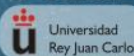
PROCESADORES GRÁFICOS

07 CUDA: MODELO DE PROGRAMACIÓN Y JERARQUÍA DE MEMORIA

Máster Oficial en Informática Gráfica, Juegos y Realidad Virtual

Escuela Técnica Superior de Ingeniería Infomática



Índice

- Introducción a CUDA
- Segunda personalidad: GPGPU
- Grids, Bloques y Hebras
- Jerarquía de Memoria
- Ej: Multiplicación de matrices
- Patrones típicos de programación
- Modelo SPMD
- API de CUDA
- Compilación, enlazado y debugging

“"If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?"
-- Seymore Cray

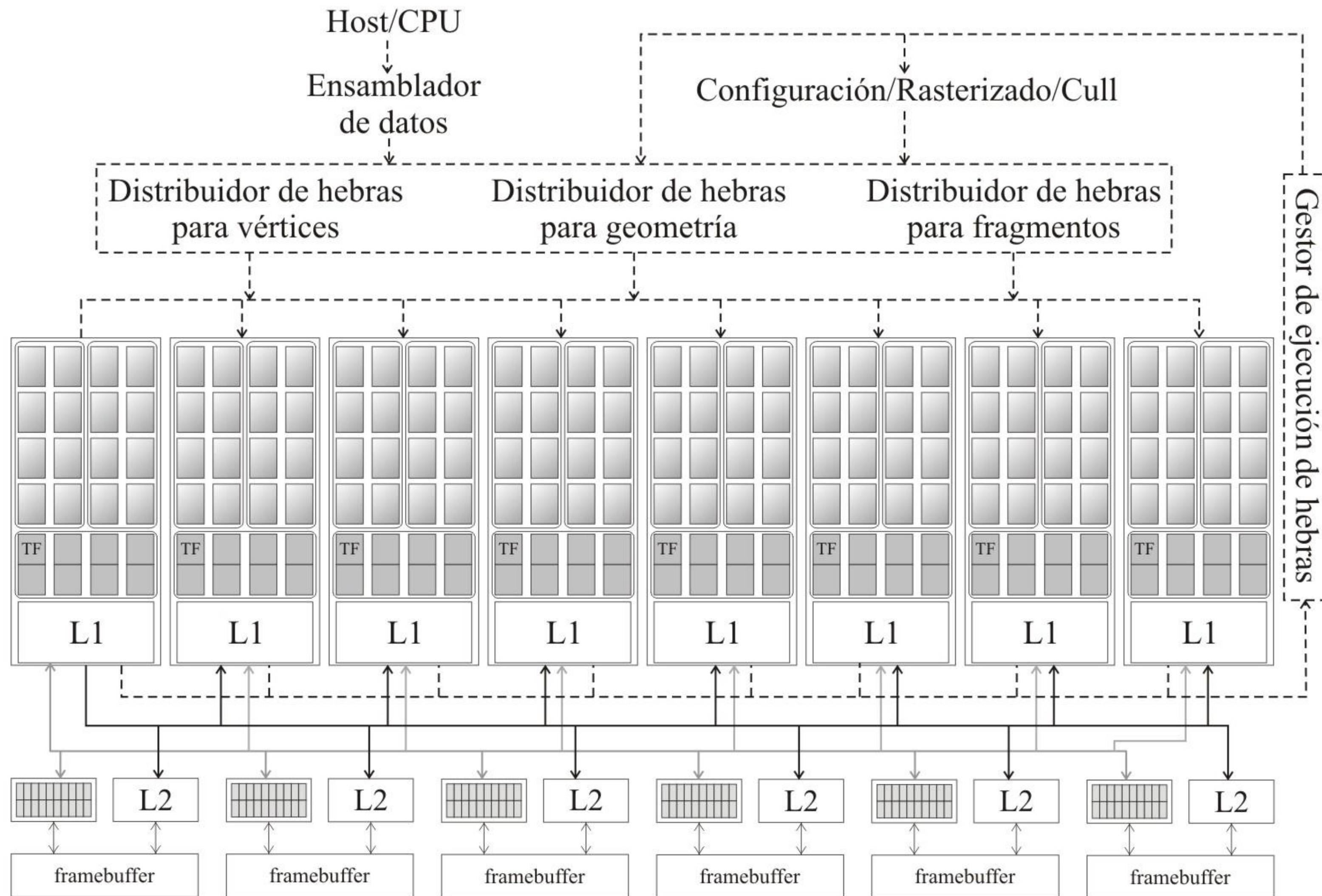
"640 K ought to be enough for anybody."
-- Bill Gates, 1981

Aunque en esta clase me voy a apoyar en las transparencias de D. Kirk y WM Hwu la explicación será **muy diferente**

© Todas las marcas y productos mencionados en estas trasparencias están registradas por sus respectivas compañías, y su uso es de carácter descriptivo con fines docentes.

Parte de las tablas y gráficos están basados en las presentaciones de GPGPU y streaming computing de NVidia y ATI-AMD, en los libros mencionados en la bibliografía, en el curso de David Kirk de la Universidad de Illinois, y el de Simon Green en ARCS08

Qué tenemos dentro del G80/G92?



(c) David Miraut Andrés

Universidad Rey Juan Carlos

Solicitud de patente de CUDA

US 20070159488A1

(19) **United States**
(12) **Patent Application Publication** (10) **Pub. No.: US 2007/0159488 A1**
Danskin et al. (43) **Pub. Date: Jul. 12, 2007**

(54) **PARALLEL ARRAY ARCHITECTURE FOR A GRAPHICS PROCESSOR**

(75) Inventors: **John M. Danskin**, Providence, RI (US); **John S. Montrym**, Los Altos Hills, CA (US); **John Erik Lindholm**, Saratoga, CA (US); **Steven E. Molnar**, Chapel Hill, NC (US); **Mark French**, Raleigh, NC (US)

Correspondence Address:
TOWNSEND AND TOWNSEND AND CREW LLP
TWO EMBARCADERO CENTER
8TH FLOOR
SAN FRANCISCO, CA 94111-3834 (US)

(73) Assignee: **NVIDIA Corporation**, Santa Clara, CA (US)

(21) Appl. No.: **11/611,745**

(22) Filed: **Dec. 15, 2006**

Related U.S. Application Data

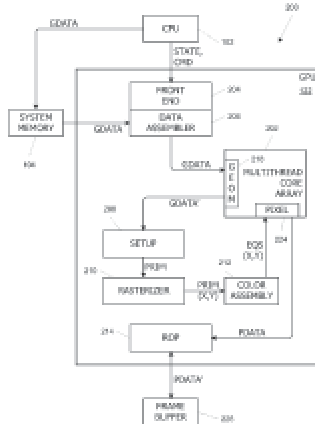
(60) Provisional application No. 60/752,265, filed on Dec. 19, 2005.

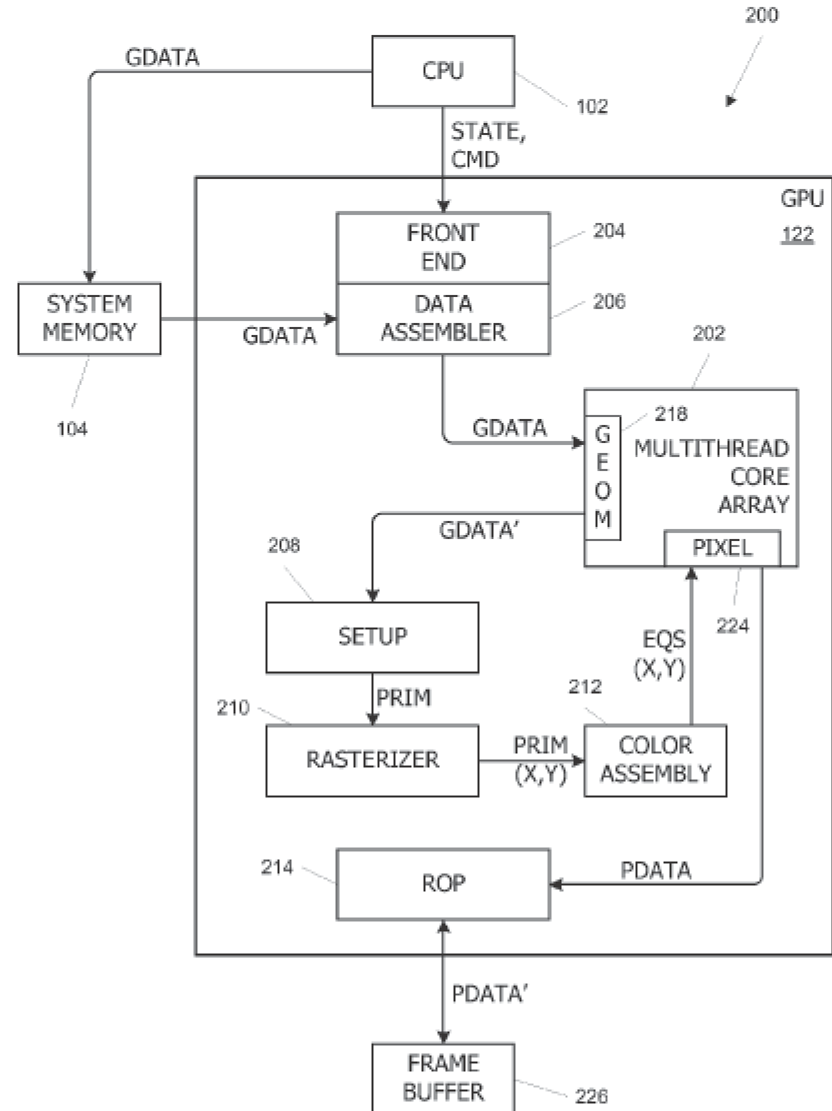
Publication Classification

(51) **Int. Cl.**
G06F 15/80 (2006.01)
(52) **U.S. Cl.** 345/505

(57) **ABSTRACT**

A parallel array architecture for a graphics processor includes a multithreaded core array including a plurality of processing clusters, each processing cluster including at least one processing core operable to execute a pixel shader program that generates pixel data from coverage data; a rasterizer configured to generate coverage data for each of a plurality of pixels; and pixel distribution logic configured to deliver the coverage data from the rasterizer to one of the processing clusters in the multithreaded core array. The pixel distribution logic selects one of the processing clusters to which the coverage data for a first pixel is delivered based at least in part on a location of the first pixel within an image area. The processing clusters can be mapped directly to the frame buffers partitions without a crossbar so that pixel data is delivered directly from the processing cluster to the appropriate frame buffer partitions. Alternatively, a crossbar coupled to each of the processing clusters is configured to deliver pixel data from the processing clusters to a frame buffer having a plurality of partitions. The crossbar is configured such that pixel data generated by any one of the processing clusters is deliverable to any one of the frame buffer partitions.





Características del G80

Aún mayor en modelos de gama alta y serie 9

- 367 GFLOPS de rendimiento pico (25-50 veces los procesadores genéricos más potentes actuales)
- 265 GFLOPS sostenido para aplicaciones como VMD
- Masivamente paralelo, 128 cores, 90W
- Se pueden utilizar miles de hebras simultáneamente por kernel (de hecho es necesario para aprovechare estas características
- Las aplicaciones científicas y multimedia alcanzan mejoras en el tiempo de ejecución entre 30 a más de 100 veces lo que se puede conseguir en procesadores genéricos de última generación

"I think they're right on the money, but the huge performance differential (currently 3 GPUs ~= 300 SGI Altix Itanium2s) will invite close scrutiny so I have to be careful what I say publically until I triple check those numbers."

-John Stone, VMD group, Physics UIUC

[Adaptada de David Kirk y W-M. Hwu]

Aplicaciones en un mundo concurrente

- Las aplicaciones que ya están siendo demandadas de forma masiva en el mercado corresponden a lo que hasta ahora se clasificaba como "aplicaciones de supercomputación"
  - Manipulación de audio y vídeo
  - Simulación de dinámica molecular
  - Simulación de física
  - Productos de realidad virtual de consumo (juegos)
- Estas "superaplicaciones" representan y modelan un mundo concurrente
- Existen varias granularidades de paralelismo, pero…
  - El modelo de progrmación no debe estorbar la implementación paralela
  - El manejo y envío de datos se ha de gestionar con cuidado: es nuestro bien más escaso (y el cuello de botella)

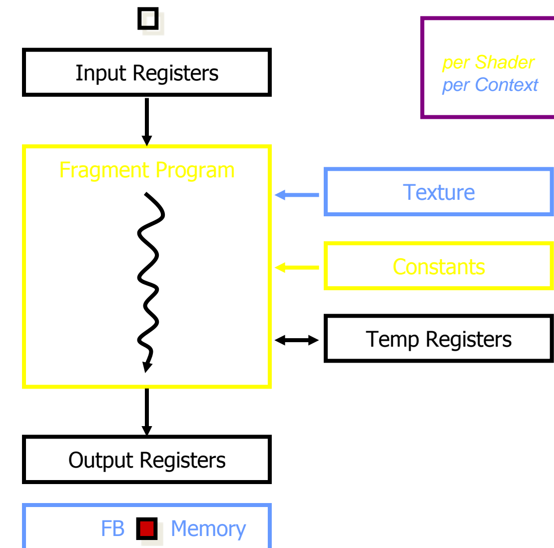
Vídeo de la semana: Dave Patterson

[Adaptada de David Kirk y W-M. Hwu]

(c) David Miraut Andrés

Antiguas restricciones en GPGPU

- "Lidia" con la API gráfica
  - Se trabaja con los casos extremos de la API, los que no han sido optimizados, ya que se hace un uso de los para el que originalmente no estaba pensada
- Modos de direccionamiento (y tipos de datos)
  - Dimensiones y tamaños limitados para texturas
- Capacidades de los shaders
  - Salida de datos muy ortopédica
- Conjuntos de instrucciones
  - Sin operaciones sobre enteros ni a nivel de bit
- Comunicación muy restringida
  - Entre píxeles
  - Scatter a[i] = p
- Y muchas más…



Lo hemos vivido en durante la práctica 4

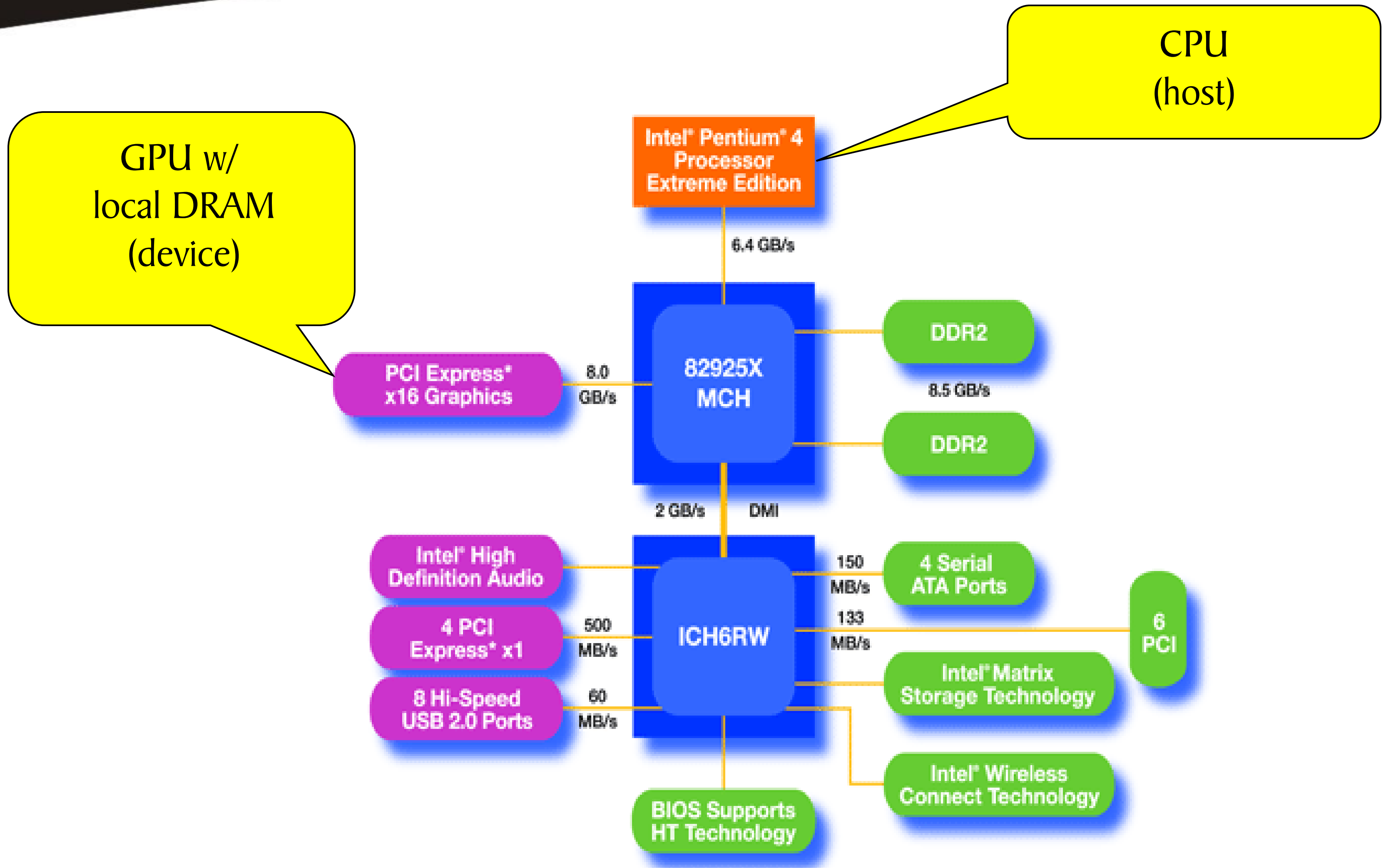
[Adaptada de David Kirk y W-M. Hwu]

CUDA

- "Compute Unified Device Architecture"
- Modelo de programación de propósito general
  - La GPU se puede convertir en un procesador específico masivamente paralelo, con miles de hebras que trabajan sobre un mismo problema simultáneamente (fácil división del problema por datos)
- Driver, lenguaje y herramientas específicos
- Driver para la carga y puesta en marcha de programas
  - Optimizado para problemas de computación (no para gráficos)
  - No requiere que se use a través de una API gráfica
  - Puede compartir datos con OpenGL y/o DirectX
  - Garantiza el mejor BW en carga y descarga de datos de la memoria de vídeo
  - Gestión explícita de memoria de vídeo (y toda la jerarquía)
- Solución propia de nVIDIA (que puede extenderse a ATI-AMD si se acogen al modelo PTX de máquina virtual)

[Adaptada de David Kirk y W-M. Hwu]

¿Puente Norte?



[Adaptada de David Kirk y W-M. Hwu]

Segmentación de productos para GPGPU

- NVIDIA GPU Computing Architecture
  - Mediante una interfaz separada
  - En laptops, sobremesa, estaciones de trabajo, servidores

GeForce 8800

- Con tarjetas de la serie 8 se pueden conseguir ratios entre 50 y 200 GFLOPS en aplicaciones escritas con fuentes integrados de C (CUDA) de forma muy sencilla

Tesla D870

- El nivel de paralelismo en GPUs se duplica cada año(*)
- El modelo de programación está pensado para que todo escale de forma transparente y se aproveche esa nuava capacidad anual

- El modelo multihebra SPMD aprovecha el paralelismo de datos y el de hebras (en cada multiprocesador)

Tesla S870

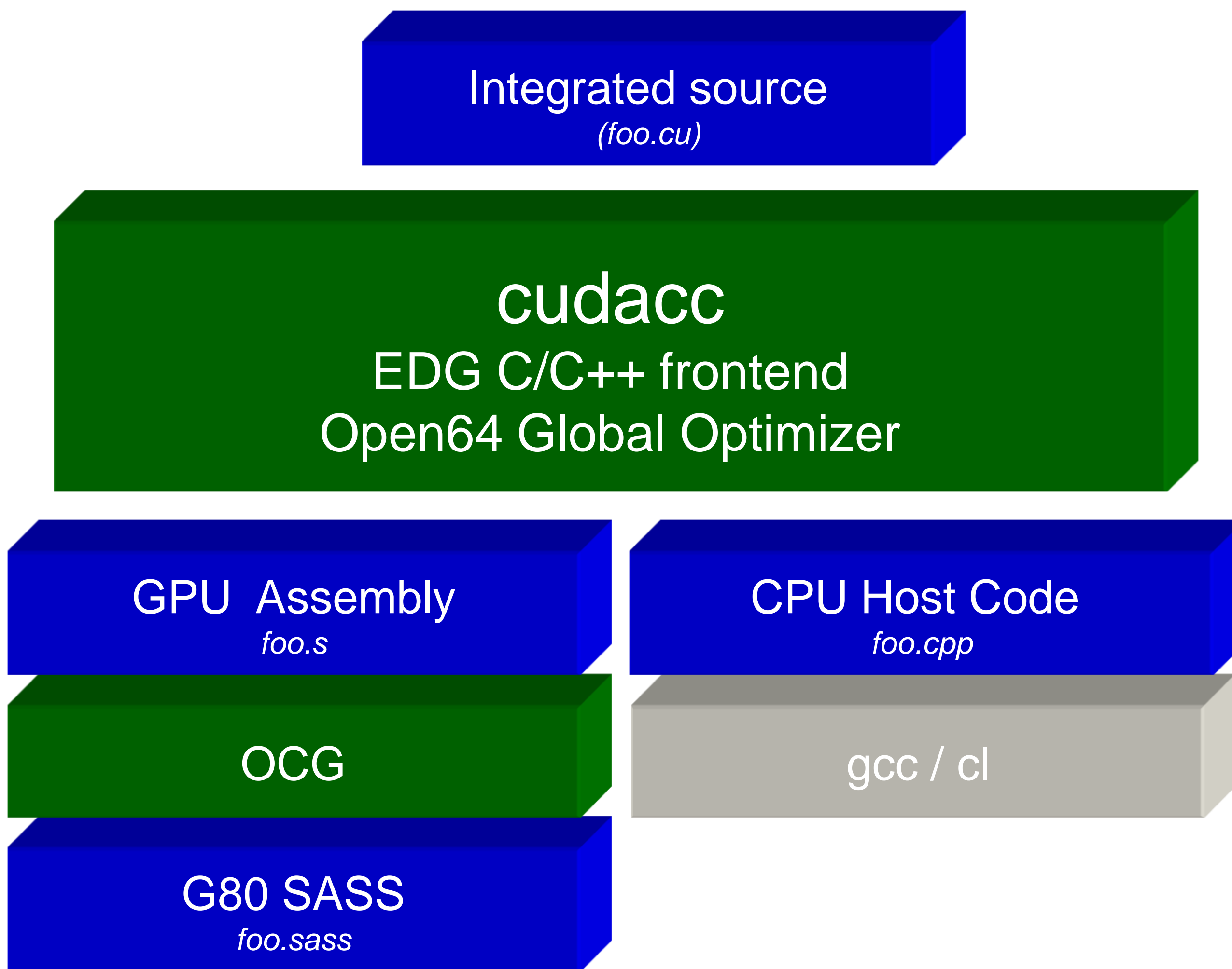
Extensiones de C

- Declspecs
  - global, device, shared, local, constant
- Palabras reservadas
  - threadIdx, blockIdx
- Intrínsecos
  - __syncthreads
- Runtime API
  - Memoria, símbolos, gestión de la ejecución
- Invocación del kernel en la GPU

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

[Adaptada de David Kirk y W-M. Hwu]

Extended C



[Adaptada de David Kirk y W-M. Hwu]

A vista de pájaro (hoy)

- Modelo de programación de CUDA -- conceptos básicos y tipos de datos
- Application Programming Interface de CUDA - básica
- Ejemplos muy sencillos para ilustrar estos conceptos y funcionalidades iniciales
- Todos los trucos y el conocimiento de la arquitectura que nos permitirá lograr grandes factores de aceleración los veremos en las dos próximas clases.

[Adaptada de David Kirk y W-M. Hwu]

(c) David Miraut Andrés

Modelo de programación de CUDA

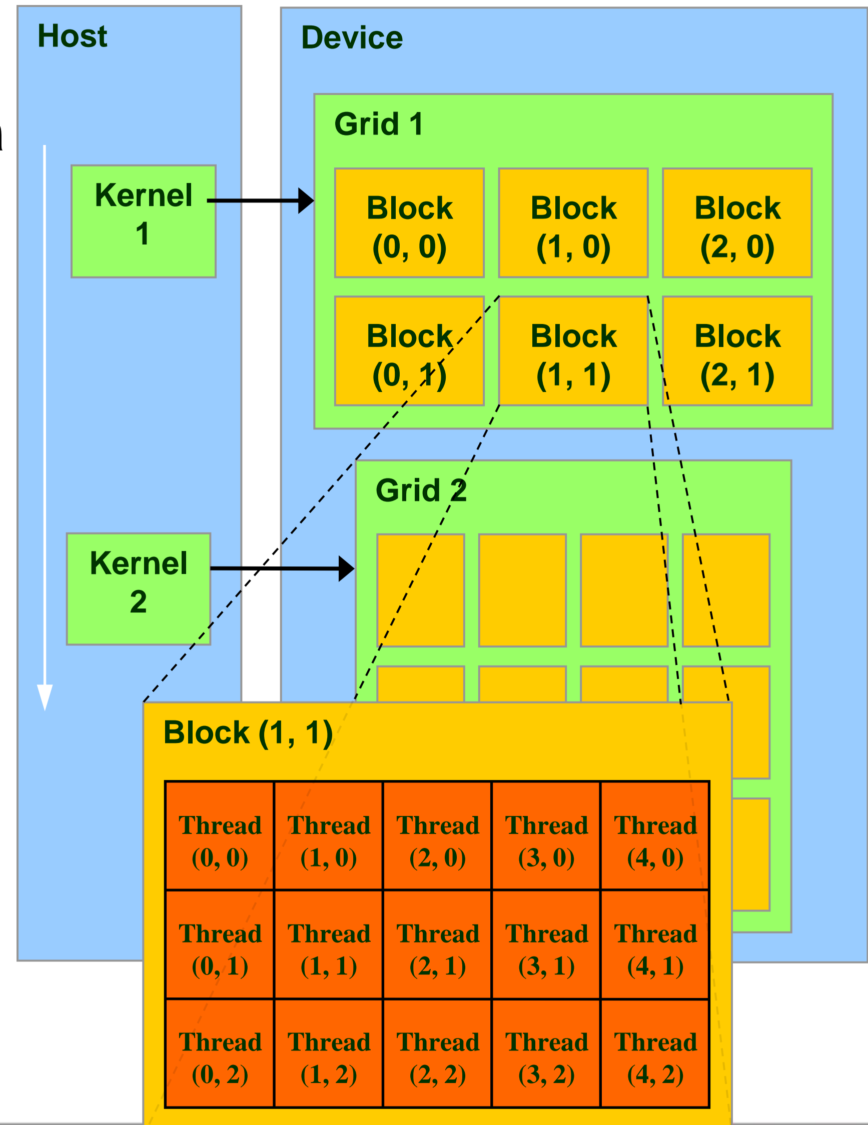
- Podemos ver la GPU como un dispositivo (device) que:
  - Es un coprocesador de la CPU (host)
  - Tiene su propia DRAM (memoria de vídeo ⇨ device memory)
  - Y que puede ejecutar muchas muchas hebras en paralelo
- Las porciones de código en las que se encuentre concurrencia en función de los datos pueden ser ejecutadas en la GPU en forma de kernels (hebras en paralelo) con facilidad
- Las hebras en GPU y CPU son muy diferentes:
  - Las hebras en la GPU son muy ligeras
    - Apenas hay sobrecarga por crearlas
    - El intercambio entre hebras es instantáneo (0 ciclos)
  - La GPU necesita de miles de hebras para ser eficaz
    - Las CPUs multicore sólo soportan unas pocas

[Adaptada de David Kirk y W-M. Hwu]

(c) David Miraut Andrés

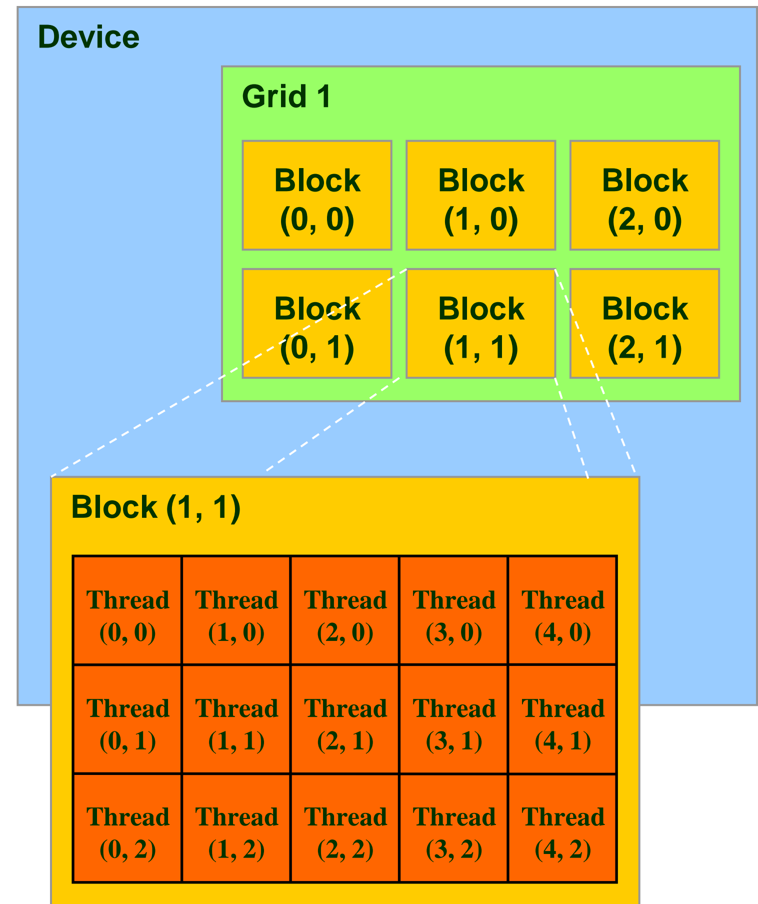
Conjuntos de hebras: Grids y Bloques

- La ejecución de un kernel en múltiples hebras es organizada "espacial y temporalmente" como un grid de bloques de hebras
  - Esta organización tiene que ver con la estructura en multiprocesadores del array de procesadores
- Las hebras dentro de un bloque serán ejecutadas en el mismo multiprocesador, y gracias a esta localidad pueden cooperar entre ellas:
  - Sincronizar su ejecución
    - Paar evitar riesgos en los accesos a memoria
  - Compartir datos de forma eficiente mediante una memoria local de baja latencia
- Hebras de diferentes multiprocesadores no pueden cooperar (no es trivial hacerlo en CUDA)



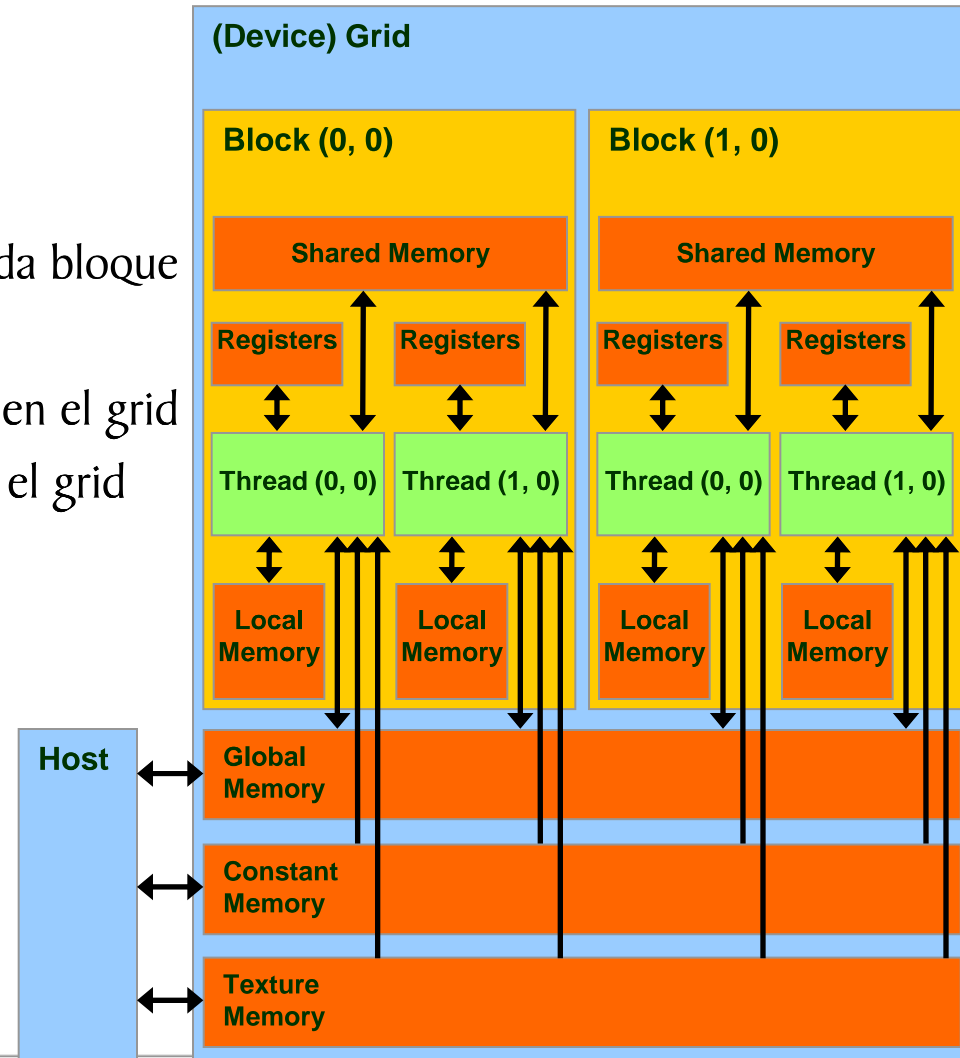
IDs de bloques y hebras

- Las hebras y los bloques tienen identificadores
  - De modo que cada hebra pueda decidir sobre qué datos trabajar
  - Block ID: 1D ó 2D (3D no se usa por ahora)
  - Thread ID: 1D, 2D, ó 3D
- Simplifica el acceso a memoria al procesar datos multidimensionales
  - Procesado de imagen
  - Resolución de PDEs en volúmenes
  - …



Jerarquía de memoria en CUDA

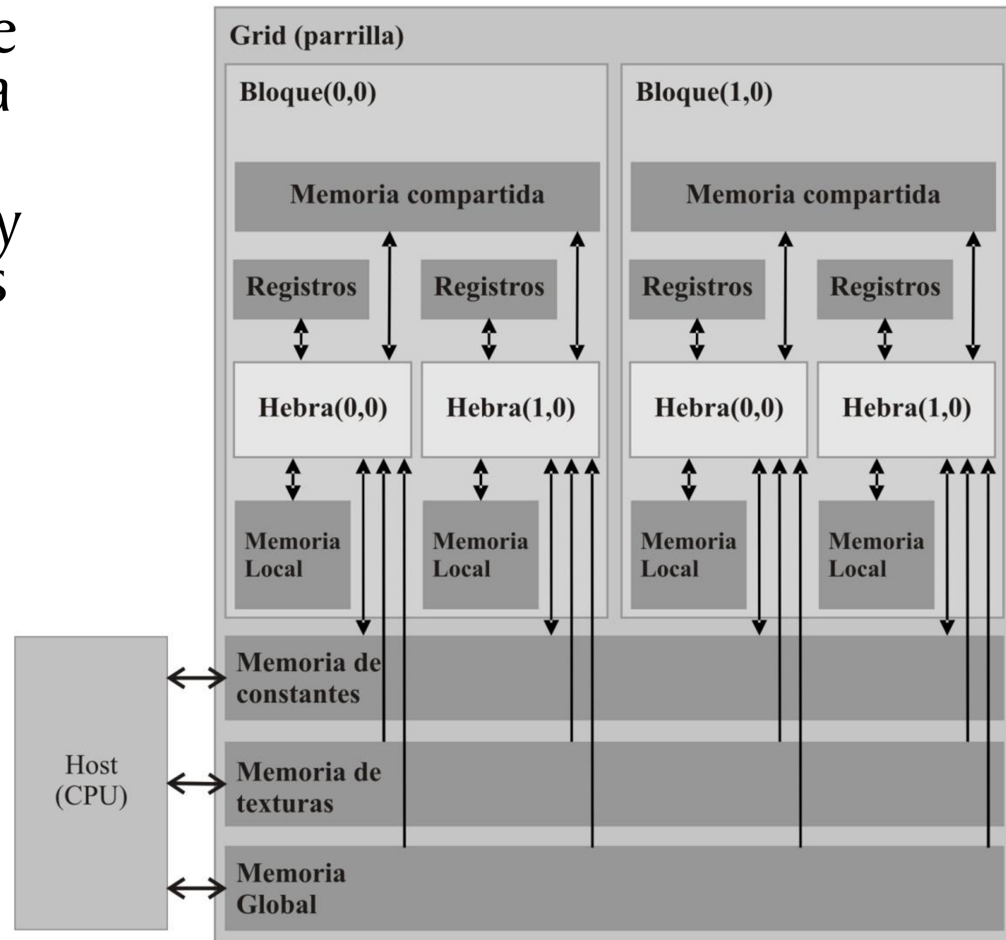
- Cada hebra puede:
  - R/W registros para cada hebra
  - R/W memoria local a cada hebra
  - R/W memoria compartida para cada bloque
  - R/W memoria global en el grid
  - Consultar memoria de constantes en el grid
  - Consultar memoria de texturas en el grid

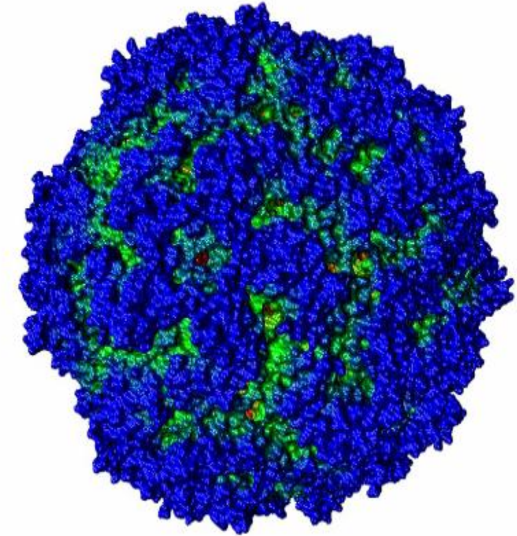


- La CPU (host), puede leer y escribir en la memoria global, de constantes, y de textura

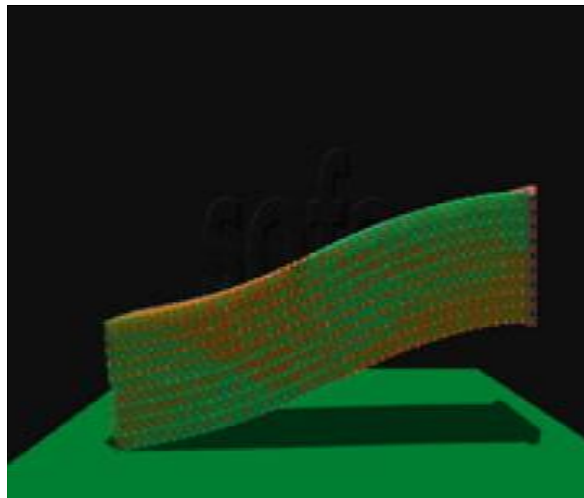
Memoria de gran latencia:

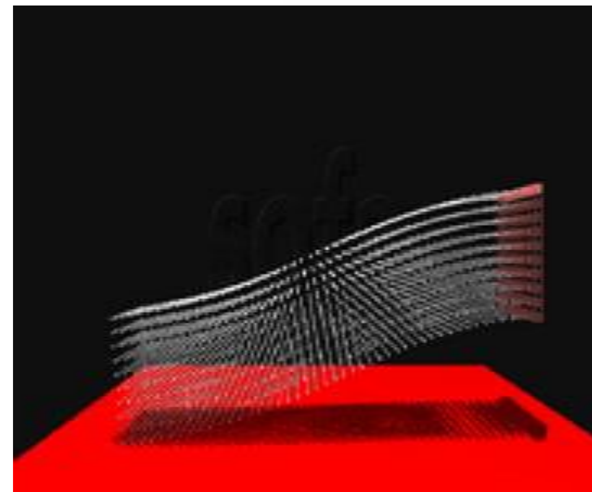
- Memoria global
  - Se utiliza principalmente para comunicar (R/W) la CPU con la GPU
  - Su contenido es visible y modificable desde todas las hebras
- Memorias de texturas y constantes
  - Su contenido es inicializado por la CPU
  - Sólo lectura (por todas las hebras)
  - Tienen mecanismos de caché que hace más eficiente su lectura





CUDA
a través de ejemplos





Ventajas: Facil y ligero

- La API es una extensión del lenguaje de programación ANSI C
  - ➡ Curva de aprendizaje asequible
- El hardware está pensado para que el driver y la parte runtime sean muy ligeras para la CPU
  - ➡ Alto rendimiento

[Adaptada de David Kirk y W-M. Hwu]

Ejemplo: Tipo de datos Matriz

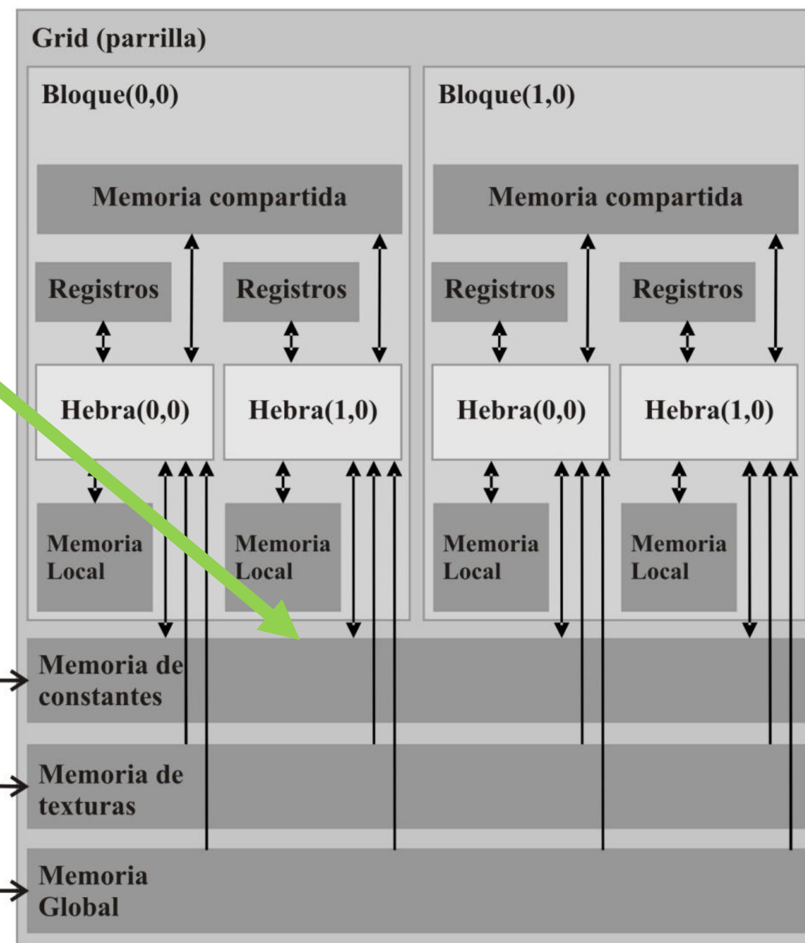
- No es parte de CUDA
- Lo utilizaremos a menudo en los ejemplos de clase
  - Matriz 2D
  - Elementos float (simple precisión)
  - width * height elementos
  - pitch tendrá significado cuando la matriz sea una submatriz de otra más grande
  - La memoria relativa a los elementos de datos se gestiona a partir del puntero elements

```
typedef struct {
    int width;
    int height;
    int pitch;
    float* elements;
} Matrix;
```

[Adaptada de David Kirk y W-M. Hwu]

Reserva de memoria de dispositivo con CUDA

- cudaMalloc()
  - Reserva espacio de memoria en el dispositivo (memoria de vídeo)
  - Requiere dos parámetros
    - Un puntero en el que se guardará la posición de memoria del objeto
    - El tamaño de memoria a reservar
- cudaFree()
  - Libera el espacio de memoria que comienza en el puntero dado (parámetro requerido)
  - Muy importante, sino entre ejecuciones de kernels el espacio permanece ocupado



Reserva de memoria de dispositivo con CUDA

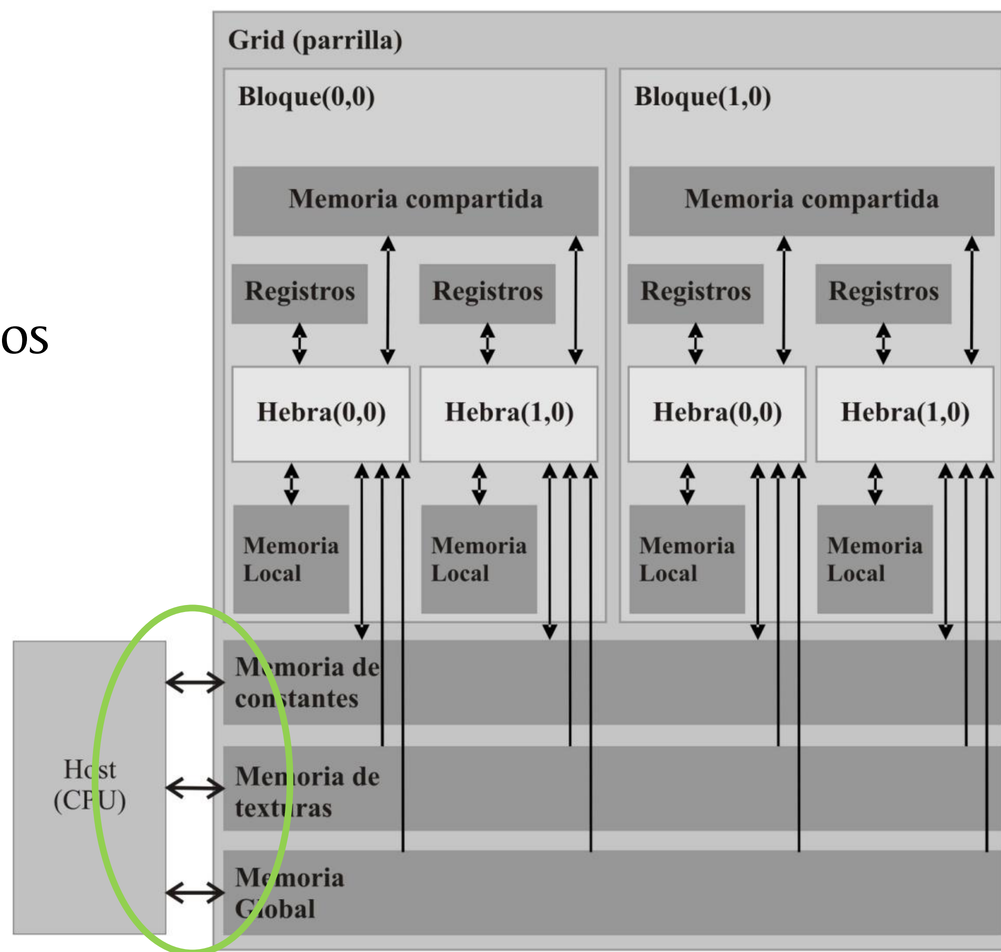
- Ejemplo de código:
  - Reservamos espacio para 64 * 64 floats (precisión simple)
  - Asociamos el espacio reservado a Md.elements

```
BLOCK_SIZE = 64;
Matrix Md
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)&Md.elements, size);
cudaFree(Md.elements);
```

[Adaptada de David Kirk y W-M. Hwu]

Transferencia de datos mediante CUDA

- cudaMemcpy()
  - Transfiere datos
  - Necesita 4 aparámetros
    - El puntero a la fuente
    - El puntero al destino
    - El número de bytes copiados
    - El tipo de transferencia
      - Host a Host
      - Host a Device
      - Device a Host
      - Device a Device
- Asíncrono en CUDA 1.0



Transferencia de datos mediante CUDA

- Ejemplo de código:
  - Copia un array de 64 * 64 floats
  - M está en la memoria principal y Md en memoria de vídeo
  - cudaMemcpyHostToDevice y cudaMemcpyDeviceToHost son constantes simbólicas

```
cudaMemcpy(Md.elements, M.elements, size,
    cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md.elements, size,
    cudaMemcpyDeviceToHost);
```

[Adaptada de David Kirk y W-M. Hwu]

Declaraciones de funciones CUDA

Ejemplo

	Se ejecuta en:	Sólos e peude llamar desde:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` define una función del Kernel
  - Debe devolver `void`
- `__device__` y `__host__` pueden ser utilizadas juntas (pero no revueltas)

[Adaptada de David Kirk y W-M. Hwu]

Declaraciones de funciones CUDA

- Las direcciones de las funciones `__device__` no se pueden tomar
- Para las funciones ejecutadas en la GPU:
  - No se permite la recursión
  - No podemos tener variables estáticas
  - Ni un número variable de argumentos

[Adaptada de David Kirk y W-M. Hwu]

(c) David Miraut Andrés

Llamada a la función kernel y generación de hebras

Ejemplo

- La función del kernel debe ser invocada con la junto con la configuración de ejecución:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);     // 5000 thread blocks
dim3   DimBlock(4, 8, 8);    // 256 threads per
   block
size_t SharedMemBytes = 64; // 64 bytes of shared
   memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes
   >>>(...);
```

- Las llamadas son asíncronas en CUDA 1.0, hace falta sincronismo explícito para bloqueo

[Adaptada de David Kirk y W-M. Hwu]

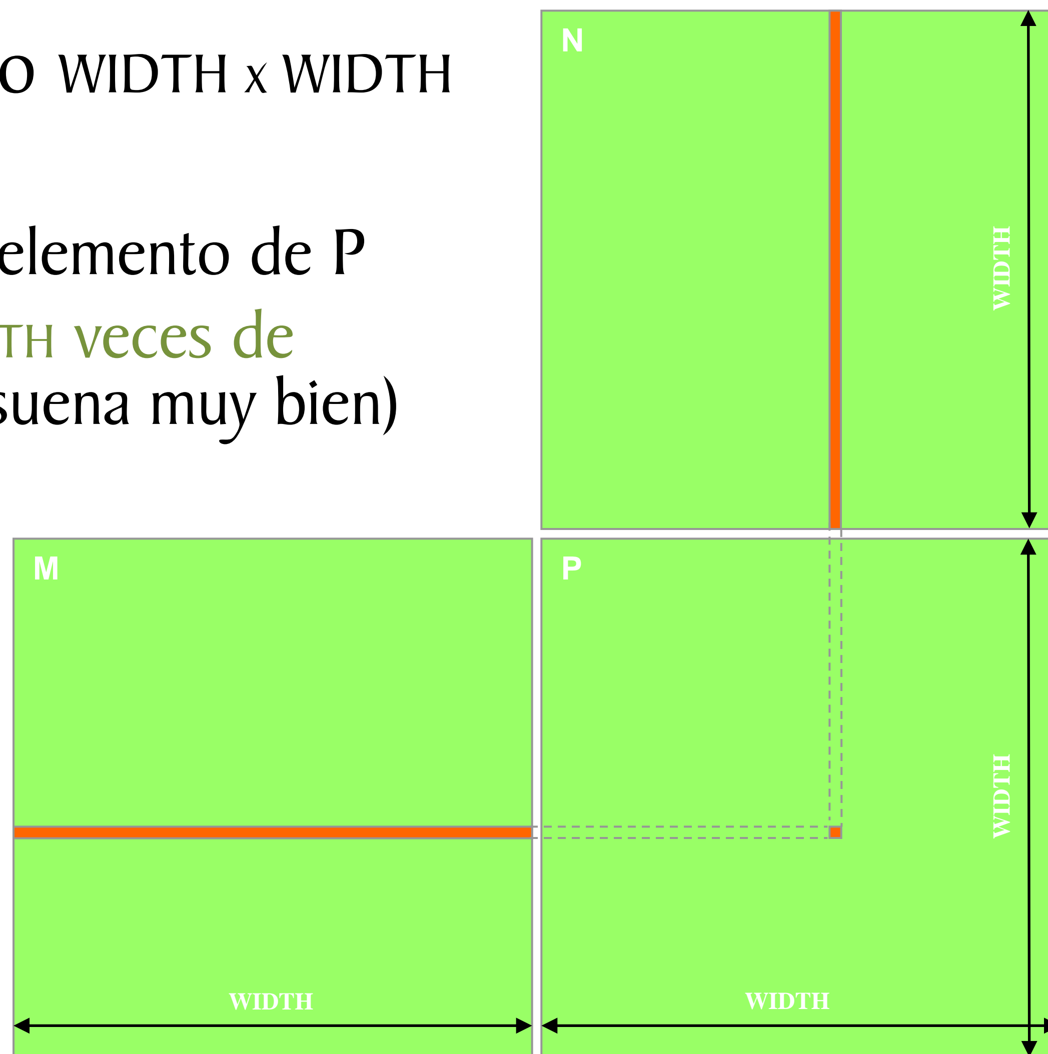
Ejemplo de multiplicación de matrices simple

- Un ejemplo muy simple (no optimizado) de multiplicación de matrices que muestra las características básicas de gestión de memoria y hebras en CUDA
  - Más adelante veremos cómo mejorarlo con memoria compartida
  - Uso de regitros locales
  - Uso de Thread IDs
  - Uso de la API en la transferencia de memoria entre CPU y GPU

[Adaptada de David Kirk y W-M. Hwu]

Ejemplo de multiplicación de matrices cuadradas

- P = M * N de tamaño WIDTH x WIDTH
- Sin subdivisiones:
  - Una hebra por cada elemento de P
  - M y N se cargan WIDTH veces de memoria global (no suena muy bien)



Paso 1: Transferencia de datos de matrices


```
// Allocate the device memory where we will copy M to
Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size,
    cudaMemcpyHostToDevice);

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size,
    cudaMemcpyDeviceToHost);
...
// Free device memory
cudaFree(Md.elements);
```

[Adaptada de David Kirk y W-M. Hwu]

Paso 2: Código del Host (CPU)

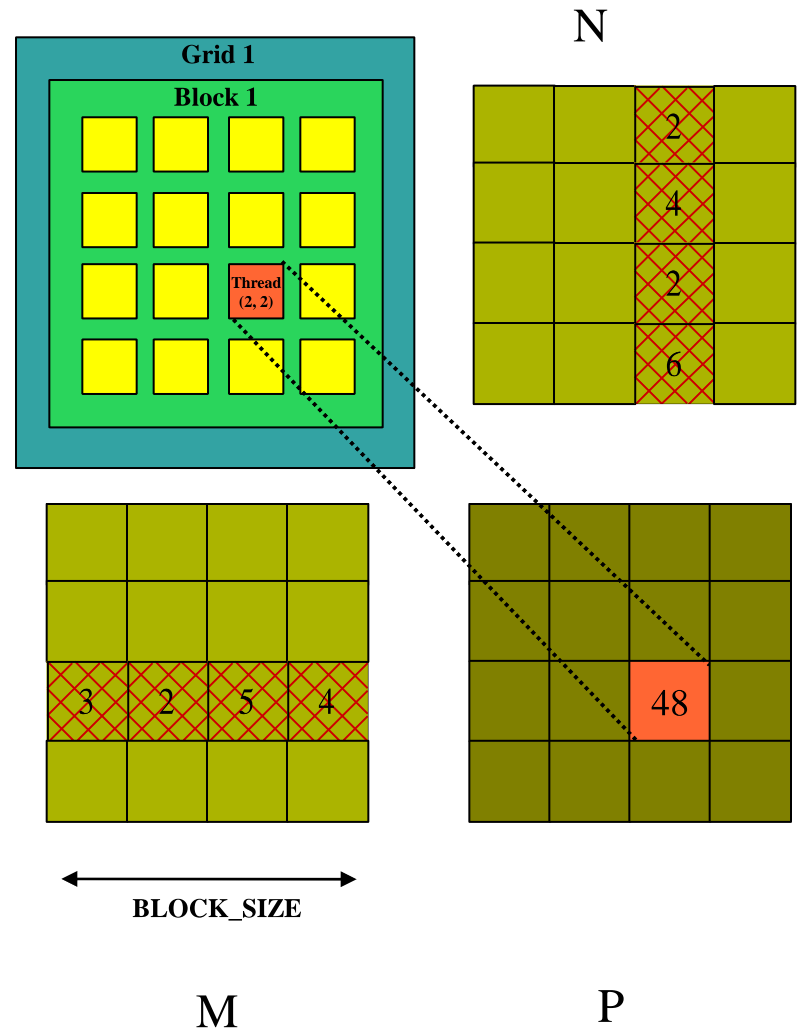
Ejemplo

```
// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i)
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];
                double b = N.elements[k * N.width + j];
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
}
```

[Adaptada de David Kirk y W-M. Hwu]

Multiplicación con un sólo Bloque

- Un bloque de hebras calcula la matriz P
  - Cada hebra se ocupa de un elemento de P
- Cada hebra
  - Carga una fila de la matriz M
  - Carga una columna de la matriz N
  - Lleva a cabo una multiplicación y una suma por cada par de elementos cargados de M y N
  - Tiene un ratio de cálculo / acceso a memoria cercano a 1:1 (bastante malo)
- El tamaño de la matriz está limitado por el número de hebras que pueden ser gestionadas en un bloque (multiprocesador). La mayor parte de la GPU se queda sin hacer nada.



Paso 3: Código en la CPU


```
int main(void) {
// Allocate and initialize the matrices
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

// M * N on the device
    MatrixMulOnDevice(M, N, P);

// Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
return 0;
}
```

[Adaptada de David Kirk y W-M. Hwu]

Paso 3: Código en la CPU

```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);
    CopyToDeviceMatrix(Pd, P); // Clear memory
```

[Adaptada de David Kirk y W-M. Hwu]

Paso 3: Código en la CPU

Ejemplo

```
// Setup the execution configuration
    dim3 dimBlock(WIDTH, WIDTH);
    dim3 dimGrid(1, 1);

    // Launch the device computation threads!
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```

[Adaptada de David Kirk y W-M. Hwu]

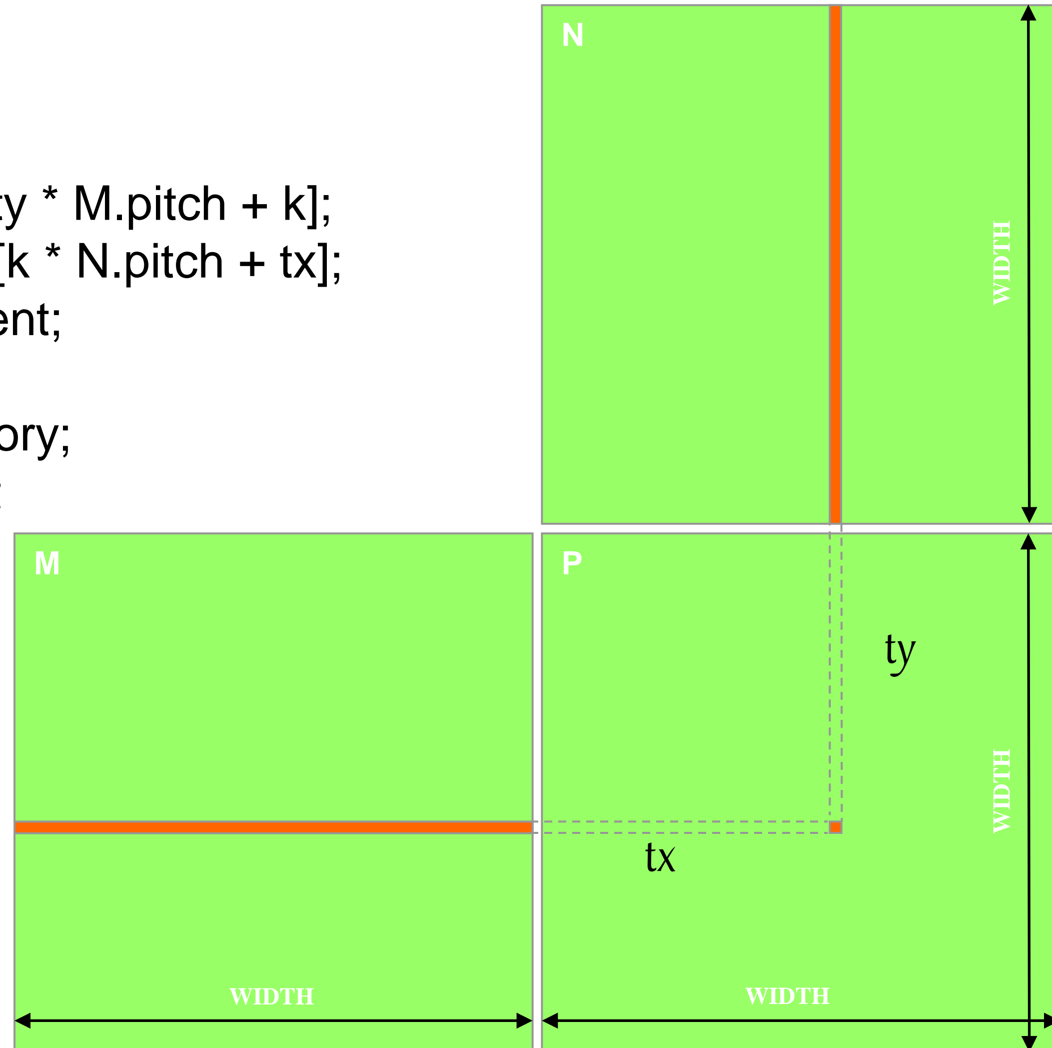
Paso 4: Código en la GPU


```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

[Adaptada de David Kirk y W-M. Hwu]

Paso 4: Código en la GPU


```
    for (int k = 0; k < M.width; ++k)
    {
        float Melement = M.elements[ty * M.pitch + k];
        float Nelement = Nd.elements[k * N.pitch + tx];
        Pvalue += Melement * Nelement;
    }
    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty * P.pitch + tx] =
                    Pvalue;
}
```



Paso 5: Algunos cabos sueltos

Ejemplo

```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M)
{
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

[Adaptada de David Kirk y W-M. Hwu]

(c) David Miraut Andrés

Paso 5: Algunos cabos sueltos

Ejemplo

```
// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost)
{
   int size = Mhost.width * Mhost.height * sizeof(float);
   cudaMemcpy(Mdevice.elements, Mhost.elements, size,
           cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice)
{
   int size = Mdevice.width * Mdevice.height * sizeof(float);
   cudaMemcpy(Mhost.elements, Mdevice.elements, size,
           cudaMemcpyDeviceToHost);
}
```

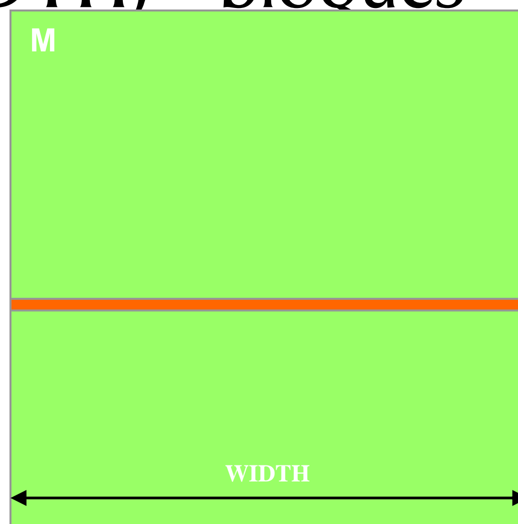
[Adaptada de David Kirk y W-M. Hwu]

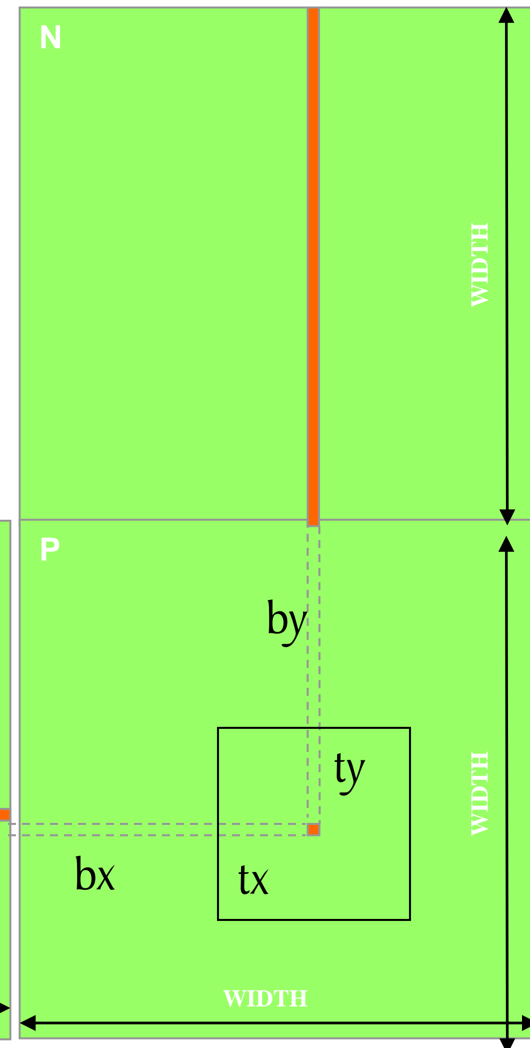
(c) David Miraut Andrés

Paso 6: Manejo de matrices cuadradas de tamaño arbitrario

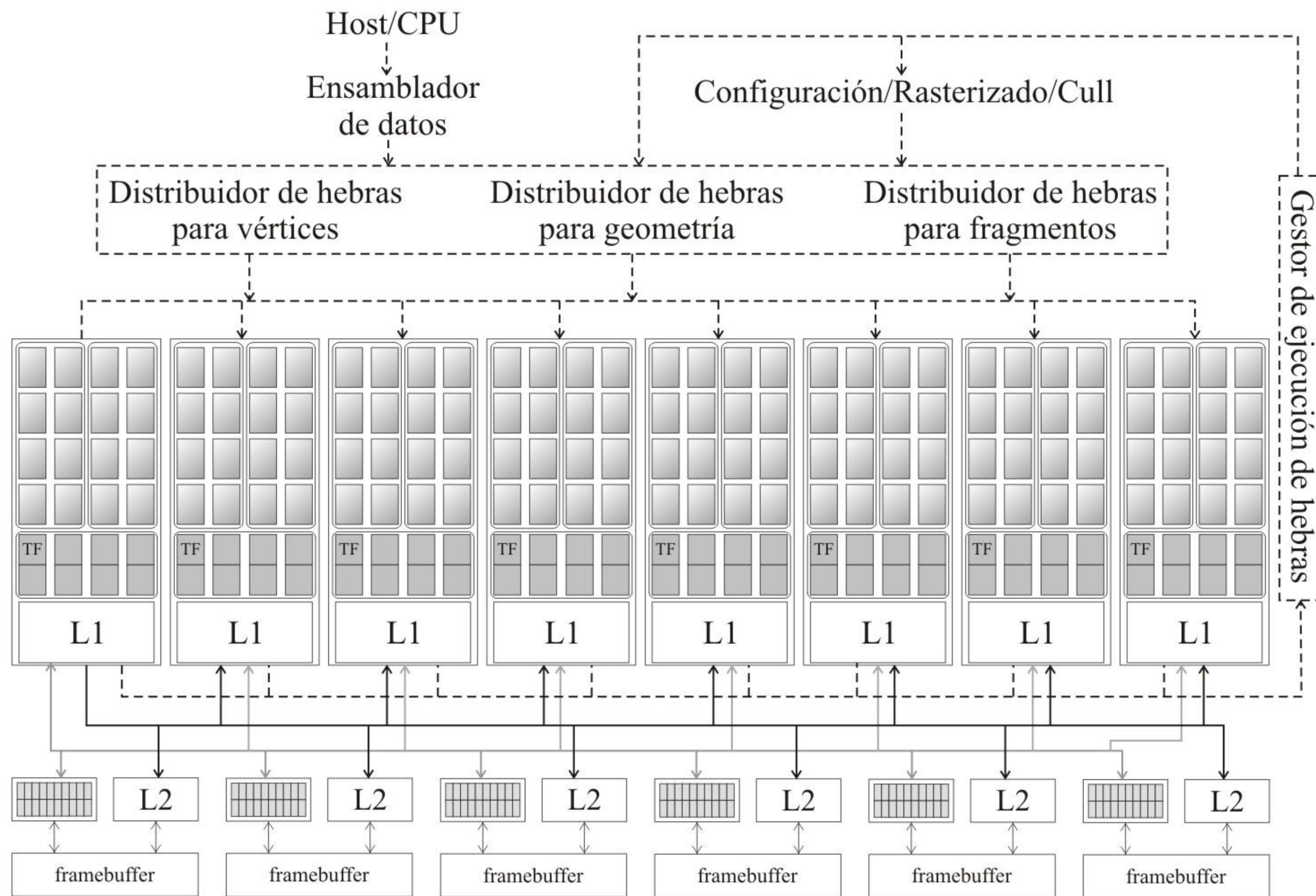
- Cada bloque de hebras 2D se encarga de calcular una submatriz de $(\text{BLOCK_WIDTH})^2$
  - Y tendrá $(\text{BLOCK_WIDTH})^2$ hebras
- Se genera un grid 2D de $(\text{WIDTH/BLOCK_WIDTH})^2$ bloques

Ejercicio práctico en el laboratorio

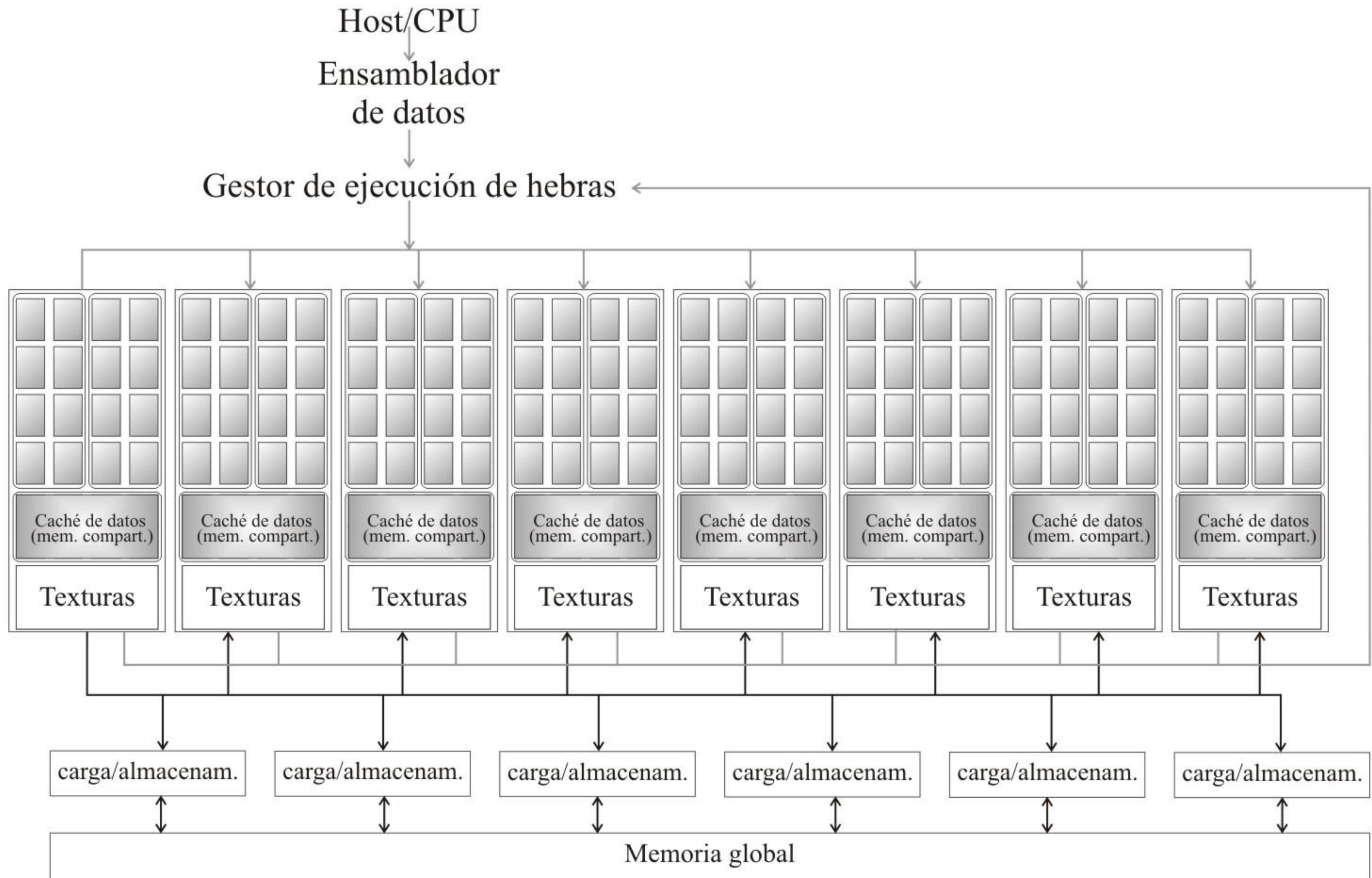




Personalidad Gráfica



Personalidad GPGPU



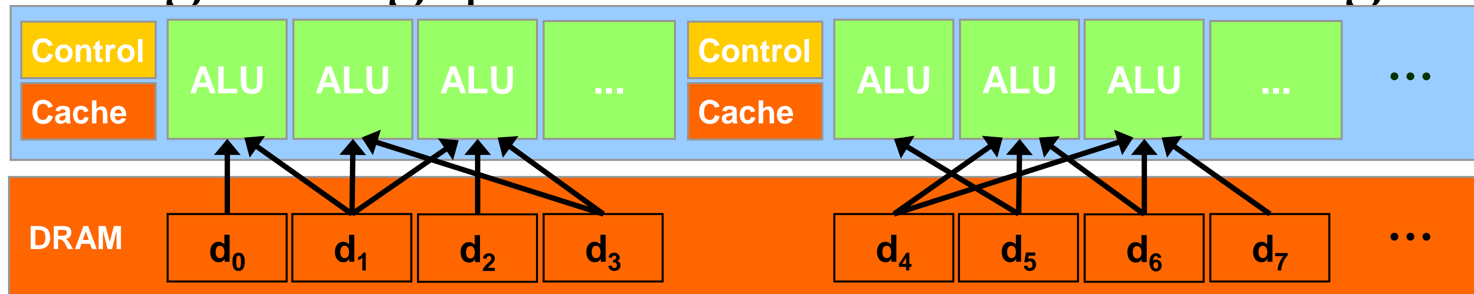
Especificaciones técnicas del G80

- Máximo número de hebras por bloque: 512
- Máximo número en cada dimensión del un Grid : 65,535
- Numero de multiprocesadores streaming (SM):
  - GeForce 8800 GTX: 16 @ 675 MHz
  - GeForce 8800 GTS: 12 @ 600 MHz
- Memoria de vídeo:
  - GeForce 8800 GTX: 768 MB
  - GeForce 8800 GTS: 640 MB
- Memoria compartida en cada multiprocesador: 16KB distribuida en 16 bancos
- Memoria de constantes: 64 KB
- Tamaño de Warp: 32 hebras (16 Warps/Bloque)

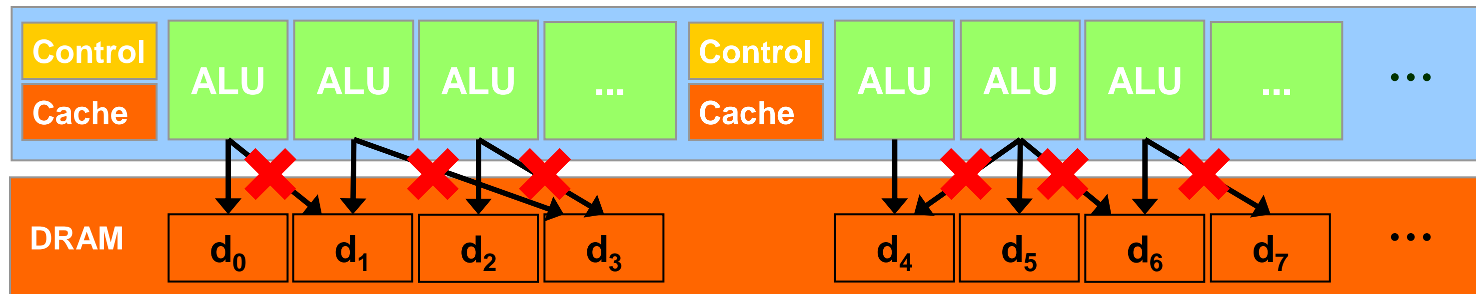
[Adaptada de David Kirk y W-M. Hwu]

Antiguas limitaciones hardware

- Accesos a memoria (ej. Con fragmentos) (de una pasada anterior)
  - Sólo gathering: podemos leer datos de otros fragmentos



No hay scattering: sólo podemos escribir en el propio fragmento

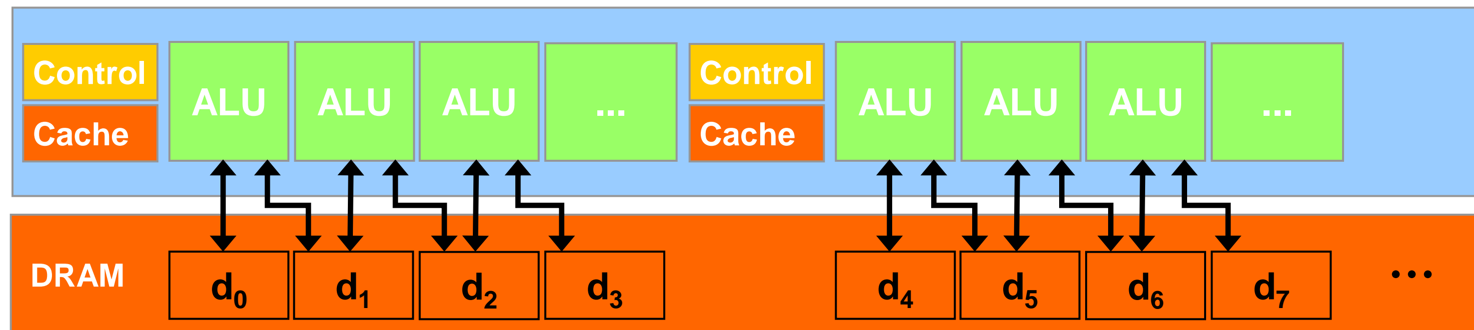


Poca flexibilidad

[Adaptada de David Kirk y W-M. Hwu]

Antiguas limitaciones hardware

- Las aplicaciones a menudo estaban limitadas por BW en alguna de las etapas (cuellos de botella)



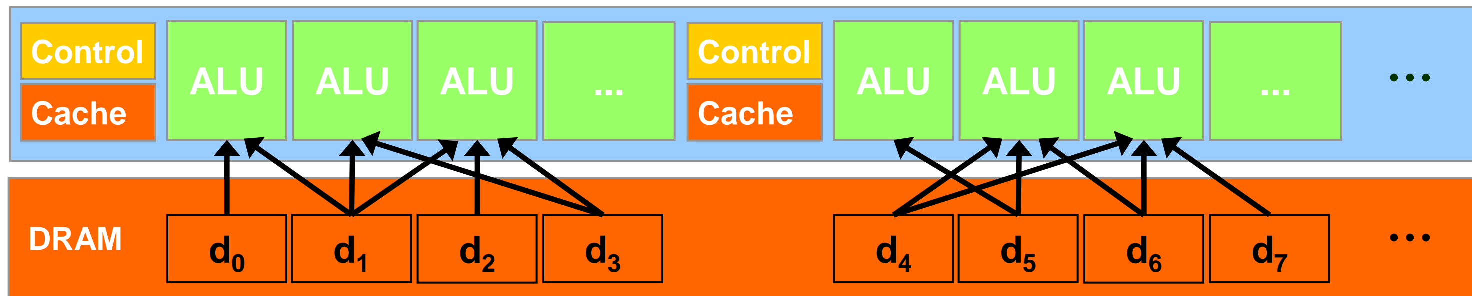
➡ Malgasto de la capacidad de cálculo debido a la inanición en las etapas siguientes

[Adaptada de David Kirk y W-M. Hwu]

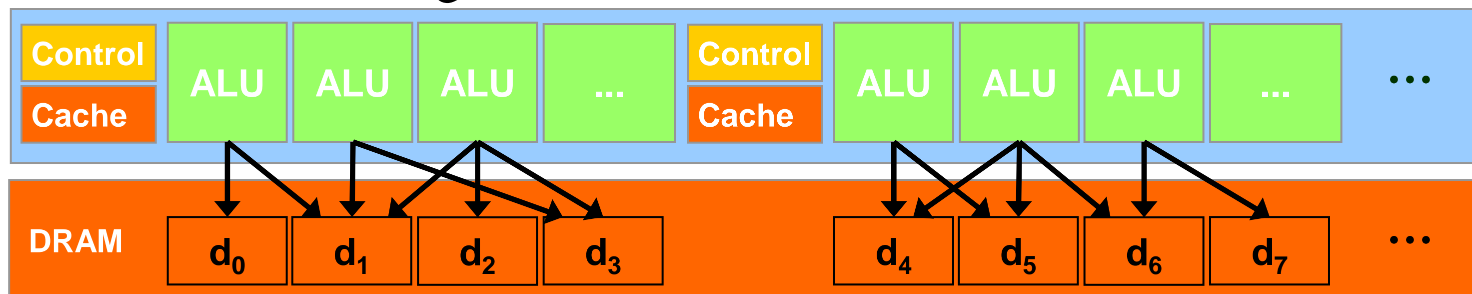
Novedades de CUDA: Scatter

CUDA nos da acceso y direccionamiento a la memoria de vídeo a nivel de byte

Gather:



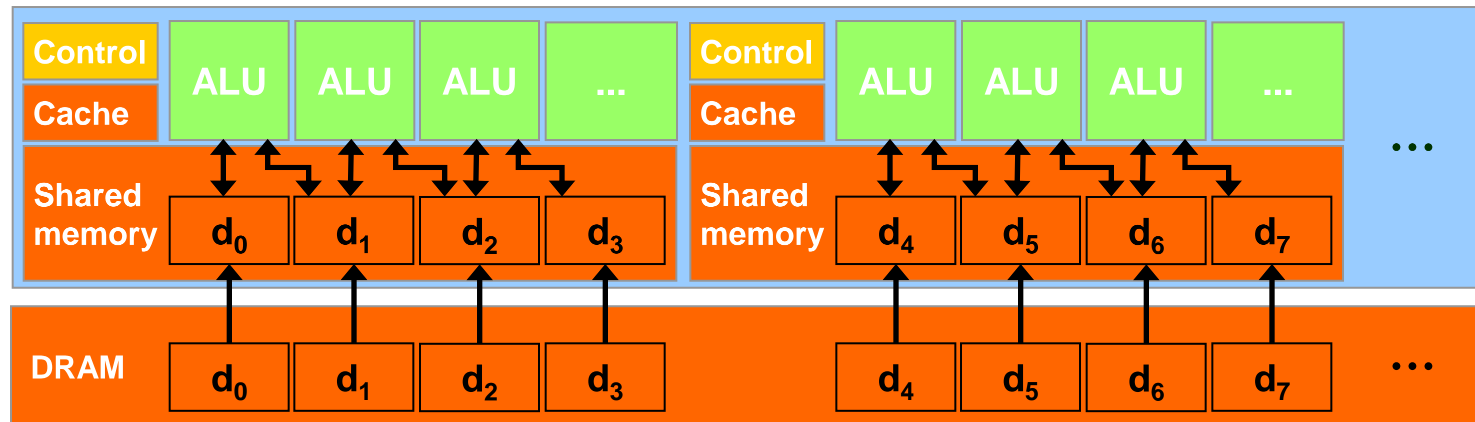
Y scatter: ya no estamos limitados a escribir sobre un único fragmento

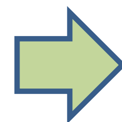


[Adaptada de David Kirk y W-M. Hwu]

Memoria compartida On-Chip

- CUDA nos da acceso a la memoria compartida de los multiprocesadores para realizar una comunicación y compartición de datos más eficiente



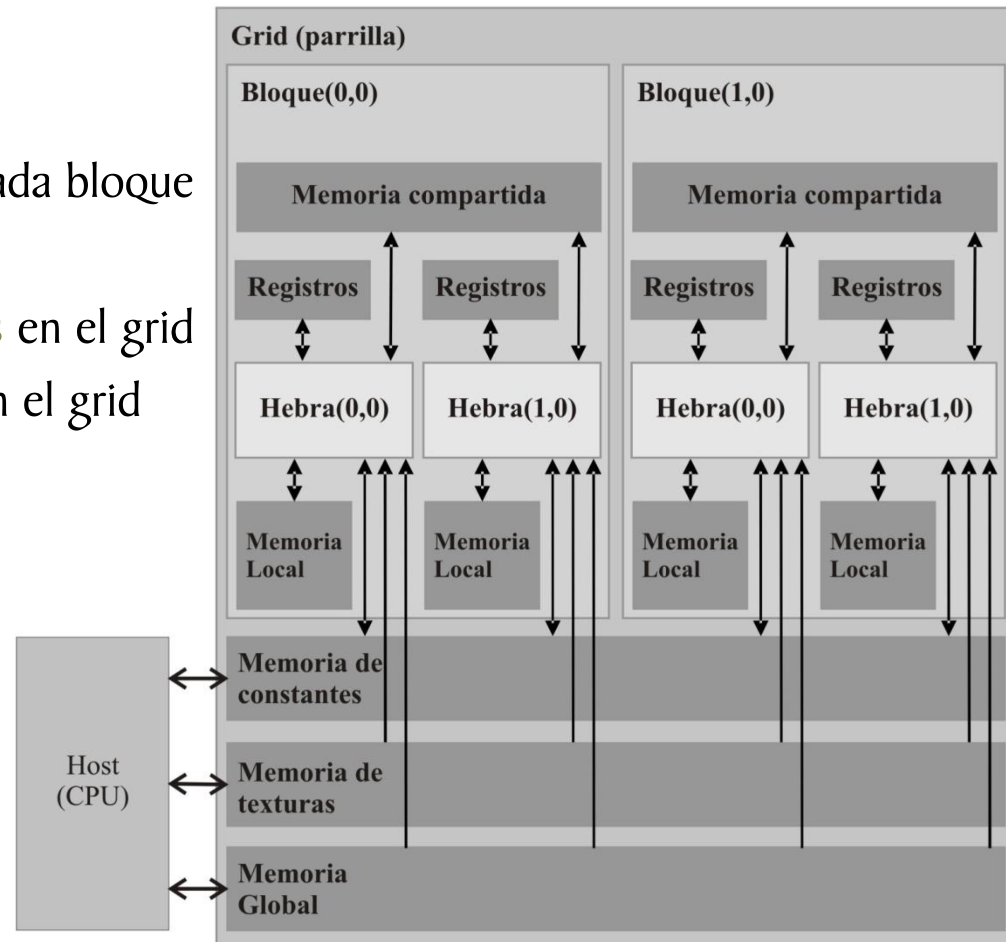


Gran ahorro de BW

[Adaptada de David Kirk y W-M. Hwu]

Jerarquía de memoria en CUDA

- Cada hebra puede:
  - R/W registros para cada hebra
  - R/W memoria local a cada hebra
  - R/W memoria compartida para cada bloque
  - R/W memoria global en el grid
  - Consultar memoria de constantes en el grid
  - Consultar memoria de texturas en el grid
- La CPU (host), puede leer y escribir en la memoria global, de constantes, y de textura



Consejos de programación en CUDA

Cuidado!

- La memoria local y global se encuentran en memoria de vídeo (DRAM) – cuyo acceso es mucho más lento que la memoria compartida (y distribuida) de los multiprocesadores
- De manera que una forma conveniente de diseñar los programas consiste en tratar de aprovechar la memoria más cercana y más rápida:
  - Particionar los datos en subconjuntos que quepan en la memoria compartida
  - Manejar cada subconjunto sólo con un bloque de hebras para:
    - Cargar el subconjunto de memoria global a memoria compartida mediante múltiples threads que exploten el paralelismo a nivel de memoria
    - Llevar a cabo el cálculo del subconjunto en esta memoria local, en la que se pueden hacer múltiples consultas de forma eficiente
    - Copiar los resultados de memoria compartida a memoria global

[Adaptada de David Kirk y W-M. Hwu]

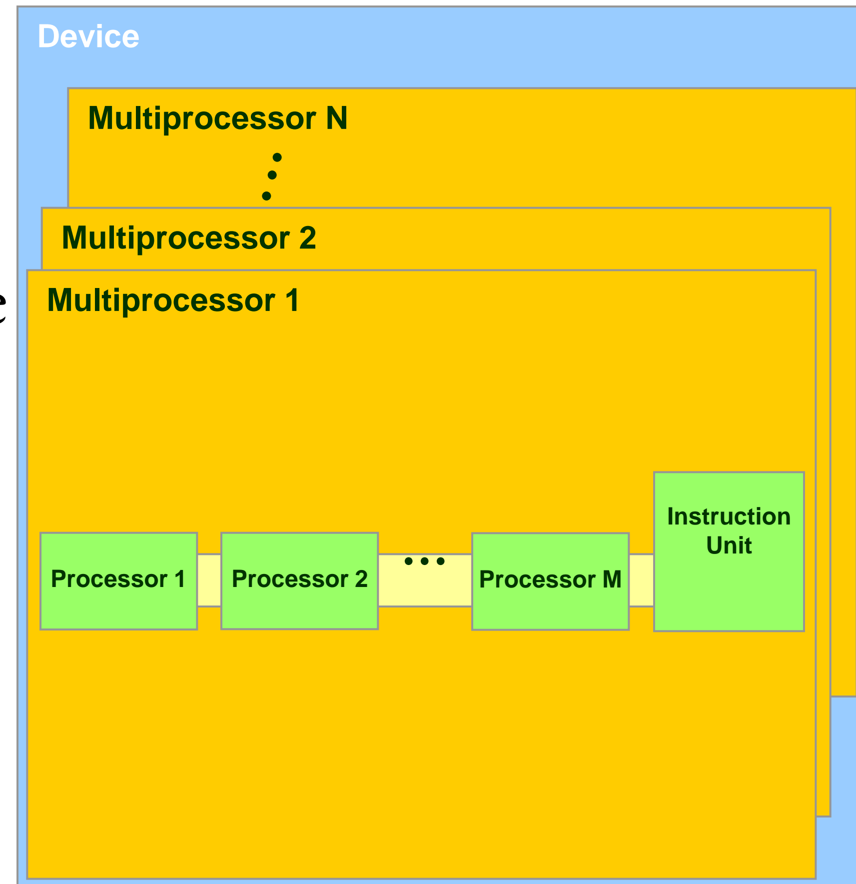
Consejos de programación en CUDA

- La memoria de texturas y de constantes se encuentran en memoria de vídeo – cuyo acceso es bastante lento
  - Pero tienen mecanismos de cachés (¿y prebúsqueda?)
  - Es bastante eficiente si se lee de forma ordenada
- Podemos adecuar nuestros datos a los distintos tipos de memoria de acuerdo con su uso:
  - R/O sin estructura → memoria de constantes
  - R/O array estructurado→ memoria de texturas
  - R/W compartida dentro de un bloque → memoria compartida
  - R/W la memoria local a menudo tendrá forma física de registros
  - R/W entradas/resultados → memoria global

[Adaptada de David Kirk y W-M. Hwu]

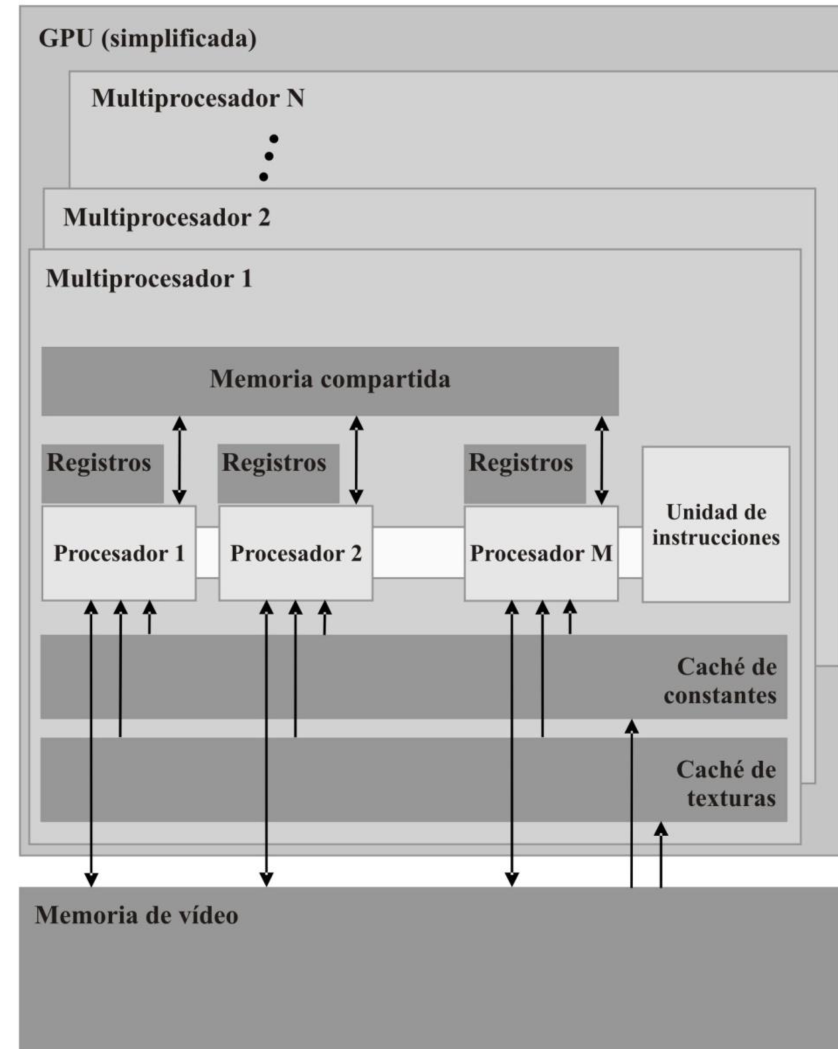
Multiprocesadores SPMD

- Actualmente la GPU (device) tiene 16 multiprocesadores
- Cada multiprocesador es un conjunto de procesadores de 32 bits con SPMD, que teóricamente comparten la unidad de instrucciones
- En cada ciclo de reloj, un multiprocesador ejecuta la misma instrucción en un grupo de hebras llamado warp
- El número de hebras en un warp es el warp size



Arquitectura de Memoria

- Los espacios de memoria de texturas, constantes y global son regiones de la memoria de vídeo
- Cada multiprocesador tiene:
  - Un conjunto de registros de 32 bits por procesador
  - Memoria compartida On-chip
  - Cache para el acceso a la memoria de constantes
  - Caché para el acceso a memoria de texturas
    - Espera cierta localidad en los accesos
    - Que también es de sólo lectura



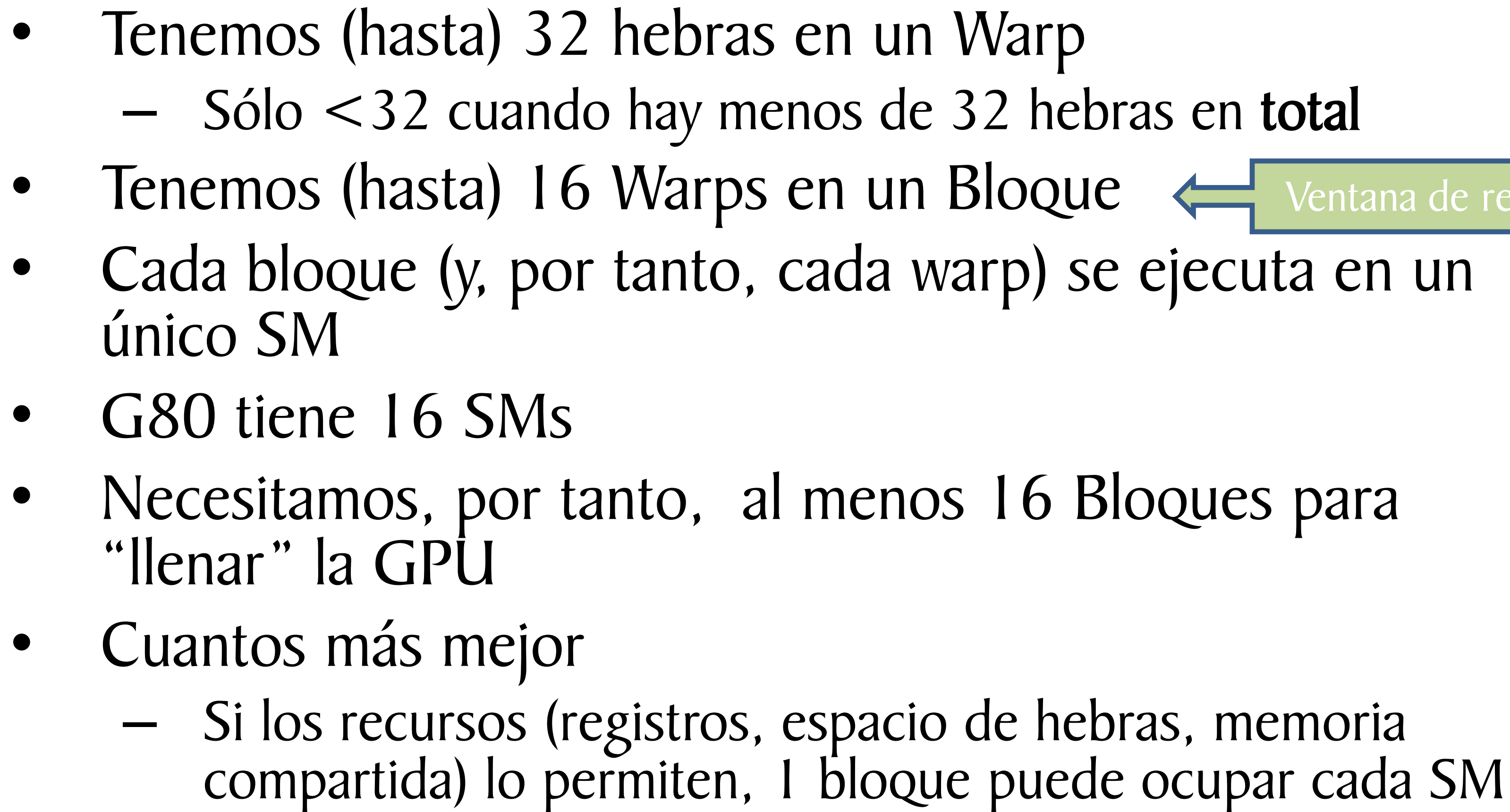
Modelo de ejecución

- Cada bloque de hebras es dividido en warps, cada uno de los cuales es ejecutado por un multiprocesador (SM)
  - La GPU con CUDA sólo procesa un Grid a la vez
- Cada bloque es ejecutado en un multiprocesador
  - De manera que la memoria compartida pueda ser aprovechada entre distintas hebras del mismo bloque
- Un multiprocesador puede ejecutar varios bloques concurrentemente
  - La memoria compartida y los registros disponibles son repartidos entre las hebras de todos los bloques concurrentes
  - De modo que al disminuir el uso de memoria compartida (por bloque) y el de registros (por thread) logramos incrementar el número de bloques que pueden ejecutarse concurrentemente

[Adaptada de David Kirk y W-M. Hwu]

(c) David Miraut Andrés

Hebras, Warps y Bloques



- Tenemos (hasta) 32 hebras en un Warp
  - Sólo <32 cuando hay menos de 32 hebras en **total**
- Tenemos (hasta) 16 Warps en un Bloque ⟸ Ventana de registros
- Cada bloque (y, por tanto, cada warp) se ejecuta en un único SM
- G80 tiene 16 SMs
- Necesitamos, por tanto, al menos 16 Bloques para "llenar" la GPU
- Cuantos más mejor
  - Si los recursos (registros, espacio de hebras, memoria compartida) lo permiten, 1 bloque puede ocupar cada SM

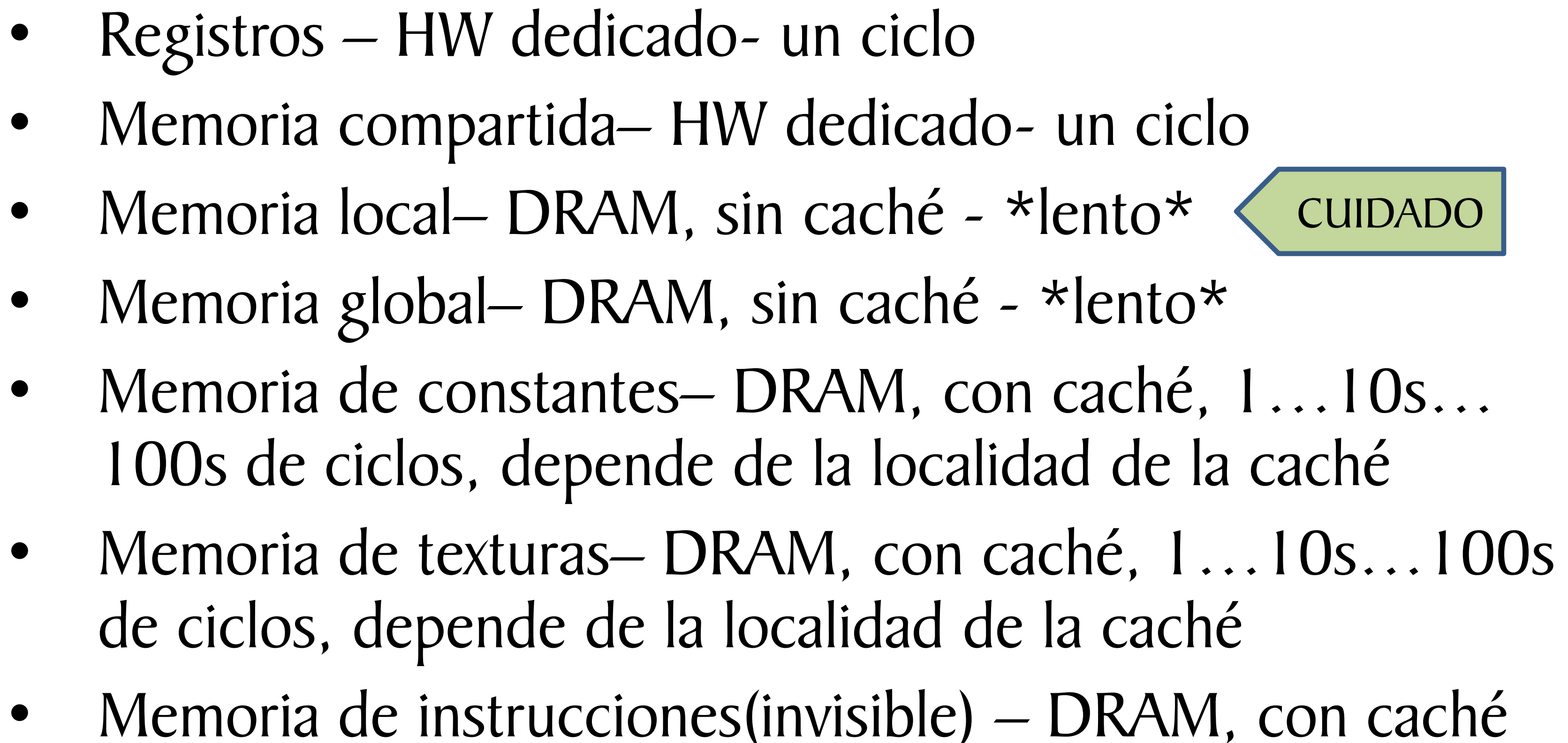
[Adaptada de David Kirk y W-M. Hwu]

Repaso de terminología

- device = GPU = conjunto de multiprocesadores
- Multiprocesador = conjunto de procesadores y memoria compartida
- Kernel = programa en la GPU
- Grid = array de bloques de hebrs que ejecutan un kernel
- Bloque de hebras = grupo de hebras SPMD que ejecutan un kernel y pueden comunicarse mediante memoria compartida

Memoria	Localización	Con caché	Acceso	Quién la usa?
Local	Fuera del chip	No	Read/write	Una hebra
Compartida	On-chip	N/A – (es local)	Read/write	Hebras del bloque
Global	Fuera del chip	No	Read/write	Todas las hebras + CPU
Constantes	Fuera del chip	Sí	Read	Todas las hebras + CPU
Texturas	Fuera del chip	Sí	Read	Todas las hebras + CPU

Tiempos de acceso



- Registros – HW dedicado- un ciclo
- Memoria compartida– HW dedicado- un ciclo
- Memoria local– DRAM, sin caché - *lento* CUIDADO
- Memoria global– DRAM, sin caché - *lento*
- Memoria de constantes– DRAM, con caché, 1…10s… 100s de ciclos, depende de la localidad de la caché
- Memoria de texturas– DRAM, con caché, 1…10s…100s de ciclos, depende de la localidad de la caché
- Memoria de instrucciones(invisible) – DRAM, con caché

[Adaptada de David Kirk y W-M. Hwu]

(c) David Miraut Andrés

Application Programming Interface

- La API está hecha de forma que nos resulte como una extensión del lenguaje C
- Tenemos:
  - Extensiones del lenguaje
    - To target portions of the code for execution on the device
  - Una librería runtime con las siguientes partes:
    - Un componente común con tipos predefinidos y un subconjunto de C que nos permite escribir código en la CPU (host) y la GPU (device)
    - Un componente en el host para controlar el acceso a uno ó más devices
    - Un componente de device que nos da la funcionalidad propia de la GPU (memoria de vídeo, arrays de procesadores…)

[Adaptada de David Kirk y W-M. Hwu]

Nuevos tipos de variables

“Cualiificadores” de las variables

	Memoria	Ámbito	Tiempo de vida
`__device__ __local__ int LocalVar;`	local	hebra	hebra
`__device__ __shared__ int SharedVar;`	compartida	bloque	bloque
`__device__ int GlobalVar;`	global	grid	aplicación
`__device__ __constant__ int ConstantVar;`	constante	grid	aplicación

- `__device__` es opcional cuando se usa con `__local__`, `__shared__`, or `__constant__`
- Las variables “automáticas” sin un “cualificador” se guardan como registros en la GPU
  - Excepto los arrays que se encuentran en memoria local

Restricciones de los tipos de variables

- Los punteros sólo pueden apuntar a zonas de memoria global:
  - Reservadas en el host y copiadas al kernel:

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

  - O como dirección obtenida de una variable global:

```
float* ptr = &GlobalVar;
```

[Adaptada de David Kirk y W-M. Hwu]

(c) David Miraut Andrés

Variables/registros útiles

- `dim3 gridDim;`
  - Las dimensiones del grid en número de bloques (`gridDim.z` no se usa)
- `dim3 blockDim;`
  - Las dimensiones del bloque en número de hebras
- `dim3 blockIdx;`
  - El índice/identificador del bloque dentro del grid
- `dim3 threadIdx;`
  - El índice/identificador de la hebra dentro del bloque

[Adaptada de David Kirk y W-M. Hwu]

Componentes comunes de Runtime

- Nos ofrece:
  - Tipos vectoriales pre-definidos (G80 es escalar)
  - Un subconjunto de la librería runtime de C para el host y el device

[Adaptada de David Kirk y W-M. Hwu]

(c) David Miraut Andrés

Tipos vectoriales pre-definidos

- `[u]char[1..4]`, `[u]short[1..4]`, `[u]int[1..4]`, `[u]long[1..4]`, `float[1..4]`
  - Las estructuras tienen los campos `x`, `y`, `z`, `w` :

```
uint4 param;
int y = param.y;
```

- `dim3`
  - Basado en `uint3`
  - Se usan para especificar dimensiones

[Adaptada de David Kirk y W-M. Hwu]

Funciones matemáticas

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- Etc.
  - Cuando se ejecutan en la CPU, las funciones utilizan la implementación runtime de C cuando está disponible
  - Sólo sirven para valores escalares, no hay soporte para tipos vectoriales

[Adaptada de David Kirk y W-M. Hwu]

Componente Runtime del host

- Nos ofrece funciones para:
  - Gestión del Device (también para sistemas con multiples GPUs)
  - Gestión de Memoria
  - Gestión de Errores
- Se inicializa la primera vez que se llama a una función del runtime
- Cada device es invocado en una hebra diferente del host (necesitamos varias hebras si tenemos varias GPUs)

[Adaptada de David Kirk y W-M. Hwu]

Gestión de memoria

- Reserva de memoria en la GPU
  - `cudaMalloc()`, `cudaFree()`
- Copia de memoria entre dispositivos (CPU, GPU…)
  - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`
- Direccionamiento de memoria
  - `cudaGetSymbolAddress()`

cudaMemcpy() se diferencia de cudaMemcpyToSymbol() en que también puede ser utilizada para copiar de las memoria del host al device.

[Adaptada de David Kirk y W-M. Hwu]

Funciones matemáticas

- Algunas funciones matemáticas (ej. `sin(x)`) tienen una versión alternativa, menos precisa pero más rápida en el device (ej. `__sin(x)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`

[Adaptada de David Kirk y W-M. Hwu]

Funciones de sincronización

- `void __syncthreads();`
- Sincroniza todas las hebras en un bloque
- Una vez todas las hebras llegan a este punto, la ejecución continúa normalmente
- Se utilizan para evitar riesgos RAW/WAR/WAW al acceder a memoria compartida ó memoria global
- Se pueden usar dentro de estructuras condicionales sólo si la condición es uniforme en todo el bloque de hebras

Los errores derivados de poner barreras en ramas Condicionales son muy difíciles de detectar

[Adaptada de David Kirk y W-M. Hwu]

Compilación

- Cualquier fichero de código fuente que tenga extensiones del lenguaje con CUDA debe ser compilado con nvcc
- nvcc es un compiler driver
  - Invoca a todas las herramientas y compiladores necesarios: cudacc, g++, cl, ...
- nvcc dar como salida:
  - Código C
    - Que tendrá que ser compilado con el resto de aplicaciones
  - O directamente código objeto

[Adaptada de David Kirk y W-M. Hwu]

Enlazado

- Los ejecutables de código escrito con CUDA necesitan al menos dos librerías dinámicas:
  - La libreria runtime de CUDA (`cudart`)
  - Las librería del núcleo de CUDA (`cuda`)

[Adaptada de David Kirk y W-M. Hwu]

Debugging con el modo de emulación

- Con el "device emulation mode" (`nvcc -deviceemu`) podemos ejecutar una aplicación de CUDA completamente en la CPU
  - No hace falta tener una tarjeta de la serie 8 ó 9, ni el driver instalado
  - Cada hebra de la GPU se convierte en una hebra de la CPU
  - Es muchísimo más lento
- Al ejecutar en "device emulation mode", podemos:
  - Utilizar el código nativo y las facilidades a las que estamos acostumbrados en la CPU (breakpoints, inspection, etc.)
  - Acceder a cualquier dato específico de "memoria de vídeo" o de los multiprocesadores desde la CPU y viceversa (esto es un arma de doble filo)
  - Llamar a cualquier función del host desde el device y viceversa (ej. printf)
  - Detectar deadlocks debido al uso inapropiado de `__synthreads`

[Adaptada de David Kirk y W-M. Hwu]

Problemas en modo de emulación

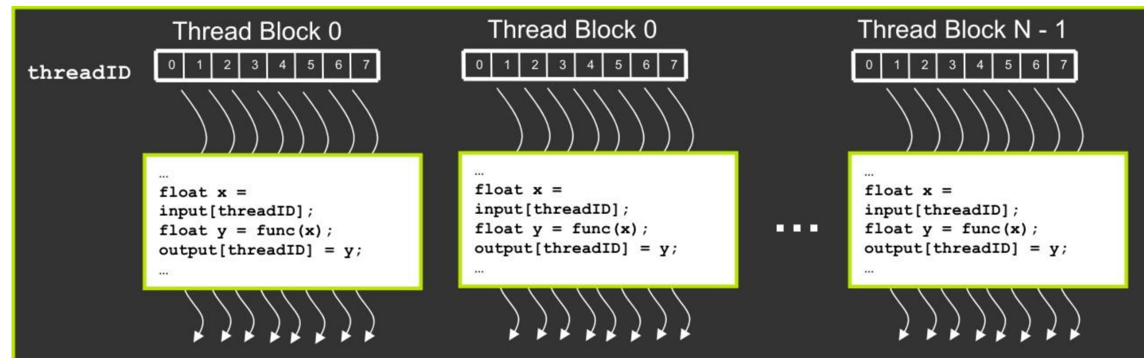
- Las hebras del device emulado se ejecutan secuencialmente, de modo que podemos tener comportamientos distintos cuendo en la GPU se produzcan accesos simultáneos a las mismas posiciones de memoria
- La dereferenciación de punteros del host en el device puede funcionar en modo emulación, pero es un error grave en la ejecución real
- Los resultados de las operaciones de coma flotante pueden ser ligeramente diferentes debido a:
  - Los distintos juegos de instrucciones, compiladores…
  - El uso de precisiñon extendida para los valores de resultados intermedios
    - Hay argumentos del compilador que pueden forzar el uso estricto de simple precisión en el host

[Adaptada de David Kirk y W-M. Hwu]

(c) David Miraut Andrés

Resumen

- GPU es un procesador masivamente paralelo
- NVIDIA G80/G92: 128 procesadores
- Soporta miles de hebras activas simultáneamente (12,288 on G80)
- CUDA nos proporciona un modelo de programación que puede aprovechar facilmente este paralelismo (SPMD)
- Sintaxis sencilla: extensiones mínimas a C/C++
- Escalabilidad transparente en el futuro harware y su evolución
- Un kernel de CUDA se ejecuta en un array de hebras que se organizan por multprocesadores
  - Los identificadores de las hebras nos ayudan en decisiones de control (ej. Eleccion de datos a procesar)



Resumen: ventajas

- Acceso aleatorio a una memoria direccionable a nivel de byte
  - Una hebra puede acceder a cualquier posición de memoria
- Acceso ilimitado a posiciones de memoria
  - Se puede leer/escribir donde sea necesario
- Jerarquía de memoria para optimizar el uso del BW
  - Memoria compartida por bloque y sincronización de hebras
- Curva de aprendizaje suave
  - Solo unas extensiones de C
- No hace falta ser un "gurú" de la programación de shaders para programar GPGPU
  - No tenemos sobrecarga por utilizar la API gráfica

Laboratorio del miércoles

- Explicaremos los secretos de la Multiplicación eficiente de matrices
- Pasaremos la barrera de los 45 GFLOPS