INTRODUCCIÓN A LA COMPUTACIÓN PARALELA CON GPUs

Sergio Orts Escolano sorts@dtic.ua.es

Vicente Morell Giménez vmorell@dccia.ua.es

Universidad de Alicante

Departamento de tecnología informática y computación

DEPARTAMENTO DE TECNOLOGÍA INFORMÁTICA Y COMPUTACIÓN

Contenido

- **GPU**
- Paralelismo
- GPGPU
- Introducción a CUDA
- Aplicaciones
- Conclusiones

GPU

- **Unidad de procesamiento gráfico** (GPUs) son procesadores de varios núcleos que ofrecen alto rendimiento

- Procesador dedicado al procesamiento de gráficos u operaciones de coma flotante

GPU

- Hoy en día las **GPU** son muy potentes y pueden incluso superar la frecuencia de reloj de una **CPU** antigua (**más de 800MHz**)

- Arquitecturas Many-core con una cantidad ingente de núcleos que realizan operaciones sobre múltiples datos

Contenido

- GPU
- **Paralelismo**
- GPGPU
- Introducción a CUDA
- Aplicaciones
- Conclusiones

Paralelismo

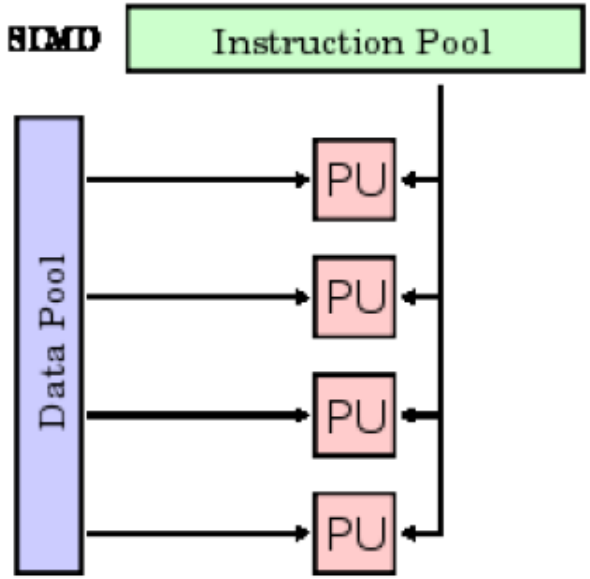
- El **paralelismo** es una forma de computación en la cual varios cálculos pueden realizarse simultáneamente.

- Basado en el principio de dividir los problemas grandes para obtener varios problemas pequeños, que son posteriormente solucionados en paralelo. (Divide y vencerás)

SIMD

Taxonomía de Flynn

	Una instrucción	Múltiples instrucciones
Un dato	SISD	MISD
Múltiples datos	SIMD	MIMD



- **SIMD:** Una instrucción múltiples datos

- Todos los núcleos ejecutan la misma instrucción al mismo tiempo.
- Solo se necesita decodificar la instrucción una única vez para todos los núcleos
- Ejemplo: Unidades vectoriales: suma, producto de vectores, etcétera

Paralelismo

- *Computación paralela sobre GPUs* explicado por los cazadores de mitos: CPU vs GPU

- VÍDEO INCRUSTADO
- NVIDIA_ Adam and Jamie explain parallel processing on GPU's(360p_H.264-AAC)

Contenido

- GPU
- Paralelismo
- **GPGPU**
  - Introducción
  - CPU y GPU
  - Evolución
- Introducción a CUDA
- Aplicaciones
- Conclusiones

GPGPU

- Computación de propósito general sobre procesadores gráficos

- Cómputo de aplicaciones tradicionalmente ejecutadas sobre CPUs
- Aprovechamiento capacidades GPU:
  - Grandes vectores de datos
  - Paralelismo de grano fino SIMD
  - Baja latencia en operaciones en punto flotante

Históricamente GPGPU

- Restricciones:
  - APIs gráficas complejas
  - Capacidades de los Shader reducida
  - Comunicación limitada
    - Entre píxeles
  - Conjunto de instrucciones
    - Falta de operaciones con enteros

Evolución GPU

1. Ordenadores sin tarjeta gráfica.
2. Aparición de **GPU**s para la representación en pantalla.
3. GPUs para procesamiento gráfico: industria del videojuego (**3dfx,NVIDIA,ATI**).
4. En **2004** surge la idea de utilizar las GPU para computación de alto rendimiento (**HPC**).
5. **2007**. NVIDIA ve oportunidad en el mercado del **HPC** y desarrolla **CUDA**. (Lenguaje específico para aprovechar la capacidad de las **GPU**).
6. Actualmente **NVIDIA** ha tenido éxito en la computación **GPGPU**. Los supercomputadores top utilizan **clústers** de **GPUs** para aumentar su potencia computacional.

Oportunidad de mercado

- Si hay un mercado suficientemente grande, alguien desarrollará el producto.
- La oportunidad de ganar dinero dirige la computación, así como la tecnología.
- Se necesitan economías de escala para producir de forma barata, algo que solo pocas empresas pueden permitirse.
- Observar tendencias de mercado

El gran desarrollo de las GPU

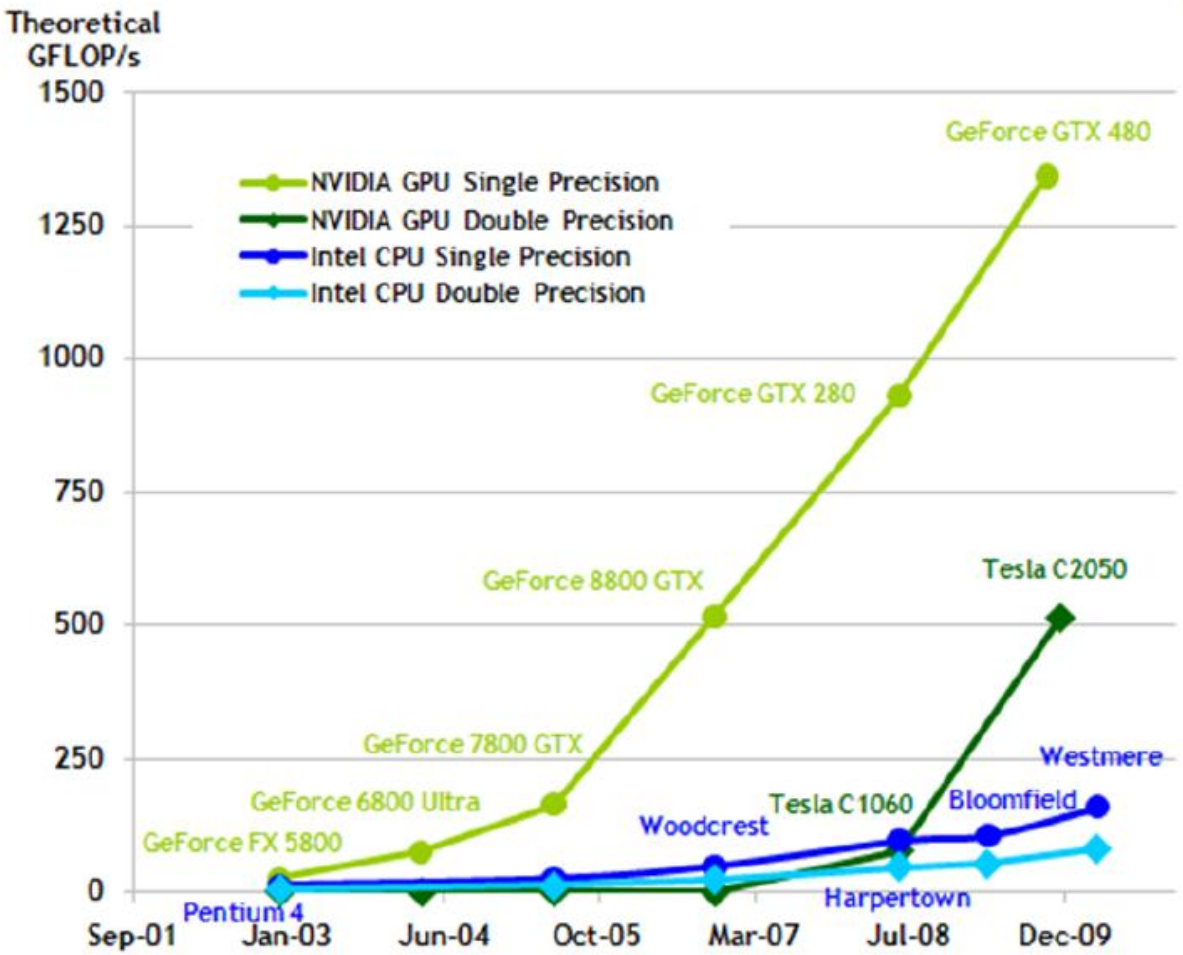
- La evolución de las tarjetas gráficas ha venido acompañado de un gran crecimiento en el mundo de los videojuegos y las aplicaciones 3D
- Gran producción de chips gráficos por parte de grandes fabricantes:
  - NVIDIA
  - AMD(ATI)
  - IBM: desarrollo en colaboración con Sony y Toshiba procesadores Cell
  - Intel: Desarrollo chips GPU Larrabee

Comparación CPU y GPU

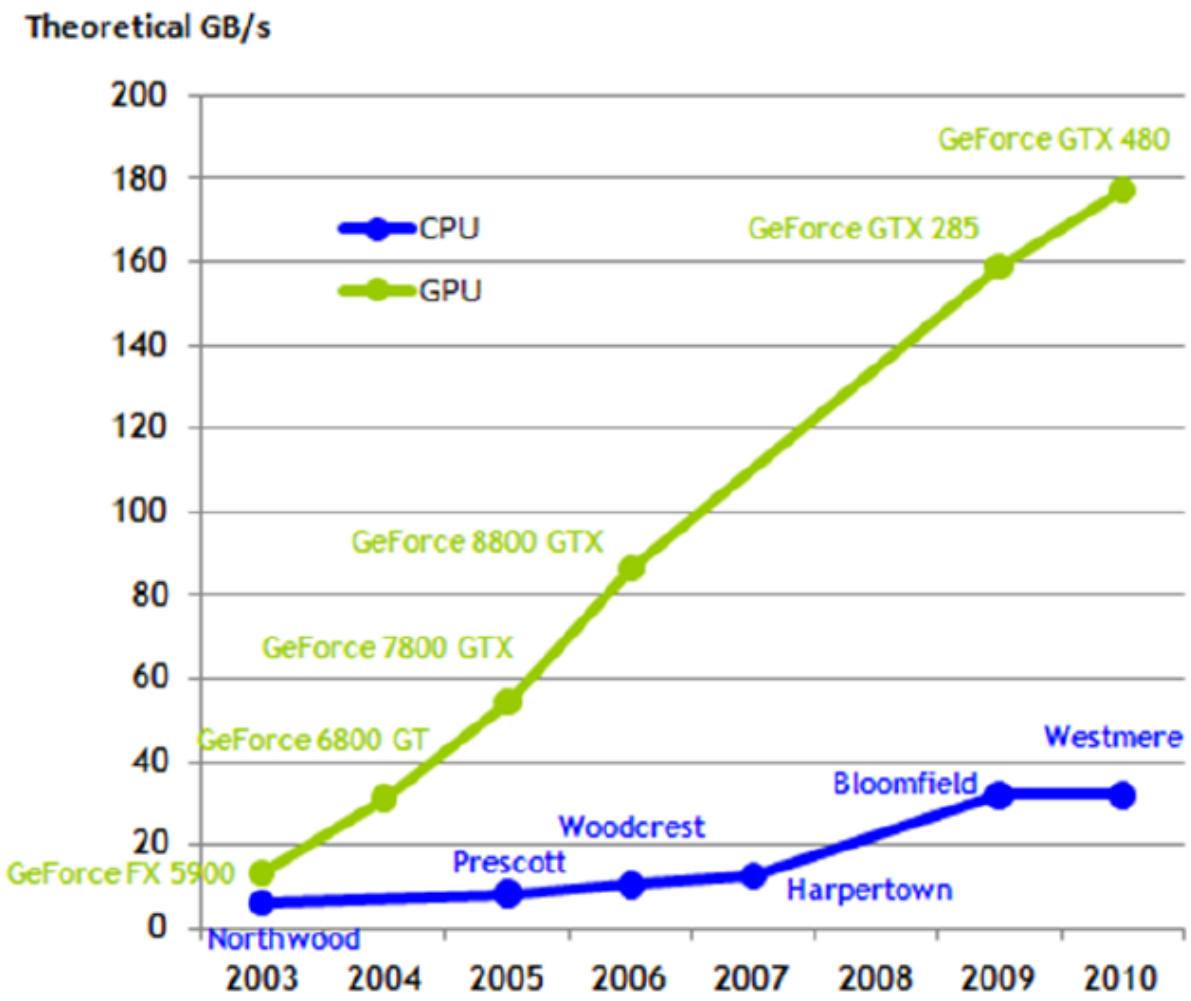
- **Intel Core 2 / Xeon / i7**
  - 4-6 núcleos MIMD
  - Pocos registros, cache multi-nivel
  - **10-30 GB/s** ancho de banda hacia la memoria principal

- **NVIDIA GTX480**
  - 512 núcleos, organizados en 16 unidades SM cada una con 32 núcleos.
  - Muchos registros, inclusión cachés nivel 1 y 2.
  - 5 GB/s ancho de banda hacia el procesador HOST.
  - 180 GB/s ancho de banda memoria tarjeta gráfica.

Evolución GPU: GFLOPS/s



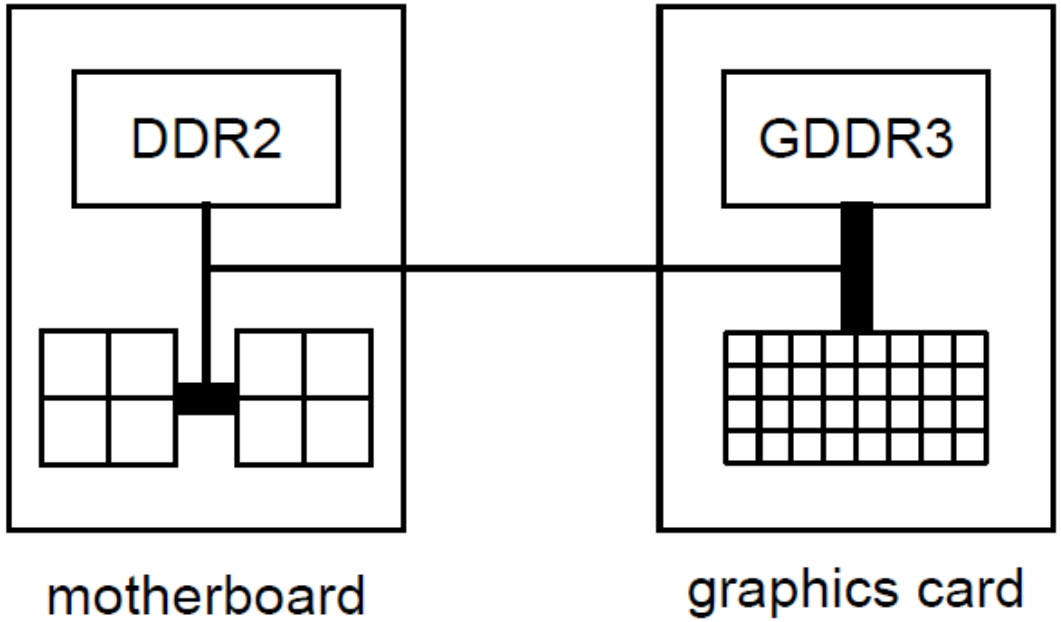
Evolución GPU: GB/s



Contenido

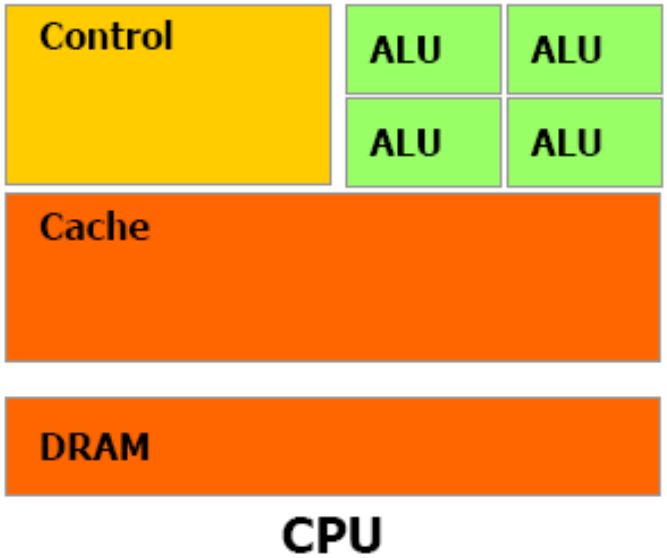
- **Introducción a CUDA**
  - **Hardware**
  - Software
  - Manejo de la memoria
  - Manejo de los hilos
  - Librerías
  - Cómo afrontar el desarrollo de una nueva aplicación

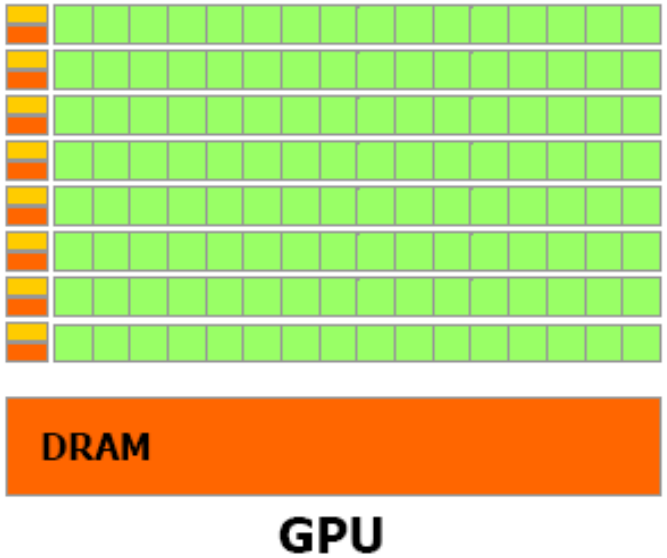
Hardware: Ubicación GPU



La GPU se sitúa sobre una placa gráfica pci-e dentro de un computador con uno o varios núcleos.

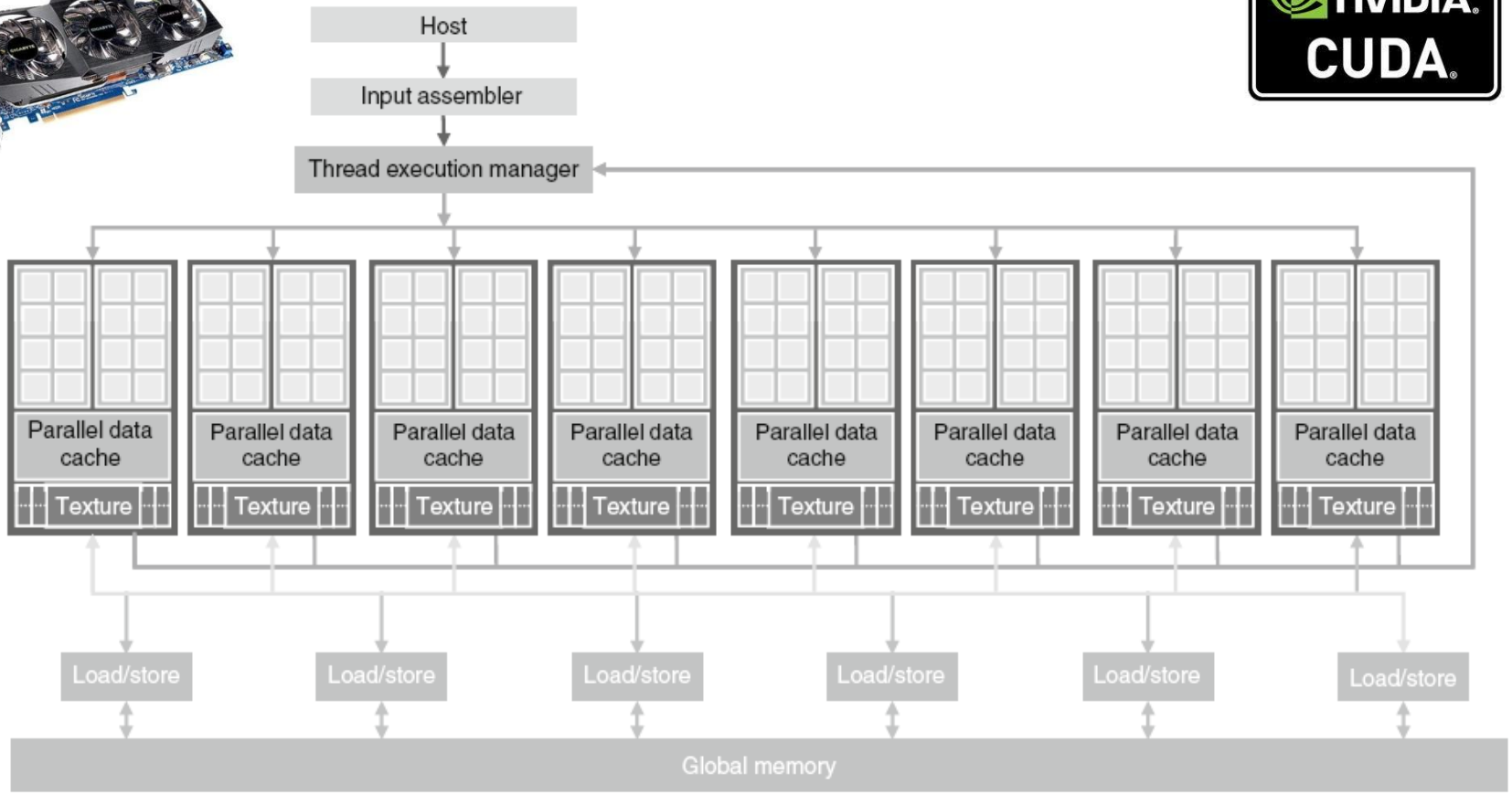
Hardware: Arquitectura



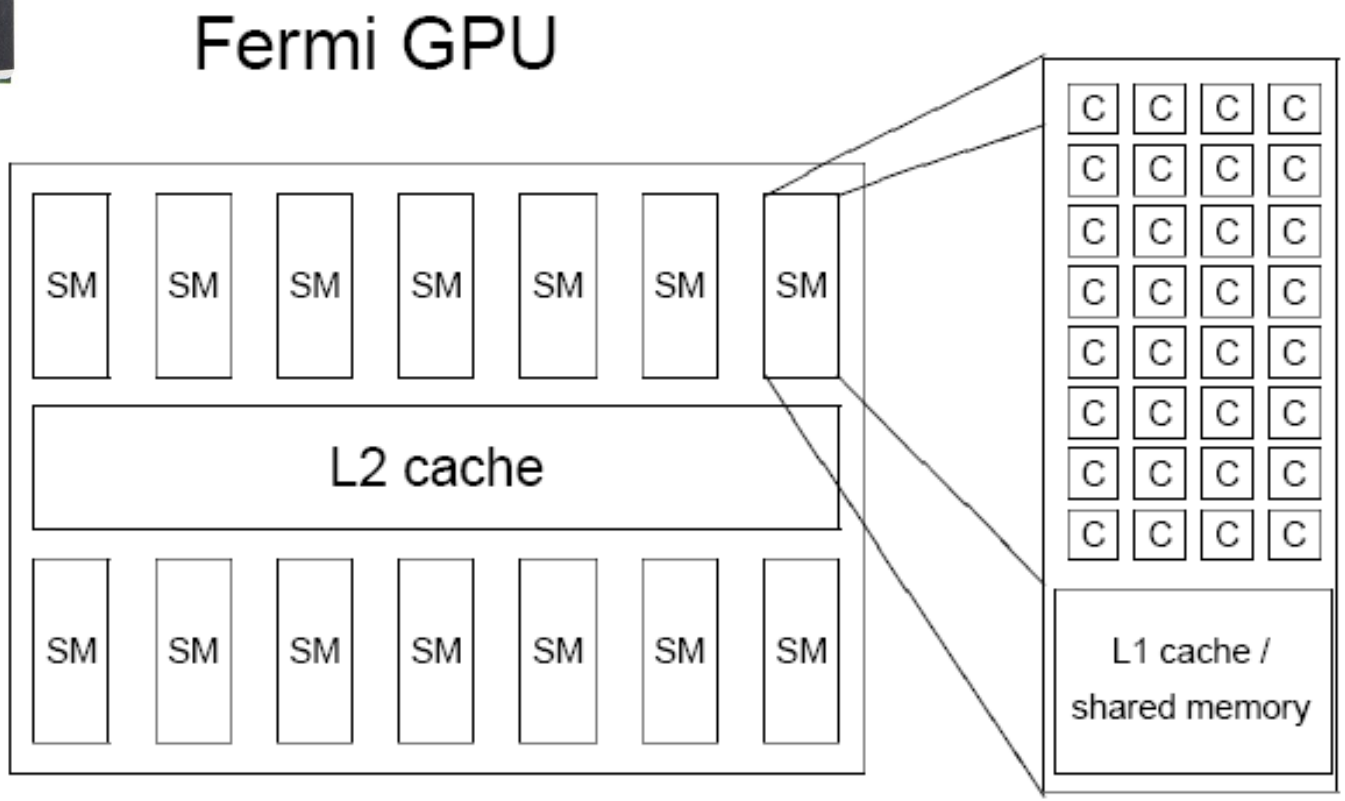


- La GPU dedica más transistores al procesamiento de datos

Hardware: GPU compatible CUDA



Hardware:
Nueva arquitectura GPU NVIDIA



Hardware: Generaciones de chips gráficos

- A nivel de GPU, los detalles dependen de la generación de los chips.

- Primeras generaciones
  - GeForce Series 8,9, GTX2XX
  - Tesla C1060, S1070 para HPC, sin salida gráfica, más memoria.
- Generaciones actuales
  - Nueva arquitectura Fermi: GeForce GTX4XX
  - Tesla C2050,S2050 para HPC.

Hardware: Características 1ª generación

- Primeras generaciones:
  - Compuestas de SM "Streaming Multiprocessor":
    - 8 núcleos, cada uno con 2048 registros
    - Hasta 128 hilos por núcleo
    - 16 KB de memoria compartida
    - 8 KB de caché para constantes
  - Diferentes chips tienen distinto número de SM,s

Modelo	**SM,s**	**Ancho de banda**
GTX260	27	110 GB/S
GTX285	30	160 GB/S
Tesla C1060	30	102 GB/S

Hardware: Características generación actual

- Generación actual: Fermi
  - Cada SM "Streaming Multiprocessor":
    - 32 núcleos, cada uno con 1024 registros
    - Hasta 48 hilos por núcleo
    - 64 KB de memoria compartida / Caché L1
    - Cache L2 común a los SM.
    - 8 KB de caché para constantes

Modelo	**SM,s**	**Ancho de banda**
GTX470	14	134 GB/s
GTX480	15	180 GB/s
Tesla C2050	14	140 GB/S

SIMT: Una instrucción múltiples hilos

- Característica clave de los núcleos dentro de un SM.
  - Todos los núcleos ejecutan la misma instrucción simultáneamente pero con distintos datos
  - Similar a la computación en los supercomputadores CRAY
  - Mínimo de 32 hilos realizando la misma tarea (casi) al mismo tiempo
  - Técnica tradicional en el procesamiento gráfico y en muchas aplicaciones científicas

Múltiples hilos

- Gran cantidad hilos clave alto rendimiento
  - **No penalización cambios de contexto**. Cada hilo tiene sus propios registros lo cual limita el número máximo de hilos activos.
  - Los hilos se ejecutan en los SM en grupos de 32 denominados “WARPS”. La ejecución alterna entre “warps” activos y “warps” temporalmente inactivos.

Múltiples hilos

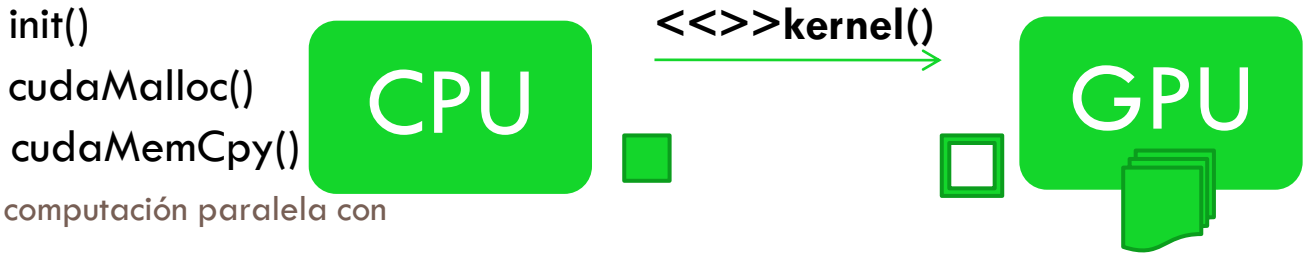
- Gran cantidad hilos clave alto rendimiento
  - **No penalización cambios de contexto**. Cada hilo tiene sus propios registros lo cual limita el número máximo de hilos activos.
  - Los hilos se ejecutan en los SM en grupos de 32 denominados “WARPS”. La ejecución alterna entre “warps” activos y “warps” temporalmente inactivos.

Contenido

- **Introducción a CUDA**
  - Hardware
  - **Software**
  - Manejo de la memoria
  - Manejo de los hilos
  - Librerías
  - Cómo afrontar el desarrollo de una nueva aplicación

Software: Flujo de ejecución

- En el nivel más alto encontramos un proceso sobre la CPU (Host) que realiza los siguientes pasos:
  1. Inicializa GPU
  2. Reserva memoria en la parte host y device
  3. Copia datos desde el host hacia la memoria device
  4. Lanza la ejecución de múltiples copias del kernel
  5. Copia datos desde la memoria device al host
  6. Se repiten los pasos 3-5 tantas veces como sea necesario
  7. Libera memoria y finaliza la ejecución proceso maestro



Software: Dentro de la GPU

- En el nivel más bajo dentro de la GPU:
  - Cada copia del programa se ejecuta sobre un SM
  - Si el número de copias excede el número máximo de SMs, entonces más de uno se ejecutará al mismo tiempo sobre cada SM hasta ocupar todos los recursos, en ese caso esperarán para su posterior ejecución.
  - No hay garantía en el orden de ejecución en el cual cada copia se ejecuta

Software: CUDA

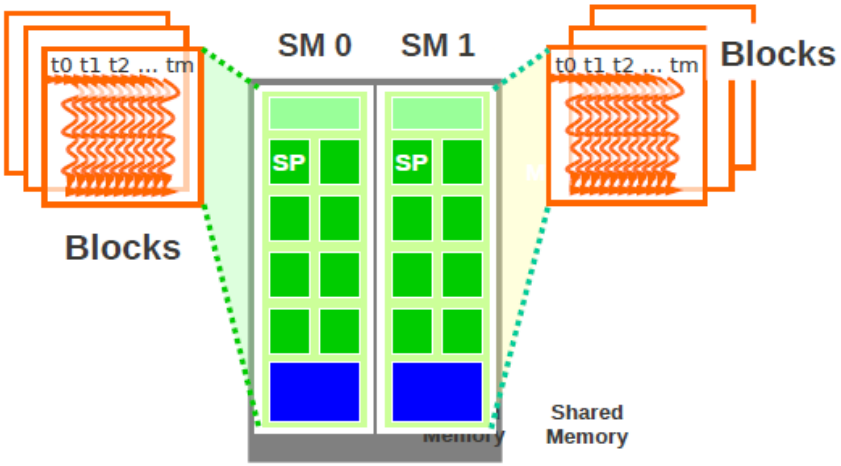
- **CUDA (Compute Unified Device Architecture)**
  - Hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVIDIA
  - **Basado en C** con algunas extensiones
  - Soporte C++ , Fortran. Envoltorios para otros lenguajes: *.NET, Python, Java, etcétera*
  - **Gran cantidad de ejemplos** y buena documentación, lo cual reduce la curva de aprendizaje para aquellos con experiencia en lenguajes como OpenMPI y MPI.
  - Extensa comunidad de usuarios en los **foros de NVIDIA**

Software: Programación CUDA

- Un programa en cuda tiene dos partes:
  - Código Host en la CPU que hace interfaz con la GPU
  - Código Kernel que se ejecuta sobre la GPU
- En el nivel host, existen dos APIs
  - Runtime: Simplificada, más sencilla de usar.
  - Driver: más flexible, más compleja de usar.
    - La versión driver no implica más rendimiento, sino más flexibilidad a la hora de trabajar con la GPU

Software: Conceptos básicos

- **Bloque:**
  - Agrupación de hilos
  - Cada bloque se ejecuta sobre un solo SM
  - Un SM puede tener asignados varios bloques



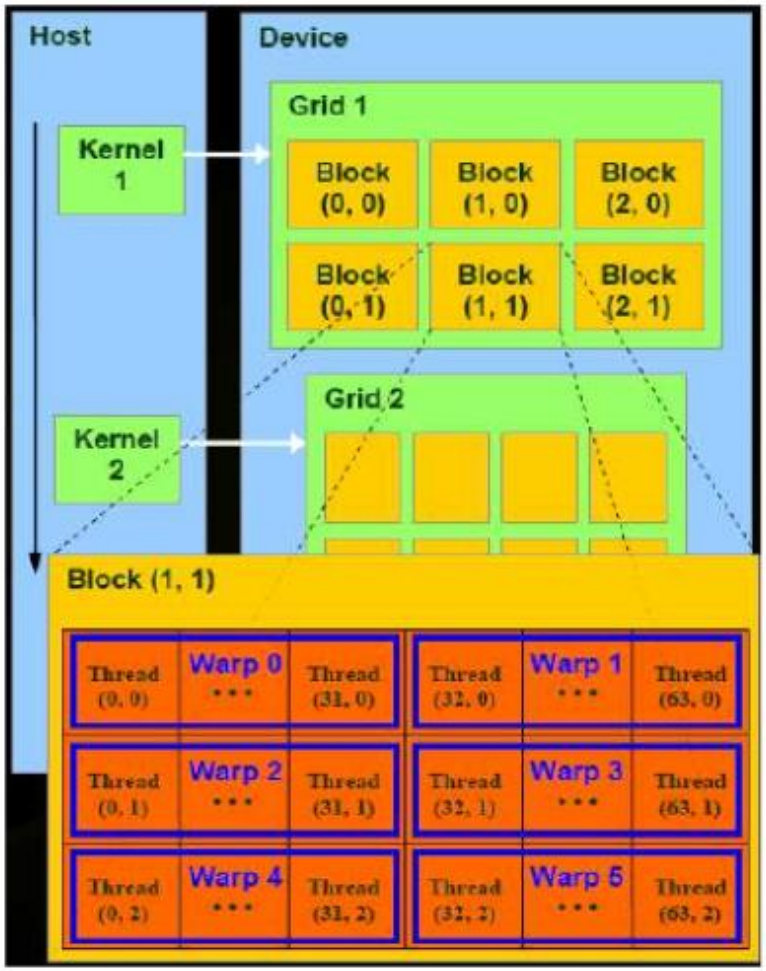
Software: Conceptos básicos

- **Kernel**: es una función la cual al ejecutarse lo hará en N distintos hilos en lugar de en secuencial.

- **Grid**: forma de estructurar los bloques en el kernel
  - Bloques: 1D,2D
  - Hilos/Bloque: 1D,2D o 3D

Software: Ejecución

- Cada bloque se divide en warps de 32 hilos
- Los hilos de un warp se ejecutan físicamente en paralelo
  - Aprovechamiento pipeline gpu
  - Ejecución paralela sobre los 32 núcleos del SM



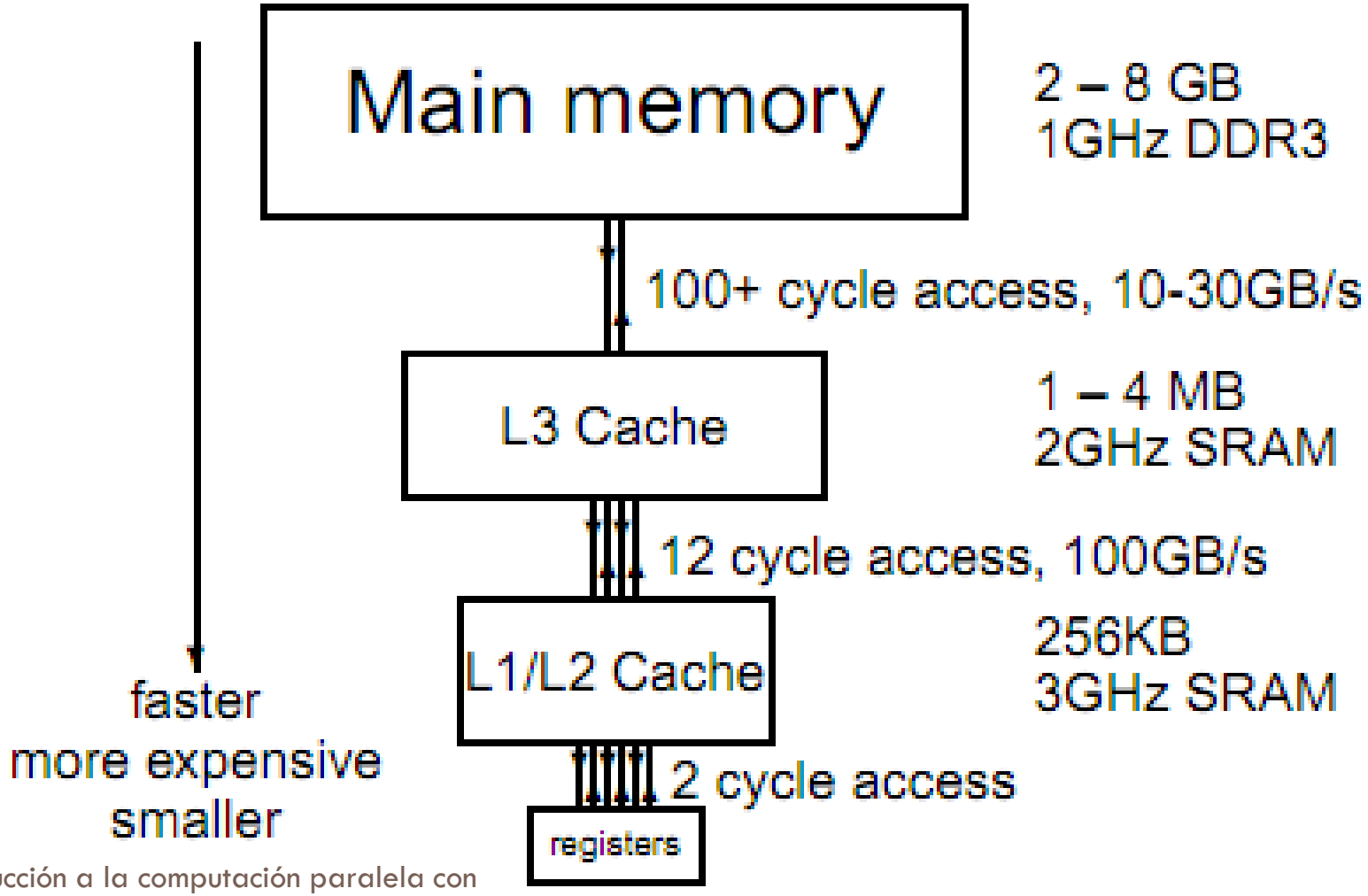
Contenido

- **Introducción a CUDA**
  - Hardware
  - Software
  - **Manejo de la memoria**
  - Manejo de los hilos
  - Librerías
  - Cómo afrontar el desarrollo de una nueva aplicación

Memoria

- Desafío clave en las arquitectura modernas de computadoras
  - No tiene sentido aumentar la capacidad de computo si la memoria no está a la altura
  - Las aplicaciones grandes trabajan con mucha memoria
  - La memoria rápida es muy cara de fabricar
  - Aspectos que obligan a un sistema jerárquico de memoria

Memoria


Main memory
2 – 8 GB
1GHz DDR3
100+ cycle access, 10-30GB/s
L3 Cache
1 – 4 MB
2GHz SRAM
12 cycle access, 100GB/s
L1/L2 Cache
256KB
3GHz SRAM
2 cycle access
registers
faster
more expensive
smaller

Jerarquía de memoria: Principio de localidad

- La velocidad de ejecución se basa en la explotación de la localidad de los datos:
  - Localidad temporal: un dato usado recientemente es probable que se use otra vez a corto plazo
  - Localidad espacial: es probable usar los datos adyacentes a los usados, por ello se usan caches para guardar varios datos en una línea del tamaño del bus

Caché Arquitectura Fermi

- Líneas de caché de 128 bytes (32 floats o 16 doubles)
- Bus de memoria de 384 bits hacia L2
- Ancho de banda de mas de 160 GB/s
- Una caché L2 de 384kB compartida por todos los multiprocesadores (SM)
- Cada SM tiene una caché L1 de 16kB o 48 kB (64 kb a repartir entre L1 y la memoria compartida)
- No hay coherencia global de cachés como en las CPUs así que debería evitarse que varios bloques actualicen los mismos elementos de la memoria global

Importancia de la localidad GPU

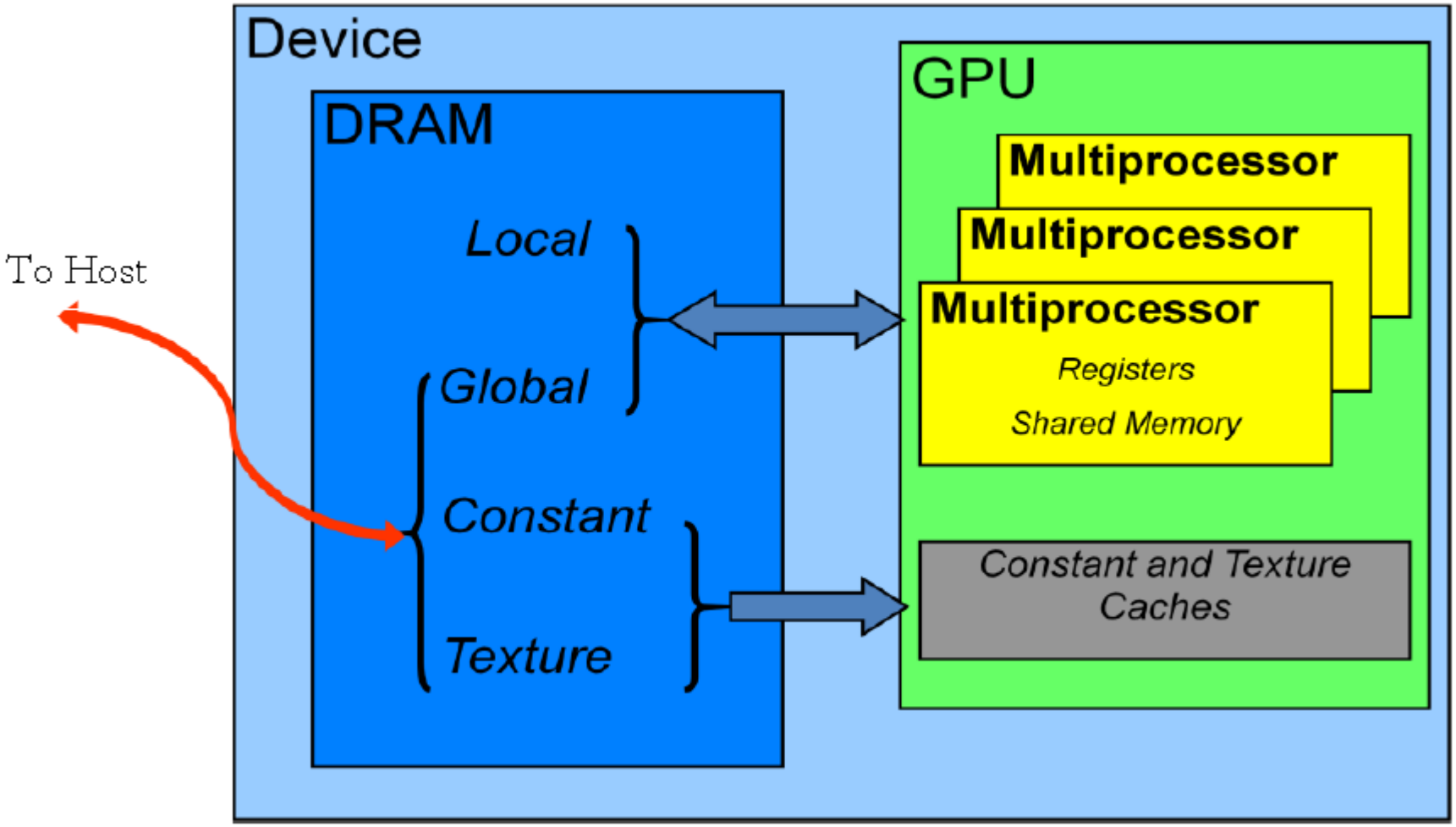
- 1 Tflops GPU (para precisión simple)
- 100 GB/s entre la memoria y la caché L2
- 128 bytes/línea

- 100 Gb/s ≈ 800M líneas/s ≈ 24 Gfloat/s

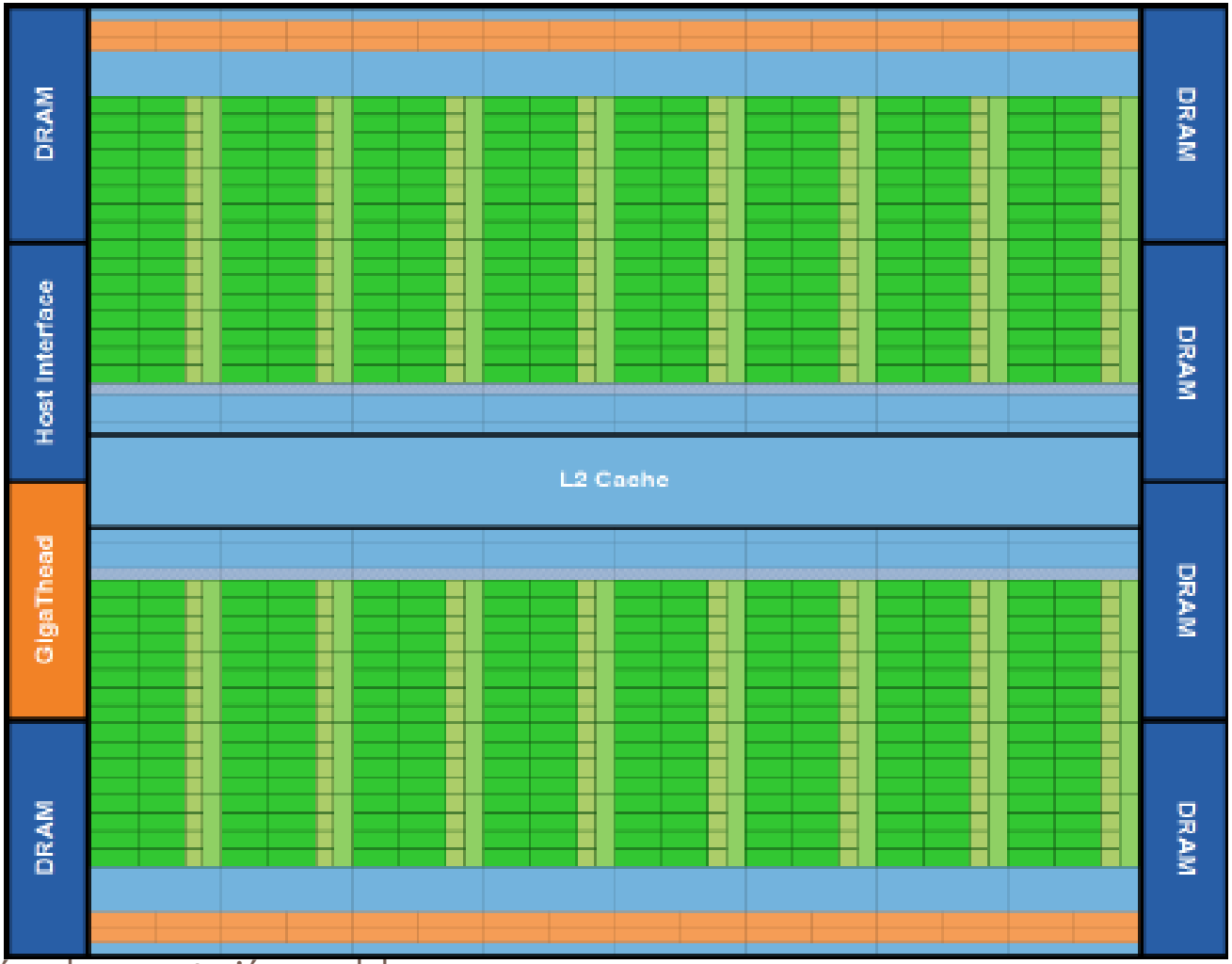
- En el peor caso, cada operación(flop) necesita 2 entradas y una salida, si pertenecen a líneas diferentes entonces ≈ 260 Mflops

- para conseguir una velocidad de 500Gflops son necesarios unas 20 operaciones por cada float transferido.
- Incluso con una buena implementación, muchos algoritmos no logran superar esta limitación.

Tipos de memoria

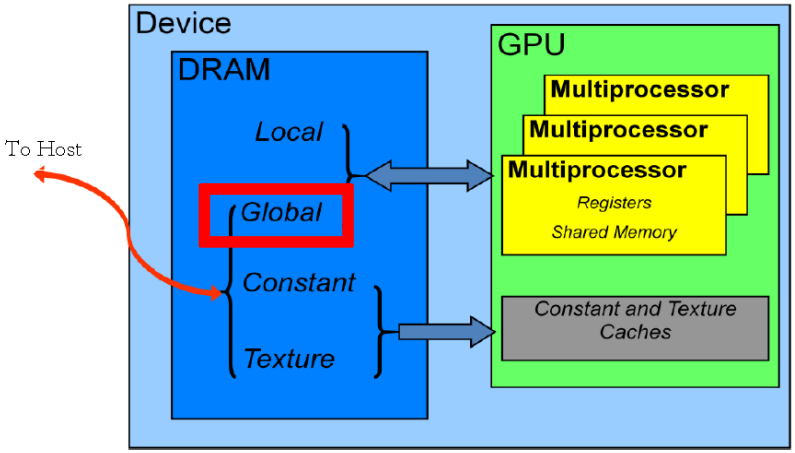


Tipos de memoria



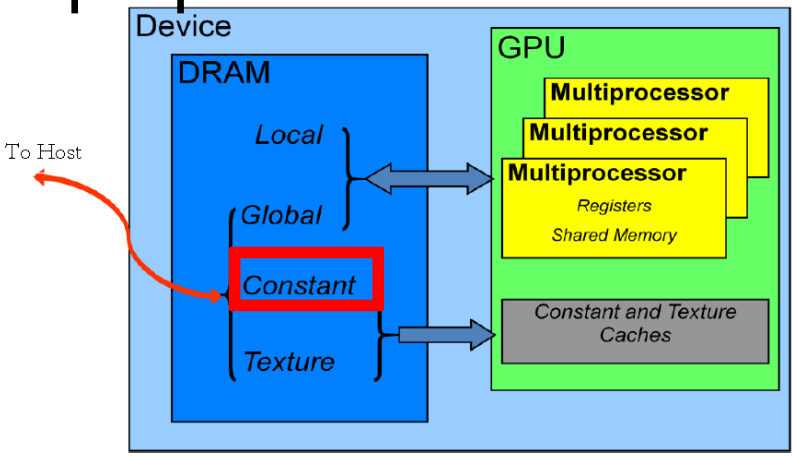
Tipos de memoria: Memoria Global

- Declarada y manejada desde la parte Host
- Ubicación para los datos de la aplicación, normalmente vectores y las variables globales del sistema.
- Es la memoria compartida por todos los SM



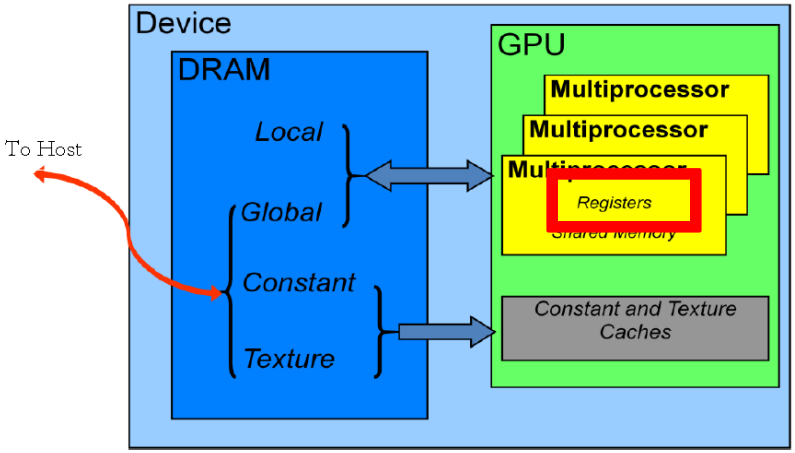
Tipos de memoria: Memoria Constante

- Sus valores no pueden ser cambiados.
- Optimizada para lecturas, su uso aumenta notablemente la velocidad del sistema.
- Su velocidad es similar a los registros sin necesidad de ocuparlos
- debido a que disponen de su propia mem. caché.



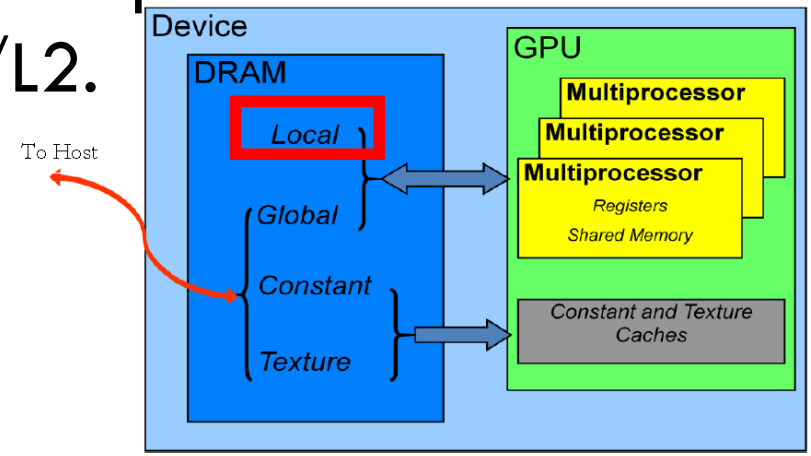
Tipos de memoria: Registros

- Ubicación para las variables usadas en los kernels
- Cada SM dispone de un numero de registros a repartir entre el numero de hilos
- En la arquitectura Fermi:
  - 32k registros de 32 bits por SM
  - 63 registros por hilo, unos 1536 hilos
- Para grandes aplicaciones, su manejo es esencial



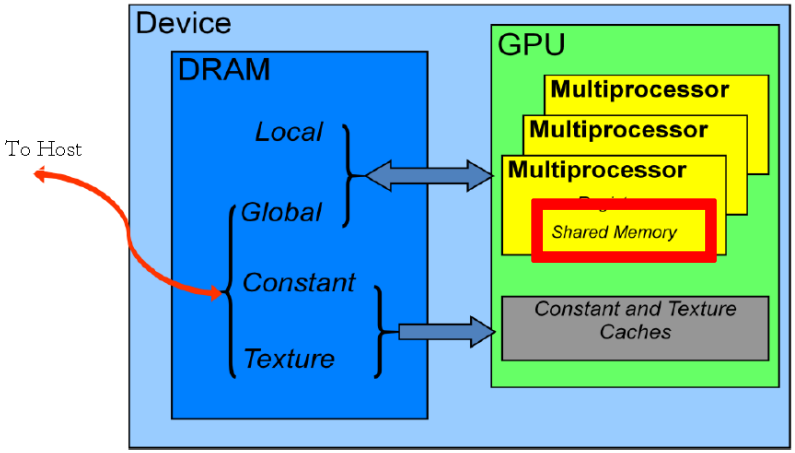
Tipos de memoria: Memoria Local

- Destinada a la información local de cada hilo.
- Para variables o vectores locales que no caben en sus respectivos registros
- Tiene bastante capacidad y su velocidad es como la mem. Global.
- Las nuevas arquitecturas han mejorado bastante su uso mediante las caches L1/L2.



Tipos de memoria: Memoria Compartida

- Memoria dedicada a cada multiprocesador
- Compartida por todos los hilos de un mismo bloque, útil para compartir información
- Puede necesitar instrucciones de control de acceso y flujo para controlar los accesos.
- Normalmente se copian los datos desde la mem. global en la memoria compartida



Contenido

- **Introducción a CUDA**
  - Hardware
  - Software
  - Manejo de la memoria
  - **Manejo de los hilos**
  - Librerías
  - Cómo afrontar el desarrollo de una nueva aplicación

Hilos CUDA

- Los hilos de CUDA se distribuyen automáticamente entre los SM

- Para identificar el hilo y poder trabajar con su parte se usan las variables:

  *int mi_ID=blockIdx.x * blockDim.x + threadIdx.x*

Control de flujo

- Todos los hilos de un mismo warp ejecutan al mismo tiempo la misma instrucción
- Los kernels con varios posibles caminos se preparan para que todos ejecuten la misma instrucción pero cada hilo ejecute su respectiva funcionalidad

```
if (x < 0.0)
        z=x-2.0;
else
        z=sqrt(x);
```

Control de flujo

- Para resolver los diferentes caminos, el compilador predica las instrucciones:

```
if (x < 0.0)                p = (x<0.0);
   z=x-2.0;              p:  z = x-2.0;
else                     !p: z = sqrt(x);
   z=sqrt(x);
```

- De esta forma todos los hilos ejecutan las mismas instrucciones.

Sincronización

- Para sincronizar los hilos de forma interna al kernel, se utiliza la instrucción:

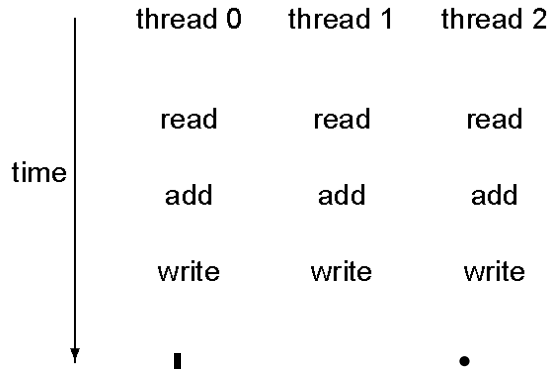
  __syncthreads()

- Si se utiliza código condicional hay que asegurarse que todos los hilos alcancen la instrucción de sincronización

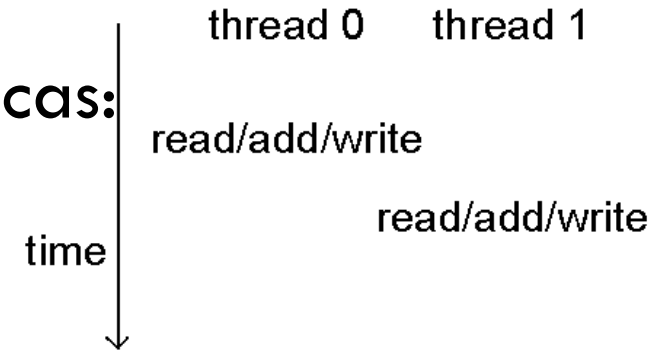
Operaciones Atómicas

- Hay operaciones que necesitan asegurar que se van a ejecutar una tras otra en los diferentes hilos

```
__shared__ int cont;
...
cont++; //dentro de un kernel
```



- necesarias para trabajar con la memoria compartida
- Existen varias operaciones atómicas:
  - Incrementos, mínimos, máximos…



Ejemplo de 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

Contenido

- **Introducción a CUDA**
  - Hardware
  - Software
  - Manejo de la memoria
  - Manejo de los hilos
  - **Librerías**
  - Cómo afrontar el desarrollo de una nueva aplicación

Librerías basadas en CUDA

- Se desarrollaron muchas aplicaciones especificas
- Mas tarde, comenzaron a aparecer varias librerías sobre CUDA para varios ámbitos:
  - CUBLAS: operaciones sobre algebra lineal
  - CUFFT : transformadas rápidas de Fourier
  - CUDPP: operaciones paralelas primitivas

Librerías basadas en CUDA

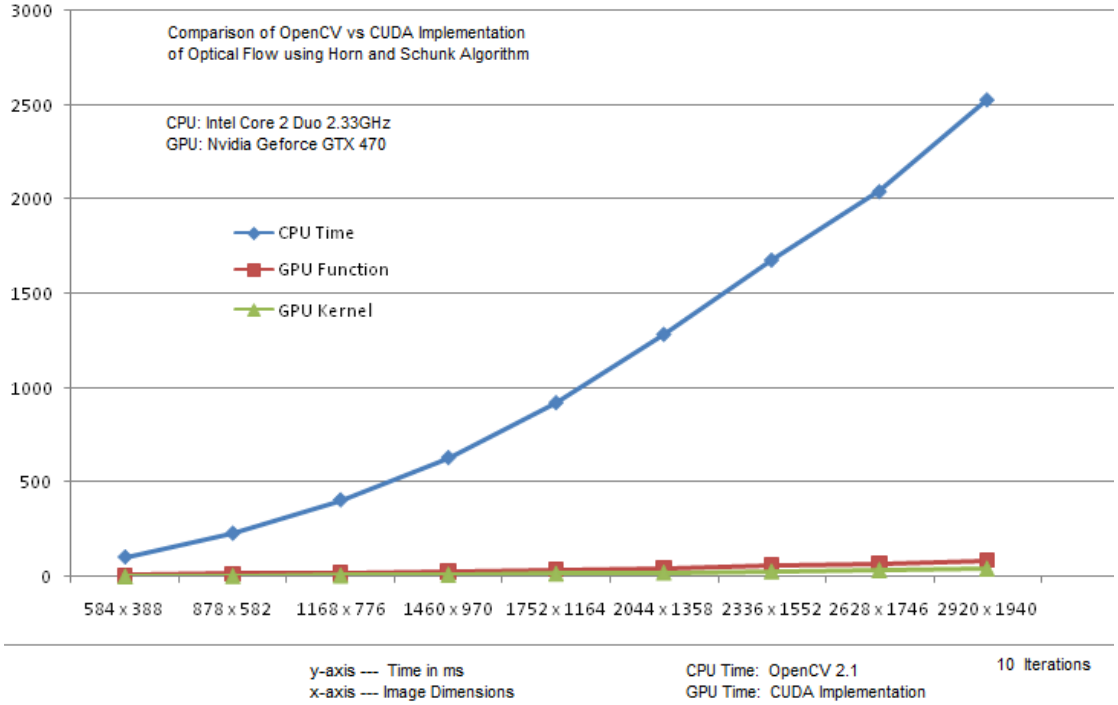
- **Thrust:** desarrollada por uno de los desarrolladores de CUDA e incluida en la versión 4.
  - Ofrece una abstracción de la programación de los kernel
  - Implementa varios algoritmos paralelos de datos
    - Máximo, suma, Multiplicación de vectores, …

```
int main(void)
{
  // generate random data on the host
  thrust::host_vector<int> h_vec(100);
  thrust::generate(h_vec.begin(), h_vec.end(), rand);

  // transfer to device and compute sum
  thrust::device_vector<int> d_vec = h_vec;
  int x = thrust::reduce(d_vec.begin(), d_vec.end(), 0, thrust::plus<int>());
  return 0;
}
```

Librerías basadas en CUDA

- **CUVILib:** librería que implementa sobre GPU algoritmos de *visión por computador* y *procesamiento de imágenes*.



Cálculo de flujo de movimiento:

Lucas and Kanade Algorithm

Contenido

- **Introducción a CUDA**
  - Hardware
  - Software
  - Manejo de la memoria
  - Manejo de los hilos
  - Librerías
  - **Cómo afrontar el desarrollo de una nueva aplicación**

Diseñar aplicación con CUDA

- 1) *La aplicación a desarrollar…*
  - Hay algo similar en el SDK de CUDA?
  - En los foros de CUDA hay algún proyecto relacionado?
  - Hay algo en la red ? (google)
- 2) *¿Qué partes son paralelizables?*
  - ¿tiene suficiente paralelismo?
  - CUDA es eficiente cuando trabaja con miles de hilos

Diseñar aplicación con CUDA

- 3) *Dividir el problema en funcionalidades diferenciadas:*
  - Alguna de las partes esta ya implementada?
  - Diseñar un kernel para cada funcionalidad..
  - Puede servir alguna de las librerías ya implementadas?
- 4) *Va a existir mucha divergencia en los warp?*
  - Hay mucho código condicional a las entradas?
  - La máxima eficiencia se consigue con la poca divergencia

Diseñar aplicación con CUDA

- *5) Habrá un problema con el ancho de banda?*
  - El tiempo de trasferir la información entre la CPU y la GPU se verá recompensado por la ventaja de paralelizar el problema?
  - Para aplicaciones para varios kernels, es apropiado utilizar ciertas funciones en la GPU o conviene mover toda la funcionalidad en la GPU
  - Siempre tener en cuenta que el número de operaciones por dato debe ser alto para compensar las transferencias

Diseñar aplicación con CUDA

- 6.a) *¿El problema es de computación intensiva?*
  - No hay que preocuparse tanto por la optimización de memoria
  - Cuidado si los cálculos son doubles
- 6.b) *¿O sin embargo es para trabajar con muchos datos?*
  - Hay que utilizar muy bien los niveles de memoria

Contenido

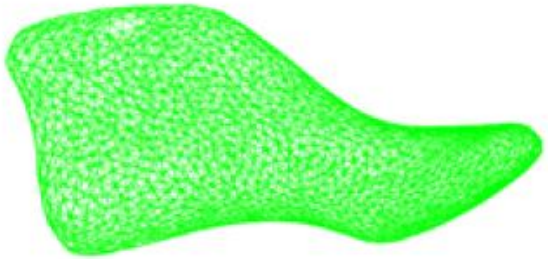
- GPU
- Paralelismo
- GPGPU
- Introducción a CUDA
- **Aplicaciones**
- Conclusiones

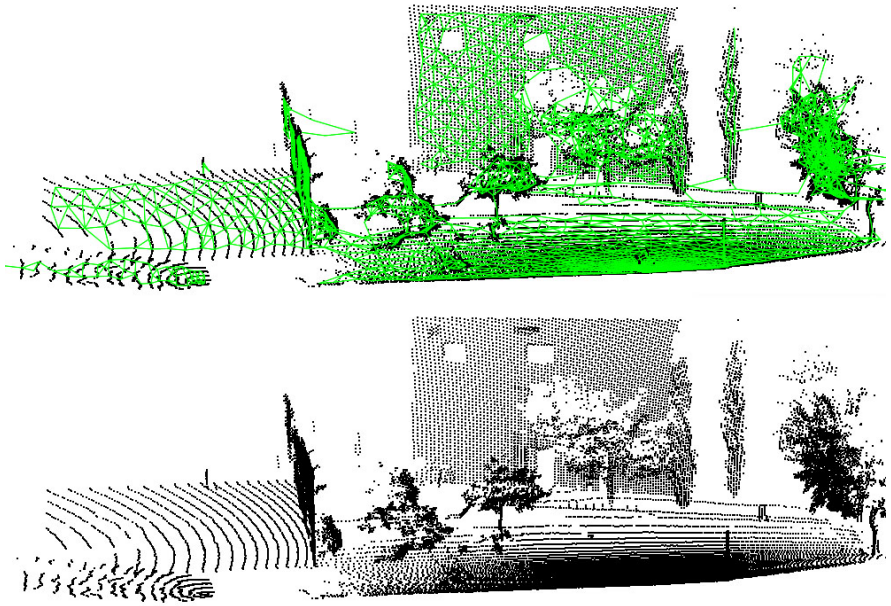
Aplicaciones en CUDA

- En la pagina de www.nvidia.com podemos encontrar una larga lista de aplicaciones especificas y modificaciones de programas para aprovechar esta tecnología, para diferentes ámbitos de aplicación:
  - Gobierno y defensa
  - Dinámica molecular
  - Bioinformática
  - Electrodinámica y electromagnetismo
  - Imágenes medicas
  - Combustibles y gases
  - Computación financiera
  - Extensiones de Matlab
  - Tratamiento de video y visión por computador
  - Modelado del tiempo y los océanos

Aplicaciones en CUDA

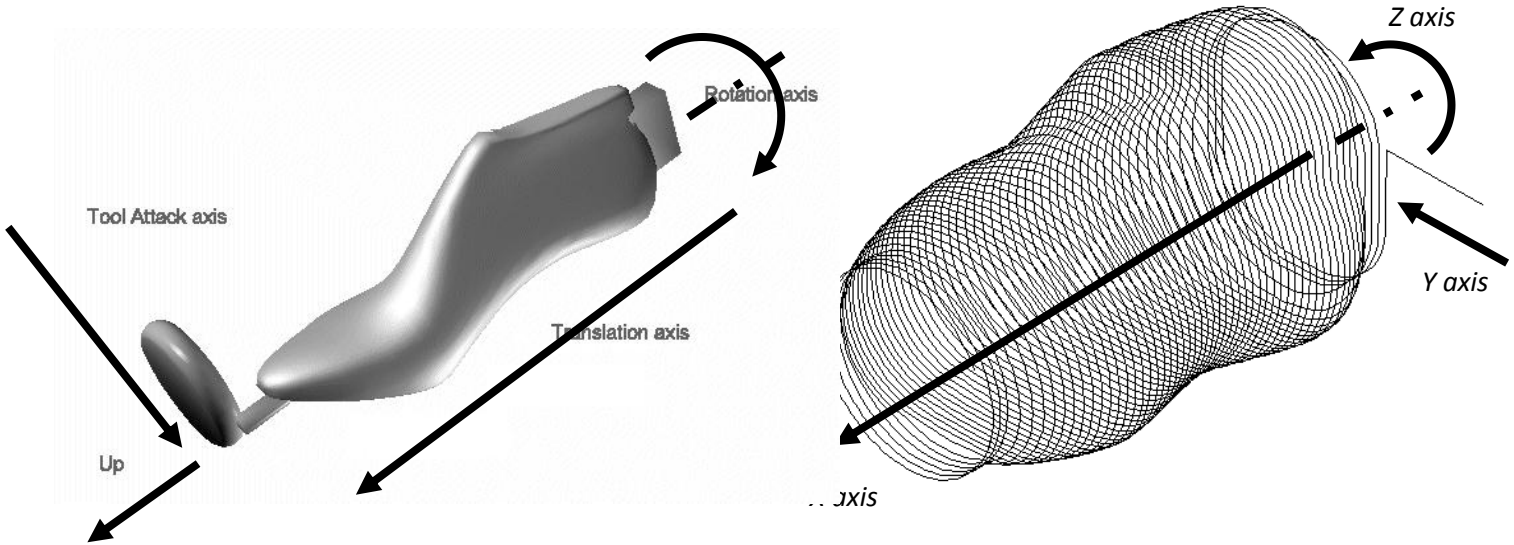
- *Actualmente estamos trabajando en:*
- **Visión:**
  - Aceleración algoritmos redes neuronales auto-organizativas (GNG,NG)
    - Reconstrucción escenarios 3D
    - Representación 3D objetos





Aplicaciones en CUDA

- *Actualmente estamos trabajando en:*
- **CAD/CAM:** Calculo de rutas de maquinas-herramienta para recrear modelos digitales



Contenido

- GPU
- Paralelismo
- GPGPU
- Introducción a CUDA
- Aplicaciones
- **Conclusiones**

CUDA para acelerar algoritmos

- **Buenas prácticas :**
  - Paralelizar la mayor parte posible del algoritmo.
  - Disminuir las operaciones de memoria entre device y host
  - Organizar la memoria de forma que los accesos paralelos sean óptimos:
    - Copiando partes de la memoria global a la compartida
    - Transformando los datos para que los accesos sean concurrentes
    - Alinear y conjuntar la memoria para que en menos accesos se lean los datos necesarios.
  - Usar medidas tanto de tiempo como de ancho de banda utilizado para optimizar los algoritmos.

Conclusiones

- CUDA es una tecnología que permite obtener grandes rendimientos para problemas con un alto paralelismo.
- Hay que tener claro su funcionamiento para saber si es adecuado y obtener el mayor rendimiento posible.
- ***Programar en CUDA es fácil, pero no lo es obtener rendimiento.***

¿Preguntas?