UNIVERSIDAD DE ALMERÍA

Computación en procesadores gráficos

Programación con CUDA

José Antonio Martínez García
Francisco M. Vázquez López
Manuel Ujaldón Martínez
Ester Martín Garzón

Contenidos

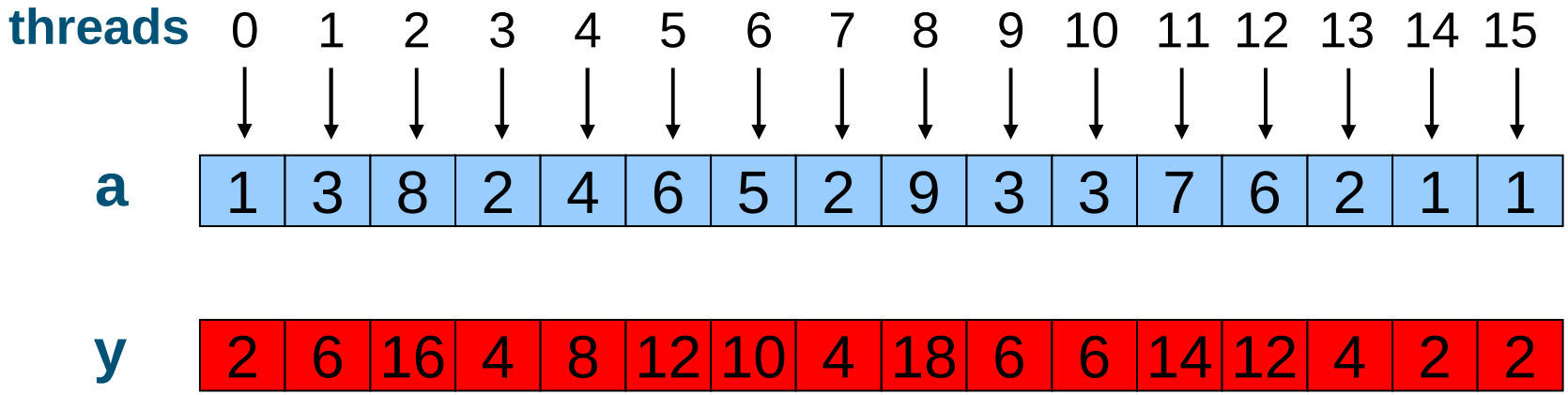
- Arquitectura de las GPUs
- **Modelo de programación SIMT**
- **Claves computacionales para la programación de GPUs**
- **Programación con CUDA**
- Supercomputación gráfica y arquitecturas emergentes

Modelo de programación SIMT

- **SIMT**: Single Instruction Multiple Threads
  - Extensión de SIMD (Single Instruction Multiple Data)
  - Programación vectorial
  - Paralelismo de grano fino
  - Unidad mínima de ejecución: **thread**
  - Agrupación en bloques y grid
  - Cada thread es asignado a un SP
  - Los distintos bloques se asignan a los SMs. Cola de bloques por SM.
  - Para la ejecución, los bloques se dividen en agrupaciones de 32 threads: **warp**
  - El acceso a memoria se realiza en grupos de 16 threads: **half-warp**
  - Esta organización tiene el objetivo de aprovechar al máximo las posibilidades de paralelismo de la arquitectura, ocultando latencias mediante la ejecución concurrente de tareas.

Modelo de programación SIMT

Operación y = α a, α = 2



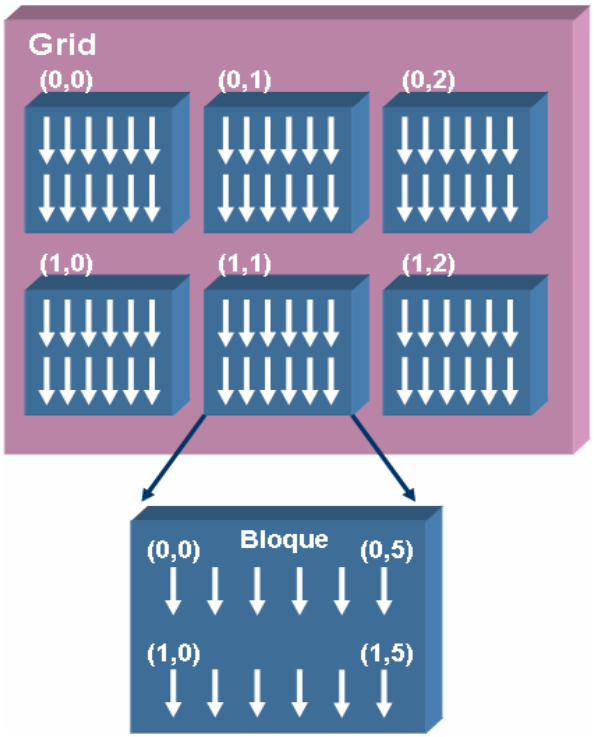
Speed – up* = 16x

Modelo de programación SIMT

- Bloques:
  - Dimensiones: 1D, 2D, 3D
  - Máximo 512 threads por bloque
  - Dimensión máxima: 512 x 512 x 64
- Grid:
  - Dimensiones: 1D, 2D
  - Dimensión máxima: 65535 x 65535 x 1

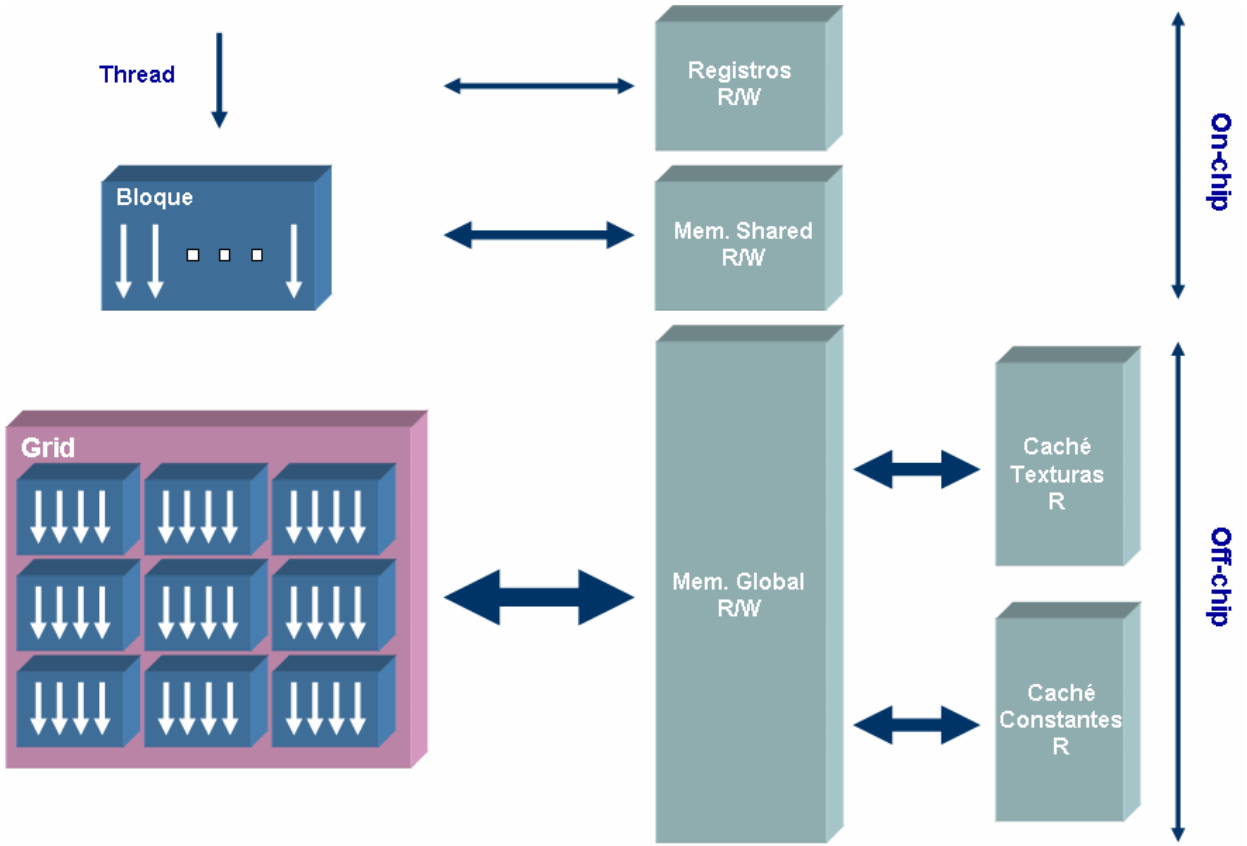
Modelo de programación SIMT

- **¿ Cómo se determina la topología del bloque ?**
- Mapear los datos del problema a los threads
- **Problemas**:
  - Situaciones en las que se accede a una misma estructura de distinta forma
  - Limitados por las operaciones que definen el algoritmo
  - Dependencias de datos
- **Soluciones**:
  - Cambiar la forma de almacenar las estructuras
  - Dividir en múltiples kernels



Modelo de programación SIMT

- **Jerarquía de memoria**



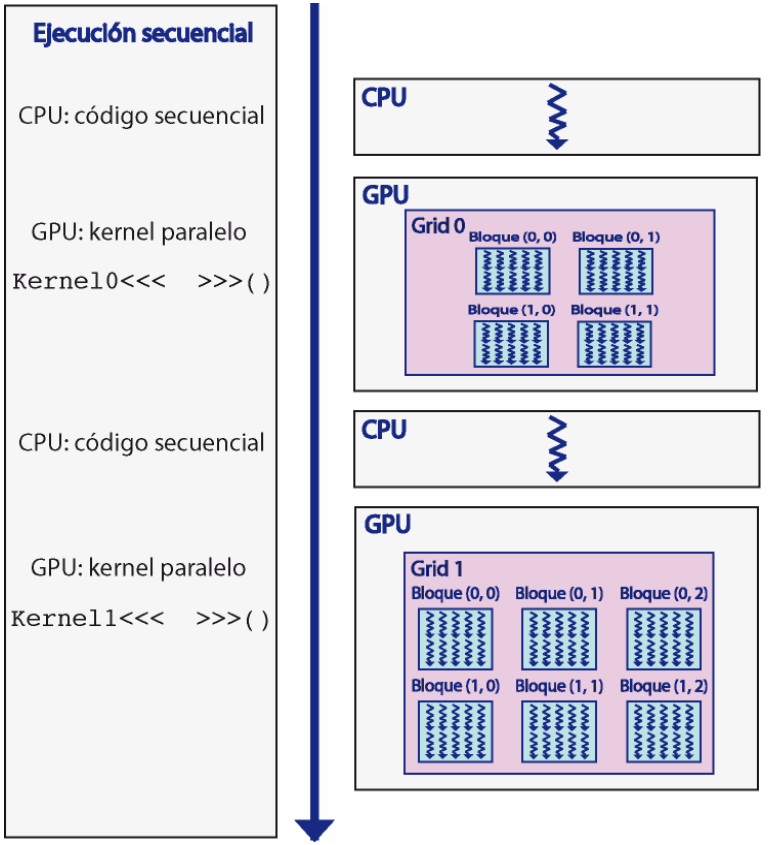
Modelo de programación SIMT

- **Jerarquía de memoria**

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	No	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	No	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

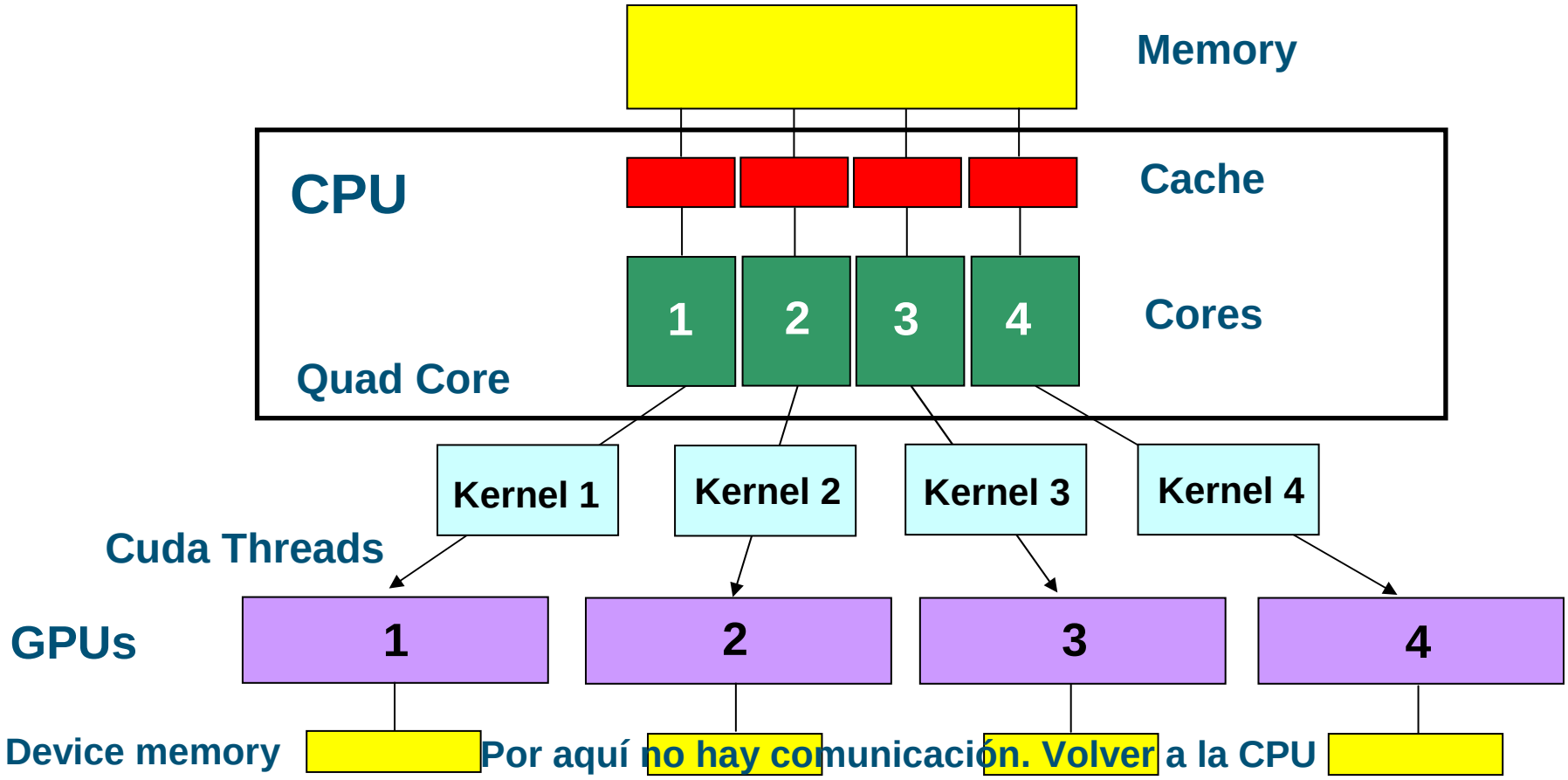
Modelo de programación SIMT

- **Programación heterogénea CPU-GPU**
  - CPU: **Host**. Código secuencial
  - GPU: **Device**. **Kernels**. Código paralelo
  - Ejecución asíncrona
  - Multi GPU: Cudathreads
- **Etapas:**
  - Transferencia CPU -> GPU: PCIX
  - Ejecución del kernel
  - Transferencia GPU -> CPU: PCIX



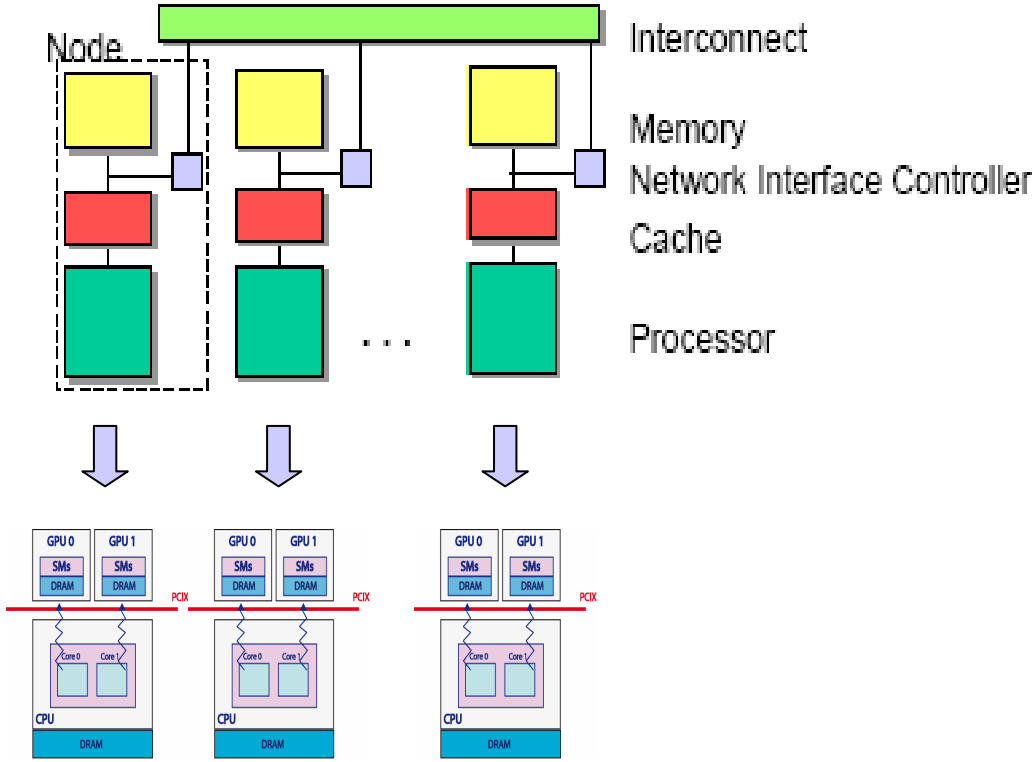
Modelo de programación SIMT

- Multi GPU



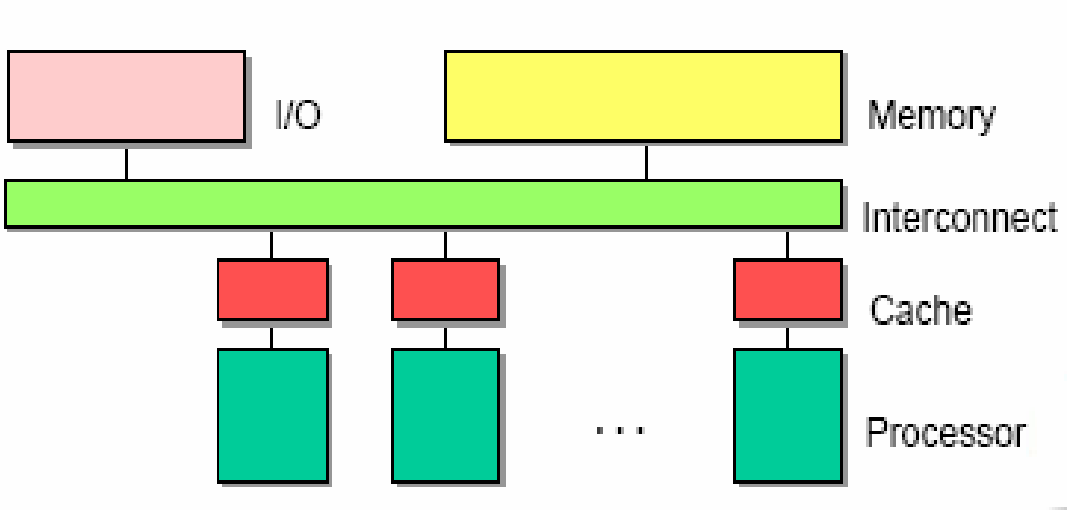
Modelo de programación SIMT

- **Modelo memoria distribuida: MPI**



Modelo de programación SIMT

- **Modelo memoria compartida: Posix-Threads, Open-MP**



Modelo de programación SIMT

- Compute Capability 1.0, 1.1, 1.2, 1.3 y 2.0
  - Define características de la GPU, especificaciones, operaciones que puede realizar, tamaños de memoria, grid y bloques…
  - Chip G80: **1.0**
  - Chip G92, G96, G98: **1.1**
  - Chip GT215, GT216, GT218: **1.2**
  - Chip GT200: **1.3**
  - Chip GT300 “Fermi”: **2.0**

Modelo de programación SIMT

	Capacidad de cómputo				
Operación	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Funciones atómicas sobre enteros de 32 bits en memoria global	No	Si			
Funciones atómicas sobre enteros de 64 bits en memoria global	No		Si		
Funciones atómicas sobre enteros de 32 bits en memoria shared					
Funciones de predicado para los threads de un warp					
Números en coma flotante en doble precisión	No			Si	
Suma atómica en coma flotante para palabras de 32 bits en memoria global y memoria shared	No				Si
Funciones avanzadas de sincronización de threads: __syncthreads_count(), __syncthreads_and(), __syncthreads_or()					

Modelo de programación SIMT

	Capacidad de cómputo				
Especificaciones técnicas	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Dimensión máxima (x, y) de un grid	65535				
Número máximo de threads por bloque	512				1024
Dimensión máxima (x, y) de un bloque	512				1024
Dimensión máxima z de un bloque	64				
Tamaño del warp	32				
Número máximo de bloques activos por SM	8				
Número máximo de warps activos por SM	24		32		48
Número máximo de threads activos por SM	768		1024		1536
Número de registros de 32 bits por SM	8 K		16 K		32 K
Cantidad máxima de memoria shared por SM	16 Kb				48 Kb
Número de bancos en memoria shared	16				32
Cantidad de memoria local por thread	16 Kb				512 Kb
Tamaño de memoria de constantes	64 Kb, 8Kb por SM				
Tamaño de memoria de texturas por SM	Según dispositivo, 6 – 8 Kb				
Número máximo de instrucciones por kernel	$2x10^6$				

Contenidos

- Arquitectura de las GPUs
- **Modelo de programación SIMT**
- **Claves computacionales para la programación de GPUs**
- **Programación con CUDA**
- Supercomputación gráfica y arquitecturas emergentes

Claves computacionales

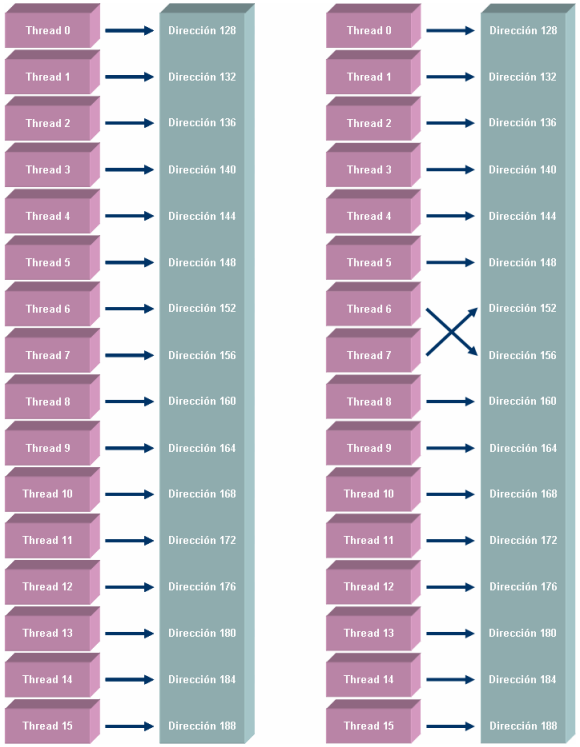
- **Coalescing**: Acceso coalescente o fusionado a memoria global
  - Capacidad de la arquitectura para obtener 16 palabras de memoria **simultáneamente** (en un único acceso) por medio de los 16 threads del half-warp
  - Reduce la latencia de memoria a 16 veces menos
  - Primera cuestión de eficiencia en el desarrollo del kernel
  - Se consigue bajo ciertos patrones de acceso
  - Estos patrones dependen de la capacidad de cómputo de la GPU

Claves computacionales

- **Coalescing compute capability 1.0, 1.1**
  - El acceso a memoria de los 16 threads de un half-warp puede realizarse en **uno** o **dos** accesos a memoria si:
    - Los 16 threads acceden a palabras de 4 bytes (float, int) con lo que resulta en **una** operación de 64 bytes
    - Los 16 threads acceden a palabras de 8 bytes (float2, int2) con lo que resulta en **una** operación de 128 bytes
    - Los threads acceden a palabras de 16 bytes (float4, int4) con lo que resulta en **dos** operaciones de 128 bytes
    - Las 16 palabras accedidas por los 16 threads del half-warp han de estar en el **mismo segmento de memoria**
    - Los threads han de acceder en secuencia a las 16 palabras de memoria. O sea, el ***k-th* thread** accede a la ***k-th* palabra**
    - ¿ Qué pasa con el tipo double ?

Claves computacionales

- **Coalescing compute capability 1.0, 1.1**



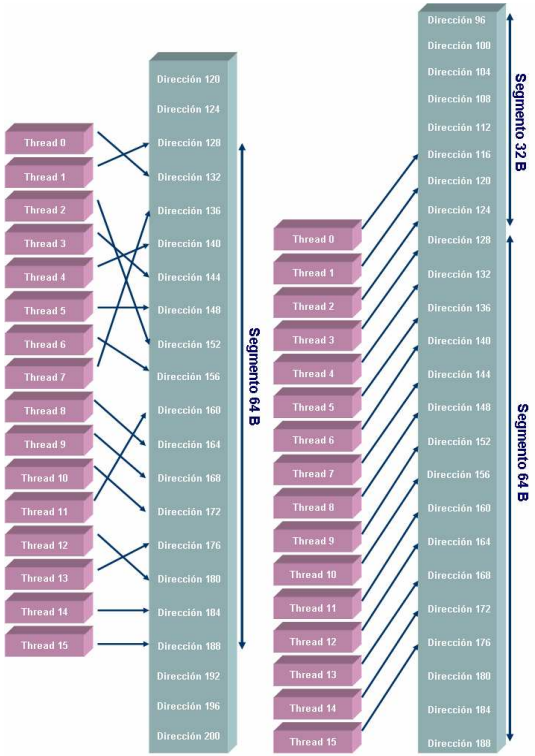
1 acceso **16 accesos**

Claves computacionales

- **Coalescing compute capability 1.2, 1.3**
  - El acceso a memoria de los 16 threads de un half-warp puede realizarse en **un** acceso a memoria si:
    - Los 16 threads acceden a palabras dentro del mismo segmento de memoria, siendo éste de tamaño:
      - 32 bytes, si los threads acceden a palabras de 1 byte
      - 64 bytes, si los threads acceden a palabras de 2 bytes
      - 128 bytes, si los threads acceden a palabras de 4 ó 8 bytes
    - El tamaño de segmento ha de ser el **doble** de las palabras a las que se accede salvo en el caso en el que las palabras son de 8 bytes
    - Cuando el patrón de acceso a memoria requiere el uso de ***n*-segmentos** distintos, se realizarán ***n*-accesos**

Claves computacionales

- **Coalescing compute capability 1.2, 1.3**



1 acceso **2 accesos**

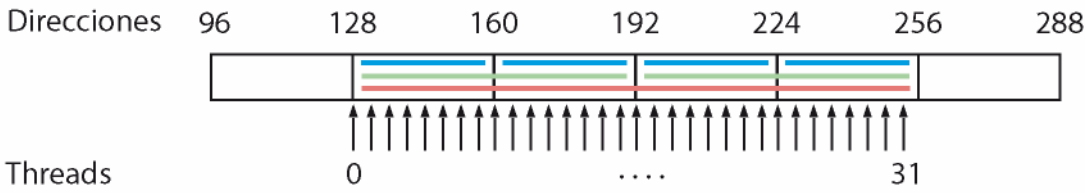
Claves computacionales

- **Coalescing compute capability 2.0**
  - El acceso a memoria se realiza a segmentos alineados de 128 bytes:
    - **Un** acceso si los 32 threads del **warp** acceden a palabras de 4 bytes
    - **Dos** accesos si los 16 threads de cada **half-warp** acceden a palabras de 8 bytes
    - **Cuatro** accesos si los 8 threads de cada **quarter-warp** acceden a palabras de 16 bytes

Claves computacionales

Ejemplo coalescing I

Caso 1: Memoria alineada y acceso secuencial

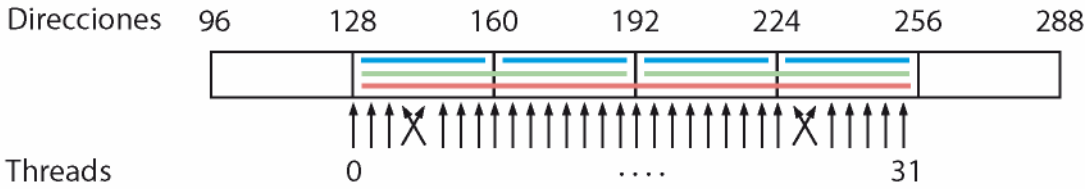


Capacidad de cómputo	1.0 y 1.1	1.2 y 1.3	2.0
Transacciones de memoria	Sin caché		L1 y L2
	1 x 64B en 128 1 x 64B en 192	1 x 64B en 128 1 x 64B en 192	1 x 128B en 128

Claves computacionales

Ejemplo coalescing II

Caso 2: Memoria alineada y acceso no secuencial

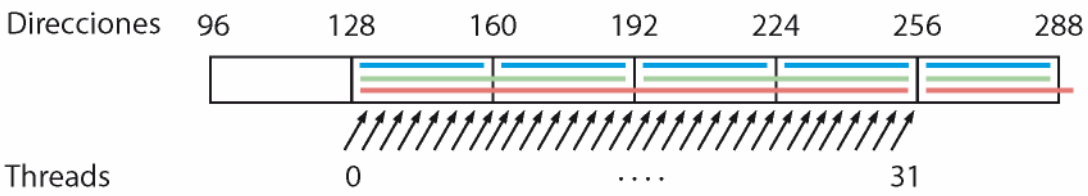


Capacidad de cómputo	1.0 y 1.1	1.2 y 1.3	2.0
Transacciones de memoria	Sin caché		L1 y L2
	8 x 32B en 128 8 x 32B en 160 8 x 32B en 192 8 x 32B en 224	1 x 64B en 128 1 x 64B en 192	1 x 128B en 128

Claves computacionales

Ejemplo coalescing III

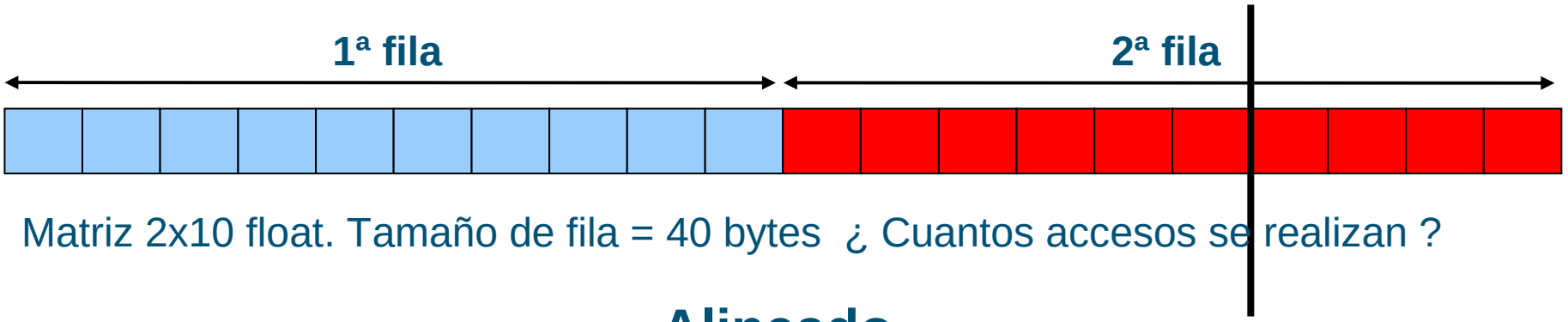
Caso 3: Memoria no alineada y acceso secuencial



Capacidad de cómputo	1.0 y 1.1	1.2 y 1.3	2.0
Transacciones de memoria	Sin caché		L1 y L2
	7 x 32B en 128 8 x 32B en 160 8 x 32B en 192 8 x 32B en 224 1 x 32B en 256	1 x 128B en 128 1 x 64B en 192 1 x 32B en 256	1 x 128B en 128 1 x 128B en 256

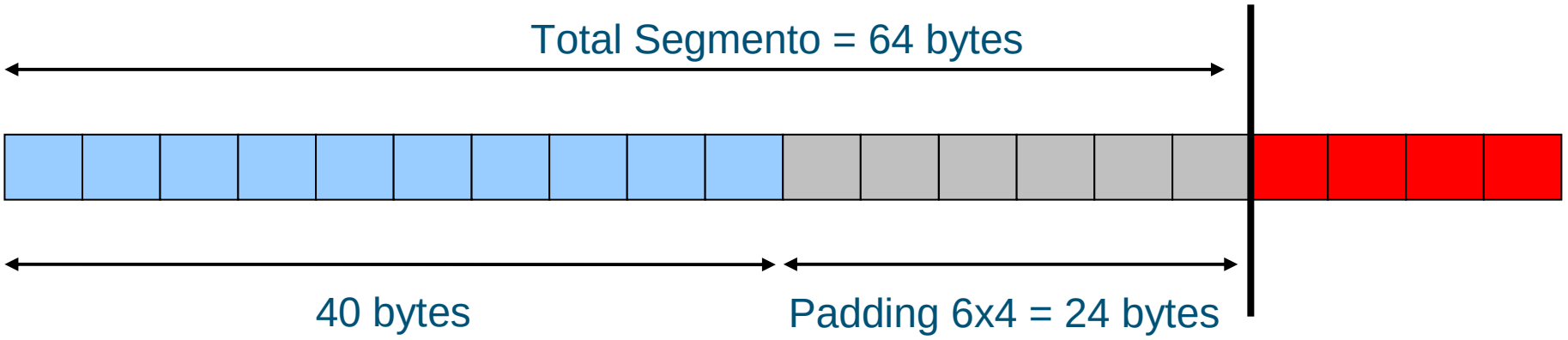
Claves computacionales

Alineación de segmentos



Matriz 2x10 float. Tamaño de fila = 40 bytes ¿ Cuantos accesos se realizan ?

Alineado



Claves computacionales

- **Registros – Shared memory**
  - Uso beneficioso al ser memorias **on-chip** de baja latencia
  - A compartir entre todos los threads del bloque
  - Estamos limitados por sus tamaños
  - Si nos excedemos se hace uso de una memoria especial gestionada sólo por el compilador: **local memory**
  - La local memory está ubicada en la device memory (**off-chip**), por lo que tiene alta latencia
  - **¿ Cómo controlamos el espacio gastado en registros y shared memory ?**
    - **Mediante un flag al compilador –Xptxas=-v**

Claves computacionales

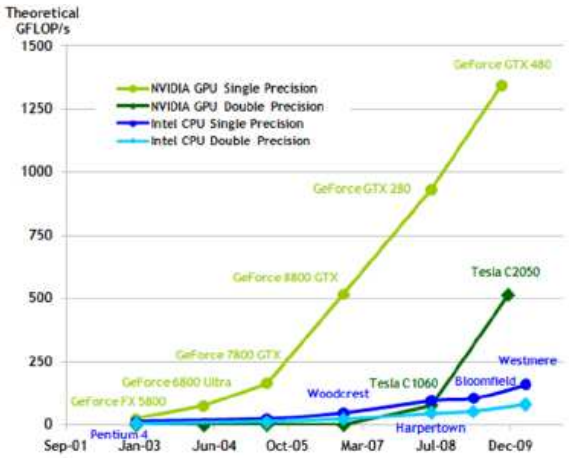
- **Uso de cachés: constant cache y texture cache**
  - Uso beneficioso en **cache hit**
  - Ante un **cache miss**, la latencia es la misma que la del acceso a memoria global
  - Válidas cuando el problema presente cierta **reutilización** de datos
  - Limitaciones
    - **Tamaño reducido**
    - **Sólo lectura**
    - **Reserva estática en la caché de constantes**
    - **La misma palabra de la caché de constantes ha de ser accedida por todos los threads del half-warp**
  - En la práctica, el uso de la caché de texturas produce un aumento considerable el rendimiento del kernel

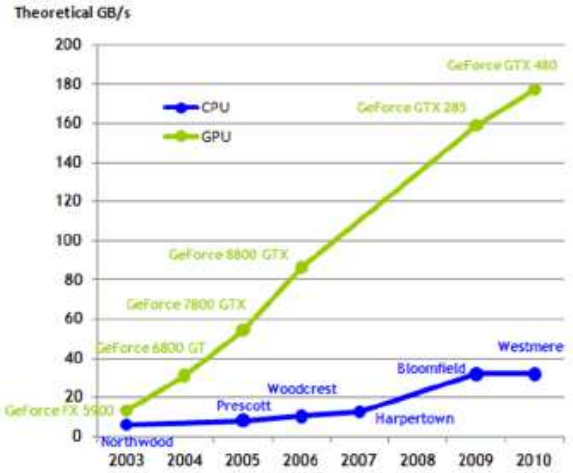
Claves computacionales

- **Cachés en GT300 “Fermi”**
  - Se dispone de dos niveles de caché L1 y L2
    - L1 ligada a **shared memory**
    - L2 ligada a **device memory**
  - Líneas de caché de 128 bytes
  - L1 configurable:
    - Por defecto: 16Kb L1 y 32 Kb shared memory
    - Opcional: 32Kb L1 y 16Kb shared memory (**cudaFuncSetCacheConfig**)
    - Opcional: Sin L1, 48Kb shared memory (flag compilación -dlcm)
  - **L2 de 768Kb siempre activa**
  - Rendimiento*:
    - No usar la caché de texturas, la L2 es automática
    - La mejor configuración es la que se toma por defecto

Claves computacionales

- **Bandwidth**
  - Medida que define “el buen uso” que realizamos de la memoria global
  - Cada GPU tiene un **peak bandwith** que representa el **máximo teórico** que se puede alcanzar





Claves computacionales

- **Bandwidth**
  - La GPU GTX 480 tiene un rendimiento pico de 1.3 TFLOPS
  - ¿ En qué condiciones se puede alcanzar ?

Claves computacionales

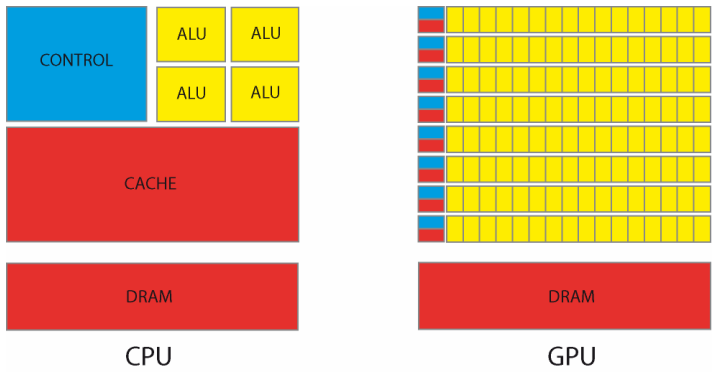
- **Bandwidth**
  - **Bandwidth teórico** para Geforce GTX280:
    - (1107 Mhz x 10^6 x (512/8) x 2)/10^9=141,6 Gbps
    - 512-bit: wide memory interface, DDR: Double Data Rate
  - **Bandwidth efectivo**:
    - ((Br + Bw) / 10^9) / time (secs) (Gbps)
    - Br: Bytes leídos Bw: Bytes escritos por el kernel
  - Para su cálculo, a veces, no se tiene en cuenta el acceso a cachés
  - Un valor bajo indica que el punto de partida en la optimización del kernel es revisar el cómo se realiza el acceso a memoria

Claves computacionales

- **Bandwidth**
  - **Ejemplo**: Kernel con un acceso aleatorio a uno de sus vectores, este se encuentra en caché de texturas
    - Un **valor alto** de Bandwidth puede indicar que hay bastante reutilización y un patrón de acceso a memoria coalescente
    - Un **valor menor**, pero suficientemente alto indica la poca reutilización pero mantiene el acceso coalescente
    - Un **valor muy bajo** indica que no se consigue coalescencia

Claves computacionales

- **Divergencia**
  - Ocurre cuando se producen saltos condicionales en el código
    - Uso de sentencias **if condicion then … else …**
    - Serialización hasta que se acaba el **if**
    - Se pierde paralelismo al estar muchos threads parados
    - Las GPUs no están diseñadas para sentencias de control



Claves computacionales

- **Transferencias CPU ← → GPU**
  - Comunicación por medio del bus PCI Express 16x, 8x
  - Alta latencia en comparación con el acceso a la memoria del dispositivo: GDDR4, GDDR5
  - Peak Bandwidth = 8 GBPS con PCIX 16x Gen2
  - Optimizar las transferencias:
    - Comunicación CPU → GPU
    - Procesamiento
    - Comunicación GPU → CPU
  - En ciertos problemas, los datos a procesar pueden generarse directamente en la GPU sin necesidad de la comunicación inicial

Claves computacionales

- **Transferencias CPU ← → GPU**
  - Otras optimizaciones:
    - **Ejecución concurrente de kernels**(Capacidad de cómputo 1.3 y 2.0)
      - Permite solapar la ejecución de un kernel con la transferencia CPU → GPU de otro kernel
      - Mediante “pinned memory”
    - **Acceso directo a memoria de la CPU** (Fermi)
      - Permite mapear un espacio de direcciones de la memoria de la CPU en la GPU

Claves computacionales

- **Ocupación**
  - Mide el grado en el que el kernel que se ejecuta mantiene ocupados a los SMs
  - Ratio entre los **warps activos** por SM y el máximo posible
  - La ejecución de un warp mientras otro está pausado es la única forma de **ocultar latencias** y **mantener el hardware ocupado**
  - **Situaciones:**
    - **Divergencia**
    - **Accesos a memoria**
  - Una alta ocupación no implica siempre un alto rendimiento, pero es condición indispensable
  - Una baja ocupación imposibilita la ocultación de latencias, por lo que resulta en una reducción del rendimiento
  - Se calcula con: Cuda Occupancy Calculator (xls), Cuda Visual Profiler

Claves computacionales

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)		
1.) Select Compute Capability (click):	1,3	(Help)
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	16	
Shared Memory Per Block (bytes)	1024	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1024	(Help)
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	1,3
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2
Maximum Thread Block Size	512

Allocation Per Thread Block	
Warps	8
Registers	4096
Shared Memory	1024

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	4
Limited by Registers per Multiprocessor	4
Limited by Shared Memory per Multiprocessor	16

Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator	
Version:	2,3
Copyright and License	

Se indica:

Threads por bloque

Registros usados por thread

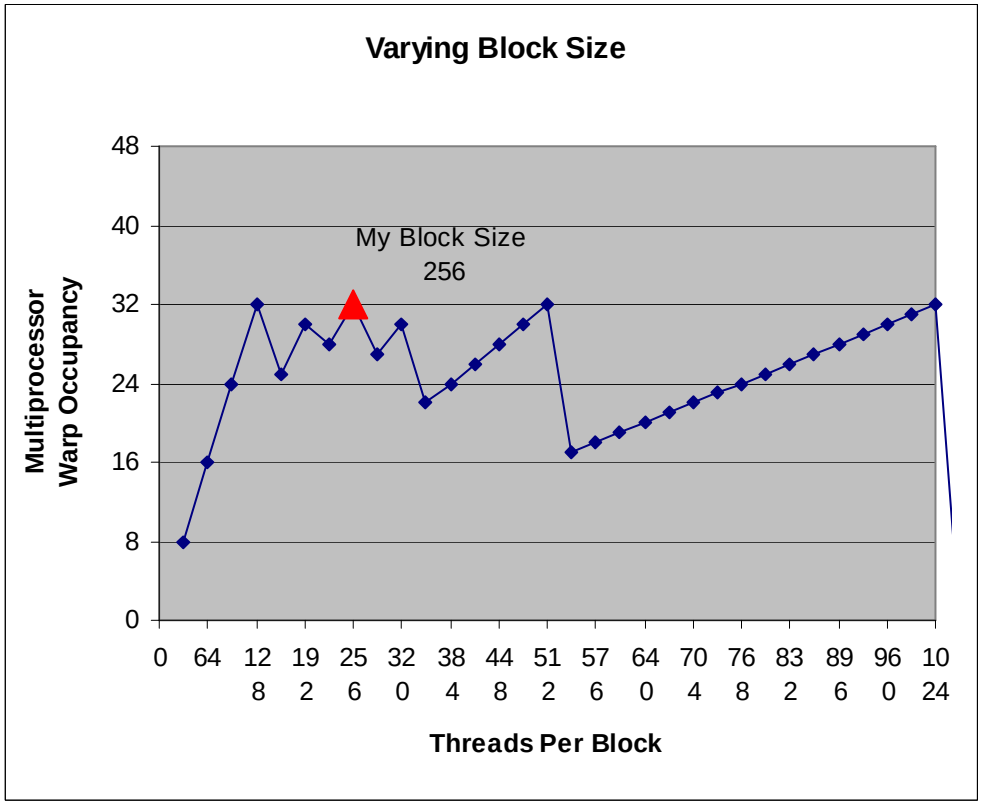
Shared memory por bloque

Se obtiene:

Ratio de ocupación = 100%

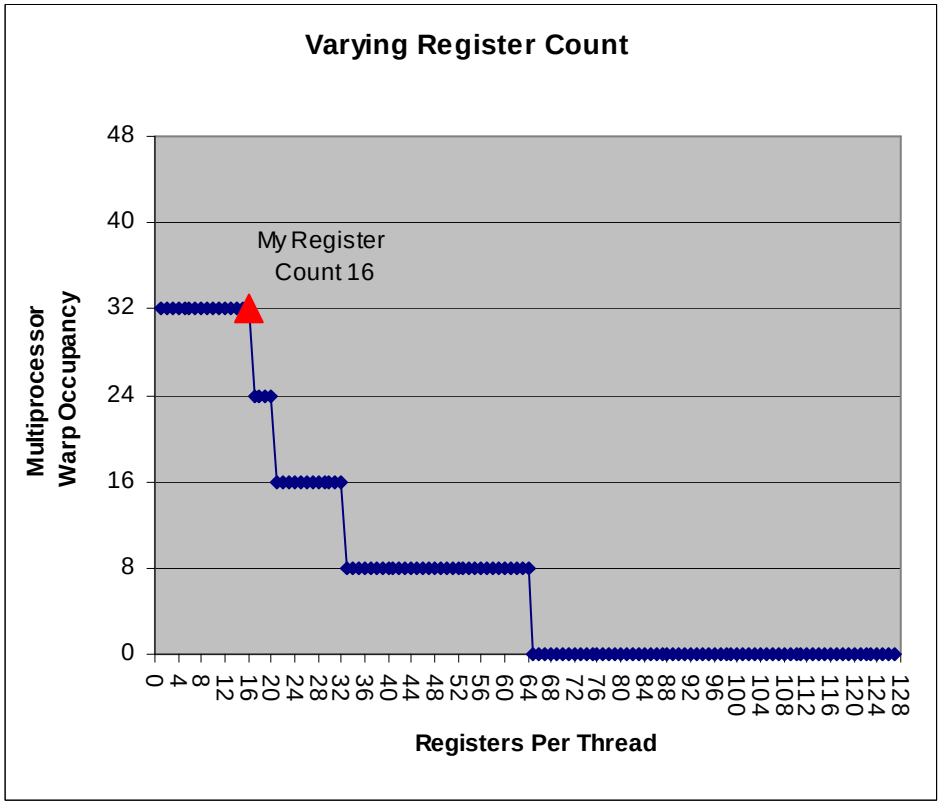
¿ Cuál es el número máximo de warps activos por SM para compute capability 1.3 ?

Claves computacionales

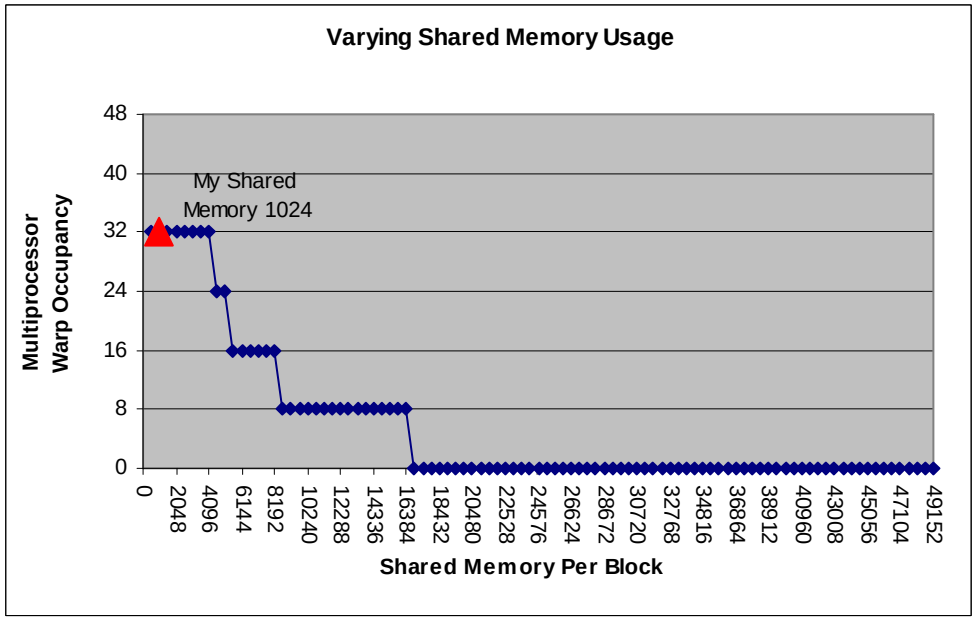


Máxima ocupación
32 Warps x SM

Claves computacionales



Claves computacionales



Claves computacionales

- **Reglas para determinar el tamaño de bloque**
  - Número de threads múltiplo del tamaño del warp (32):
    - Facilita coalescencia. ¿ Porqué ?
    - Evitar warps incompletos
  - Si existen suficientes bloques por SM comenzar con un tamaño mayor o igual a 64 threads por bloque
  - Entre 128 y 256 threads por bloque es un buena elección
  - Comprobar experimentalmente con 512, 768, 1024

Claves computacionales

- **Cuda Visual Profiler**
  - Herramienta para medir parámetros de rendimiento de un kernel
    - # Accesos coalescentes
    - # Saltos condicionales, Divergencia
    - Bandwidth
  - Obtiene valores de la estadística sobre varios SMs, no sobre el total
  - Extrapola los resultados al total de SMs. Es una aproximación
  - Bandwidth incluye todas las estructuras de memoria, no sólo las relevantes para el algoritmo

Claves computacionales

- **Optimización en operaciones**
  - **+**, ***, add-multiply: 8 operaciones/ciclo (Fermi 32 op/ciclo)
  - **/**: 0.88 operaciones/ciclo
  - Uso de operaciones **fast_math**: Comprobar precisión requerida

Operator/Function	Device Function
x/y	__fdividef(x,y)
sinf(x)	__sinf(x)
cosf(x)	__cosf(x)
tanf(x)	__tanf(x)
sincosf(x,sptr,cptr)	__sincosf(x,sptr,cptr)
logf(x)	__logf(x)
log2f(x)	__log2f(x)
log10f(x)	__log10f(x)
expf(x)	__expf(x)
exp10f(x)	__exp10f(x)
powf(x,y)	__powf(x,y)

Claves computacionales

- Precisión en resultados: Cumplimiento del estándar IEEE 754 - 2008
  - Capacidades de cómputo < 2.0. Limitaciones:
    - No detecta una excepción tras una operación en coma flotante
    - Resultado NaN si algún operando es NaN
    - Desbordamientos se transforman a 0
    - Otras implementaciones más lentas si lo cumplen: __fmad_r, __fdiv_r, __fadd_r,…
  - Capacidad de cómputo 2.0:
    - Lo cumple enteramente en operaciones en coma flotante de 32 y 64 bits
    - Memoria ECC para la corrección de errores

Contenidos

- Arquitectura de las GPUs
- **Modelo de programación SIMT**
- **Claves computacionales para la programación de GPUs**
- **Programación con CUDA**
- Supercomputación gráfica y arquitecturas emergentes

Programación con CUDA

- **C**omputed **U**nified **D**evice **A**rchitecture
- NVIDIA, Nov 2006: Arquitectura para computación paralela de propósito general
- Conjunto de extensiones al lenguaje C
- Soporta otros lenguajes de alto nivel: C++, OpenCL, Fortran, DirectX

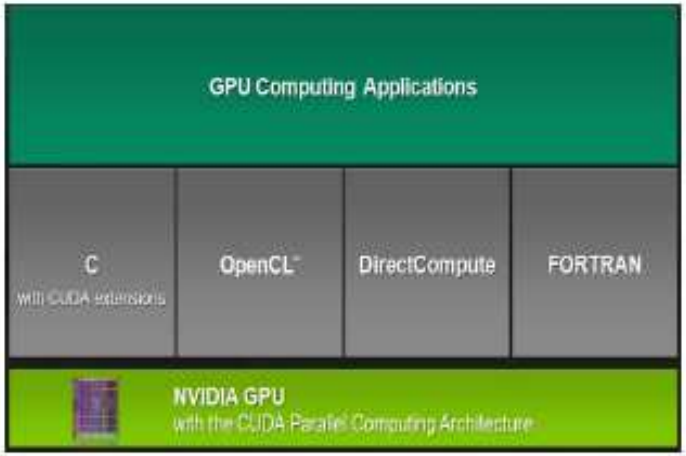
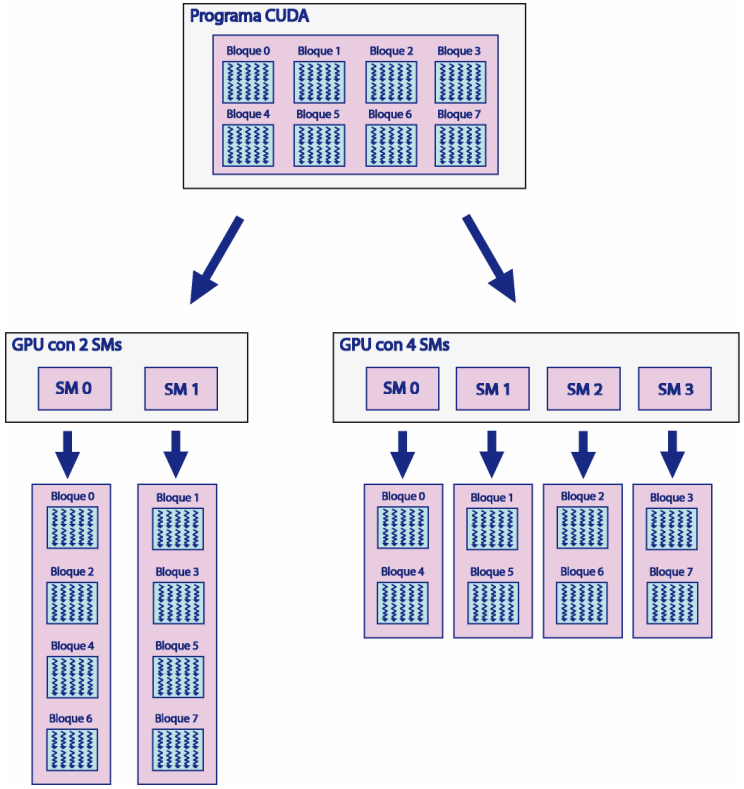


Figure 1-3.CUDA is Designed to Support Various Languages or Application Programming Interfaces

Programación con CUDA

Escalabilidad



Programación con CUDA

- **Identificación de threads, bloques y grid:**
  - **int threadIdx.x, threadIdx.y, threadIdx.z**
    - Identificador (x, y, z) del thread dentro del bloque
  - **int blockIdx.x, blockIdx.y, blockIdx.z**
    - Identificador (x, y, z) del bloque dentro del grid
  - **int blockDim.x, blockDim.y, blockDim.z**
    - Tamaño (x, y, z) del bloque

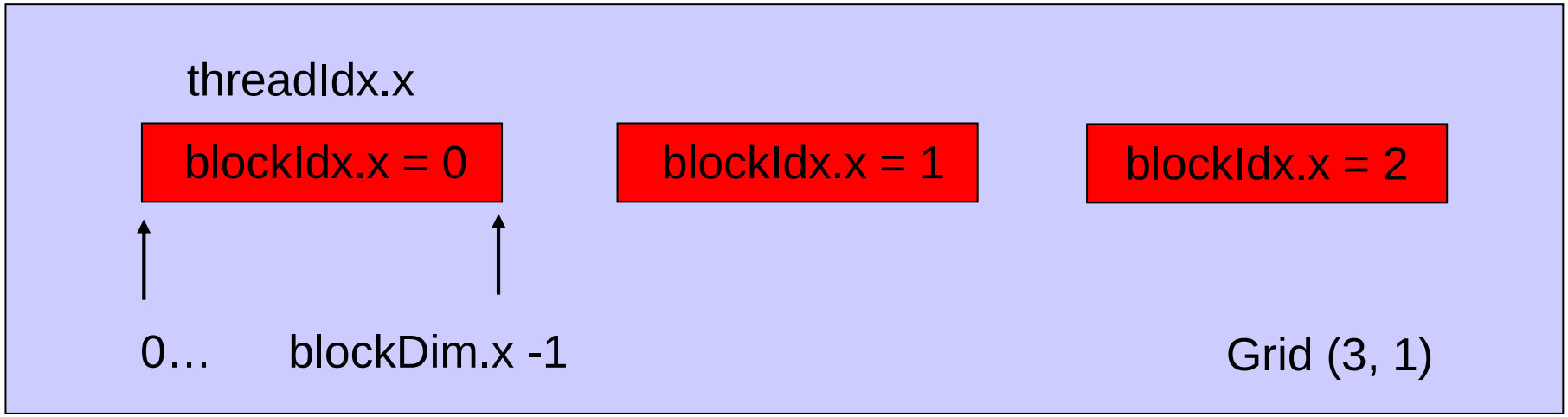
Programación con CUDA

- **Identificador único de thread:**
  - **threadIdx.x**: Número de thread dentro de un bloque 1-D
  - **threadIdx.x + (blockIdx.x * blockDim.x)**: Identificador de thread único entre todos los bloques 1-D (grid)
  - **(threadIdx.x, threadIdx.y):** Coordenadas del thread dentro de un bloque 2-D
  - **(threadIdx.x + (blockIdx.x * blockDim.x), threadIdx.y + (blockIdx.y * blockDim.y))**: Coordenadas del thread dentro del grid (con bloques 2-D)

Programación con CUDA

- **Bloques 1-D**

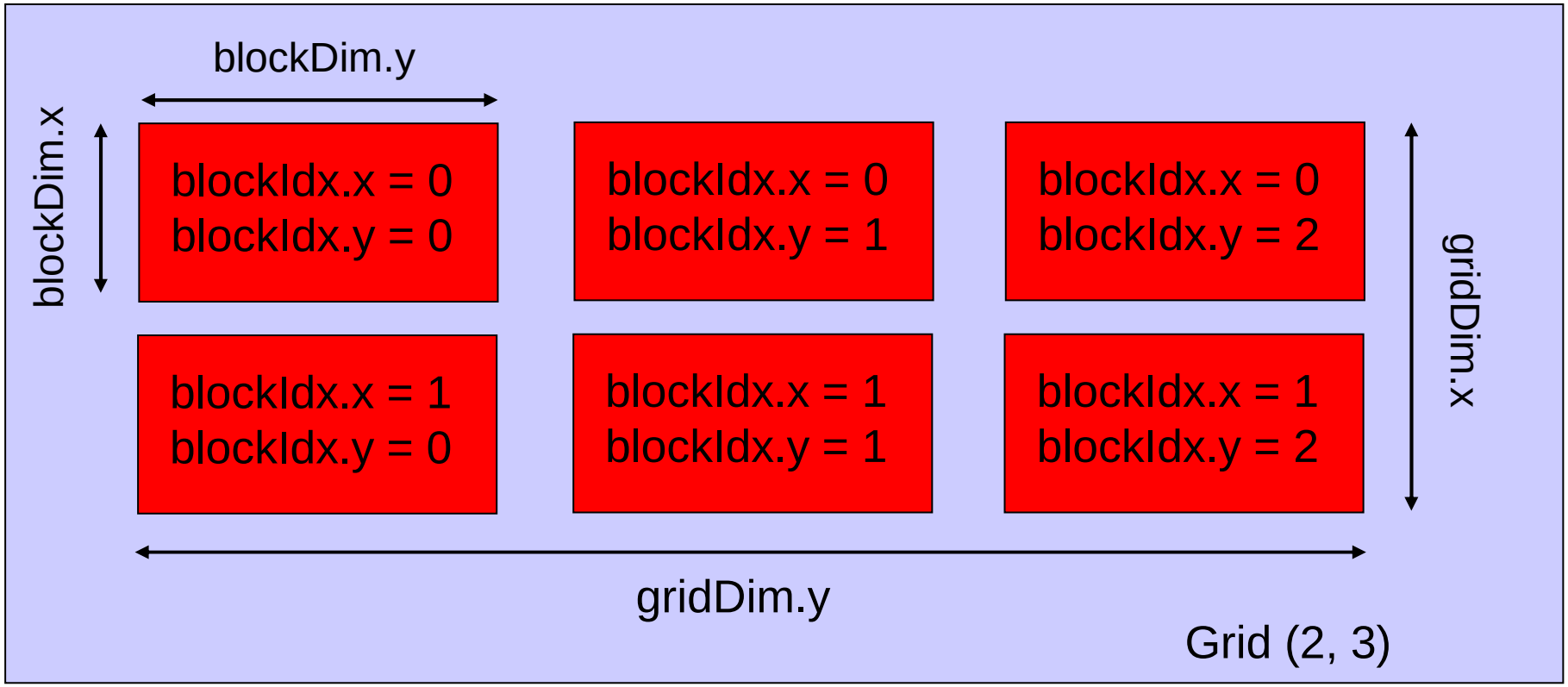
thid = threadIdx.x + (blockIdx.x * blockDim.x)



Si blockDim.x = 10
Los identificadores únicos de los threads, son:
Bloque 0: 0..9
Bloque 1: 10..19
Bloque 2: 20..29

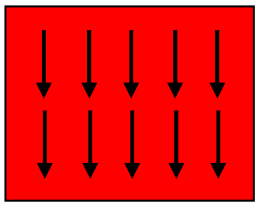
Programación con CUDA

- **Bloques 2-D**



Programación con CUDA

- **Bloques 2-D**



blockDim.x = 2
blockDim.y = 5

(threadIdx.x,threadIdx.y)

(0,0) (0,1) (0,2) (0,3) (0,4)
(1,0) (1,1) (1,2) (1,3) (1,4)

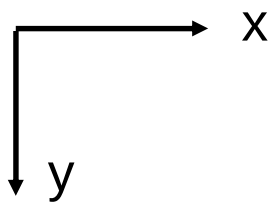
(blockIdx.x, blockIdx.y) = (1,2)

(2,10) (2,11) (2,12) (2,13) (2,14)
(3,10) (3,11) (3,12) (3,13) (3,14)

thidx = threadIdx.x + (blockIdx.x * blockDim.x)
thidy = threadIdx.y + (blockIdx.y * blockDim.y)

Programación con CUDA

- **Identificadores**
  - ¿ Porqué la notación es distinta para el caso de bloques 1-D y 2-D ?
  - 1-D Eje cartesiano
  - 2-D (fila, columna) matriz
    - Normalmente cuando se definen bloques 2-D es para simplificar la referencia a los elementos de una matriz
    - También se podrían haber referenciado mediante coordenadas cartesianas



Programación con CUDA

- **Identificadores**
  - Ejemplo 1-D:

```
dim3 blocksize(10, 1);
dim3 gridsize(3, 1);
```

  - Ejemplo 2-D:

```
dim3 blocksize(2, 5);
dim3 gridsize(2, 3);
```

Programación con CUDA

- **Invocación a kernels**

```
dim3 blocksize(2,5);
dim3 gridsize(2,3);
example_kernel<<<gridsize,blocksize>>>(param1, param2,..)
```

- **Opcionalmente, se puede indicar el tamaño de la memoria shared (si su reserva es dinámica):**

```
size_t sharedmemsize=256;   // bytes
example_kernel<<<gridsize,blocksize,sharedmemsize>>>(params...)
```

Programación con CUDA

- **Invocación a kernels**
  - En el ejemplo anterior sharedmemsize=256 bytes
    - **¿ Cuantos bytes hay disponibles por thread ?**
  - **Código host: *fichero.cu***
  - **Código device: *fichero_kernel.cu***

```
__global__  void example_kernel(params..)
{
.........
}
```

Programación con CUDA

- **__device__**
  - Función que se ejecuta en el device
  - Invocable solamente desde el device
- **__global__**
  - Función que se ejecuta en el device
  - Invocable solamente desde el host
- **__host__**
  - Función que se ejecuta en el host
  - Invocable solamente desde el host
- **Limitación kernels**:
  - Recursividad
  - Número variable de parámetros
  - Declaración de variables estáticas

Programación con CUDA

- Manejo de memoria
  - Normalmente: Reserva lineal y dinámica
  - Ej. Array *N* elementos float
  - CPU: Notación ***h_Nombre***

```
size_t size=N*sizeof(float);
float *h_A=malloc(size);
```

  - GPU: Device memory. Notación ***d_Nombre***

```
size_t size=N*sizeof(float);
float *d_A;
cudaMalloc((void **), &d_A,size);
```

Programación con CUDA

- Manejo de memoria
  - Cualquier memoria reservada en el dispositivo sólo es referenciable desde los kernels
    - Depuración complicada: printf's en host o modo emulación
  - Copia de datos CPU → GPU

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

  - Copia de datos GPU → CPU

cudaMemcpy(h_A, d_A, size, cudaMemcpyDeviceToHost);

Programación con CUDA

- **Ejemplo: Suma de dos vectores**
- Código device

```
// Kernel
__global__ void VecAdd_kernel(float *d_A, float * d_B, float *d_C, int N)
{
        int i = threadIdx.x + (blockIdx.x * blockDim.x);
        if (i < N)
           d_C[i] = d_A[i] + d_B[i];
}
```

- ¿ Porqué comprobar **i < N** ? ¿ Cuantos threads se ejecutan ?
- ¿ Ocurre divergencia ?

Programación con CUDA

Código host

```
int main(){

int N = 1000;
size_t size = N * sizeof(float);
int MAX_BLOCKSIZE = 256;

//….. Asignar valores a los vectores, etc…

// Reserva en device

float *d_A; cudaMalloc((void **) &d_A, size);
float *d_B; cudaMalloc((void **) &d_B, size);
float *d_C; cudaMalloc((void **) &d_C, size);
```

Programación con CUDA

```
//Copiar datos a la GPU

cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_A, size, cudaMemcpyHostToDevice);

//Topología bloques, grid

int BLOCKSX=ceil((float) N/MAX_BLOCKSIZE); //3.90 -> 4
dim3 blocksize(MAX_BLOCKSIZE,1);
dim3 gridsize(BLOCKSX,1);

//Llamada al kernel

VecAdd_kernel<<<gridsize, blocksize>>>(d_A, d_B, d_C, N);
```

Programación con CUDA

```
//Obtener resultados de la GPU

cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

//Liberar memoria GPU

cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);

//Liberar memoria CPU

free(h_A);
free(h_B);
free(h_C);                                    }
```

Programación con CUDA

- **Inicialización de memoria**

```
//Inicializa size bytes de d_A a 0
cudaMemset(d_A, 0, size);

//Inicializa size bytes de d_A a 1
cudaMemset(d_A, 1, size);

//d_A[1]=4294967295.0
```

- **Sincronización**
  - Establece un punto de sincronización (barrier) a un conjunto de threads
  - La sincronización dentro de un warp es implícita
  - Las operaciones cudaMemcpy conllevan una sincronización implícita

Programación con CUDA

- **Sincronización**
  - Host: ***cudaThreadSynchronize();***
  - Kernel: ***__syncthreads();***

```
// Kernel
__global__ void VecAdd_kernel(float *d_A, float * d_B, float *d_C, int N)
{
        int i = threadIdx.x + (blockIdx.x * blockDim.x);
        if (i < N){
            d_C[i] = d_A[i] + d_B[i];
             __syncthreads();
         }
}
```

¿ Qué ocurre aquí ?

Programación con CUDA

- Ejercicio 1: ~ejercicios/cudaMallocAndMemcpy
  - Ejemplo de transferencia host → device, device → device y device → host
  - cudaMallocAndMemcpy.cu
  - nvcc cudaMallocAndMemcpy.cu –o cudamalloc
  - 1. Inicializar h_a a n
  - 2. Declarar d_a, d_b
  - 3. Transferir h_a → d_a
  - 4. Transferir d_a → d_b
  - 5. Inicializar h_a a 0
  - 6. Transferir d_b → h_a
  - 7. Comprobar resultado

Programación con CUDA

- Ejercicio 2: ~ejercicios/myFirstKernel/a
  - Ejemplo de asignación en memoria de un valor
  - myFirstKernel.cu, myFirstKernel_kernel.cu
  - nvcc myFirstKernel.cu –o mykernel
  - 1. Declaración h_a, d_a
  - 2. Definición grid, bloques
  - 3. Ejecución del kernel
  - 4. Sincronización
  - 4. Transferencia d_a → h_a
  - 5. Comprobar resultados
- ¿ De qué tamaño son los bloques ?
- ¿ Qué hace cada thread ?
- ¿ Porqué se realiza una sincronización ?

Programación con CUDA

- Ejercicio 3: ~ejercicios/myFirstKernel/b
  - Ejemplo de funcionamiento en modo emulación
  - myFirstKernelemu.cu, myFirstKernelemu_kernel.cu
  - nvcc –myFirstKernelemu.cu –o mykernel -**deviceemu**

Programación con CUDA

- Shared memory
  - **Reserva estática**: Dentro del kernel

```
__shared__ float d_sdata[256];

//Cada bloque dispone de 256 floats
```

  - **Reserva dinámica**: Desde el host, en la llamada al kernel se indica el tamaño

```
extern __shared__ float d_sdata[];
```

Programación con CUDA

- Texture memory

```
//Host

cudaBindTexture(NULL,d_texture,d_A);
…
cudaUnbindTexture(d_texture);
```

```
//Declaración en el kernel como variable global
texture<float, 1> d_texture;

//Acceso
float value = tex1Dfetch(d_texture, index);
```

Programación con CUDA

Constant memory

```
//Declaración en el kernel como variable global
extern __constant__ int d_cdata[256];

// Copiar desde el host

size_t size=256 * sizeof(int);
size_t offset=0; //Offset en bytes a partir de donde se copia en d_cdata
cudaMemcpyToSymbol(d_cdata, h_A, size, offset,
cudaMemcpyHostToDevice);

//Acceso
int temp = d_cdata[i];
```

Programación con CUDA

- **Medida de tiempos**
  - **GPU timers**
  - Milisegundos
  - Precisión de medio microsegundo

```
cudaEvent_t start, stop;
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);

example_kernel<<<gridsize,blocksize>>>……

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Programación con CUDA

- **Medida de tiempos**
  - **C timers <sys/time.h>**
  - Microsegundos
- En la práctica, ambos métodos son equivalentes
- Necesaria la sincronización de threads antes de poner ***start*** y de medir ***stop***

```
struct timeval start, stop;
double usecs;

cudaThreadSynchronize();
gettimeofday(&start, NULL);

example_kernel<<<gridsize,blocksize>>>…

cudaThreadSynchronize();
gettimeofday(&stop, NULL);

usecs=(stop.tv_sec-start.tv_sec)*1e6+
(stop.tv_usec-start.tv_usec);
```

Programación con CUDA

- **Librería CUTIL <cutil.h>**
  - Incluída en la SDK de desarrollo
  - Macros
    - CUT_INIT_DEVICE
      - Inicializa el primer device que se encuentra
    - CUDA_SAFE_CALL(call)
      - Realiza una llamada a una función cuda y muestra un texto con el tipo de error que se ha producido
    - CUT_EXIT
      - Finaliza la ejecución de un programa forzando a pulsar una tecla
  - Funciones
    - Parsing de los parámetros indicados en la línea de comandos
    - Comparación de arrays para comparar CPU y GPU
    - Timers
    - Conflictos en shared memory

Programación con CUDA

- El toolkit de CUDA incluye:
  - Librería runtime de CUDA: *cudart*
  - Librería FFT: *cufft*
  - Librería BLAS (Algebra lineal): *cublas*
  - Librería CUSPARSE: BLAS para matrices sparse
- Librerías de terceros
  - cuDPP: CUDA Data Parallel Primitives
  - CULA: LAPACK (Sistemas de ecuaciones lineales,..)
  - GPULib: Librerías matemáticas para IDL y MATLAB
  - VSIPL: Procesamiento de señales

Programación con CUDA

- **Compilación**
  - **nvcc**: Compilador de NVIDIA
  - Flags
    - -deviceemu: Modo emulación. Admite printf's en kernels
    - -O2, -O3: Flags optimización gcc, sólo aplicables al código del host
    - -use_fast_math: Usar implementación de funciones rápidas. El compilador las cambia automaticamente
    - -Xptxas=-v. Ver ocupación de registros y shared memory
    - -arch=sm11 sm12 sm13. Compilar para una determinada compute capability
      - Si se compila < sm13, el tipo double se transforma en float dentro del kernel, pero no dentro del host
- **scons**
  - Útil para linkar códigos C++, C, con .cu
  - Necesita script cuda.py en phyton

Programación con CUDA

- Cuda Visual Profiler
  - **Gpu time stamp**: Marca de tiempo para el kernel a ejecutar
  - **Method**: Operación memcpy* o ejecución de un kernel
  - **GPU Time**: Tiempo en GPU
  - **CPU Time**: Tiempo en CPU
  - **occupancy**: Ratio entre warps activos por SM y máximo
  - **gld uncoalesced**: Número de lecturas de memoria no coalescentes
  - **gld coalesced**: Número de lecturas de memoria coalescentes
  - **gld request**: Número total de lecturas en memoria
  - **gst uncoalesced**: Número de escrituras en memoria no coalescentes
  - **gst coalesced**: Número de escrituras en memoria coalescentes
  - **gst request**: Número total de escrituras en memoria
  - **gst_32/64/128b**: Número de transacciones de escritura en memoria de 32, 64 ó 128 bytes

Programación con CUDA

- **tlb hit**: Aciertos en caché de constantes
- **tlb miss**: Fallos en caché de constantes
- **sm cta launched**: Número de bloques ejecutados en un SM
- **branch**: Número de saltos realizados por los threads en la ejecución de un kernel
- **divergent branch**: Número de saltos divergentes dentro de un warp
- **warp serialize**: Número de warps serializados debido a un conflicto de direccionamiento en la memoria shared o caché de constantes.
- **cta launched**: Número de bloques ejecutados
- **grid size X, Y**: Número de bloques en X, Y
- **block size X, Y, Z**: Número de threads por bloque en X, Y, Z
- **dyn sm per block**: Tamaño de memoria shared dinámica por bloque (bytes)
- **sta smem per block**: Tamaño de memoria shared estática por bloque (bytes)
- **reg per thread**: Número de registros por thread

Programación con CUDA

- **Cuda Visual Profiler**
  - Desafortunadamente, no todos los valores están disponibles para todas las GPUs
  - Saber interpretar los valores, ya que se corresponden con la ejecución de varios SMs, no del total
  - Útil en la comparación de un mismo parámetro después de la optimización del código
  - Parámetros resúmen:
    - GPU usec
    - CPU usec
    - % GPU time
    - Bandwidth read, write, overall Gbps
    - Instruction throughput

UNIVERSIDAD DE ALMERÍA