# Programación de Procesadores Masivamente Paralelos

## EVOLUCIÓN ¿Por qué usar GPUs?

CPU + Cache + Control - Cores

GPU + Cores – Cache - Control

**CPU →** hardware de control Menos complejo + Flexibilidad + rendimiento - Caro en términos de poder

- Hasta mediados de los '00 se buscó aumentar la velocidad de las CPUs.
- Por problemas con la disipación del calor y el consumo de energía, se comenzó a aumentar el número de núcleos.
- CPU (multi-core): pocos núcleos – instrucciones complejas y están dominadas por memorias de cache
  - Cada núcleo tiene una arquitectura x86 completa.
  - Están diseñados para ejecutar de manera óptima código secuencial.

**GPU→** control más simple de hardware, + Más HW para el cálculo, + Más eficiente de la energía

- Modelo de programación más restrictiva

- Ejecución de aplicaciones paralelas por naturaleza.
- GPU (many-core): muchos núcleos – instrucciones sencillas
- Acceden a direcciones de memoria contigua y tienen un uso eficiente del ancho de banda Implementan núcleos más pequeños (sencillos) cuyo número se duplica en cada generación.
- Cada núcleo puede ejecutar multitud de hilos y el control y el cache se comparte entre varios núcleos
- la carga de trabajo sobre los pixel shader y vertex shader es relativamente baja.

Posible cuestión: Memoria gráfica dedicada o compartida

el caso es que queria saber si da el mismo resultado un portatil con 8GB de RAM y gráfica compartida de 4GB o un portatil con 8GB de RAM y 2 GB de gráfica dedicada.

La potencia de cálculo(Gigaflops) de las GPUs crece de manera exponencial frente a la de las CPUS.

El pico máximo del procesador de las GPUs es mayor y más fácil de sostener debido a la arquitectura:

Las memorias empleadas en la GPUs están diseñadas para tener un gran ancho de banda (muchos datos ligeramente más lento). => alta latencia

Las memorias de las CPUs están diseñadas para tener una baja ancho de banda (pocos datos muy rápido). => baja latencia

**Latencia**: suma de retardos temporales dentro de una red →Tiempos de acceso a la memoria

**Ancho de banda**: carga de trabajo.

**El parseo de lenguajes** no se beneficiaría, a priori, de este tipo de arquitecturas **ni los problemas que tengan una naturaleza secuencial**, como por ejemplo **Lenguages & APIs**

Fue Nvidia la primera en desarrollar, en 2007, unas GPUS completamente programables y con SDK.

**LLVM:** Compilador de estructura de código abierto, con diseño modular que hacen más sencillo agregar el soporte para lenguajes de programación y arquitecturas de procesador

soporta un amplio rango de lenguajes de programación, incluyendo C/C++, Objective-C, Fortran, Ada, Python, Ruby,…

# Evolución APIs

-Antes se trabajaba directamente con registros

-CUDA genera un lenguaje específico para su programación y no requiere de un API intermediaria (2006)

-Cg o C for Graphics es un lenguaje de alto nivel desarrollado por Nvidia en colaboración con Microsoft1 2 para la programación de vertex y pixel shaders.

**Paralelismo de datos:** permite al programador especificar cómo realizar las mismas operaciones en paralelo sobre diferentes datos.

**Intensidad aritmética:** les da a los programadores el poder para minimizar la comunicación global de las operaciones y maximizar la comunicación local de las mismas

## Historia Abreviada de las GPU

**Renderizado:** Proceso de **generar una imagen desde un modelo.** Convertir imágenes de 2D a 3D .

**- NVidia Riva 128 :** Fue una de las primeras tarjetas en incluir **aceleración 2D/3D.**

**- Nvidia GeForce 256**

**-GeForce3 (64/128 MB DDR)**

**-GeForce FX 5200 pag 28**

- **Soporte hardware de Transform &Lighting**: Da textura a las imágenes. **actualmente está en desuso**, al haber implementado los **pixel shaders en DirectX 8.0**

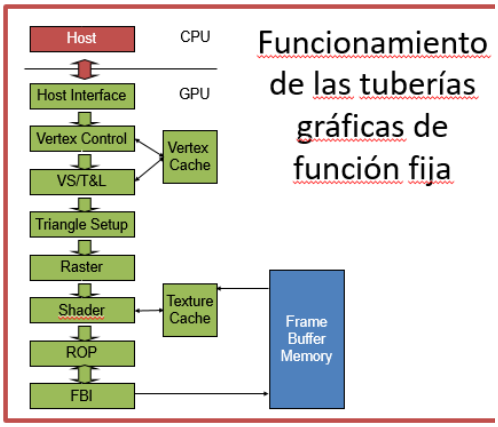
**- Clipping:** Consiste en sólo dibujar las partes de la escena que va a estar presente en la imagen después de completar la renderización.

-Los que no se van a ver se eliminan --clucking?

**IMPORTANTE**

SON LOS MODELOS ANTIGUOS DE TARJETAS

### Funcionamiento de las tuberías gráficas de función fija PAG 16




-La malla de triángulos es enviada a la GPU, los servidores comprueban cómo están colocados dichos triángulos.

-Cada esquina de los triángulos contiene la información de la textura.

**Host Interface**

- Recibe comandos y datos desde la CPU, dados mediante llamadas a la API.
- Contiene DMA especializado para las transferencias de datos entre el host y la GPU.
- Comunica los resultados y el estado de vuelta al host.

**Vertex Control**

- Los objetos se representan como colecciones de triángulos.
- Recibe la imagen, en triángulos parametrizados por la CPU y los convierte a un formato adecuado y los coloca en el cache de vértices.

**VS/T&L*(**vertex shading, transform, and lighting )

- Transforma vértices y les asigna valores ( colores, coordenadas de texturas, tangentes…).
- El sombreado se realiza por el hardware de sombreado de píxeles.
- Puede asignar colores, pero no se aplican a los triángulos hasta más tarde.

**Vertex Cache**

Almacén temporal para los vértices, que se utilizan para obtener mayor eficiencia.

• La reutilización de vértices entre los primitivos **ahorra** AGP / PCI-E **de ancho de banda de bus.**

• La reutilización de vértices entre los primitivos **ahorra** GPU **recursos computacionales.**

• Un intento de caché de vértices entre triángulos para generar reutilización de vértices.

• Muchas aplicaciones no utilizan ordenamiento triangular eficiente.

**Triangle Setup** : Crea ecuaciones para las aristas que son usadas para interpolar colores y otra información per-vértice (tales como coordenadas de las texturas) a lo largo de los píxeles en contacto con el triángulo.

**Raster:** Determina qué píxeles están contenidos en cada triángulo. Para cada pixel, interpola valores per-pixel necesarios para el sombreado, incluyendo posición, color y textura que será pintado en el pixel.

La **rasterización** es el proceso por el cual una imagen descrita en un formato gráfico vectorial se **convierte en un conjunto de** píxeles **o puntos para ser desplegados** en un medio de salida digital

**IMPORTANTE**

**Shader** (En infografía es un programa informático ejecutado en GPU **para hacer el sombreado.)**

- Determina el color final de cada pixel, combinando varias técnicas: interpolación de vértices de colores, mapeo de texturas, iluminación, reflejos, y más.

**Cache de textura:** Almacena temporalmente los valores locales texel a reducir los requisitos de ancho de banda

**ROP(Raster Operation → Anti-Aliasing** lo que hace es difuminar

Lleva acabo el rasterizado final (conversión de información vectorial a píxeles):

- mezcla el color de objetos superpuestos/adyacentes y efectos de antialiasing raster operations that blend the color of overlapping/
- determina la visibilidad de un pixel desde un punto de vista
- **EL numero de vertex que tiene indica si una tarjeta es buena, al igual que el número de shaders**

FBI (Frame Buffer Interface)

- El interfaz del buffer de memoria maneja la lectura y la escritura al frame buffer.
- Requiere un gran ancho de banda, alcanzado mediante:
    - Diseños especiales de memoria
    - Manejo de múltiples canales que acceden a distintos bancos de memoria simultáneamente.
- El **multisampling** puede suavizar los vértices de los triángulos y los vértices que se forman cuando un triángulo intersecta a otro. Texturas, sombras e iluminación producen los mismos resultados.
- **Aliasing** es el efecto que causa que señales continuas distintas se tornen indistinguibles cuando se muestrean digitalmente

## Los shaders trabajan de la siguiente manera.

- El programador envía un conjunto de vértices que forman su escena gráfica a través de un lenguaje de propósito general. Todos los vértices pasan por el **vertex shader** donde pueden ser **transformados y se determina su posición final**. El siguiente paso es el **geometry shader** donde se pueden **eliminar o añadir vértices**. Posteriormente los vértices son ensamblados formando **primitivas que son rasterizadas**, proceso en el cual las **superficies se dividen en puntos que corresponden a píxeles de la pantall**a. El Píxel/Fragment shader **se encarga de modificar estos puntos**. Por último, se producen cierto tests entre ellos el de profundidad que determina que punto es dibujado en pantalla.
- Vertex shader**: Permite transformaciones sobre coordenadas, normal, color, textura, etc. de un vértice.**

Geometry shader**: Es capaz de generar nuevas primitivas dinámicamente, así como de modificar existentes.** Un ejemplo claro es la decisión de utilizar o eliminar vértices en una malla polígonal . **Procesa todos los vértices de una primitiva en vez de los de un vértice aislado.**

- Píxel/Fragment shader: **En primer lugar, aclarar la difererencia entre fragmento y píxel.**

A la GPU le llega la información de la CPU en forma de vértices, lo que evita enviar toda la información de la imagen, **por lo que el tráfico en el bus (PCI o AGP) es mucho menor.**

**SABER CARACTERÍSTICAS DE ESTA TARJETA DE MEMORIA: REGISTRO E HILOS POR BLOQUE**

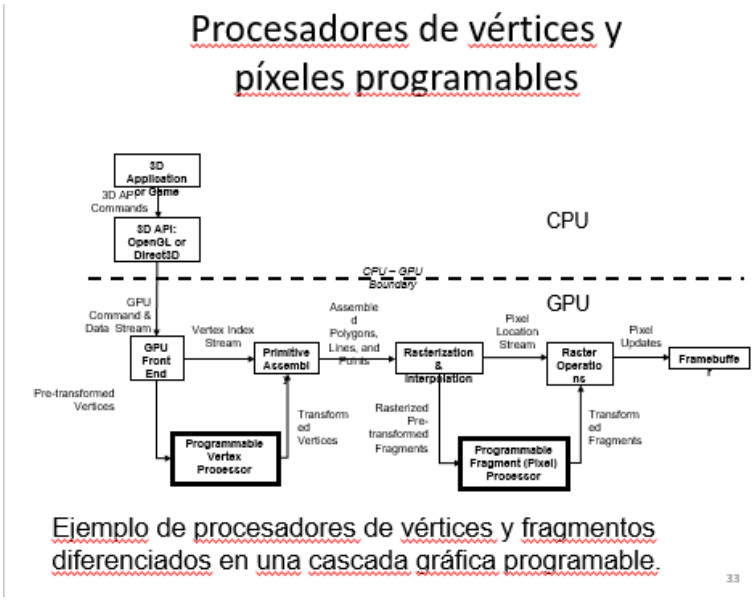
**ES LA BASE DE TODAS .**

**32 Procesadores, la 8800 posee 128**

**8 Procesadores de vértices**

**Nvidia GTX 7800 (256 MB GDDR3)**
Año de lanzamiento: 2005
Versión DirectX: 9.0c
Número de transistores: 302 millones (G70), 281 millones (G71)
Características destacadas:
- Arquitectura de 24 pixel pipelines paralelos
- Arquitectura de 7 vertex pipelines paralelos
- Bus de Transferencia: PCI-Express & AGP
- El rendimiento de una 7800 GTX se incrementaba un 25-35% frente al modelo 6800 Ultra.

Extienden las posibilidades de programación a las fases de pixel y vertex shading.




**Los procesadores escalares** →Cada instrucción de un procesador escalar opera sobre un dato cada vez. En contraposición, en un **procesador vectorial** una sola instrucción opera simultáneamente sobre un conjunto de datos.

# Inconvenientes de la Arquitectura

Programar vía APIs gráficas → Formular los problemas en términos entendibles por las APIs gráficas

Modos de direccionamiento → Limited texture size/dimension

Capacidades de los shader → Salida limitada – acceso memoria

Juego de instrucciones → Faltan enteros y operaciones sobre bits

Comunicaciones limitadas → Entre pixeles y Accesos raros

la única manera de obtener un resultado de un paso de cálculo a la siguiente era

escribir todos los resultados en paralelo a un frame buffer(muy restrictiva) de píxeles, y luego usar esa memoria de vídeo como una entrada de mapa de textura para el pixel shader

los shaders no pueden acceder a localizaciones arbitrarias de memoria.

## Características del modelo CUDA (Compute Unified Device Architecture)

**-Modelo de programación general** Emplea lotes de hilos y Coprocesador masivamente paralelo (GPU)

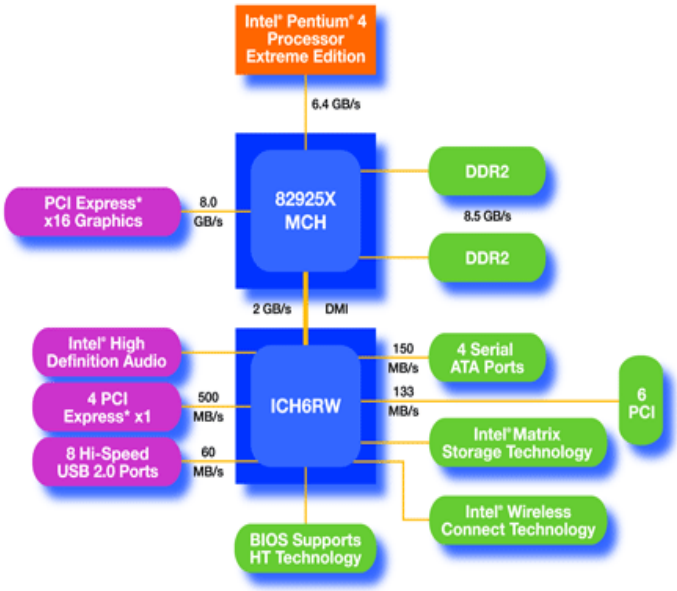
**-Software de desarrollo específico:** Drivers, lenguajes y herramientas orientados a computación

**Driver para la carga de programas en la GPU**

- Driver independiente, optimizado para la computación
- Interfaz no dependiente de librerías gráficas
- Compartición de datos con objetos OpenGL
- Manejo explícito de la memoria de la GPU

# Arquitectura de sistema CUDA

## LAS TRANSMISIONES SON IMPORTANTES



# Taxonomía de Flynn (contexto de arquitecturas paralelas)

Las GPUs actuales no encajan exactamente dentro de dicha clasificación. **Pueden verse como un subtipo de SIMD**. La taxonomía de Flynn clasifica la arquitectura de acuerdo al número de instrucciones concurrentes que se ejecutan y al número de flujos de datos sobre los que se opera. De acuerdo a ello hay las siguientes categorías:

**SISD (única instrucción/único flujo de datos)** - Son máquinas secuenciales que no tienen ningún paralelismo. La unidad de control (coge la instrucción)→CPU → único flujo de datos.

**SIMD (única instrucción/múltiples flujos de datos)** - Esto permite paralelizar numerosos problemas. Las modernas GPUs se basan en ideas similares.Ej:extensiones MMX ,SSEx 3DNow

**MISD (múltiples instrucciones/único flujo de datos)** – Este tipo de arquitecturas tienen sus aplicaciones en entornos donde se necesita una alta tolerancia a fallos.

**MIMD (múltiples instrucciones/único flujo de datos)** – Múltiples procesadores autónomos ejecutan de manera simultánea instrucciones sobre datos diferentes. Los sistemas distribuidos son clasificados generalmente como MIMD, bien usen memoria compartida o distribuida.

# CUDA y la taxonomía de Flynn

**SPMD (Single Program Multiple Data) -** Un único programa se ejecuta en múltiples hilos que acceden a diferentes flujos de datos

■ Tipos: Memoria compartida (Hilos) y Memoria distribuida (Master & Slave)

■ Los núcleos son más complejos → CPU

**SIMD (Single Instruction Multiple Data)**

■ Todos los hilos ejecutan la misma instrucción (debe ser muy sencilla)

■ Cantidad ingente de hilos (hasta 65536 en la G80) en arquitecturas many-cores.

## Vistazo rápido al modelo CUDA

Un dispositivo de cómputo (device)

- Es un coprocesador de la CPU (host)
- Tiene su propia DRAM
- Ejecuta múltiples hilos en paralelo

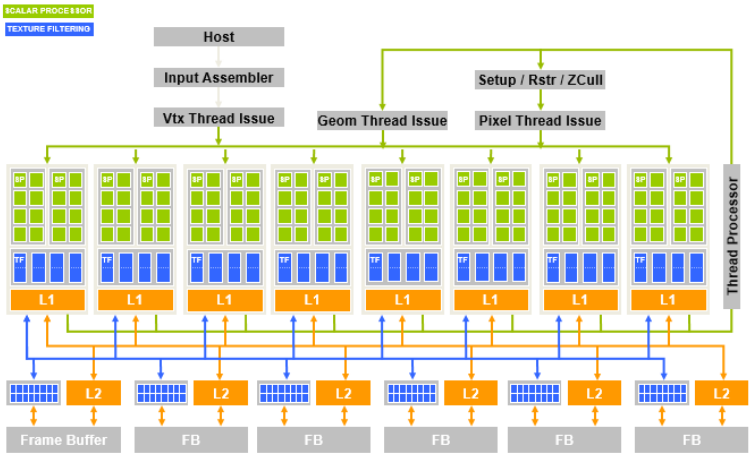
Las partes paralelas se expresan mediante núcleos (kernels) que se ejecutan en múltiples hilos

Características de los hilos de la GPU → ligeros y se usan miles para mayor rendimiento

**La API** es una extensión al ANSI C ,Una librería de ejecución

- Extensiones del lenguaje
- Una librería de ejecución que se divide en:
  - Un componente común que suministra vectores
  - Un componente del anfitrión para controlar uno o más dispositivos desde el anfitrión y Un componente del dispositivo

## G80 modo Gráfico




Hay programas que permiten crear código ejecutado en la GPU, pero hay que tener en cuenta que no todas las etapas del pipeline son programables.

Existen **shaders para vértices** y **shaders para fragmentos** ejecutados en sus procesadores.

Un **vertex shader** es una función que recibe como parámetro un vértice. Sólo trabaja con un vértice a la vez, y no puede eliminarlo, sólo transformarlo. Ejemplo, el movimiento de una ola.

Un **fragment shader** especifica el color de un píxel. Este tratamiento individual de los píxeles permite que se realicen cálculos principalmente relacionados con la textura del elemento y en tiempo real.

| | |
|---|---|
| **Declspecs**<br>**global, device, shared, local, constant** | **__device__ float filter[N];**<br>**__global__ void convolve (float *image) {**<br>**__shared__ float region[M];** |
| Keywords<br>threadIdx, blockIdx | region[threadIdx] = image[i];<br>… |
| Intrinsics<br>__syncthreads | __syncthreads()<br>…<br>image[j] = result; } |
| Runtime API<br>Memory, symbol, execution management | // Allocate GPU memory<br>void *myimage = cudaMalloc(bytes) |
| Function launch | // 100 blocks, 10 threads per block convolve<<<100, 10>>> (myimage); |

Se trabaja siempre con 2^n

2048hilos

Cuántos bloques y cuántos hilos por bloque2048 / 512 con lo cual son 4 bloques

Dividimos los 512 entre 8 SP que da 64 hilos por cada Stream Multiprocesador

## Arrays de hilos paralelos

Un kernel de CUDA se ejecuta en un array de hilos

- Todos los hilos ejecutan el mismo código (SPMD)
- Cada hilo tiene una única ID empleada para acceder a la memoria y controlar las decisiones sobre los datos con los que trabajar.
- Simplifica el acceso a la memoria al procesar datos multidimensionales

El array de hilos se divide en múltiples bloques

- Los hilos dentro de un bloque colaboran mediante memoria compartida, operaciones atómicas y sincronizaciones de barrera.
- Los hilos de diferentes bloques no pueden colaborar

Ejemplo tengo 32 hilos en una dimensión que hilo es el 30

Si tengo dos dimensiones cuatro bloques 8 hilos cada uno y 32 hilo en total ¿qué posicion ocupa 15? ¿y qué posicion ocupa 31?

(256,2,1) = 512

***Los bloques son siempre de 2 dimensiones , los hilos son de 3 dimensiones**

***Siempre movemos los datos de la CPU a la GPU**

## Gestión de la memoria

**cudaMalloc()**

- Asigna objetos en la memoria global
  - Requiere dos parámetros Puntero y Tamaño del objeto a asignar

**cudaFree()**

- Libera un objeto de la memoria global → Puntero al objeto a liberar

Código de ejemplo:

```
T_WIDTH = 64;
Float* Md;
int size = T_WIDTH * T_WIDTH * sizeof(float);
cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

Asigna 64 * 64 floats

Adscribir al espacio asignado a Md

“d” se emplea normalmente para indicar la estructura de datos en el dispositivo

# Transferencia de datos pag 67

cudaMemcpy()

- Transfiere datos
  - Require 4 parámetros →Puntero al destino y origen, nºbytes, tipo de transferencia.

Código ejemplo:

Transfiere 64 * 64 floats

M está en el host y Md en el device

**Constantes simbólicas**

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
-

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```

**IMPORTANTE**

## Declaración de funciones

| | Se ejecuta en | Ejecutable desde |
|---|---|---|
| `__device__ float DeviceFunc()` | device | device |
| `__global__ void KernelFunc()` | device | host |
| `__host__ float HostFunc()` | host | host |

- La directiva `__global__` define un kernel y debe devolver `void`
- Las directivas `__device__` y `__host__` pueden usarse simultáneamente

-No se puede obtener la dirección de las funciones __device__

Las funciones que se ejecutan en el device:

- No pueden ser recursivas, ni declarar variables estáticas
- No pueden tener un número variable de argumentos

# Ejecución de kernels/Creación de Hilos

-Las llamadas a kernels deben tener una configuración de ejecución.

-Las llamadas a los kernels son asíncronas, **se necesita sincronización explícita para bloquear.**

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50); // 5000 bloques de hilos
dim3   DimBlock(4, 8, 8); // 256 hilos por bloque
size_t SharedMemBytes = 64; // 64 bytes de memoria comp.
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

***Dim Grid indica la cantidad de bloques de funcionamiento**

Hilos totals = 5000 * 256

DIM GRID 256

1X 256 X 256 = 65535 DIM BLOCK

N es el nº de hilos

Lanzamiento de threads con un solo bloque

// configuración de la ejecución

dim3 tamGrid(1, 1); //Grid dimensión

dim3 tamBlock(1,N,1); //Block dimensión

// lanzamiento del kernel

MatrixSumKernel

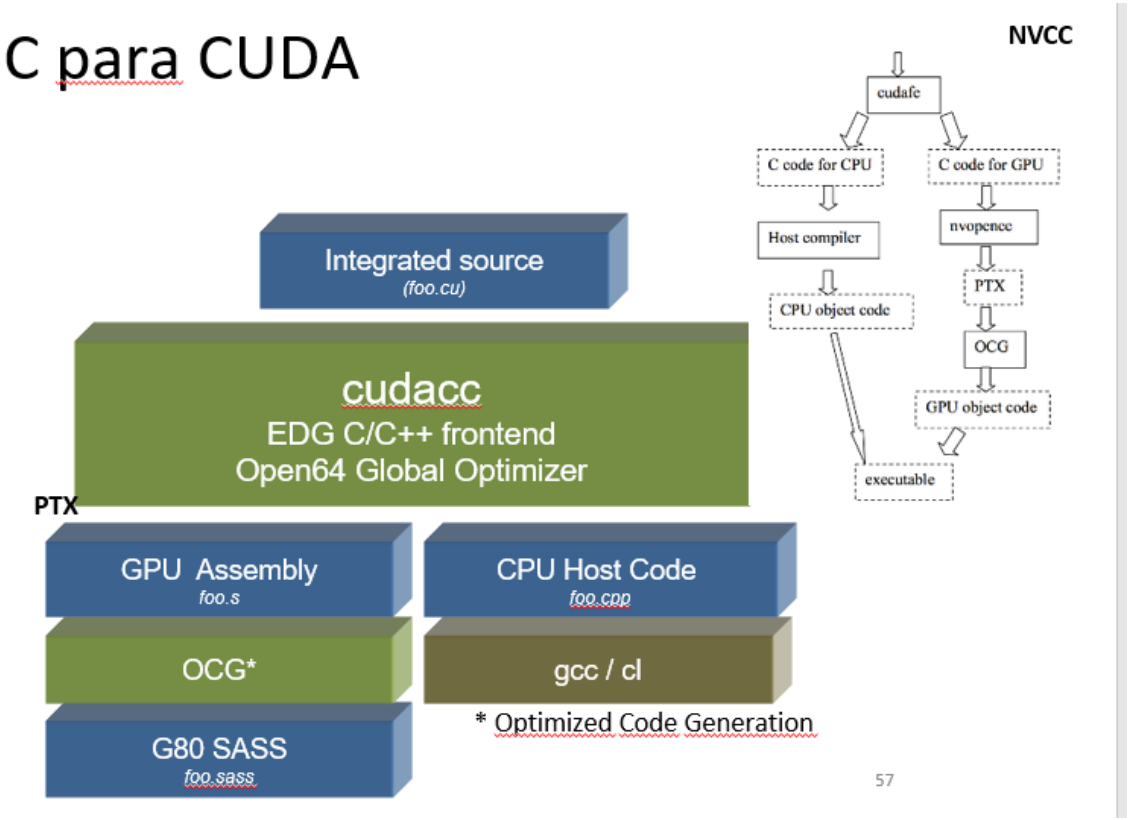
<<<tamGrid, tamBlock>>>(M, Md, Nd);

## Ejemplo: Multiplicación de matrices

- Dejar el uso de la memoria compartida hasta más tarde.
- Uso de registro, local.
- Uso del ID del hilo.
- API para la transferencia de datos en memoria entre host y device.
- Asumimos una matriz cuadrada para simplificar.

Mirar de la página 72 a la 78

**IMPORTANTE – SEGURO QUE ENTRA**




Un programa CUDA consiste en una o más fases que se ejecutan bien en la CPU/host o bien en la GPU/device.

El código destinado al host es C/C++ estándar, mientras que el destinado al device es ANSI C extendido con algunos keywords para etiquetar los Kernels. Durante la compilación, el compilador nvcc separa ambos tipos de códigos y los compila por separado. En el caso en que no haya device alguno en el sistema, se puede compilar para ejecutarlo en el propio host vía emulación.

Cuando se ejecuta un kernel, se indica el número de hilos que ejecutarán el kernel (también el número de bloques que ya veremos qué son más adelante). Cuando todos los hilos terminan se continúa ejecutando el código serie en la CPU hasta que se vuelve a llamar a un nuevo kernel.

La fuente está integrada en un fichero foo.cu

Este es compilado con cudacc y se obtienen dos ficheros foo.s (lenguaje ensamblador) para GPU y foo.cpp para CPU

El controlador de CUDA compilador (NVCC) utiliza un preprocesador (cudafe) que divide la aplicación en la CPU y la GPU partes, creando un archivo preprocesado C para la entrada de GPU.

La GPU de entrada es procesada por nuestro compilador Open64 (llamado nvopencc), que emite un lenguaje ensamblador que hemos creado llamado PTX (Ejecución de subprocesos en paralelo).

PTX ofrece un modelo de máquina virtual que múltiples herramientas pueden orientar, y que es independiente del procesador subyacente. Algunas características importantes de PTX son que tiene:

La PTX se pasa entonces a OCG, que asigna los registros y la lista de las instrucciones de acuerdo con el chip en particular que se está utilizando(programación) y optimización.

# Compilación programas CUDApág 80

- Cualquier fichero fuente conteniendo extensiones CUDA debe compilarse con NVCC

## Enlazado

Cualquier ejecutable CUDA requiere dos librerías dinámicas:

La CUDA runtime library (**cudart**) y la CUDA core library (**cuda**)

- Un ejecutable compilado en device emulation mode (**nvcc –deviceemu**) se ejecuta en el host vía CUDA runtime
  - No necesita un dispositivo CUDA y cada hilo se corresponde con un hilo del host
  -
- Durante la ejecución en device emulation mode, uno puede:
  - Emplear depuración nativa en el host (puntos de ruptura, inspección,…)
  - Acceder cualquier dato específico del dispositivo desde el host y viceversa
  - Llamar cualquier función del host desde el dispositivo (p.e. **printf**) y vice-versa
  - Detectar situaciones de deadlock causadas por un uso inapropiado de **__syncthreads**

## Fallos en Device Emulation Mode

- Los hilos emulados se ejecutan secuencialmente, por lo tanto, acceso simultáneo de las mismas direcciones de memoria por múltiples hilos pueden producir resultados diferentes.
- Diferenciar punteros del dispositivo en el host o punteros del host en el dispositivo pueden producir resultados correctos en device emulation mode, pero generarán un error al ejecutarse en el dispositivo

Coma flotante→ diferentes salidas y juegos de instrucciones y empleo de precision extendida

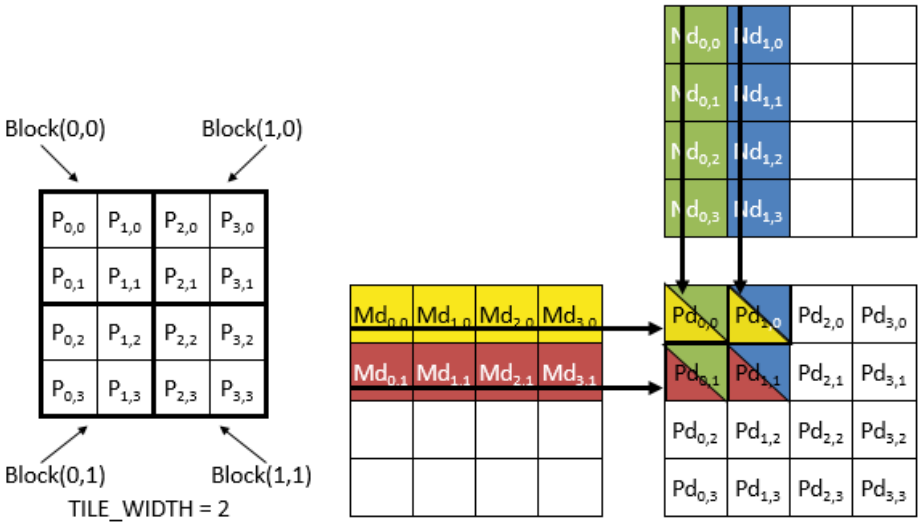
HILOS EN CUDA -Pág 86

## Calling a kernel function

- **kernel<<<dim3 dG, dim3 dB>>>(...)**
  - **Execution Configuration ("<<< >>>")**
  - **dG - dimension and size of grid in blocks**
    - **Two-dimensional: x and y**
    - **Blocks launched in the grid: dG.x * dG.y**
  - **dB - dimension and size of blocks in threads:**
    - **Three-dimensional: x, y, and z**
  - **Threads per block: dB.x * dB.y * dB.z**
- **Unspecified dim3 fields initialize to 1**

## Multiplicación de matrices usando múltiples bloques

- Partimos Pd en teselas
- Cada bloque calcula una tesela de Pd
  - Cada hilo calcula un elemento
  - Bloques de tamaño igual a las teselas




IMPORTANTE

## Bloques de hilos en CUDA

- Todos los hilos de un bloque ejecutan el mismo kernel (SPMD)
- El programador declara bloques:
  - Tamaño de los bloques: de 1 a 768* hilos concurrentes (EN LA G80)
  - Forma de los bloques: 1D, 2D, ó 3D
  - Dimensiones de los bloques en hilos

Los hilos tienen un ID dentro del bloque→ El programa usa esa ID para seleccionar la tarea y los datos con los que trabajar

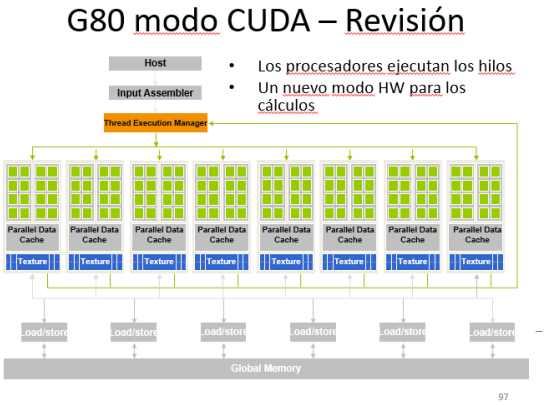
- Los hilos de un mismo bloque comparten datos y se pueden sincronizar mientras realizan sus tareas

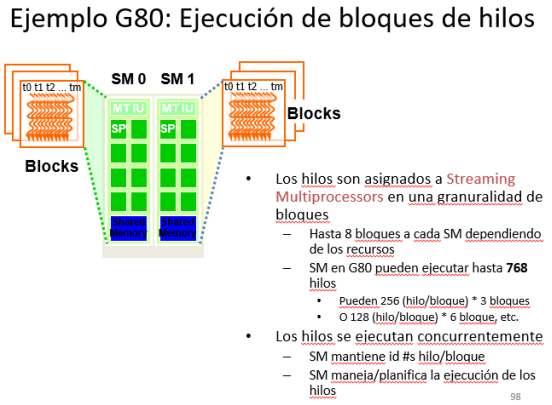
- Hilos de diferentes bloques no cooperan
  - Cada bloque puede ejecutarse en cualquier orden en relación con otros bloques

## Escalabilidad transparente

El Hardware puede asignar bloques a cualquier procesador en cualquier momento

- Un kernel se escala a lo largo de cualquier número de procesadores paralelos
- **Cada bloque se puede ejecutar en cualquier orden relativo a otros bloques.**




**IMPORTANTE**

## Ejemplo G80: Ejecución/Planificación de bloques de hilos

A la hora de ejecutarse, todos los hilos de un mismo bloque se agrupan en conjuntos de 32 → "warp".

Los hilos de un warp se planifican en los 8 cores de un SM para aprovechar el pipeline de 4 etapas.

*Resultado*: ejecución paralela de 32 hilos en el mismo instante de tiempo → SIMD

Ejemplo

1 bloque de 256 hilos se divide en 256/32=8 warps que son planificados para la mejorar la eficiencia.

Cada bloque activo se divide en warps.

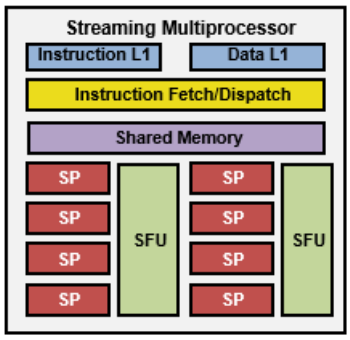
Los hilos de un warp son ejecutados *físicamente en paralelo.*

Warps y bloques se *planifican* en el SM. (Los warps son unidades de planificación en SM.

Si 3 bloques son asignados a una SM y cada bloque tiene 256 hilos, ¿cuántos warps hay en cada SM?

Cada bloque se divide en 256/32 = 8 warps

Hay 8 * 3 = 24 warps




Los SM implementan la planificación de warps sin sobrecarga:

- Los warps cuya siguiente instrucción tiene sus operandos listos para su uso son elegibles para su ejecución
- Los warps elegibles son seleccionados para ejecución de acuerdo a una política de planificación priorizada
- Todos los hilos de un mismo warp ejecutan la misma instrucción cuando son seleccionados

# Ejemplo G80: Consideraciones sobre la granularidad de bloques

**¿Para la multiplicación de matrices debemos usar bloques de 8X8, 16X16 ó 32X32?**

- **Para 8X8, tenemos 64 hilos por bloque. Como cada SM puede ejecutar hasta 768 hilos, hay 12 bloques. Pero como cada SM puede albergar sólo 8 bloques, ¡sólo 512 hilos irán a cada SM!**
- **Para 16X16,tenemos 256 hilos por bloque. Cada SM acoge a 3 bloques y conseguimos máxima capacidad (salvo que otras consideraciones indiquen otras estrategias).**
- **Para 32X32, tenemos 1024 hilos por bloque. Ni siquiera caben en un SM.**
-

**Estrategia "tiling":** técnica de aprovechamiento de los recursos de la shared memory (16KB).

Se divide el problema en bloques de computación con subconjuntos que quepan en shared memory.

Se maneja a nivel de bloque de hilos:

1. Carga de un subconjunto de global memory a shared memory, usando varios hilos para aprovechar el paralelismo a nivel de memoria.

2. Se realizan las operaciones sobre cada subconjunto en shared memory.

3. Se copian los resultados desde shared memory a global memory

## Extensiones del lenguaje: Variables (built-in)

- **dim3 gridDim**;
  - Dimensiones de un grid en boques -**gridDim.z no se usa**-.
- **dim3 blockDim**; → define la cantidad de hilos
  - Dimensiones de un bloque en hilos
- **dim3 blockIdx**;

  - Índice de un bloque dentro de la cuadrícula. Definido como una pero podemos meter 3 datos

- **dim3 threadIdx**;
  - Índice de un bloque dentro del bloque

## Componentes del Runtime Común: Funciones Matemáticas

  - Cuando se ejecutan en el anfitrión, si está disponible, se ejecuta la del runtime de C
  - Sólo están soportadas en tipos escalares, no en los vectoriales.

## Operaciones enteras en la GPU

Una **operación atómica** es una operación en la que un procesador puede simultáneamente leer una ubicación y escribirla en la misma operación del bus. Esto previene que cualquier otro procesador o dispositivo de E/S escriba o lea la memoria hasta que la operación se haya completado.

Los **procesadores escalares** son el tipo más simple de procesadores. Cada instrucción de un **procesador escalar** opera sobre un dato cada vez. En contraposición, en **un** procesador vectorial una sola instrucción opera simultáneamente **sobre un conjunto** de datos.

Se denomina **tiempo de ejecución** (*runtime* en inglés) al intervalo de tiempo en el que un programa de computadora se ejecuta en un sistema operativo. Este tiempo se inicia con la puesta en memoria principal del programa, por lo que el sistema operativo comienza a ejecutar sus instrucciones. El intervalo finaliza en el momento en que éste envía al sistema operativo la señal de terminación,

Componente del Runtime del anfitrión

- Proporciona funciones para:
  - Manejo del dispositivo, memoria y errores.
- Inicializa el runtime la primera vez que una función es llamada
- Un hilo del anfitrión puede ejecutar código en un único dispositivo
  - Se necesitan múltiples hilos en el anfitrión para ejecutar código en múltiples dispositivos

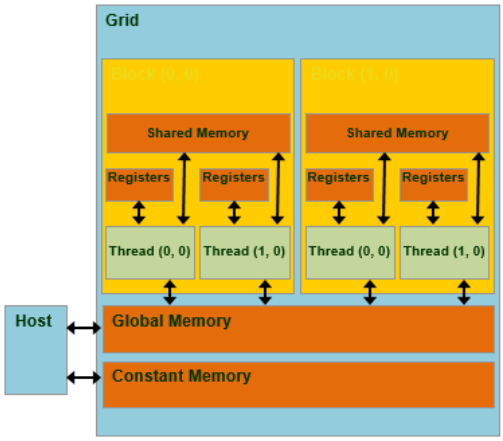
**void __syncthreads(); → Sincroniza los hilos para evitar problemas RAW y WAW al usar memoria compartida**

# Memoria en CUDA – p112

# Ejemplo G80: Implementación de la Memoria en CUDA

Cada hilo puede:

- Leer/escribir registros por hilo
- Leer/escribir memoria local por hilo
- Leer/escribir memoria compartida por bloque
- Leer/escribir memoria global por cuadrícula
- Leer/escribir memoria constante por cuadrícula

- 


# IMPORTANTE

**Memoria global →768 MB – 1024 MB (G80)  Por aplicación por ahí ponía también 1,5GB**

- Todos los hilos que se ejecutan en la GPU tienen acceso R/W al mismo espacio global de memoria
- Comunicación entre cuadrículas (grid)
- Es off-chip
- El acceso es el más lento y no es cacheado
- Reside en la memoria del dispositivo
- Copiar el resultado de la memoria compartida a la global
- -Son contenidos visibles desde la CPU
- -No son visibles en la memoria compartida desde la GPU
- -Memoria de constants y texturas son siempre y solo de lectura

**Memoria Local→ 16KB (G80)**

- **Cada hilo** tiene su propia memoria local(privada
- Es de las más lentas de la GPU y no es cacheada
- Es off-chip (y no como pone en la imagen)
- Este espacio de memoria es gestionado por el compilador
- Variables automáticas y registros
- Usada por el compilador automáticamente para alojar variables cuando hace falta
- Cada hilo tiene su propio conjunto de registros, el programador no tiene control sobre estos, y son utilizados igual que los de propósito general de la CPU

**Memoria Compartida →16KB (G80)**

- **Cada bloque** tiene su espacio de memoria tan rápida como los registros prácticamente
- Puede ser accedida por cualquier hilo del bloque, del mismo SM vamos(lectura/escritura)
- Los hilos de un bloque se pueden comunicar a través de ella.
- No se usa explícitamente por los shader
- Los píxeles no deben hablar entre ellos
- Es on-chip
- Su tiempo de vida es igual al del tiempo de vida del bloque
- Organizar los datos de manera que quepan en la memoria compartida

**Registros→ 8 KB por multiprocesador (G80)**

- Es on chip

- Es la más rápida de la GPU
- Solo son accesibles por cada hilo
- Este espacio de memoria es gestionado por el compilador

**Memoria Constante→ 64KB (8KB por TPC) (G80)**

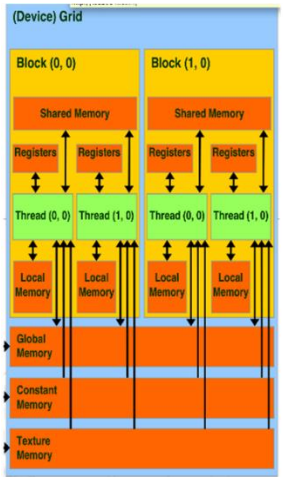
- Es rápida y off-chip, aunque cacheada
- Para el dispositivo es solamente de lectura
- Puede ser vista como una caché a memoria global más que como un espacio distinto
- **La memoria de texturas** cuenta con características similares **(8KB por TPC) (G80)**
- **Es la DRAM y es más lenta que la compartida, eficiente para sólo lectura**
- **Es parte de la memoria global**
- **La CPU puede leer y escribir y solo es de lectura para los hilos de la GPU**
- **Ofrece mayor ancho de banda cuando grupos de hilos acceden al mismo dato**

**Memoria de texturas (8KB por TPC) (G80)**

**-La memoria de texturas** cuenta con características similares a la constante

-Controlada por el programador y puede beneficiar aplicaciones con localidad espacial donde el acceso a memoria global es un cuello de botella

**Ejemplo →** 3 programas de 5Kb caben en la memoria compartida de 16kb




# TIEMPOS DE ACCESO Instrucciones

A nivel de instruccion, cada multiprocesador o cluster toma los siguientes ciclos de reloj para llevar a cabo el lanzamiento de instrucciones:

Instrucciones aritmeticas: 4 ciclos de reloj para suma, producto y MAD sobre operandos en coma flotante, suma y operaciones a nivel de bit en operandos enteros, y otras operaciones como máximo, mínimo o conversión de tipo. Las operaciones logarítmicas o de cálculo de inversa conllevan 16 ciclos de reloj.

La alta especialización de este tipo de procesadores en operaciones sobre coma flotante queda

demostrada por el coste de una operación de multiplicación sobre datos enteros de

32 bits: 16 ciclos de reloj.

**Los accesos a memoria** tienen un sobrecoste de 4 ciclos de reloj, tanto a nivel de memoria global como de memoria compartida; además, el acceso a memoria global suele acarrear entre 400 y 600 ciclos de reloj

debido a la latencia propia del acceso a niveles inferiores en la jerarquía de memoria.

**Instrucciones de sincronización**: una instrucción de sincronización a nivel de threads

supone un sobrecoste de 4 ciclos de reloj, más el tiempo de espera propio de la

sincronizaci´on de todos los hilos de ejecuci´on.

# IMPORTANTE

## Tipos de variable cualificadoras en CUDA

| Declaración de variable | Memoria | Ambiente | Vida |
|---|---|---|---|
| `__device__ __local__ int LocalVar;` | local | thread | hilo |
| `__device__ __shared__ int SharedVar;` | compartida | block | bloque |
| `__device__ int GlobalVar;` | global | cuadrícula | aplicación |
| `__device__ __constant__ int ConstantVar;` | constante | cuadrícula | aplicación |

- **`__device__`** es opcional al emplear **`__local__`**, **`__shared__`**, o **`__constant__`**
- Variable automáticas, sin cualificar, residen en registros
  - Los arrays residen en la memoria local

Las variables automáticas, excepto los array, son escalares y son declarados en el kernel y en las funciones del dispositivo y son colocadas dentro de los registros. El ámbito de las variables escalares se limita a los hilos individuales.

Las variables “array” automáticas no se almacenan en registros sino en la memoria global, por ello los tiempos de acceso son largos y se pueden producir congestiones.

**Ejemplo del Kernel**

Indicación kernel

```
__global__ void sharedABMultiply(float *a,float* b,float *c,int N){
    __shared__ float aTile[TILE_DIM][TILE_DIM],
    bTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x]=a[row*TILE_DIM+threadIdx.x];
    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
    __syncthreads();
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
    }
    c[row*N+col] = sum;
}
```

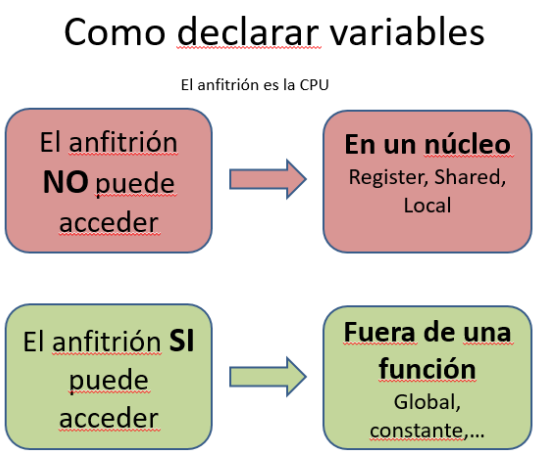
Mem compartida

registros

Identifadores hilo

Barrera para todos los hilos del mismo bloque

# IMPORTANTE




## Restricciones en el tipo de las variables

Punteros sólo pueden apuntar a memoria situada o declarada en la memoria global:

Situada en el anfitrión y pasada a un núcleo:

**__global__ void KernelFunc(float* ptr)**

Obtenida como dirección de una variable global:

**float* ptr = &GlobalVar;**

# Multiplicación de matrices vía memoria compartida

# pag 130

Cada elemento es leido por WIDTH hilos

Carga cada elemento en la memoria compartida y varios hilos emplean la versión local para reducir el ancho de banda → Algoritmos teselados

Se usa la memoria compartida para reutilizar datos de la memoria global.

### Multiplicación teselada

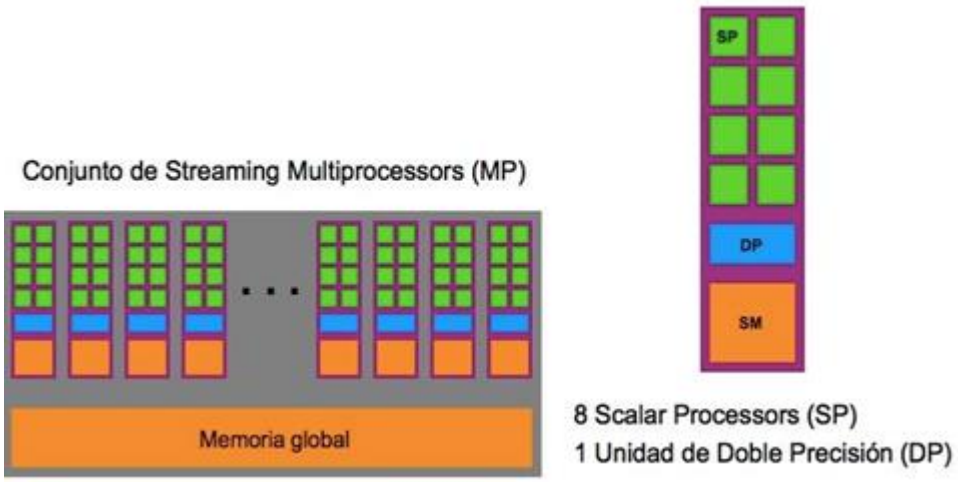
Rompemos la ejecución del kernel en fases de manera que los datos accedidos en cada fase se centran en un subconjunto (tesela) de Md y Nd

Cada bloque calcula una submatriz cuadrada Pdsub de tamaño TILE_WIDTH

Cada hilo calcula un elemento de Pdsub

# Sobre el rendimiento en la G80

# Arquitectura Cuda →IMPORTANTE




Arquitectura many-cores (240 cores simples en la G80)

1 core es 1 SP O SM ¿? Yo creo que SM.--> puede gestionar 512 hilos La agrupación de 8 SPs es 1 SM.

1 SM comparte memoria privada, caché y unidad de control. Puede gestionar 768 hilos

Puede tener 8192 registros

MÁXIMA MEMORIA COMPARTIDA POR UN STREAM MULTIPROCESADOR

En cuanto a una memoria compartida de 16Kb

-Sólo se puede acceder a una memoria de 64Kb

Grid Conjunto de hilos que genera un Kernel

¿Si tengo 3 kernel de 256 hilos ?

En el multiprocesador si , en el procesador no porque se pasa

¿Si tengo 3 kernel de 1024 hilos ?

- El algoritmo de baldosa reduce por un factor de TILE_WIDTH accesos a la memoria global
- Todos los hilos acceden la memoria global para la entrada de elementos
- 2 accesos a la memoria (8 bytes) por cada multiplicación/adición en coma flotante
- G80 soporta 86,4GB/s de acceso memoria global
- Se necesitan 4 ciclos de reloj para ejecutar la misma instrucción para todos los hilos del Warp (en una G80)
- No se puede cargar más de 86,4/4=21,6 Gb/s
- [(86.4/4) x 16] = 345.6 gigaflops, < Theoretical peak
- El código de hecho corre a unos 15 GFLOPS
- Necesita cortar los accesos a memoria para acercarse al pico de 346.5 GFLOPS

ANOTACIONES

- Cada bloque de hilos se divide en Warps y 1 Warp(Unidad de planificación de los SMs) → 32 hilos
-
- 4 ciclos/maquina (para ejecutar la misma instrucción en todos los hilos de un Warp → 86,4 /4 *16 = 346,5 gigaflops
- 512 - SP
- 768 – SM (hilos) → 2 SFU, instrucciones compartida entre 2 hilos
- 16KB – Memoria compartida en cada SM(usada por bloques y en un SM caben 8 bloques)
- Cada bloque no debe usar más de 2KB de memoria compartida
- 8192 – Reg
- 86,4 GB/s de velocidad de acceso a memoria global.
- 346,6 gigaflops
- 367 pico - gigaflops
- Bloque –Compartida
- Hilo – Registros
- Texturas y acceso a memoria global

- **reducir los accesos de memoria global por un factor de tesela**

En un momento, sólo uno de los Warps de un SM será elegido para la obtención y ejecución de una instrucción

El número de Warps en ejecución concurrente en una SM depende del número máximo de hilos por SM.

Los hilos se ejecutan concurrentemente

SM asigna/mantiene las ids de los hilos

SM maneja/planifica la ejecución de los hilos

**IMPORTANTE**

operaciones de coma flotante por segundo son una medida del rendimiento de una computadora, especialmente en cálculos científicos que requieren un gran uso de operaciones de coma flotante. Es más conocido su acrónimo, FLOPS, por el inglés floating point operations per second

MEMORIA GLOBAL  UNA SUMA EN GLOBAL TIENE UN COSTE DE 4 CICLOS LA MEMORIA GLOBAL ESTÁ ENTRE 404 Y 604

MEMORIA COMPARTIDA  4 CICLOS/MÁQUINA, LLEVAR DATOS DE LA MEMORIA GLOBAL A LA COMPARTIDA,  3 ELEMENTOS (X Y Z),

3*4 = 12 CICLOS, SI LO MOVEMOS A MEMORIA  COMPARTIDA COSTE 4 = 12 + 4 = 20 CICLOS.

¿QUÉ ES TESELAR?

EJEMPLO CON WARPS  Pag 166

Si se necesita hacer un acceso a memoria global por cada 4 instrucciones …  y que ese acceso a memoria tiene un coste de 4 ciclos reloj

Necesitamos un mínimo de 13 warps en el mismo bloque para poder esconder el efecto de la latencia de una memoria que tarde 200 ciclos

200/4(acceso a memoria global )* 4 (ciclos reloj)=12.5 aprox 13 warp

Acceden los hilos que estén preparados para ejecutarse, se basan en el que primero llega accede o por edad.

Los registros se reparten dinámicamente a lo largo de los bloques asignados a un SM

Una vez asignados a un bloque, no son accesibles por hilos de otros bloques

Cada hilo del mismo bloque sólo puede acceder a sus registros

# Buffer de Instrucciones (SM)
# Planificación de Warps

Coge una instrucción por ciclo para el warp

-De la caché de instrucciones L1

-En cualquier slot de instrucciones del buffer

Se da una instrucción lista “limpia” para ejecutar en el warp por ciclo

-Desde slot del buffer de instrucciones de un warp por ciclo

-se intentan prevenir riesgos mediante scoreboarding (marcación) de operandos

La selección esta basada en un criterio de round-robin/edad del warp

El SM envía la misma instrucciones a las 32 hebras de un warp

# Puntuación

Todos registros para operandos de todas las instrucciones en el buffer de instrucciones están marcadas (scoreboarded)

Las instrucciones están listas después de que todos los valores necesarios están depositados

Previene/evitan problemas

Las instrucciones listas son elegibles para su ejecución

Separación en cascadas memoria/procesadores

- Cualquier hilo puede continuar ejecutando instrucciones hasta que la puntuación(marcación) previene su ejecución
- Permite a las operaciones de memoria/procesador continuar en la sombra de otras operaciones a la espera

TIPO EXAMEN

¿Cuá de las siguiente matrices pueden correr en un stream multiprocesador?

Número de hilos por bloque

8*8 □ 64 □1 SM □ 768/ 64 = 12 pero como hablamos de SM solo caben 8

16*16 □ 256□1 SM □ 768 / 256 = 3

24*24 □ 576□1 SM □ 768/ 576

32*32 □ 1024□2 SM

1 SM son 8 SP

# Consideraciones de Granularidad

En el ejemplo de las matrices

Para 4x4, tenemos 16 hilos por bloque. Como cada SM puede ejecutar 8 bloques, sólo tenemos 128 hilos en cada SM.

Tenemos 8 warps, cada uno a medio usar

Para 8X8, tenemos 64 hilos por bloque. Sólo utilizamos 512 hilos en cada SM.

Hay 16 warps en cada SM

Cada warp ocupa 4 fragmentos de fila

Para 16X16, tenemos 256 hilos por bloque. En cada SM tenemos 3 bloques con un total 768 hilos (máxima capacidad).

Hay 24 warps por SM

Cada warp ocupa 2 fragmentos de fila

Para 32X32, tenemos 1024 hilos que no caben en la SM

**EJEMPLO**

- Si cada bloque utiliza más de 2 KB de memoria, el número de bloques que pueden residir en cada SM es tal que la cantidad total de memoria compartida usada por estos bloques no exceda de 16 kB, por ejemplo, si cada bloque utiliza 5 kB de la memoria compartida, no más de 3 bloques se pueden asignar a cada SM.
-

Cada bloque de hilos puede tener muchos hilos:
TILE_WIDTH de 16 da 16*16 = 256 hilos

Debe haber muchos bloques de hilos
Una Pd de 1024*1024 da (16*64)*(64*16) = 4096 bloques de hilos
TILE_WIDTH de 16 da para cada SM 3 bloques, 768 hilos (full capacity)

Cada bloque de hilos ejecuta 2*256 = 512 lecturas de float de la memoria global por 256 * (2*16) = 8,192 operaciones de mul/sum.
El ancho de banda de la memoria no es un problema ya

# Más consideraciones sobre la G80

Cada SM en una G80 tiene 16KB de memoria compartida
El tamaño de una SM depende de la implementación!
Para TILE_WIDTH = 16, cada bloque de hilos usa 2*256*4B = 2KB de memoria compartida
La memoria compartida puede tener hasta 8 bloques de hilos ejecutándose activamente
Esto permite hasta 8*512 = 4,096 lecturas. (2 por hilo, 256 hilos por bloque)
El modelo de hilos limita el número de bloques de hilos a 3, la memoria compartida no es un problema
Para TILE_WIDTH = 32, tendríamos 2*32*32*4B= 8KB de memoria compartida por bloque, permitiendo sólo 2 bloques activos al mismo tiempo
Usar teselas de tamaño 16x16 reduce el acceso a la memoria global por un factor de 16
El ancho de banda de  86.4B/s puede soportar (86.4/4)*16 = 347.6 GFLOPS!

# COMPUTE CAPABILITIES

Para hacer uso óptimo de las GPUs, Nvidia utiliza un formato estandarizado para especificar estas características

Dos cifras: La primera indica la versión y la segunda la revisión (las actuales están en 2.0 y 3.0)

| Feature Support (Unlisted features are supported for all compute capabilities) | Compute Capability 1.0 | 1.1 | 1.2 | 1.3 | 2.x, 3.0 | 3.5 |
|---|---|---|---|---|---|---|
| Atomic functions operating on 32-bit integer values in global memory (Atomic Functions) | No | Yes | | | | |
| atomicExch() operating on 32-bit floating point values in global memory (atomicExch()) | | | | | | |
| Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions) | No | | Yes | | | |
| atomicExch() operating on 32-bit floating point values in shared memory (atomicExch()) | | | | | | |
| Atomic functions operating on 64-bit integer values in global memory (Atomic Functions) | | | | | | |
| Warp vote functions (Warp Vote Functions) | | | | | | |
| Double-precision floating-point numbers | No | | | Yes | | |
| **Feature Support (Unlisted features are supported for all compute capabilities)** | **Compute Capability 1.0** | **1.1** | **1.2** | **1.3** | **2.x, 3.0** | **3.5** |
| Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions) | No | | | | Yes | |
| Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd()) | | | | | | |
| __ballot() (Warp Vote Functions) | | | | | | |
| __threadfence_system() (Memory Fence Functions) | | | | | | |
| __syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions) | | | | | | |
| Surface functions (Surface Functions) | | | | | | |
| 3D grid of thread blocks | | | | | | |
| Funnel shift (see reference manual) | No | | | | | Yes |

# Consideraciones sobre el rendimiento

# Parallel Computing Patterns

- Map (one to one)
- Transpose (one to one)
- Gather (many to one)
- Scatter ( one to many)
- Stencil (several to one)
- Reduce (all to one)
- Scan/sort (all to all)

# Sobre los Hilos

# Single-Program Multiple-Data (SPMD)

CUDA integra programas CPU + GPU en C

El código secuencial se ejecuta en la CPU

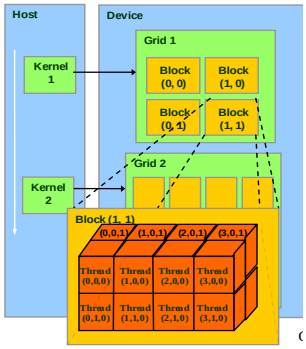
El núcleo paralelo se ejecuta en bloques de hilos en la GPU

# Cuadrículas e hilos

- Un núcleo se ejecuta en una cuadrícula de hilos
  - Todos los hilos acceden a la memoria compartida
- Un bloque es una tanda de hilos que pueden cooperar:
  Sincronizando su ejecución usando barreras

  Compartiendo datos vía memoria de baja latencia

  hilos de diferentes bloques no pueden cooperar




Ciclo de vida de un hilo en HW

Se lanza la cuadrícula en la SPA

Los bloques de hilos se distribuyen secuencialmente a todos los SMs

- Potencialmente >1 bloque de hilos por SM

Cada SM lanza Warps de hilos

2 niveles de paralelismo

Los SMs planifican y ejecutan los Warps que estén listos para ejecutarse

A medida que Warps y bloques de hilos se completan, los recursos se liberan

- SPA distribuye más bloques de hilos

El TPC Texturre processor es un concepto que encontramos en las GPU NVIDIA. El G80 y GT200 arquitecturas, TPC, o un racimo Textura / procesador, es un grupo formado por varios SMs Stream Multiprocesador, una unidad de textura y un poco de lógica de control.

El SM es un multiprocesador Streaming y se compone de varios productos especiales (o procesadores de streaming), SFUs varias (o Unidad Función especial - la unidad que se utiliza para las funciones trascendentales como el seno o coseno).

Un procesador de Streaming Multiprocessor se llama también un núcleo CUDA (en la nueva terminología Fermi).

El TPC de una GPU G80 tiene 2 SM, mientras que el TPC de un GT200 tiene 3 SMS.

El SP es el elemento de procesamiento real que actúa sobre los datos de vértice o pixel.

Varios TPCs pueden ser agrupados en un alto nivel de entidad llamada  Streaming Processor Array.

Pero en la nueva GPU de NVIDIA, el GF100 / Fermi, el TPC ya no es válida: sólo quedan el SMs. También podemos decir que en la arquitectura Fermi, un TPC = a SM.

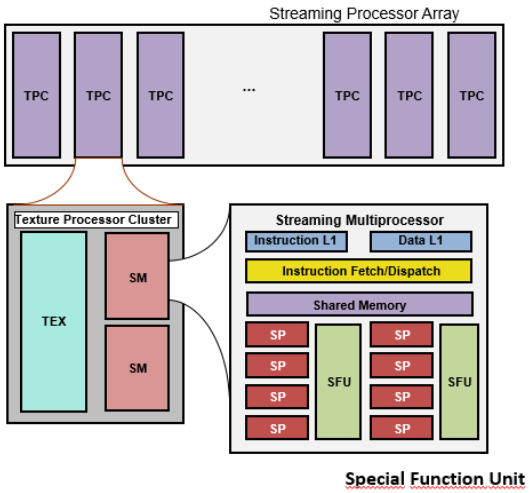
En arquitectura Fermi, un SM se compone de dos SIMD 16-forma de unidades. Cada SIMD de 16 vías tiene 16 SP entonces un SM en Fermi tiene 32 SP o 32 núcleos CUDA.

Unidad Textura

Procesa un grupo de 4 hilos por ciclo

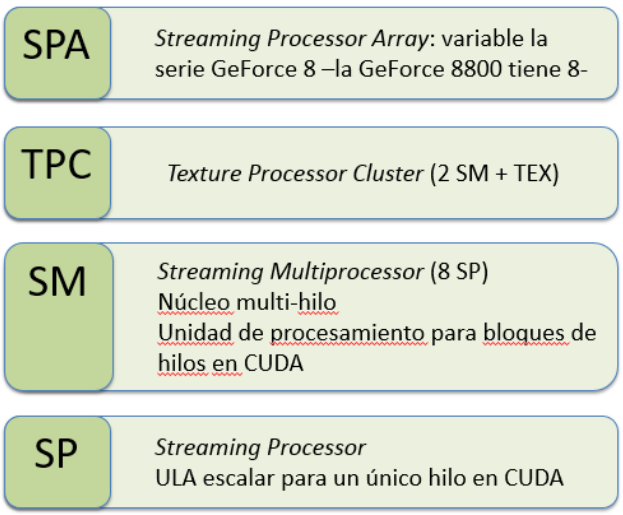
Instruciones de textura: las fuentes son coordenadas de textura y las salidas filtran las muestras

Cada unidad de textura tiene 2 SMs





# Terminología de Procesadores CUDA




## Thread Execution Manager → Genera cuadrículas basándose en llamadas a núcleos

## Arquitectura de la memoria de las SMs

Los hilos de un bloque comparten datos & resultados

En memoria global y memoria compartida

Se sincronizan mediante barreras

La Memoria Compartida se asigna por Bloques

Mantiene los datos próximos al procesador

Minimiza las idas y venidas a la memoria global

La memoria compartida es dinámica y se asigna por bloques

## IMPORTANTE

## Fichero de registros de las SMs

Fichero de registros (RF)

32 KB (8K entradas) para cada SM en G80

Cascada TEX (TEX pipe) puede leer/escribir RF

2 SMs comparten 1 TEX

Cascada de carga/almacenamiento puede leer/escribir RF

Fichero de registros (RF)

32 KB por cada registro de memoria (8K entradas) para cada SM en G80 (8192 registros de un maximo de 32KB = 4b)

Puede dar 4 operandos/ciclo de reloj

El cauce de texturas (TEX pipe) también puede leer/escribir en RF

2 SMs comparten 1 TEX

El cauce de carga/ Almacenamiento tambien puede leer y escribir RF

## EJERCICIOS

¿Cuántos bloques de hilos puedo ejecutar en un SM si tengo bloques de 16*16 que consumen 3KB de memoria?

¿Y si en lugar de 3KB tengo 6 KB de memoria?

En memoria compartida hay 16KB

Bloques de hilos de 256 □ por cada bloque consume 3 KB

Si tengo 2 Bloques de 256 □entran en 1SM

Si tengo 3 También □ 768 --< 9KB

2ºCaso

256 □ 6KB

512 □ 12 KB

768 □ 18KB

En memoria RAM falla pero en bloques entra

## EJEMPLO MULTIPLICACIÓN DE MATRICES

Si cada bloque tienen 16X16 hilos y cada hilo usa 10 registros, ¿cuántos hilos pueden ejecutarse en cada SM?

Cada bloque necesita 10*256 = 2560 registros

8192 = 3 * 2560 + change

Luego, 3 bloques pueden correr en una SM teniendo en cuenta los registros

¿Qué pasa si cada hilo incrementa su consumo de registros por 1?

Cada bloque requiere 11*256 = 2816 registros

8192 < 2816 *3

Sólo pueden ejecutarse 2 bloques en una SM, una reducción de 1/3 en el paralelismo

**Procesador de grano fino(operaciones elementales)**

**Procesador de grano grueso(operaciones complejas)**

# Ejemplo: ILP* vs. TLP**

Supongamos que un núcleo tiene bloques de 256 hilos, 4 instrucciones independientes de acceso a la memoria global por hilo y que cada hilo usa 10 registros

Los accesos a la memoria global consumen 200 ciclos

cada instrucción tarda 4 ciclos de reloj para proceso , por lo que las 4 instrucciones independientes dan una holgura de 16 ciclos para el acceso a la memoria

Se ejecutan 3bloques en cada SM

Peor, uno necesite 200/(4*4) = 13 warps para tolerar la latencia de la memoria

hay 4 instrucciones independientes entre una carga de memoria global y su uso

cada instrucción tarda 4 ciclos

Si un compilador usa 1 registro adicional para cambiar al patrón de acceso de manera que se llevan a cabo 8 instrucciones independientes de acceso a la memoria global

Sólo pueden correr 2 bloques en cada SM

Peor, uno necesite 200/(8*4) = 7 warps para tolerar la latencia de la memoria

Dos bloques tienen 16 warps. ¡El rendimiento puede ser mayor!

**Paralelismo entre los procesadores se llama paralelismo a nivel de hilo (TLP).**

**Los procesadores se incorporan cantidades crecientes de paralelismo.**

**Las instrucciones se ejecutan de forma simultánea por diferentes unidades funcionales, y en los procesadores superescalares, incluso se inicia ("acciones") en paralelo.**

**Esto se conoce como nivel de instrucción paralelismo, o ILP.**

# IMPORTANTE

# Coalescencia

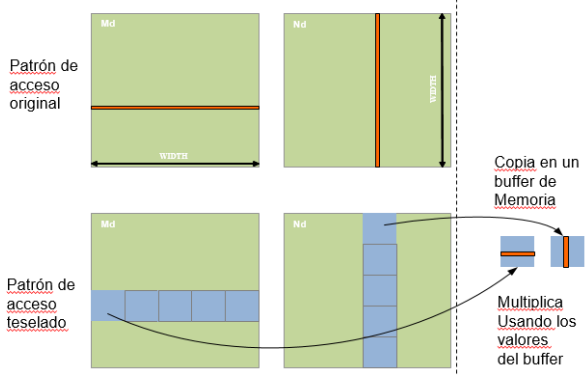
es el acto de la fusión de dos bloques libres adyacentes de memoria

Al acceder a la memoria global, el rendimiento máximo se alcanza cuando todos los hilos de medio warp acceden a direcciones contiguas.

La coalescencia y técnicas relacionadas como la compactación montón, se pueden usar en la recogida de basura.

Un algoritmo de azulejos se puede utilizar para permitir la coalescencia

Uso de la memoria compartida para mejorar la coalescencia




# Constantes

Las constantes  se almacenen en la DRAM (memoria de video), y se acceden a través de una caché en el chip

Una constante se puede enviar simultáneamente a todos los hilos de un warp

Es una manera extremadamente eficiente de acceder a valores comunes a todos los hilos de un bloque

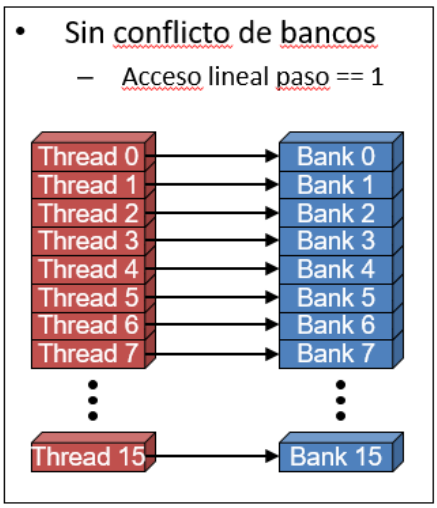
# Arquitectura de Memoria Paralela

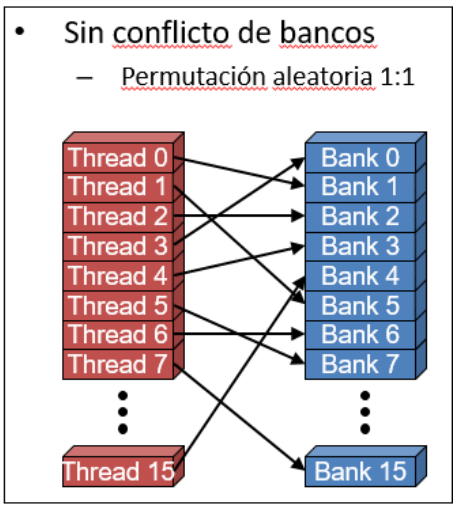
-La memoria se divide en bancos para conseguir alto rendimiento

-Cada banco puede servir una dirección por ciclo

-Los accesos simultáneos a los bancos producen conflicto → Se serializan

## Ejemplo: Direccionamiento de Bancos




Si tengo un paso igual a 1 no se producen conflictos.

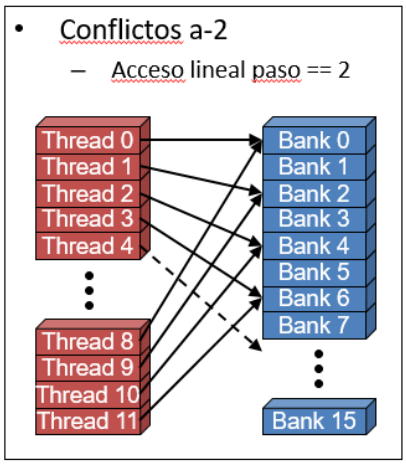
Si el número tiene factor común con el número de bancos va  a haber conflicto

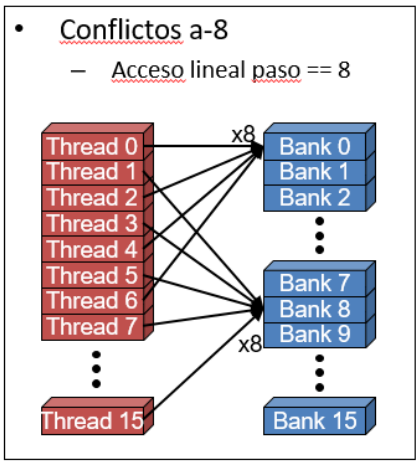
¿con el 3 va a haber conflicto ?

No porque no es un factor  de 2^4

¿y con 6?

6 = 2*3




# ¿Cómo se mapean las direcciones a los bancos en la G80?

Cada banco tiene un ancho de banda de 32 por ciclo reloj

Palabras sucesivas de 32-bit se asignan a bancos sucesivos

G80 tiene 16 bancos, luego banco que buscamos = dirección % 16

Igual que el tamaño de medio warp

No hay conflictos entre diferentes medios warps, sólo dentro de uno mismo

## Conflicto de bancos en la memoria compartida

La memoria compartida es tan rápida como los registros si no hay conflictos

**Casos sin conflictos:**

- Todos los hilos de medio warp acceden a bancos diferentes
- Todos los hilos de medio warp acceden a la misma dirección de memoria (no hay conflicto, broadcast)

**Casos con conflictos:**

- Conflicto: distintos hilos del mismo medio warp acceden al mismo banco
- Se debe serializar el acceso
- El coste es igual al número max de accesos simultaneos a un único banco

Primero divido 4/2= 2.

Entonces divido este resultado entre 2: 2/2= 1.

Luego : $4 = 2^2$. O sea (4) será igual a los factores entre los cuales hemos dividido, como se ve en los ejemplos que dividimos primero entre dos y segundo entre dos nuevamente esto es 2 por 2 $=2^2$.
Ahora hacemos lo mismo con el (6) y divido entre 2: 6/2=3. y este resultado lo divido entre( 3) esto es 3/3 =1. Luego: 6 = (2)(3) igual a los factores entre los cuales dividimos.
16/2 = 8. 8/2 = 4. 4/2 = 2. 2/2 =1. Los factores que se usaron para dividir fueron:
2, 2, 2, 2, Luego 16 = (2)(2)(2)(2)= $2^4$.

Ahora saco el factor común con su menor exponente: $4 = 2^2$.
6 = (2)(3).
16= $2^4$.

Factor común 2.

Factor común significa una cantidad que tiene que estar en cada una de ellas y la unica cantidad común es el dos en este caso.

**IMPORTANTE**

# Control de flujo

# Instrucciones de Control de Flujo

El mayor problema de rendimiento lo plantea la divergencia por saltos

- Los hilos de un mismo warp toman distintos caminos
- Los distintos caminos de ejecución se serializan en la G80
  - Los caminos de control tomados por los hilos de un warp se recorren de uno en uno hasta que se acaban

Un caso frecuente: evitar la divergencia cuando el salto depende de la ID del hilo

- Con divergencia: `If (threadIdx.x > 2){…}`
  - Crea dos caminos de control para los hilos de un bloque
  - La fineza del salto es menor que el tamaño de un warp; los hilos 0, 1 y 2 siguen caminos distintos al resto de los hilos en el primer warp
- Sin divergencia: `If (threadIdx.x / WARP_SIZE > 2){…}`
  - También crea dos caminos de control para los hilos de un bloque
  - La fineza del salto es un múltiplo del tamaño del warp; todos los hilos de un warp siguen el mismo camino

# Algoritmos fundamentales para gpu

# Reducción Paralela

Dado un array de valores, “reducirlo” a un único valor en paralelo

Ejemplo

Suma: la suma de todos los valores del array

Max: máximo de todos los valores del array

Implementación típica:

Reducir a la mitad el número de hilos, sumar dos valores por hilo

Lleva log(n) pasos para n elementos y requiere n/2 hilos

## Ejemplo: Reducción de un vector

Supongamos que se lleva a cabo una reducción in-situ usando memoria compartida

El vector original esta en la memoria global

La memoria compartida se usa para almacenar sumas parciales

Cada iteración acerca la suma parcial a la suma final

La solución final estará en el elemento 0

## Algunas Observaciones

En cada iteración, dos controles de flujo se atraviesan secuencialmente en cada warp

- Hay hilos que calculan adiciones e hilos que no
- Los hilos que no calculan adiciones pueden incurrir en ciclos extra dependiendo de la implementación de la divergencia

No más de la mitad de los hilos se están ejecutando de manera simultánea

- Todos cuyo índice sea impar se desechan desde el comienzo
- En media, menos de ¼ de los hilos están activados para todos los warps a lo largo del tiempo
- Después de 5 iteraciones, warps enteros en cada bloque son desechados, poca utilización de recursos pero sin divergencia

Esto puede continuar hasta 4 iteraciones más (512/32=16= 24), cuando cada iteración sólo tiene activo un hilo hasta que todos los warps se retiran

# Registros, ILP y mezcla de instrucciones

# Ejemplo: ILP* vs. TLP**

Supongamos que un núcleo tiene bloques de 256 hilos, 4 instrucciones independientes de acceso a la memoria global por hilo y que cada hilo usa 10 registros

- Los accesos a la memoria global consumen 200 ciclos
- Se ejecutan 3 bloques en cada SM

Si un copilador usa 1 registro adicional para cambiar al patrón de acceso de manera que se llevan a cabo 8 instrucciones independientes de acceso a la memoria global

- Sólo pueden correr 2 bloques en cada SM
- Pero, uno necesite 200/(8*4) = 7 warps para tolerar la latencia de la memoria
- Dos bloques tienen 16 warps. ¡El rendimiento puede ser mayor!

* Instruction Level Parallelism

** Thread Level Parallelism

## Ejemplo: Asignación de recursos

Supongamos ahora que el programador declara otra variable automática en el núcleo y choca el número de registros utilizados por cada hilo a 11. Suponiendo que los mismos 16 16 bloques, cada bloque ahora requiere 11 x16 x16 = 2,816 registros. El número de registros requeridos por 3 bloques es ahora 8448, que superan el límite de registro. Como se muestra en la Figura 6.12b, el SM se ocupa de esta situación mediante la reducción del número de bloques por 1, lo que reduce el número de domicilio necesaria para 5632. Esto, sin embargo, reduce el número de hilos que se ejecutan en un SM 768-512; es decir,

Centrarse en el área dentro de la primera SP. El área de la computación básica es lo que queremos maximizar el rendimiento.

Esto muestra la naturaleza impredecible de la optimización. Una optimización se aplica lo que aumenta la utilización de SP y la eficiencia del código por bloque, pero en el código de uso de los registros, dando inicio a un bloque.

## IMPORTANTE

## Precargado

Se puede usar un doble buffer para obtener mejor mezcla de instrucciones en cada hilo

RESUMIR

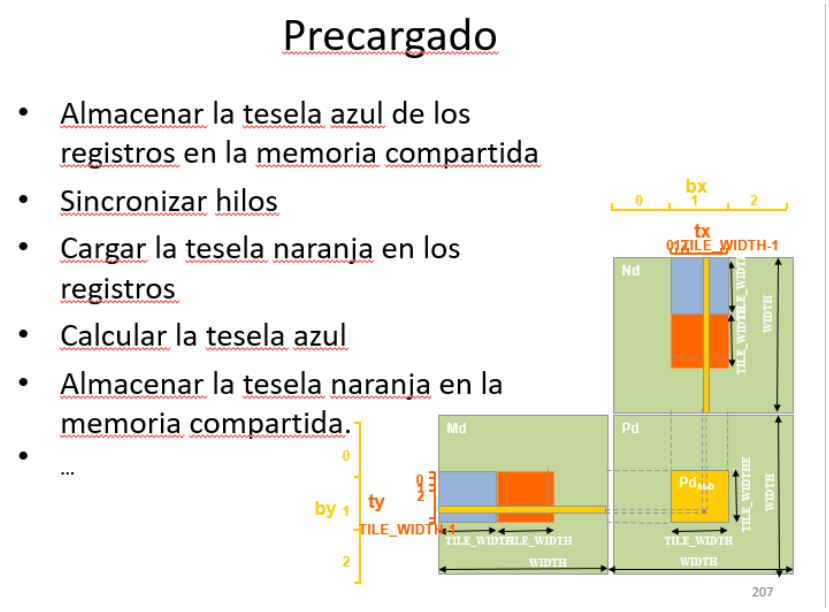
En algunas situaciones se puede dar el caso de que los hilos tengan pocas instrucciones que ejecutar entre el acceso a la memoria y el consumo de datos accedidos.

Una buena estrategia es la de pre-captar los siguientes datos mientras se están consumiendo los actuales. Así aumentamos el número de instrucciones a ejecutar entre accesos a la memoria y los consumidores de datos accedidos.

Una solución útil, complementaria al problema es precapturar los próximos elementos de datos, mientras que el consumo de los elementos de datos actuales, lo que aumenta el número de instrucciones independientes entre el accesos a memoria y los consumidores de los datos consultados. técnicas de recuperación previa se combinan a menudo con azulejos de abordar simultáneamente los problemas de ancho de banda limitado y larga latencia

Optimizamos haciendo una precarga de la segunda tesela

2 técnicas: A través de instrucción o a través de hilo




Unrolling (desenrollado)→ Eliminación de instrucciones condicionales y de salto

# Principales factores de decremento de rendimiento en la G80

Operaciones de alta latencia

- Evitar esperas ejecutando otros hilos

Esperas y burbujas en la cascada

- Sincronización de barrera
- Divergencia por saltos

Saturación de recursos compartidos

- Ancho de banda de la memoria global
- Capacidad de la memoria local

**IMPORTANTE**

# ARQUITECTURA CUDA DE ÚLTIMA GENERACIÓN, ALIAS FERMI

| | CUDA Compute Capability (CCC) | | | | Limitación | Impacto |
|---|---|---|---|---|---|---|
| | 1.0, 1.1 | 1.2, 1.3 | 2.0, 2.1 | 3.0, 3.5 | | |
| Multiprocesadores / GPU | 16 | 30 | 14-16 | 14-16 | Hardware | Escala-bilidad |
| Cores / Multiprocesador | 8 | 8 | 32 | 192 | Hardware | |
| Hilos / Warp | 32 | 32 | 32 | 32 | Software | Ritmo de salida de datos |
| Bloques / Multiprocesador | 8 | 8 | 8 | 16 | Software | |
| Hilos / Bloque | 512 | 512 | 1024 | 1024 | Software | Paralelismo |
| Hilos / Multiprocesador | 768 | 1 024 | 1 536 | 2048 | Software | |
| Registros de 32 bits / Multiprocesador | 8K | 16K | 32K | 64K | Hardware | Conjunto de trabajo |
| Memoria compartida / Multiprocesador | 16K | 16K | 16K 48K | 16K, 32K, 48K | Hardware | |

| GPU | G80 | GT200 | Fermi |
|---|---|---|---|
| **Transistors** | 681 million | 1.4 billion | 3.0 billion |
| **CUDA Cores** | 128 | 240 | 512 |
| **Double Precision Floating Point Capability** | None | 30 FMA ops / clock | 256 FMA ops /clock |
| **Single Precision Floating Point Capability** | 128 MAD ops/clock | 240 MAD ops / clock | 512 FMA ops /clock |
| **Special Function Units (SFUs) / SM** | 2 | 2 | 4 |
| **Warp schedulers (per SM)** | 1 | 1 | 2 |
| **Shared Memory (per SM)** | 16 KB | 16 KB | Configurable 48 KB or 16 KB |
| **L1 Cache (per SM)** | None | None | Configurable 16 KB or 48 KB |
| **L2 Cache** | None | None | 768 KB |
| **ECC Memory Support** | No | No | Yes |
| **Concurrent Kernels** | No | No | Up to 16 |
| **Load/Store Address Width** | 32-bit | 32-bit | 64-bit |

En la nueva versión de Fermi tiene  4 SFU’s

hasta 512 cores (16 SMs dotados de 32 cores cada uno).
Los SPs crecen de 240 a 512, los DPs de 30 a 256 y los SFUs de 60 a 64.
Doble planificador de procesos (scheduler) para los hilos que entran a cada SM.

64 KB de SRAM, repartidos entre la caché L1 y la memoria compartida.

Unidades aritmético-lógicas (ALUs):

Rediseñada para optimizar operaciones sobre enteros de 64 bits.

Admite operaciones de precisión extendida.

La instrucción “madd” (suma y producto simultáneos):

Está disponible tanto para simple como para doble precisión.

La FPU (Floating-Point Unit):

Implementa el formato IEEE-754 en su versión de 2008, aventajando incluso a las CPUs más avezadas.

## IMPORTANTE

## JERARQUÍAS

Fermi es la primera GPU que ofrece una caché L1 típica on-chip, que combina con la shared memory de CUDA en proporción 3:1 o 1:3 para un total de 64 Kbytes por cada multiprocesador (32 cores).

También incluye una caché unificada de 768 Kbytes con coherencia de datos para el conjunto total de cores.

PREGUNTA EXAMEN : COMPARAR JERARQUÍAS ???G80 Y FERNI? PERO SOBRE TODO PREGUNTARA SOBRE KERNEL Y FERMI QUE SON LAS MÁS MODERNAS

Fermi admite la ejecución concurrente de kernels, donde los diferentes núcleos del mismo contexto de aplicación se pueden ejecutar en la GPU al mismo tiempo. la ejecución concurrente de kernels permite que los programas que se ejecutan una serie de pequeños núcleos de utilizar toda la GPU. Por ejemplo, un programa de PhysX puede invocar un solucionador de fluidos y un solucionador de cuerpo rígido que, de ejecutarse de forma secuencial, utilizaría sólo la mitad de los procesadores de rosca disponibles. En la arquitectura Fermi, diferentes núcleos del mismo contexto CUDA pueden ejecutar simultáneamente, lo que permite la máxima utilización de los recursos de la GPU. Los granos de las diferentes contextos de aplicación todavía puede ejecutar secuencialmente con gran eficiencia gracias a la mejora de la conmutación de contexto de actuación.

| | FERMI GF100 | FERMI GF104 | KEPLER GK104 | KEPLER GK110 |
|---|---|---|---|---|
| **Compute Capability** | 2.0 | 2.1 | 3.0 | 3.5 |
| **Threads / Warp** | 32 | 32 | 32 | 32 |
| **Max Warps / Multiprocessor** | 48 | 48 | 64 | 64 |
| **Max Threads / Multiprocessor** | 1536 | 1536 | 2048 | 2048 |
| **Max Thread Blocks / Multiprocessor** | 8 | 8 | 16 | 16 |
| **32-bit Registers / Multiprocessor** | 32768 | 32768 | 65536 | 65536 |
| **Max Registers / Thread** | 63 | 63 | 63 | 255 |
| **Max Threads / Thread Block** | 1024 | 1024 | 1024 | 1024 |
| **Shared Memory Size Configurations (bytes)** | 16K<br>48K | 16K<br>48K | 16K<br>32K<br>48K | 16K<br>32K<br>48K |
| **Max X Grid Dimension** | 2^16-1 | 2^16-1 | 2^32-1 | 2^32-1 |
| **Hyper-Q** | No | No | No | Yes |
| **Dynamic Parallelism** | No | No | No | Yes |

**Compute Capability of Fermi and Kepler GPUs**