

**Frustum**: dentro de la imagen que se dibuja en un juego, es la parte visible en ese momento según la perspectiva.




Inconvenientes de la arquitectura

Algunos de los inconvenientes son:

- Programar vía API con un lenguaje que tengan que entender esas APIs.
- Tamaño de la memoria de textura es muy restrictiva, haciendo que sea limitada la capacidad de escritura.
- Capacidad de los shader, salida y acceso a memoria limitada
- Juego de instrucciones reducidos, faltan enteros y operaciones sobre bits. Se tenía que operar sobre matrices o vectores de 4 componentes
- Comunicaciones limitadas entre píxeles cuyas ubicaciones de memoria han sido predeterminadas, creando accesos raros.

G80 vs G200

| G80 | G200 (mejores) |
|---|---|
| Generación 1.0 - Tesla | Generación 1.3 – Tesla 2.0 |
| 128 cores CUDA | 240 cores CUDA |
| 512MB DDR3 | 1024MB |
| 754 millones de transistores | 1.400 millones de transistores |
| 112 unidades de shader | 240 unidades de shader |
| Menos rendimiento SFUs | Más rendimiento SFUs |
| TPCs agrupan 2 SM | TPCs agrupan 3 SM |
| Cada SM tiene 768 hilos | Cada SM 1024 |


SM en la arquitectura de Fermi tiene 32 SPs or 32 nucleos CUDA.

| | CUDA Compute | |
|---|---|---|
| | 1.0, 1.1 | 1.2, 1.3 |
| Multiprocesadores / GPU | 16 | 30 |
| Cores / Multiprocesador | 8 | 8 |
| Hilos / Warp | 32 | 32 |
| Bloques / Multiprocesador | 8 | 8 |
| Hilos / Bloque | 512 | 512 |
| Hilos / Multiprocesador | 768 | 1 024 |
| Registros de 32 bits / Multiprocesador | 8K | 16K |
| Memoria compartida / Multiprocesador | 16K | 16K |

Fermi vs Kepler

| Fermi | Kepler |
|---|---|
| Generación 2.0 Nvidia | Generación 3.0 Nvidia |
| 512 cores CUDA | 1536 core CUDA |
| SMs 32 cores | SMs 192 cores |
| 4 SFUs | 4 SFUs |
| 1536 hilos/SM | 2048 hilos/SM |
| 8 bloques/SM | 16 bloques/SM |
| 48 warps/SM | 64 warps/SM |
| Más voltaje para funcionar | Menos voltaje para funcionar |
| Menos rendimiento/poder | Mayor rendimiento/poder |

SFUs: Special Function Unit, para funciones especiales tales como seno y coseno.

Memoria en Fermi

Cada SM tiene dos planificadores WARPS y cada core tiene una unidad para el procesamiento de punto flotante y una ALU.

Memoria chip de 64KB como cache de primer nivel y otra parte como memoria compartida, alternando entre 16KB-48KB o 48KB-16KB.

Caché de segundo nivel de 768KB para acceso de memoria global.

Incorpora detección y corrección de errores para el acceso a datos.