I Curso de Computación Científica en *Clusters*

Febrero/Marzo 2010

Programación de GPUs

Datos generales

- Profesorado
  - Dr. D. Juan Fernández Peinador
    - Despacho: 3.32 (3ª planta de la Facultad de Informática)
    - Tfno.: 868 88**8786**
    - email: juanf@um.es
- Sesiones
  - Miércoles 24 de marzo de 17:30 a 20:30h
  - Jueves 25 de marzo de 18:30 a 20:30h
- Metodología
  - Diapositivas con ejemplos y ejercicios teórico-prácticos
- Material didáctico
  - Presentación, ejemplos, manuales y artículos en web

Datos generales

Recursos

- **NVIDIA Tesla C870***
  - 1536 MB 384-bit GDDR3
  - *Compute Capability* 1.0
  - PCIe x16
- **CUDA 2.3**

*http://www.nvidia.es/page/tesla_gpu_processor.html

Motivación

- Procesadores de propósito general
  - En las últimas décadas los procesadores comerciales han proporcionado un aumento de rendimiento año tras año
  - En los últimos años han surgido algunas dificultades:
    - El consumo de energía limita la frecuencia de reloj
    - El paralelismo de las aplicaciones secuenciales que se puede explotar de manera transparente es limitado
  - Como consecuencia, casi todos los fabricantes de procesadores han optado por arquitecturas multinúcleo
    - Intel Core 2 Duo/Quad/i3/i5/i7, AMD Phenom II X2/3/4, IBM Power 7, Cell Broadband Engine (PS3)
  - Las aplicaciones DEBEN paralelizarse para aprovechar todo el potencial de las arquitecturas subyacentes

Motivación

- *Graphics Processing Units* (GPUs)
  - Las GPUs liberan a la CPU de realizar tareas concretas de procesamiento gráfico de manera repetitiva
  - Presentes en cualquier equipo de sobremesa o servidor integradas en placa o como tarjetas externas
    - ISA →VESA | PCI → AGP 1/2/4/8x → PCIe x1/4/8/16
  - El amplio mercado de los vídeojuegos ha propiciado su consolidación, rápida evolución y precios competitivos
  - Las GPUs actuales también son procesadores multinúcleo porque el procesamiento gráfico es inherentemente paralelo
    - La necesidad de ejecutar múltiples operaciones en punto flotante para procesar cada imagen…
    - …se satisface mediante muchos *threads* capaces de ejecutarse en paralelo

Motivación

- CUESTIONES CLAVE:
  - ¿Qué diferencias hay entre un procesador multinúcleo de propósito general y una GPU?
  - ¿Qué ofrecen las GPUs que las hace atractivas para pensar que podemos aprovecharlas para realizar otras tareas?
  - ¿Qué características de las GPUs condicionan su utilización para realizar esas tareas?

Motivación

- GPUs ofrecen mayor rendimiento pico que las CPUs
  - CPUs diseñadas para optimizar la ejecución de aplicaciones de propósito general
    - Lógica de control flujo muy sofisticada
    - Memorias caché multinivel
    - # núcleos: 2 a 4 (Intel y AMD)
  - GPUs diseñadas para optimizar la ejecución de tareas de procesamiento gráfico
    - Lógica de control de flujo simple y memorias caché pequeñas
    - Múltiples unidades funcionales para punto flotante
    - Mayor ancho de banda de acceso a memoria
    - # núcleos: 8 hasta 240 (NVIDIA)
  - Por ejemplo, Intel Core i7 (55,4 Gflops -¿SP?-/25,6 GB/s) vs.
    NVIDIA Geforce GTX 280 (933 Gflops -SP-/141,7 GB/s)

Motivación

- Esta *ventaja* ha despertado el interés por explorar el uso de GPUs para acelerar aplicaciones de propósito general
  - *General-Purpose computation on Graphics Processing Units* (GPGPU)
    - Programación mediante APIs gráficas
      - Direct3D
      - OpenGL
    - Modificación de la aplicación para expresarla en función de un conjunto de llamadas a la API gráfica disponible
      - Tarea ardua y compleja que requiere un conocimiento detallado tanto de la arquitectura de la GPU como de la aplicación
      - La API limita las aplicaciones que pueden adaptarse
    - **CUDA** proporciona un modelo de programación independiente de las APIs gráficas mucho más general y flexible
      - Las aplicaciones también DEBEN paralelizarse

Motivación

- GPUs ofrecen mayor rendimiento efectivo que las CPUs[1,2] en aplicaciones basadas en CUDA de diversos campos:
  - Multiplicación de matrices densas: *speedup* de 9.3x
  - Procesamiento de vídeo (H.264): *speedup* de 12.23x
  - Cálculo de potencial eléctrico: *speedup* de 64x
  - Resolución de ecuaciones polinomiales: *speedup* de 205x
  - …
- Desafortunadamente no todas las aplicaciones son susceptibles de ser paralelizadas con éxito en una GPU

[1]Wen-mei Hwu *et al.*, Compute Unified Device Architecture Application Suitability, Computing in Science and Engineering, vol. 11 no. 3, pp. 16–26, May/June 2009

[2]Michael Garland *et al.*, Parallel Computing Experiences with CUDA, IEEE Micro, vol. 28 no. 4, pp. 13–27, July 2008

Objetivos

- Conocer la tecnología básica hardware y software que permite la paralelización de aplicaciones con CUDA
- Adquirir la habilidad de paralelizar aplicaciones sencillas con CUDA en un entorno real
- Ser capaces de determinar si una aplicación de propósito general ofrece alguna posibilidad de paralelización con éxito en CUDA a partir de su especificación algorítmica

Contenidos

- Motivación
- Objetivos
- Introducción
- Arquitectura y programación de CUDA
- Optimización y depuración de código
- Librerías basadas en CUDA
- Alternativas a NVIDIA/CUDA
- Conclusiones
- Bibliografía

Introducción

- En NOV'06 NVIDIA introduce la arquitectura unificada (CUDA) con la presentación de la NVIDIA GeForce 8800
- GPUs de NVIDIA compatibles con CUDA*
  - Modelos:
    - GeForce Series 8, 9, 100 y 200 con > 256 MB
    - **Tesla C870**, D870, S870, C1060 y S1070
    - Quadro FX
  - Diferencias:
    - Interfaz (ancho de banda) y cantidad de memoria integrada
    - # núcleos (# SMs)
    - # registros (8192 ó 16384)
    - *Compute Capability* (1.x)

*http://www.nvidia.es/object/cuda_gpus_es.html

Introducción

- *Compute Unified Device Architecture* (CUDA)*
  - Arquitectura hardware y software
    - Uso de GPU, construida a partir de la replicación de un bloque constructivo básico, como acelerador con memoria integrada
    - Estructura jerárquica de *threads* mapeada sobre el hardware
  - Modelo de memoria
    - Gestión de memoria explícita
  - Modelo de ejecución
    - Creación, planificación y ejecución transparente de miles de *threads* de manera concurrente
  - Modelo de programación
    - Extensiones del lenguaje C/C++ junto con CUDA *Runtime* API

*Erik Lindholm et al., NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE Micro, vol. 28 no. 2, pp. 39–55, March 2008

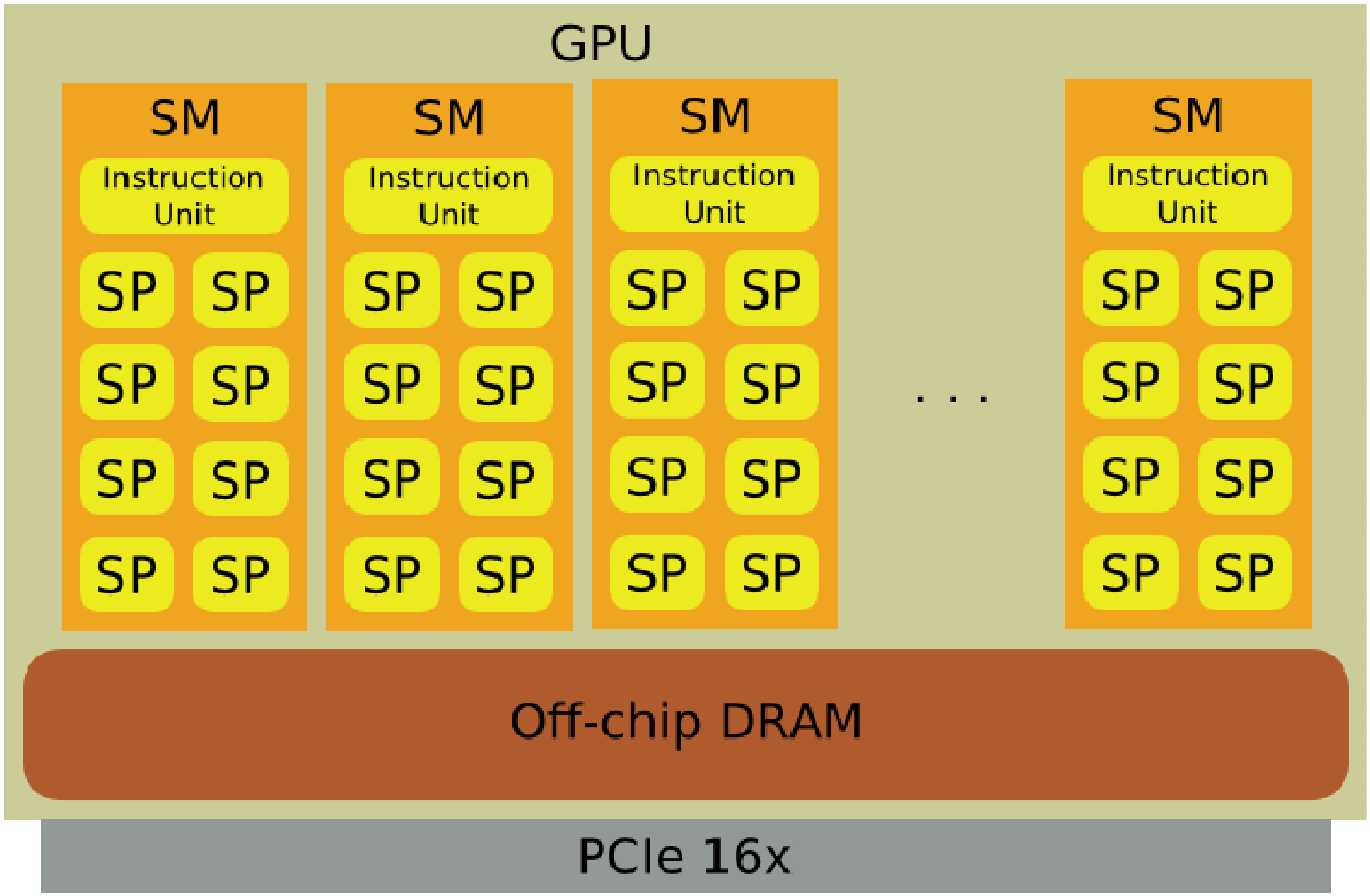
Contenidos

- Motivación
- Objetivos
- Introducción
- Arquitectura y programación de CUDA
- Optimización y depuración de código
- Librerías basadas en CUDA
- Alternativas a NVIDIA/CUDA
- Conclusiones
- Bibliografía

Arquitectura hardware y software

- GPU = vector de N x *Streaming Multiprocessors* (SMs)
  - SM = 8 x *Streaming Processors* (SPs)
    - Realizan operaciones escalares sobre enteros/reales 32 bits (compatibles con IEEE 754)
    - Ejecutan *threads* independientes pero…
    - …todos deberían ejecutar la instrucción leída por la *Instruction Unit* (IU) en cada instante para optimizar el rendimiento
      - *Single Instruction Multiple Thread* (SIMT)
        - Explotación de paralelismo de datos y, en menor medida, de tareas
- Los *threads* son gestionados por el hardware en cada SM
  - Creación y cambios de contexto con coste despreciable
  - Se libera al programador de realizar estas tareas
  - Ejecución de tantos *threads* como sea posible

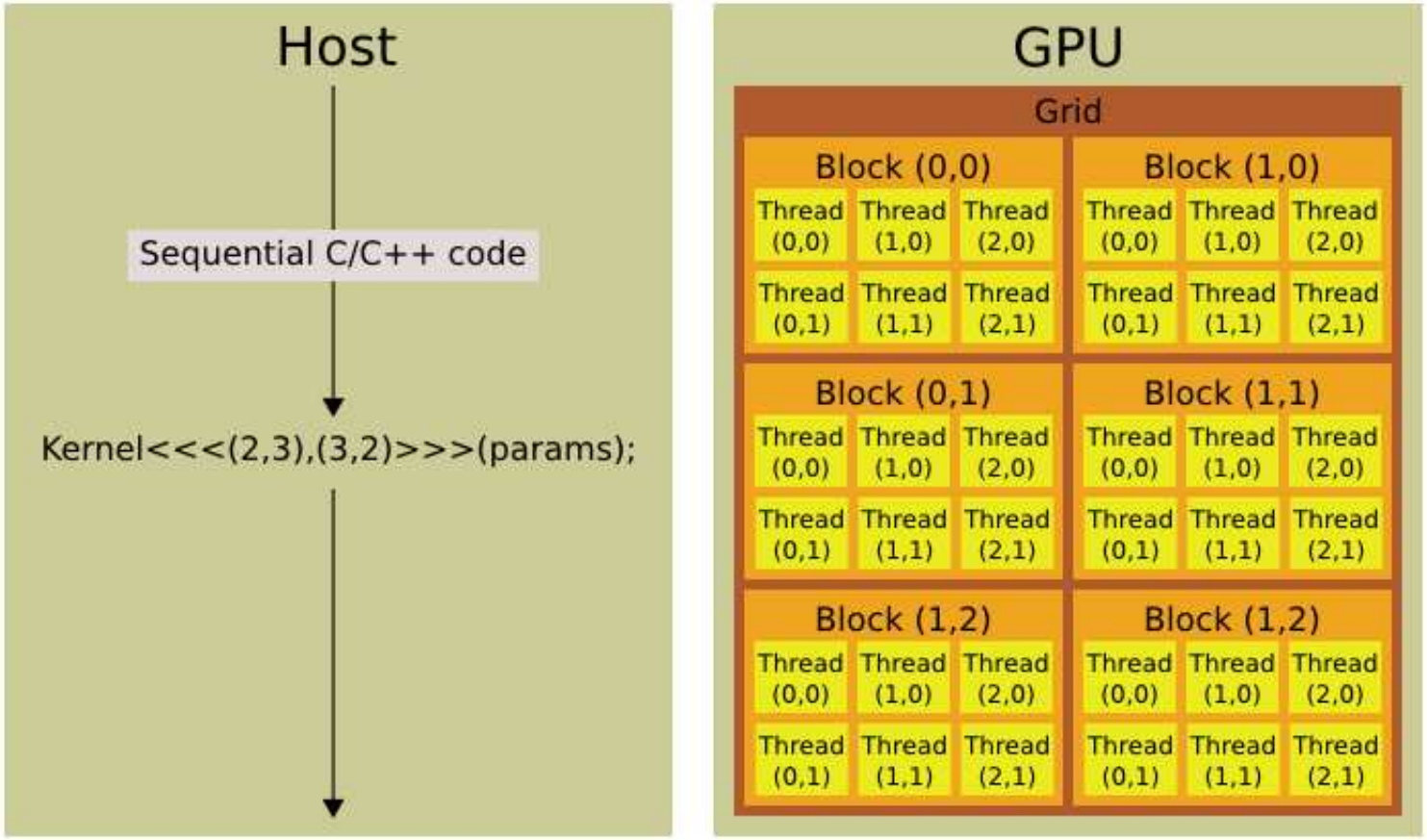
Arquitectura hardware y software



Arquitectura hardware y software

- Las partes del código secuencial paralelizadas con objeto de ser ejecutadas por la GPU se denominan ***kernels***
- Un *kernel* descompone un problema en un conjunto de subproblemas independientes y lo mapea sobre un *grid*
  - ***Grid***: vector 1D ó 2D de *thread blocks*
    - Cada *thread block* tiene su **BID (X,Y)** dentro del *grid*
  - ***Thread blocks***: vector 1D, 2D ó 3D de *threads*
    - Cada *thread* tiene su **TID (X,Y,Z)** dentro de su *thread block*
- Los *threads* utilizan su BID y su TID para determinar el trabajo que tienen que llevar a cabo como en OpenMP
  - *Single Program Multiple Data* (SPMD)

Arquitectura hardware y software



Arquitectura hardware y software

- Ejemplo: cálculo de $y=\alpha \cdot x+y$ donde x e y son vectores

```
void saxpy_serial(int n, float *y, float alpha, float *x)
{
   for(int i=0; i<n; i++)
    y[i] = alpha*x[i] + y[i];
}
/* Llamada código secuencial */
saxpy_serial(n, 2.0, x, y);

__global__ /* Código GPU */
void saxpy_parallel(int n, float *y, float alpha, float *x)
{
   int i= blockIdx.x * blockDim.x + threadIdx.x;
   if (i < n) y[i] = alpha*x[i] + y[i];
}
/* Llamada código paralelo desde código CPU */
int nblocks = (n + 255)/256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

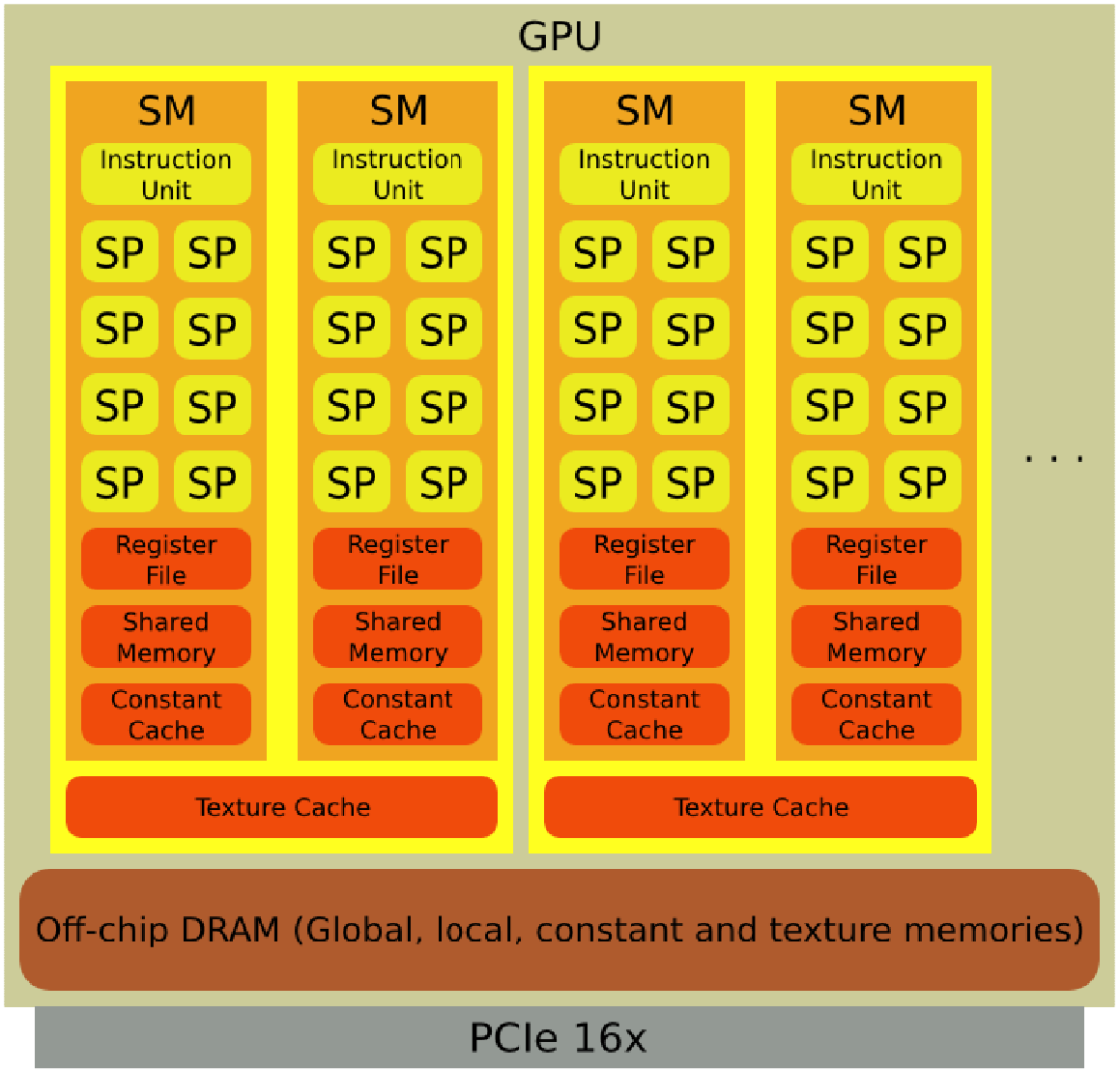
Modelo de memoria

- Banco de registros (*register file*): 8192 registros
  - Repartidos entre todos los *thread blocks* en ejecución
  - Tiempo de acceso muy pequeño
- Memoria compartida (*shared memory*): 16 KB de memoria
  - Repartida entre todos los *thread blocks* en ejecución
  - Compartida por todos los *threads* de cada *thread block*
    - Almacenamiento de datos temporal a modo de cache
  - Tiempo de acceso similar a los registros
- Memoria global (*global memory*): hasta 1,5 GB de memoria compartida por todos los *thread blocks*
  - Tiempo de acceso elevado (cientos de ciclos)

Modelo de memoria

- Memoria constante (*constant memory*): 64 KB de memoria con 8 KB de memoria cache
  - Todos los *threads* de un *warp* pueden leer el mismo valor de la memoria constante simultáneamente en un ciclo de reloj
  - Tiempo de acceso similar a los registros
  - Sólo admite operaciones de lectura
- Memoria de texturas (*texture memory*):
  - Explota localidad espacial con vectores de datos 1D ó 2D
  - Tiempo de acceso elevado pero menor que memoria global
  - Sólo admite operaciones de lectura
- Memoria local (*local memory*)
  - Memoria privada de cada *thread* para la pila y las variables locales con propiedades similares a la memoria global

Modelo de memoria



Modelo de ejecución

- Cada *thread block* de un *grid* se asigna a un único SM
- Un SM asigna a cada *thread block* en ejecución (activo) todos los recursos necesarios
  - *Thread contexts*, registros, *shm*, etc.
- Cada SM puede gestionar y ejecutar hasta 768 *threads*
  - 6 *thread blocks* de 128 *threads*, 4 *thread blocks* de 192 *threads* , 3 *thread blocks* de 256 *threads*, etc.
- Comunicación de todos los *threads* de un *thread block* mediante accesos a la memoria compartida
- Sincronización de todos los *threads* de un *thread block* mediante una única instrucción
  - `_syncthreads();`

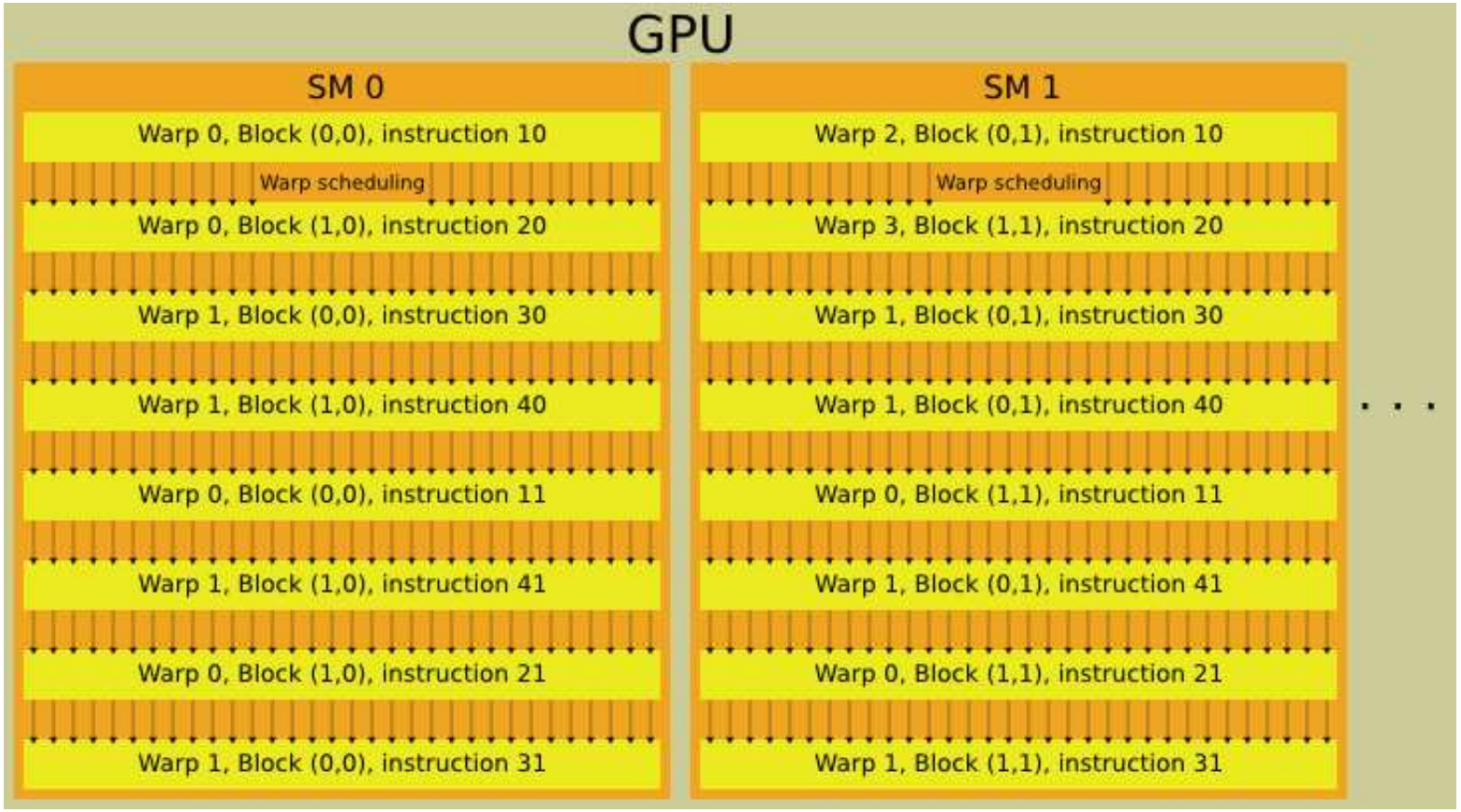
Modelo de ejecución

- Los *threads* de distintos *thread blocks* sólo se pueden comunicar vía memoria global y no se pueden sincronizar
  - La sincronización se produce de manera implícita entre la ejecución de un *kernel* y el siguiente
  - Los *thread blocks* de un *grid* deben ser independientes
  - Los resultados deberían ser correctos sin importar el orden de ejecución de los *thread blocks* del *grid*
  - Esta restricción reduce la complejidad del hardware y, sobre todo, favorece la escalabilidad pero…
  - …limita el rango de aplicaciones que pueden paralelizarse con éxito para ser ejecutadas en una GPU utilizando CUDA
- Cada *thread block* se divide en grupos de 32 threads denominados *warps*

Modelo de ejecución

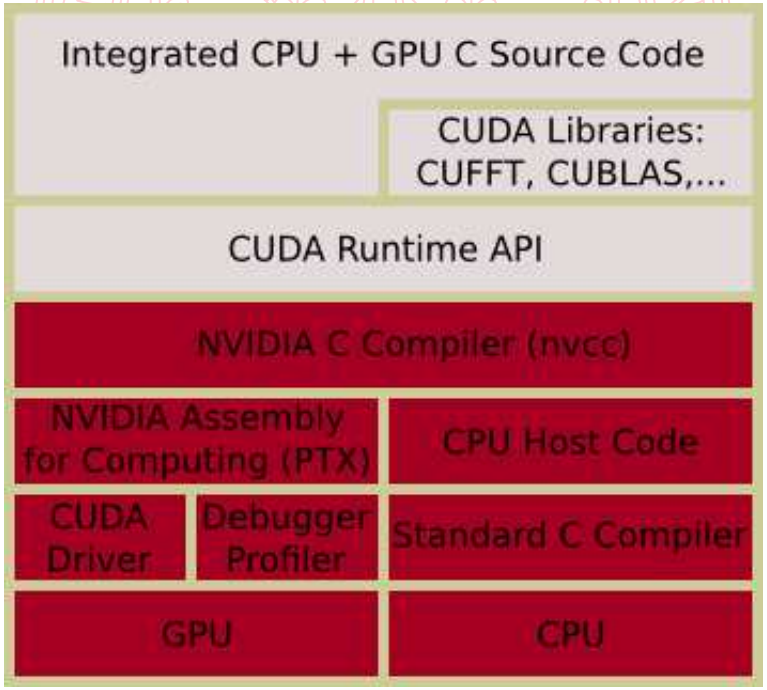
- Cada SM crea, planifica y ejecuta hasta 24 *warps* pertenecientes a uno o más *thread blocks* (768 *threads*)
- Cada *warp* ejecuta una instrucción en 4 ciclos de reloj
- Cuando un *warp* se bloquea, el SM ejecuta otro *warp* perteneciente al mismo o a otro *thread block* activo
  - Ocultación de las largas latencias de acceso a memoria
- Si los *threads* de un *warp* divergen (salto condicional), su ejecución se serializa de forma que en cada rama…
  - Todos los *threads* (SPs) ejecutan la instrucción leída por la IU inhabilitando los *threads* que siguieron otra rama distinta
  - Con degradación de rendimiento en muchos casos
- …hasta que todos convergen

Modelo de ejecución



Modelo de programación

- Código fuente integrado para CPU/GPU
  - Extensiones del lenguaje C/C++ (sólo C para GPU)
  - CUDA *Runtime* API (librería de funciones)
- NVIDIA C Compiler (nvcc) separa el código de CPU/GPU
  - Compilador convencional para el código de la CPU



Ejemplo 0

- Conectar a la máquina tesla.inf.um.es (SSH)
  - Con usuario: ccccXX y clave: ecdcXX.10
  - Donde XX Є [01..15]
  - Mediante **ssh ccccXX@tesla.inf.um.es**
  - ccccXX@tesla:~$ ...
- Cambiar clave:
  - ccccXX@tesla:~$ **passwd**
- Descargar ejemplos:
  - ccccXX@tesla:~$ **wget --http-user=sm --http-password=cuda http://ditec.um.es/~peinador/cccc/cuda-recursos.tgz**
- Extraer ficheros de ejemplo:
  - ccccXX@tesla:~$ **tar xzvf cuda-recursos.tgz**

Ejemplo 0

- Compilar ejemplo:
  - ccccXX@tesla:~$ **cat cuda.env**

```
export PATH=$PATH:/usr/local/cuda/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64
```

  - ccccXX@tesla:~$ **source cuda.env**
  - ccccXX@tesla:~$ **cd cuda_deviceQuery**
  - ccccXX@tesla:~$ **make**
- Ejecutar ejemplo:
  - ccccXX@tesla:~$ **./cuda_deviceQuery**
- Editar ejemplo:
  - ccccXX@tesla:~$ **vim|joe cuda_deviceQuery.cu**
    - Llamadas a CUDA *Runtime* API

Ejemplo 0

```
CUDA Device Query (Runtime API) version (CUDART static linking)
There is 1 device supporting CUDA

Device 0: "Tesla C870"
  CUDA Driver Version:                           2.30
  CUDA Runtime Version:                          2.30
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         0
  Total amount of global memory:                 1610350592 bytes
  Number of multiprocessors:                     16
  Number of cores:                               128
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.35 GHz
  Concurrent copy and execution:                 No
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       No
  Compute mode: Default (multiple host threads can use this device simultaneously)
```

Modelo de programación

- Extensiones del lenguaje C/C++
  - Declaración de funciones
    - `__device__`
      - Función ejecutada por GPU y llamada desde *kernels*
    - `__global__`
      - Función ejecutada por GPU y llamada desde código secuencial
    - `__host__`
      - Función ejecutada por CPU y llamada desde código secuencial
      - Por defecto si no se especifican los anteriores

Modelo de programación

Extensiones del lenguaje C/C++

- Declaración de variables
  - `__device__`
    - Variable residente en memoria global accesible por todos los *threads* de cualquier *grid* durante el tiempo de vida de aplicación
  - `__constant__`
    - Variable que reside en memoria constante accesible por todos *threads* de cualquier *grid* durante el tiempo de vida de aplicación
  - `__shared__`
    - Zona de memoria compartida accesible por todos los *threads* del mismo *thread block* durante la ejecución del *grid*

```
extern __shared__ char shared_mem[256];
__device__ void kernel()
{
  int   *array0 = (int *) shared_mem;    // int   array0[32]
  float *array1 = (float *) &array0[32]; // float array1[32]
}
```

Modelo de programación

- Extensiones del lenguaje C/C++
  - Tipos de datos vectoriales
    - `[u]char1|2|3|4`, `[u]short1|2|3|4`, `[u]int1|2|3|4`, `[u]long1|2|3|4`, `longlong1|2`, `float1|2|3|4`, `double1|2`
    - Creación: `tipo variable(int x, int y, int z, int w);`
      - `int2 var_int2 = make_int2(1,2);`
    - Acceso y modificación: `variable.x|y`
      - `var_int2.x = 1; var_int2.y = 2`
    - Las variables vectoriales deben estar alineadas al tamaño de su tipo base
      - La dirección de `int2 var_int2` debe ser múltiplo de 8
    - `dim3` equivale a `uint3` y se usa para especificar las dimensiones de *grids* y *thread blocks*

Modelo de programación

- Extensiones del lenguaje C/C++
  - Variables predefinidas
    - `gridDim`: dimensiones del *grid*
    - `blockIdx`: índice del *thread block* en el *grid* (BID)
    - `blockDim`: dimensiones del *thread block*
    - `threadIdx`: índice del *thread* en el *thread block* (TID)
    - `int warpSize`: # de *threads* en un *warp*

Modelo de programación

- Extensiones del lenguaje C/C++
  - *Intrinsics*
    - `__syncthreads()`
      - Sincronización de todos los *threads* de un *thread block*
    - `__threadfence_block()`
      - El *thread* se bloquea hasta que todos sus accesos a memoria compartida previos a la llamada sean visibles a los demás *threads* del *thread block*
    - `__threadfence()`
      - Similar a `__threadfence_block()` pero además el *thread* también espera hasta que todos los accesos a memoria global previos a la llamada sean visibles a los restantes *threads* del dispositivo

Modelo de programación

- Extensiones del lenguaje C/C++
  - Ejecución de *kernels*

```
// kernel definition
__global__ void foo(int n, float *a)
{
  ...
}
int main()
{
  dim3 dimB(8,8,4);
  dim3 dimG(4,4);
  // kernel invocation
  foo<<<dimG,dimB[,shared_mem_size]>>>(n, a);
}
```

Modelo de programación

- CUDA *Runtime* API (funciones con prefijo `cuda`) definida en el fichero `cuda_runtime_api.h`
  - Consulta de versiones de *Runtime* y *Driver*
  - Manejo de dispositivos, *threads* y errores
  - Creación y uso de flujos (*streams*) y eventos
  - Gestión de memoria
  - Manejo de texturas (CUDA *Arrays*)
  - Interacción con OpenGL y Direct3D

Modelo de programación

- CUDA *Runtime* API (funciones con prefijo `cuda`) definida en el fichero `cuda_runtime_api.h`
  - Gestión de memoria
    - `cudaMalloc(…)`: reserva zona de memoria global
    - `cudaMemSet(…)`: inicializa zona de memoria global
    - `cudaMemcpy(…)`: copia datos desde y hacia el dispositivo
    - `cudaFree(…)`: libera zonas de memoria global
    - También existen versiones equivalentes para poder manipular vectores 2D ó 3D que garantizan el cumplimiento de ciertas restricciones de alineamiento para optimizar el rendimiento (veremos estas restricciones en Optimización de código)

Ejemplo 1: Suma de matrices

- Compilar ejemplo:
  - `ccccXX@tesla:~$ source ../cuda.env`
  - `ccccXX@tesla:~$ cd cuda_matrixAdd`
  - `ccccXX@tesla:~$ make`
- Ejecutar ejemplo:
  - `ccccXX@tesla:~$ ./cuda_matrixAdd -n=N -bsx=X`
- Editar ejemplo:
  - `ccccXX@tesla:~$ vim|joe cuda_matrixAdd.cu`
  - `ccccXX@tesla:~$ vim|joe cuda_matrixAdd_kernel.cu`

Modelo de programación

- Esquema general de una aplicación con CUDA:
  - Inicializar la GPU
    - `cutilChooseCudaDevice(argc, argv);`
  - Reservar memoria en la GPU
    - `cutilSafeCall(cudaMalloc((void **) &matrix1_d, nBytes));`
  - Mover datos desde memoria del *host* a memoria de la GPU
    - `cutilSafeCall(cudaMemcpy(matrix1_d, matrix1_h, nBytes, cudaMemcpyHostToDevice));`
  - Ejecutar uno o más *kernels*
    - `matrixAdd<<<grid, block>>>(matrix1_d, matrix2_d, matrixR_d, n);`

Modelo de programación

- Esquema general de una aplicación con CUDA:
  - Realizar otras tareas
  - Esperar a que los *kernels* terminen
    - `cudaThreadSynchronize();`
  - Mover datos desde memoria de la GPU a memoria del *host*
    - `cutilSafeCall(cudaMemcpy(matrixR_h, matrixR_d, nBytes, cudaMemcpyDeviceToHost));`
  - Liberar memoria de la GPU
    - `cutilSafeCall(cudaFree((void *) matrix1_d));`
- Las llamadas `cutil*` definidas a partir de `cutil_inline.h` encapsulan una o más llamadas a CUDA *Runtime* API para realización de comprobaciones y manejo de errores

Ejemplo 1: Suma de matrices

- Código del kernel

```
__global__ /* Código GPU */
void matrixAdd(float *matrix1_d, float *matrix2_d,
               float *matrixR_d, int n)
{
   // global ID
   int gid = (blockIdx.x * blockDim.x + threadIdx.x);
   if (gid < (n * n))
     matrixR_d[gid] = matrix1_d[gid] + matrix2_d[gid];
}
/* Llamada código paralelo desde código CPU */
dim3 grid( (nPos%bsx) ? (nPos/bsx)+1 : (nPos/bsx) );
dim3 block(bsx);
matrixAdd<<<grid, block>>>(matrix1_d, matrix2_d, matrixR_d, n);
```

Ejemplo 2: Esquema general

- Compilar ejemplo:
  - `ccccXX@tesla:~$ source ../cuda.env`
  - `ccccXX@tesla:~$ cd cuda_template`
  - `ccccXX@tesla:~$ make`
- Ejecutar ejemplo:
  - `ccccXX@tesla:~$ ./cuda_template -gsx=X -gsy=Y -bsx=X -bsy=Y`
- Editar ejemplo:
  - `ccccXX@tesla:~$ vim|joe cuda_template.cu`
  - `ccccXX@tesla:~$ vim|joe cuda_template_kernel.cu`

Ejemplo 2: Esquema general

- Código del kernel

```
__constant__ int constante_d[CM_SIZE];
__global__ /* Código GPU */
void foo(int *gid_d)
{
   extern __shared__ int shared_mem[];
   int blockSize = blockDim.x * blockDim.y;
   int tidb = (threadIdx.y * blockDim.x + threadIdx.x);
   int tidg = (blockIdx.y * gridDim.x * blockSize +
               blockIdx.x * blockSize + tidb);
   shared_mem[tidb] = gid_d[tidg];
   __syncthreads();
   shared_mem[tidb] += (tidg + constante_d[tidb%CM_SIZE]);
   __syncthreads();
   gid_d[tidg] = shared_mem[tidb];
}
/* Llamada código paralelo desde código CPU */
foo<<<grid, block, shared_mem_size>>>(gid_d);
```

Modelo de programación

- Esquema general de un *kernel*:
  - Calcular GID a partir de BID y TID

```
int blockSize = blockDim.x * blockDim.y;
// global thread ID in thread block
int tidb = (threadIdx.y * blockDim.x + threadIdx.x);
// global thread ID in grid
int tidg = (blockIdx.y * gridDim.x * blockSize +
            blockIdx.x * blockSize + tidb);
```

Modelo de programación

- Esquema general de un *kernel*:
  - Mover datos desde memoria global →memoria compartida
    - `shared_mem[tidb] = gid_d[tidg];`
  - Sincronizar todos los threads del mismo bloque (opcional)
    - `__syncthreads();`
  - Procesar los datos en memoria compartida
    - `shared_mem[tidb] += (tidg + constante_d[...]);`
  - Sincronizar todos los threads del mismo bloque (opcional)
    - `__syncthreads();`
  - Mover datos desde memoria compartida → memoria global
    - `gid_d[tidg] = shared_mem[tidb];`

Ejemplo 3: Reducción

- Compilar ejemplo:
  - `ccccXX@tesla:~$ source ../cuda.env`
  - `ccccXX@tesla:~$ cd cuda_vectorReduce`
  - `ccccXX@tesla:~$ make`
- Ejecutar ejemplo:
  - `ccccXX@tesla:~$ ./cuda_vectorReduce -n=N -bsx=X`
- Editar ejemplo:
  - `ccccXX@tesla:~$ vim|joe cuda_vectorReduce.cu`
  - `ccccXX@tesla:~$ vim|joe cuda_vectorReduce_kernel.cu`

Ejemplo 3: Reducción

Código del kernel

```
__global__ /* Código GPU */
void vectorReduce(float *vector_d, float *reduce_d, int n)
{
  extern __shared__ int sdata[];

  // global thread ID in thread block
  unsigned int tidb = threadIdx.x;

  // global thread ID in grid
  unsigned int tidg = blockIdx.x * blockDim.x + threadIdx.x;

  // load shared memory
  sdata[tidb] = (tidg < n) ? vector_d[tidg]: 0;

  __syncthreads();
  . . .
```

Ejemplo 3: Reducción

Código del kernel

```
__global__ /* Código GPU */
void vectorReduce(float *vector_d, float *reduce_d, int n)
{
   . . .
   // perform reduction in shared memory
   for(unsigned int s = blockDim.x/2; s > 0; s >>= 1) {
     if (tidb < s) {
       sdata[tidb] += sdata[tidb + s];
     }
     __syncthreads();
   }
   . . .
```

Ejemplo 3: Reducción

- Código del kernel

```
__global__ /* Código GPU */
void vectorReduce(float *vector_d, float *reduce_d, int n)
{
   . . .
   // write result for this block to global memory
   if (tidb == 0) {
     reduce_d[blockIdx.x] = sdata[0];
   }
}
```

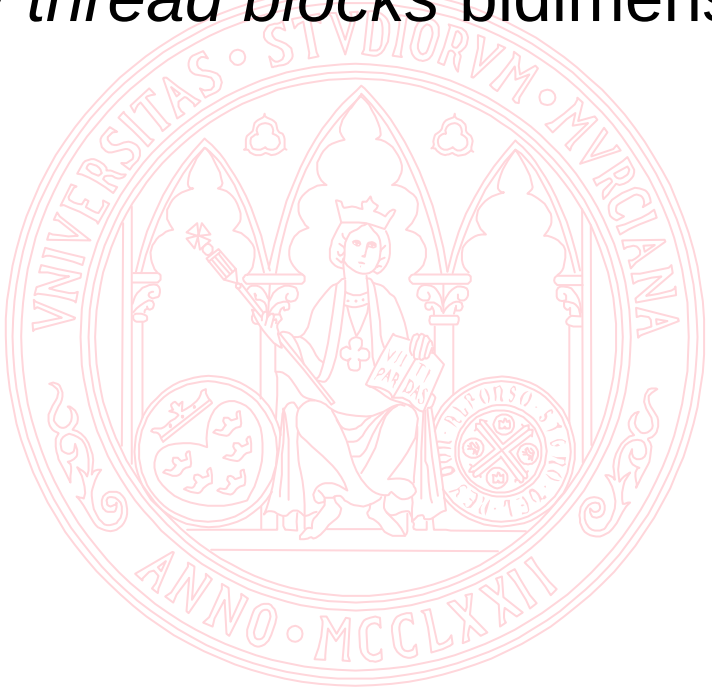
Ejemplo 3: Reducción

- Código del kernel

```
__global__ /* Código GPU */
void vectorReduce(float *vector_d, float *reduce_d, int n)
{
   . . .
   // write result for this block to global memory
   if (tidb == 0) {
     atomicAdd(reduce_d,sdata[0]); // Compute Capability ≥ 1.1
   }
}
```

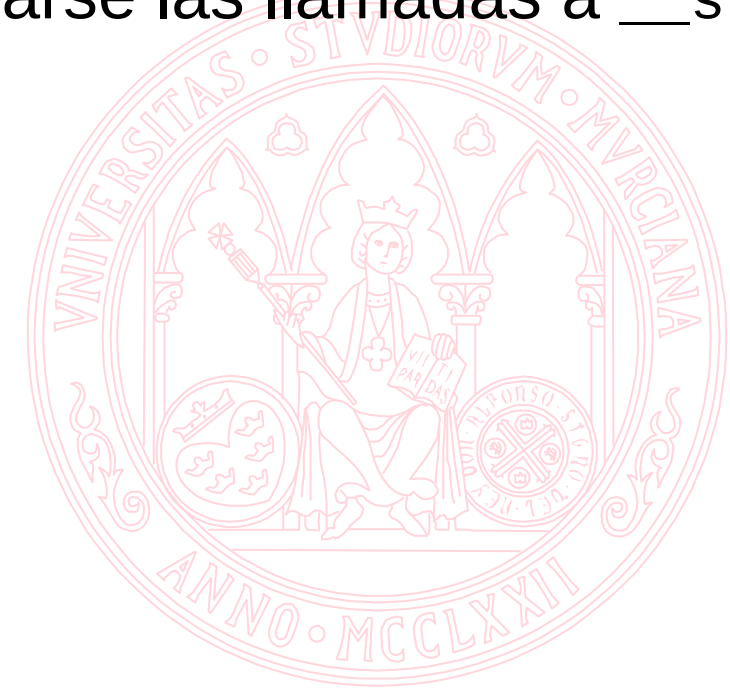
EJERCICIO 1

- Modifica el ejemplo 1 (`cuda_matrixAdd`) para que las matrices no tengan por qué ser cuadradas y el *kernel* emplee un *grid* y *thread blocks* bidimensionales



EJERCICIO 2

- Modifica el ejemplo 2 (`cuda_template`) para utilizar *thread blocks* tridimensionales en lugar de bidimensionales. ¿Podrían eliminarse las llamadas a `__syncthreads()`?



EJERCICIO 3

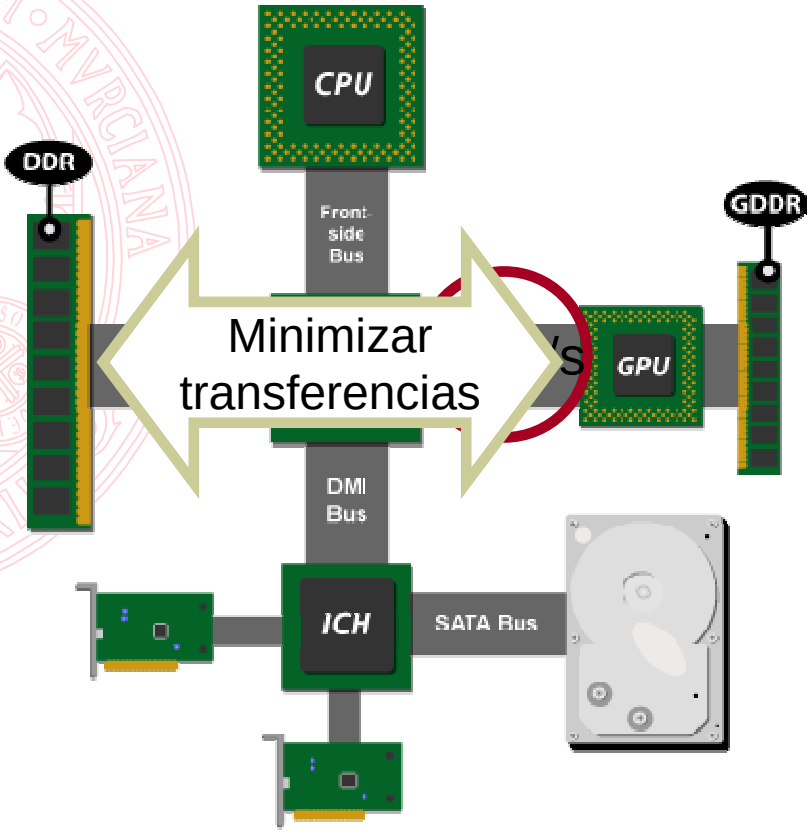
- En el ejemplo 3 (`cuda_vectorReduce`), la última fase de la reducción se realiza en la CPU. ¿Por qué? ¿Podrían eliminarse las llamadas a `__syncthreads()`?
  - Modifica el código para la reducción se complete en la GPU evitando así la intervención de la CPU.
    - ¿Qué diferencia hay entre ejecutarlo con *thread blocks* de 256 y 512 *threads* para un tamaño de 10M elementos? ¿Qué sucede si intentamos ejecutarlo con 100M elementos? ¿Por qué se produce un error? ¿Cómo se podría resolver?

Contenidos

- Motivación
- Objetivos
- Introducción
- Arquitectura y programación de CUDA
- Optimización y depuración de código
- Librerías basadas en CUDA
- Alternativas a NVIDIA/CUDA
- Conclusiones
- Bibliografía

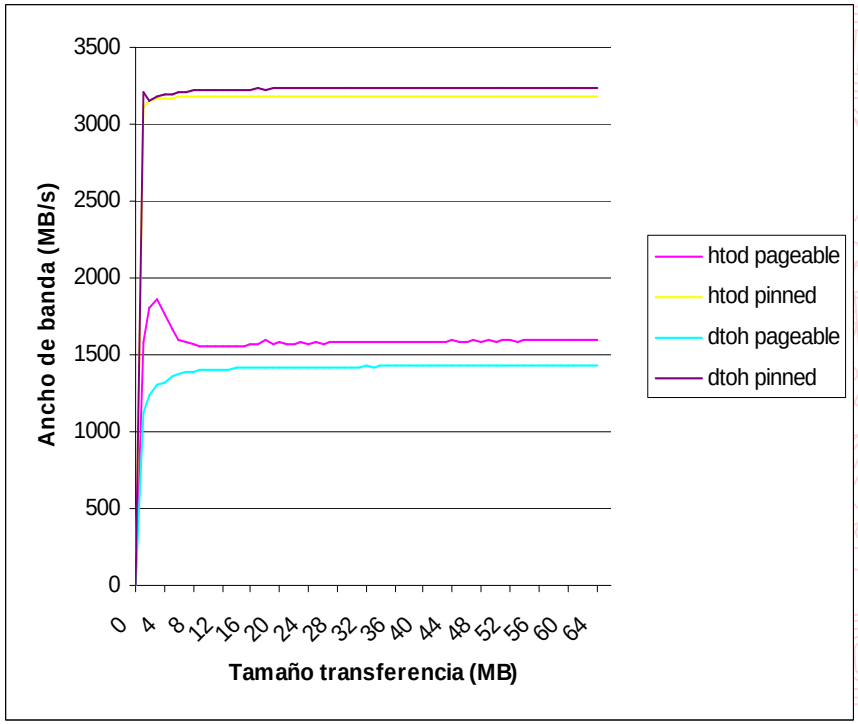
Optimización de código

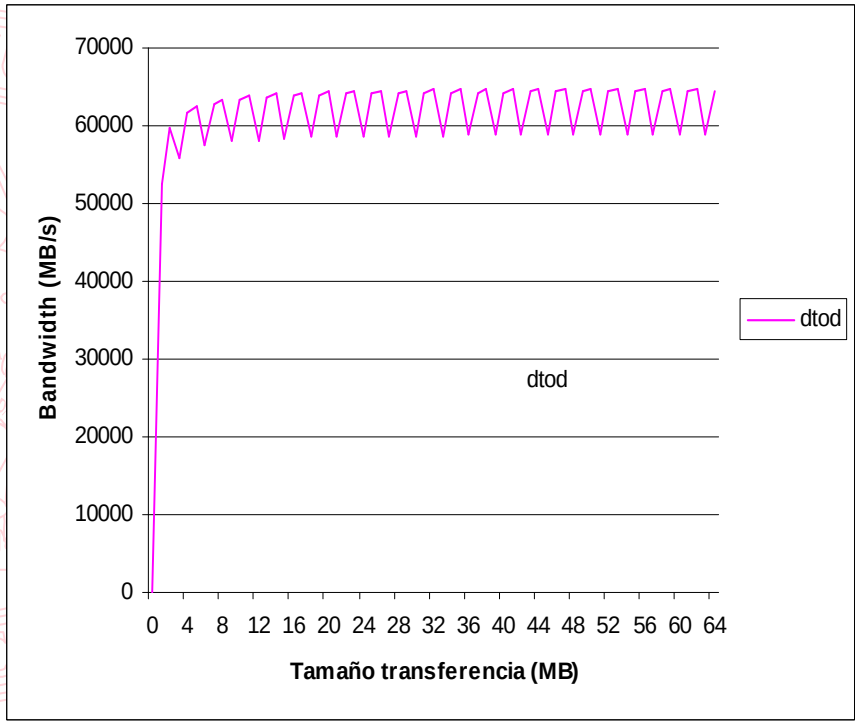
- Mover datos desde memoria del *host* a memoria de la GPU consume tiempo
  - El cuello de botella es el ancho de banda del enlace PCIe x16 (4 GB/s)
    - En la práctica, no se consigue más de 1,5 GB/s pero…
    - …si se utiliza memoria no paginable (*pinned memory*)…
      - `cudaMalloHost()/cudaFreeHost()` en lugar de `malloc()/free()`
    - …se puede duplicar el ancho de banda obtenido hasta 3 GB/s
  - El ancho de banda de la memoria de la GPU es mucho mayor (76,8 GB/s)
    - En la práctica, 65 GB/s
  - `cudaHostAlloc()` permite definir *portable memory*, *write-combining memory* y *mapped memory*



Optimización de código

- Mover datos desde memoria del *host* a memoria de la GPU consume tiempo





- Gráficas obtenidas a partir de `cuda_bandwidthTest`

Agrupando muchas transferencias pequeñas en una grande se puede reducir el impacto del movimiento de los datos

Optimización de código

- Se puede reducir el impacto de movimientos de datos...

  (véase *Concurrent copy and execution* en la salida de `cuda_deviceQuery`)

- …solapándolos con otras tareas mediante streams

```
. . .
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i) cudaStreamCreate(&stream[i]);
float* input_h, *input_d, *output_h, *output_d;
cudaMallocHost((void**)&input_h, 2 * size);
cudaMalloc((void**)&input_d, 2 * size);
cudaMallocHost((void**)&output_h, 2 * size);
cudaMalloc((void**)&output_d, 2 * size);
. . .
```

Optimización de código

```
. . .
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i)
  cudaMemcpyAsync(input_d + i * size, input_h + i * size,
                  size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
  myKernel<<<grid, block, 0, stream[i]>>>
    (input_d + i * size, output_d + i * size, size);
for (int i = 0; i < 2; ++i)
  cudaMemcpyAsync(output_h + i * size, output_d + i * size,
                  size, cudaMemcpyDeviceToHost, stream[i]);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
. . .
```

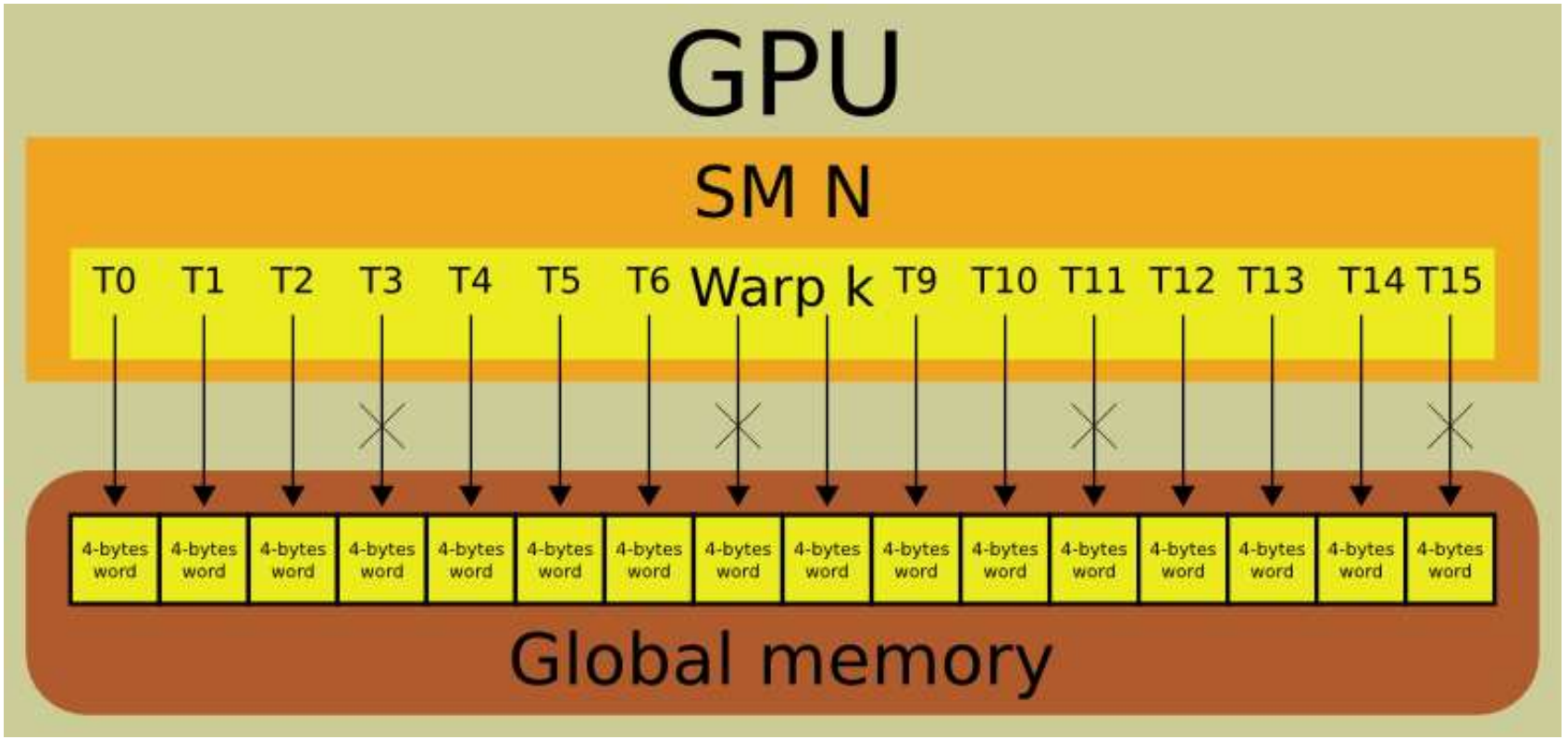
Optimización de código

```
. . .
cudaFreeHost((void *) input_h);
cudaFree((void *) input_d);
cudaFreeHost((void *) output_h);
cudaFree((void *) output_d);
for (int i = 0; i < 2; ++i)
  cudaStreamDestroy(&stream[i]);
cudaEventDestroy(start);
cudaEventDestroy(stop);
. . .
```

Optimización de código

- La memoria global no está *cacheada* por lo que se deben utilizar patrones de acceso que minimicen # transacciones
- Accesos *coalesced* a memoria global
  - Operaciones de acceso a memoria global de 4, 8 y 16 bytes
  - Memoria global como segmentos de 64 ó 128 bytes alineados a su tamaño
  - Si el k-ésimo thread de un *half warp* (16 *threads*) accede al k-ésimo elemento de un mismo segmento, los 16 accesos a memoria global se combinan en
    - Una transacción de 64 bytes para elementos de 4 bytes
    - Una transacción de 128 bytes para elementos de 8 bytes
    - Dos transacciones de 128 bytes para elementos de 16 bytes
  - No todos los *threads* tienen la obligación de participar

Optimización de código



Optimización de código

Accesos *coalesced* a memoria global

- Ignorar esta optimización penaliza el ancho de banda de acceso a la memoria global de manera significativa (aproximadamente un orden de magnitud)
  - 16 transacciones de acceso a memoria global en lugar de una
- Las variables de tipo `type`, donde `sizeof(type)` es igual a 8 ó 16 bytes, deben estar alineadas a `sizeof(type)` bytes
  - Se puede forzar el alineamiento con `__align__(8|16)`

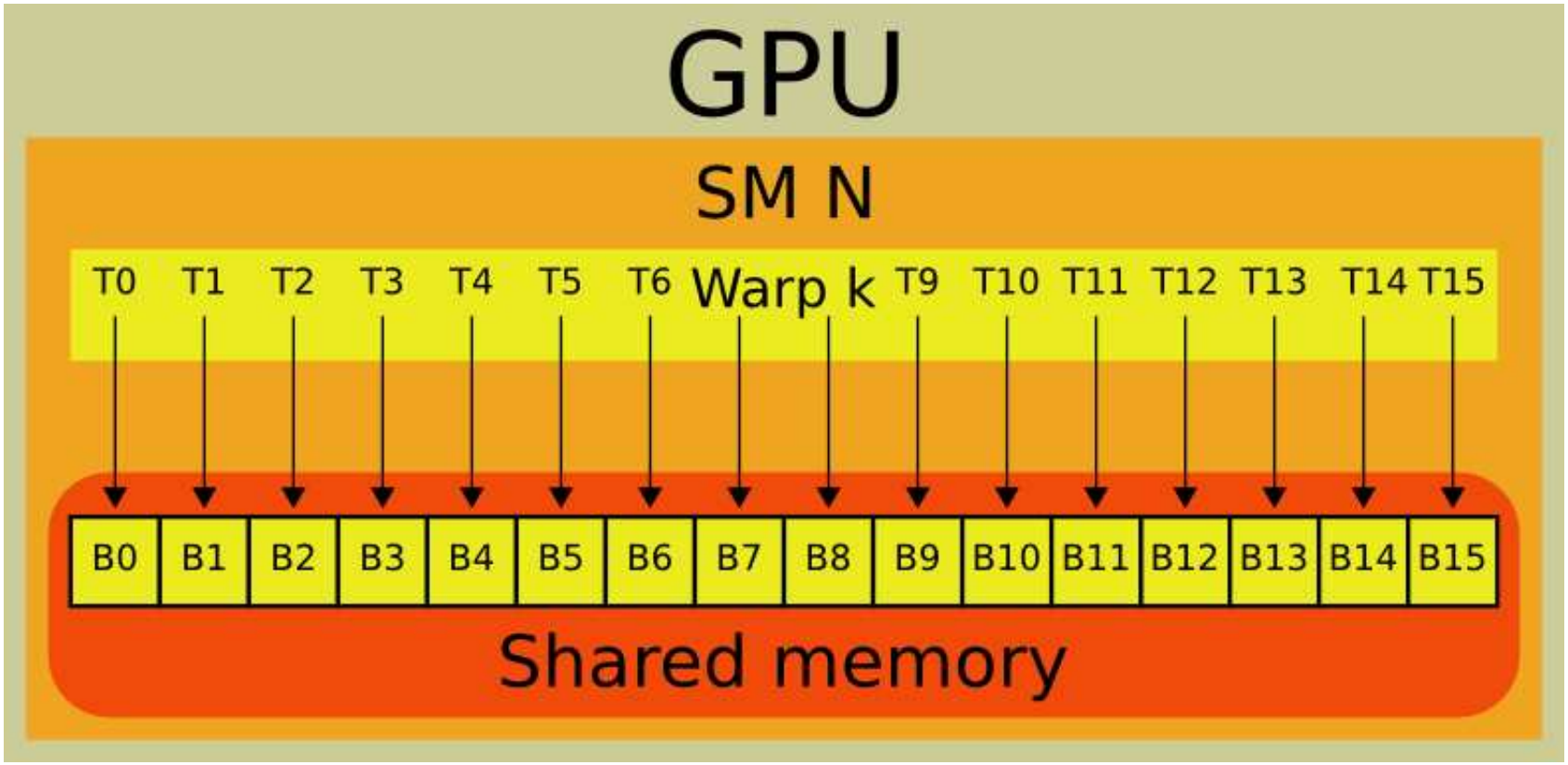
```
struct __align__(8) {
  float a;
  float b;
} align_struct;
```

- Para GPUs con *Compute Capability* ≥ 1.2 las restricciones son bastante más flexibles

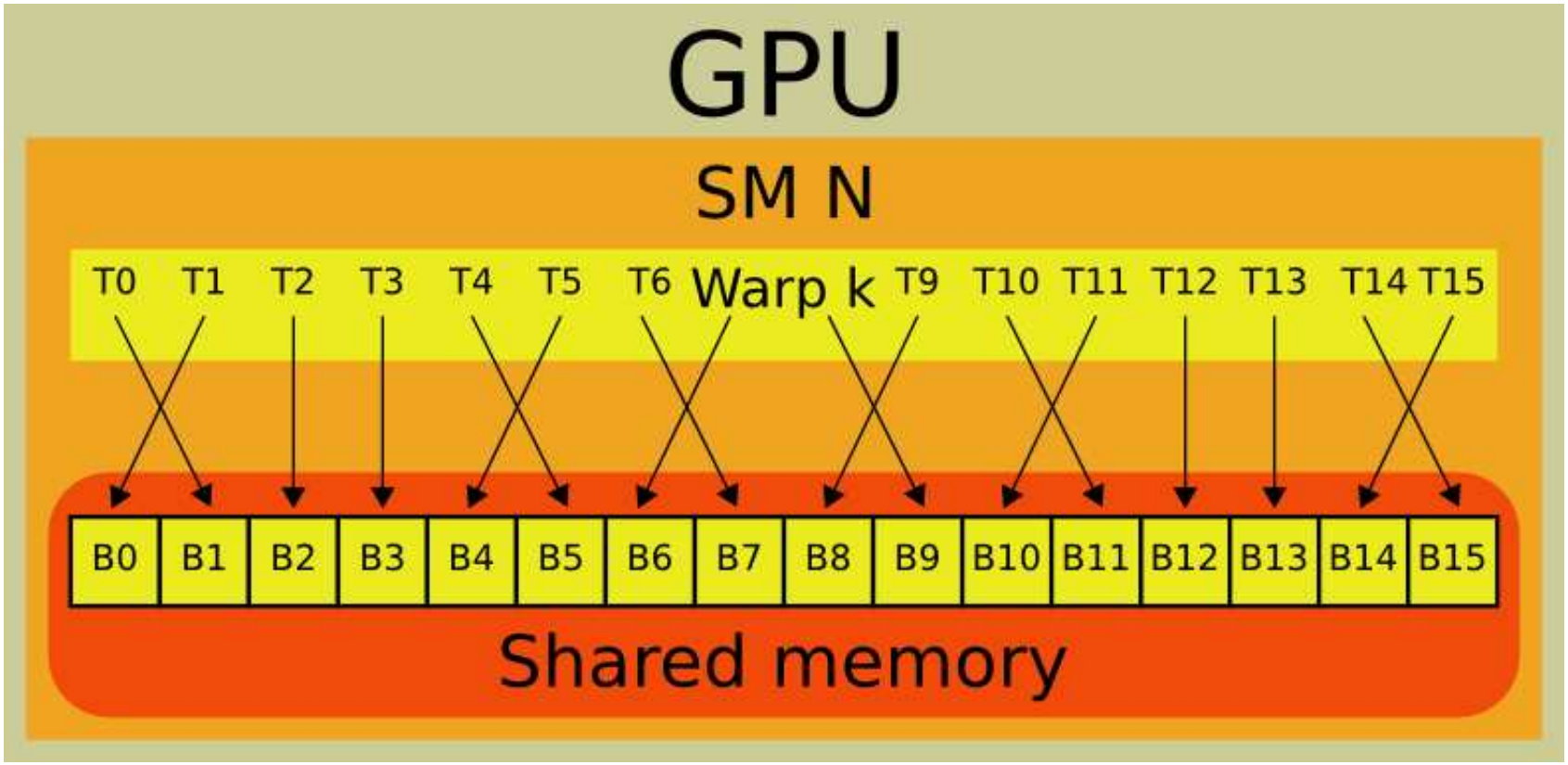
Optimización de código

- La memoria compartida tiene un tiempo de acceso similar a los registros si no hay conflictos entre los *threads*
- Accesos sin conflictos a memoria compartida
  - La memoria compartida está compuesta de 16 bancos
  - Elementos de 4 bytes consecutivos se almacenan en los bancos de memoria compartida de manera cíclica
  - Todos los accesos de un *half warp* pueden atenderse simultáneamente si todas las direcciones pertenecen a distintos bancos pudiendo repetirse éstas (*broadcast*)
- Se puede usar la memoria compartida tanto para conseguir accesos *coalesced* a memoria global como para eliminar accesos redundantes a memoria global

Optimización de código

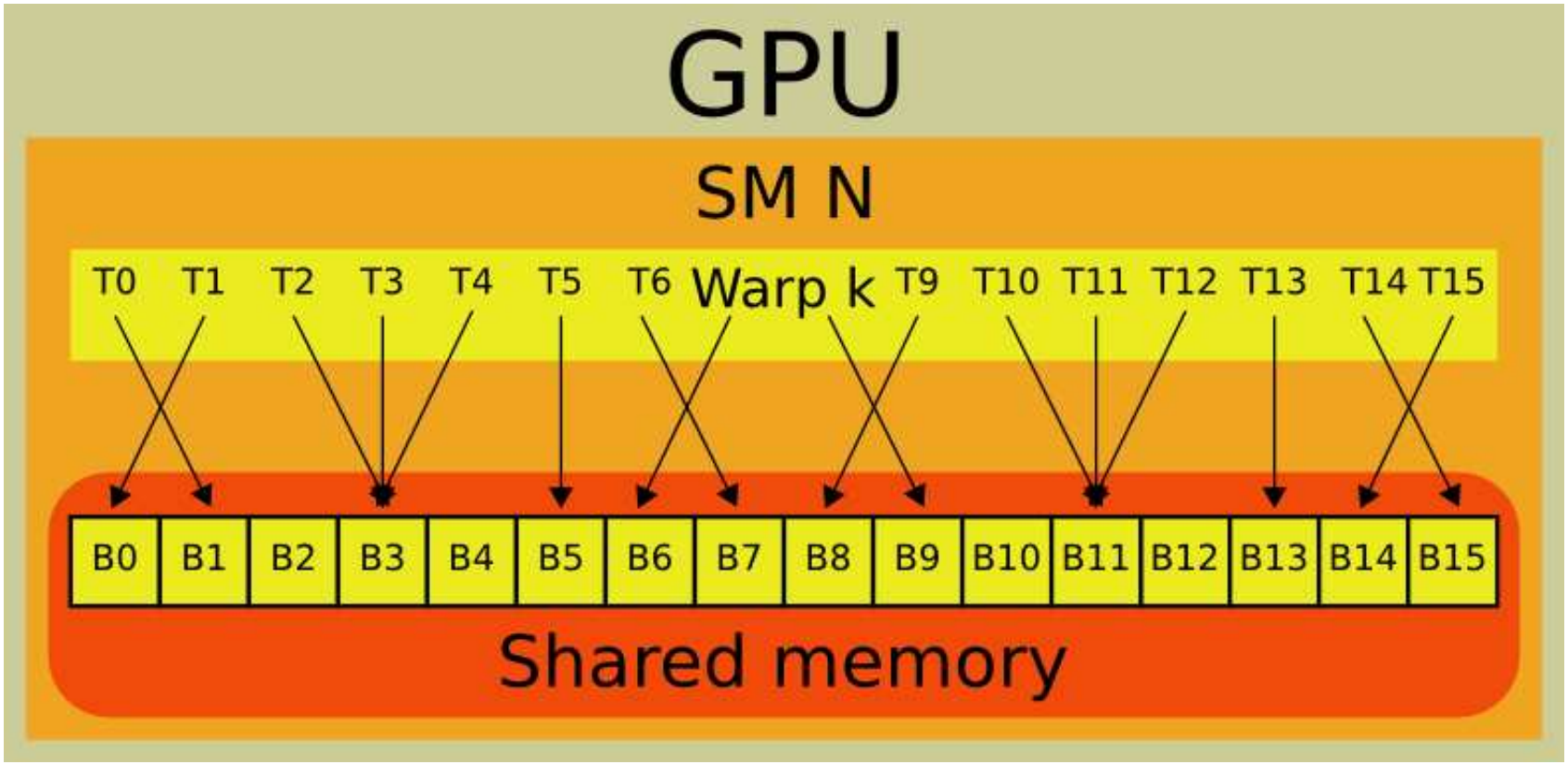


Optimización de código



Optimización de código

- ¿Libre de conflictos?



- **Solamente si T2, T3 y T4 acceden a la misma palabra**

Ejemplo 4: Trasposición de matrices

- Compilar ejemplo:
  - ccccXX@tesla:~$ **source ../cuda.env**
  - ccccXX@tesla:~$ **cd cuda_matrixTranspose**
  - ccccXX@tesla:~$ **make**
- Ejecutar ejemplo:
  - ccccXX@tesla:~$ **./cuda_matrixTranspose –dimx=1024 –dimy=1024**
- Editar ejemplo:
  - ccccXX@tesla:~$ **vim|joe cuda_matrixTranspose.cu**

Ejemplo 4: Trasposición de matrices

```
Device 0: "Tesla C870"
SM Capability 1.0 detected:
CUDA device has 16 Multi-Processors
SM performance scaling factor = 1.50
MatrixSize X = 1024
MatrixSize Y = 1024

Matrix size: 1024x1024 (32x32 tiles), tile size: 32x32, thread block size: 32x8

Kernel                  Loop over kernel    Loop within kernel
------                  ----------------    ------------------
simple copy             52.22 GB/s          58.74 GB/s
shared memory copy      53.26 GB/s          60.40 GB/s
naive transpose          3.71 GB/s           3.65 GB/s
coalesced transpose     30.96 GB/s          39.39 GB/s
no bank conflict trans  39.44 GB/s          42.26 GB/s
coarse-grained          39.76 GB/s          42.03 GB/s
fine-grained            48.48 GB/s          60.06 GB/s
diagonal transpose      34.17 GB/s          56.70 GB/s
```

A cada *thread block* se le asigna un *tile* de la matriz de 32x32 elementos.
Cada *thread* traspone 4 elementos de la misma columna.

Ejemplo 4: Trasposición de matrices

Código del kernel

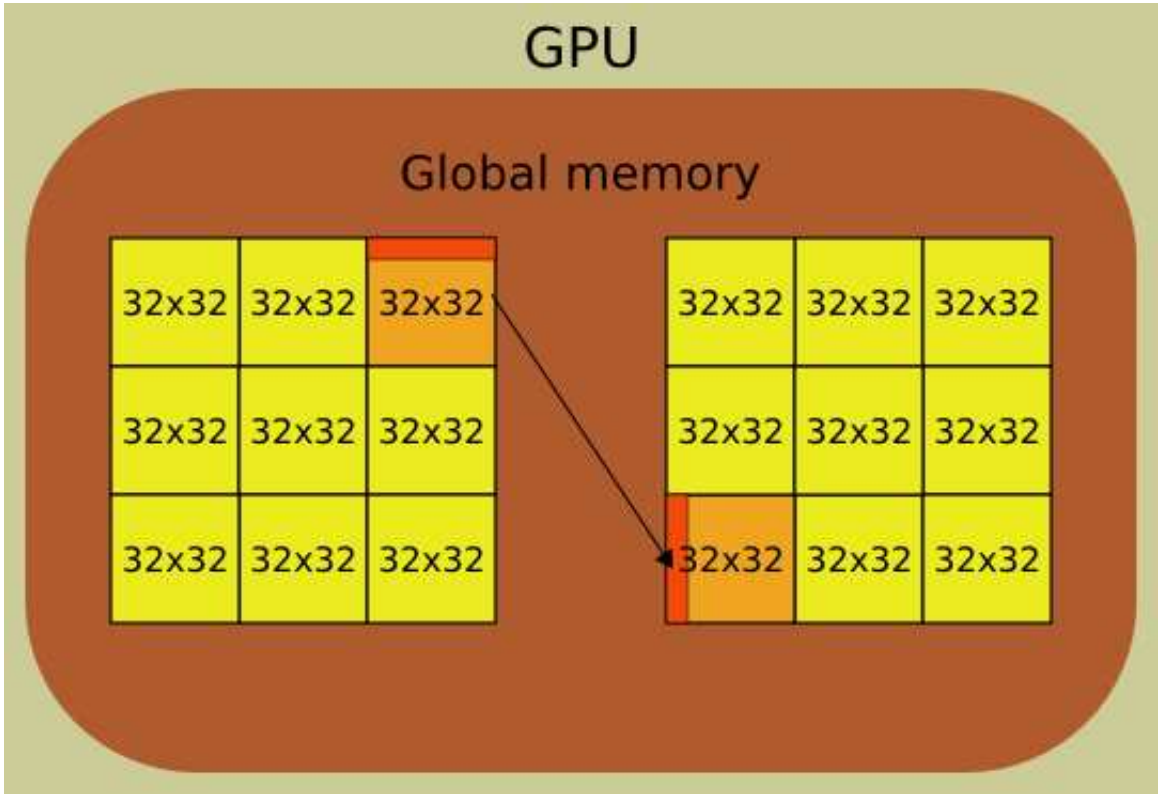
```
__global__ /* Código GPU */
void transposeNaive(float *odata,float* idata,int width,int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in  = xIndex + width * yIndex;

  int index_out = yIndex + height * xIndex;

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
      odata[index_out+i] = idata[index_in+i*width];
}
```

Ejemplo 4: Trasposición de matrices

- ¿Accesos *coalesced* a memoria global?



Lecturas de memoria global *coalesced*.

Escrituras en memoria global *non-coalesced*.

Ejemplo 4: Trasposición de matrices

Código del kernel

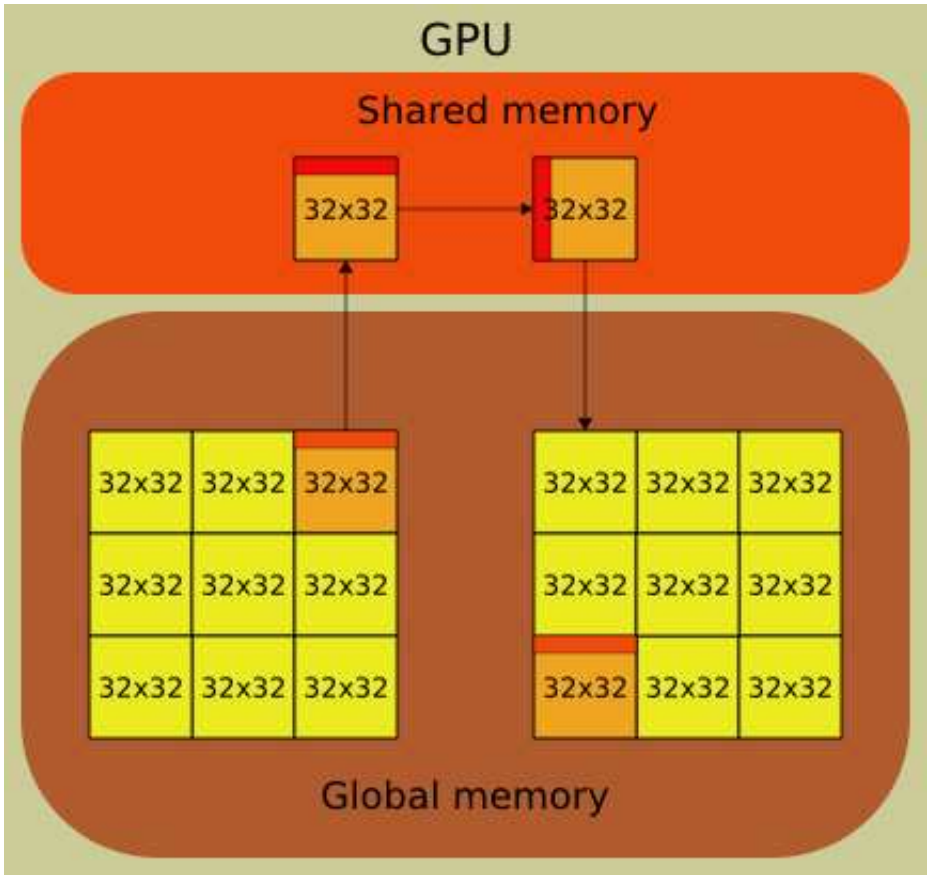
```
__global__ /* Código GPU */
void transposeCoalesced(...) {
  __shared__ float tile[TILE_DIM][TILE_DIM];
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in  = xIndex + width * yIndex;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + height * yIndex;

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
  __syncthreads();
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
}
```

Ejemplo 4: Trasposición de matrices

- ¿Accesos *coalesced* a memoria global? ¿Conflictos en los accesos a memoria compartida?



Lecturas y escrituras desde y en memoria global *coalesced*.
Conflictos en escrituras en memoria compartida.

Ejemplo 4: Trasposición de matrices

- Código del kernel

```
__global__ /* Código GPU */
void transposeNoBankConflicts(...) {
  __shared__ float tile[TILE_DIM][TILE_DIM+1];
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
```

Lecturas y escrituras desde y en memoria global *coalesced*.

Lecturas y escrituras desde y en memoria compartida libres de conflictos.

```
  int index_in = xIndex + width * yIndex;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + height * yIndex;

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
  __syncthreads();
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
    odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
}
```

Optimización de código

Configuración de la ejecución

- La **Ocupación** se define como el ratio entre el número de *threads* activos por SM y el número máximo posible
  - Mayor ocupación no garantiza mejor rendimiento pero…
  - …baja ocupación equivale normalmente a peor rendimiento
    - Incapacidad para ocultar las largas latencias de acceso a memoria global
- El número máximo de *threads* activos está limitado por el consumo de registros/memoria compartida por *thread block*
  - *Compute Capability* ≤ 1.1 → 8Kregs/768 *threads* (24 *warps*)
  - *Compute Capability* ≥ 1.2 → 16Kregs/1024 *threads* (32 warps)
- Para determinar la configuración óptima NVIDIA proporciona CUDA *Occupancy Calculator*

Microsoft Excel - CUDA_Occupancy_calculator

Archivo Edición Ver Insertar Formato Herramientas Datos Ventana ?

Escriba una pregunta

Arial 10 N K S

B9 128

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	1.0	(Help)
2.) Enter your resource usage:		
Threads Per Block	128	(Help)
Registers Per Thread	8	
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	768	(Help)
Active Warps per Multiprocessor	24	
Active Thread Blocks per Multiprocessor	6	
Occupancy of each Multiprocessor	100%	

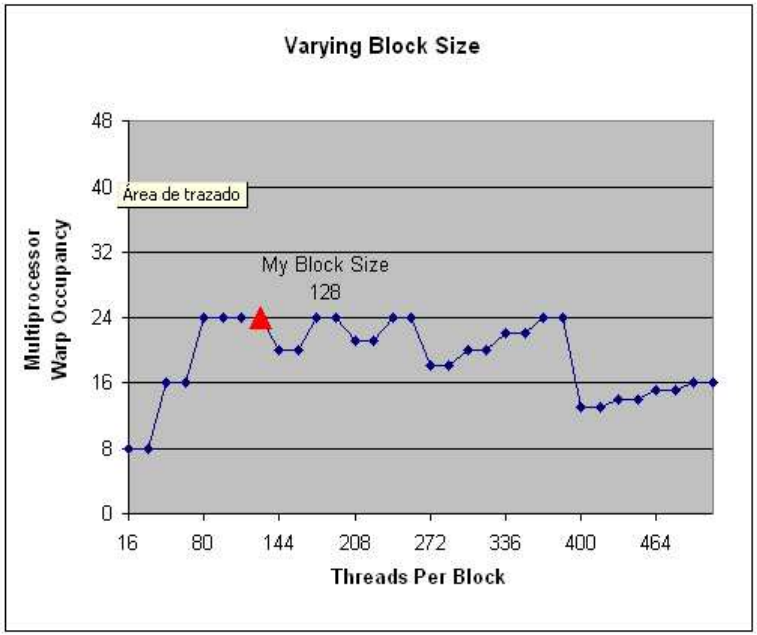
Physical Limits for GPU Compute Capability:	1.0
Threads per Warp	32
Warps per Multiprocessor	24
Threads per Multiprocessor	768
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	8192
Register allocation unit size	256
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2

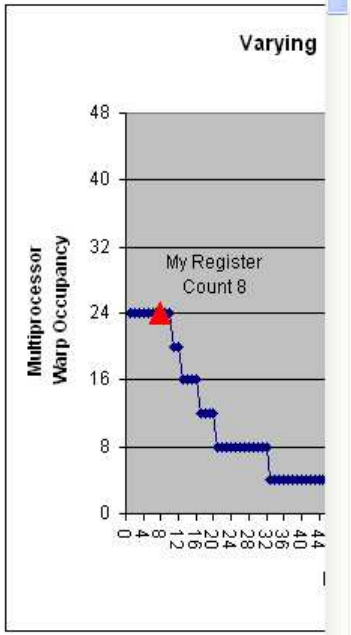
Allocation Per Thread Block	
Warps	4
Registers	1024
Shared Memory	1024

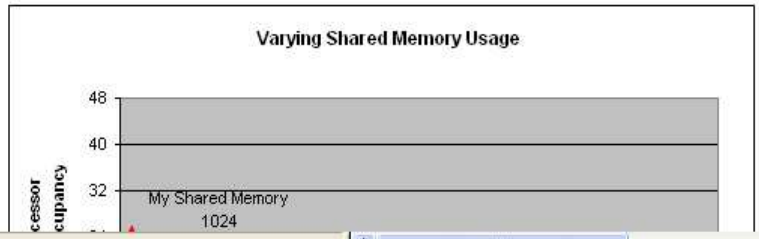
These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared mem

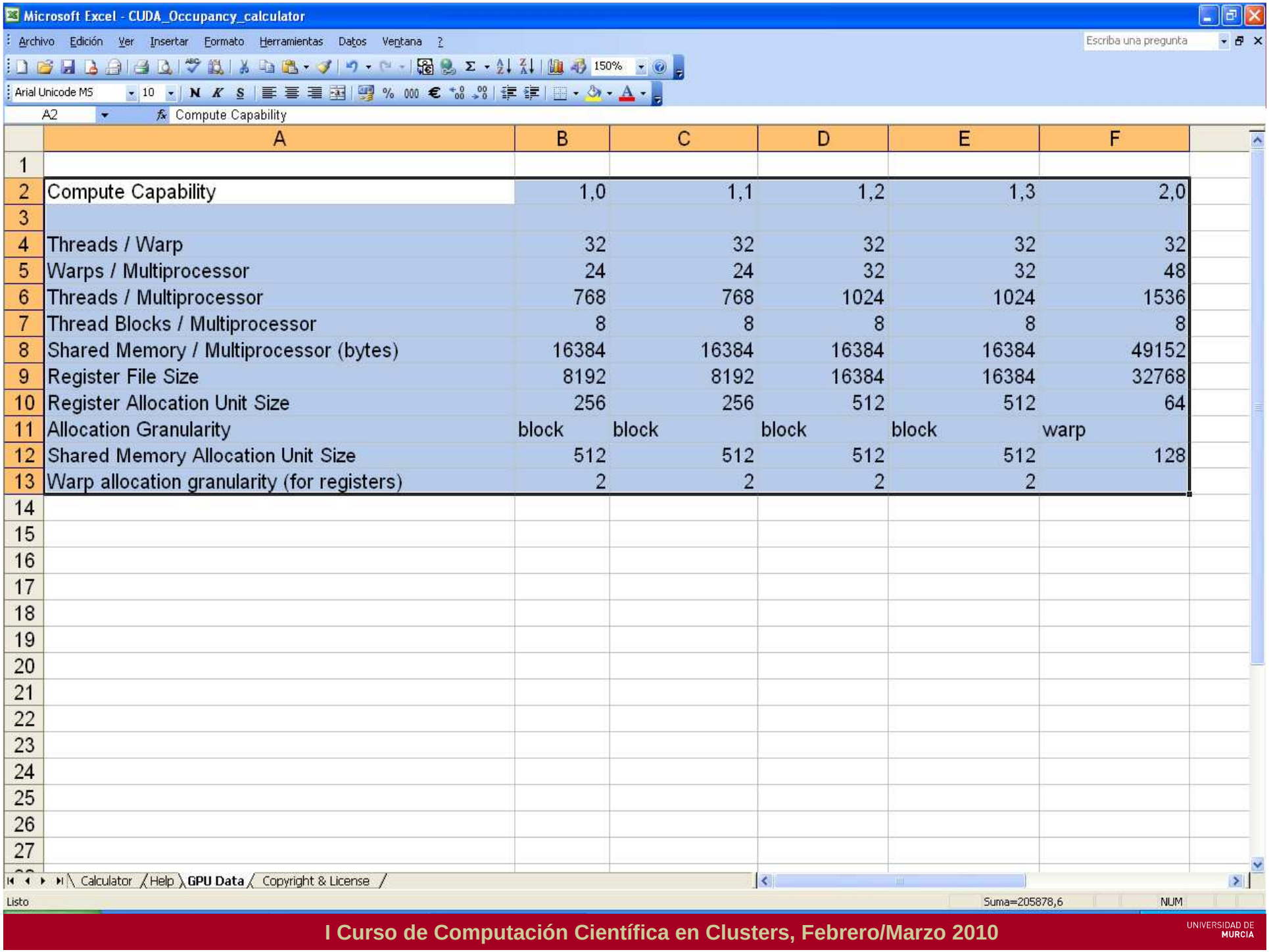






Calculator / Help / GPU Data / Copyright & License

Listo Suma=1160 NUM



	A	B	C	D	E	F
1						
2	Compute Capability	1,0	1,1	1,2	1,3	2,0
3						
4	Threads / Warp	32	32	32	32	32
5	Warps / Multiprocessor	24	24	32	32	48
6	Threads / Multiprocessor	768	768	1024	1024	1536
7	Thread Blocks / Multiprocessor	8	8	8	8	8
8	Shared Memory / Multiprocessor (bytes)	16384	16384	16384	16384	49152
9	Register File Size	8192	8192	16384	16384	32768
10	Register Allocation Unit Size	256	256	512	512	64
11	Allocation Granularity	block	block	block	block	warp
12	Shared Memory Allocation Unit Size	512	512	512	512	128
13	Warp allocation granularity (for registers)	2	2	2	2	
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						

EJERCICIO 4

- Determina el número de registros y la cantidad de memoria compartida requerida por cada *thread block* en la ejecución de los *kernels* `transposeNaive`, `transposeCoalesce` y `transposeNoBankConflicts` del ejemplo 4 (`cuda_transposeMatrix`).
  - Para ello, compila el ejemplo con `--ptxas-options=-v`
- A continuación, verifica si se usa su configuración de ejecución óptima con CUDA *Occupancy Calculator*

Optimización de código

Instrucciones aritméticas

- Rendimiento muy superior con simple precisión (SP) frente a doble precisión (DP)
  - Si *Compute Capability* ≥ 1.3 → soporte IEEE 754 DP
- En otro caso, sólo soporte IEEE 754 SP
  - El compilador transforma los `doubles` en `floats`
- CUDA no sigue completamente el estándar IEEE 754
  - Descripción de las desviaciones en [CPG2.3.1] A.2
- CUDA proporciona una versión más rápida pero menos precisa de algunas operaciones de simple precisión
  - Por ejemplo, `__sinf()` en vez de `sinf()`
  - Con la opción de nvcc `-use_fast_math` podemos forzar la sustitución de manera transparente siempre que sea posible

Optimización de código

- Control de flujo (saltos y divergencia)
  - Las instrucciones de control de flujo que implican saltos como `if, switch, do, for o while` pueden hacer que los *threads* de un *warp* diverjan
    - Se puede evitar siempre que sea posible expresar la condición en función del tamaño del *warp* y no sólo del TID
  - Todas las funciones `__device__` son *inline*
    - El programador puede evitarlo con `__noinline__`
  - El compilador desenrolla los bucles
    - El programador puede hacerlo manualmente con la directiva `#pragma unroll N`
  - También utiliza predicación cuando el número de instrucciones de cada rama es menor que o igual a 7

Depuración de código

- `nvcc` y CUDA *Runtime* API soportan un modo de emulación incluso en ausencia del dispositivo
  - `nvcc -deviceemu`
  - Modelo de ejecución emulado por el *runtime* creando en la CPU un *thread* por cada thread del *grid*
  - Uso de `printf` y herramientas de depuración nativas
  - Condiciones de carrera más difíciles de reproducir
- `cuda-gdb` (*Compute Capability* ≥ 1.1)
  - Extensión de `gdb` para soportar CUDA
- CUDA *Visual Profiler* (`cudaprof`)
  - Estadísticas de la ejecución de una aplicación CUDA incluyendo accesos *non-coalesced* a memoria global, divergencia, etc.

EJERCICIO 5

- Determina el número de accesos *non-coalesced* a memoria global que se producen en la ejecución de los *kernels* `transposeNaive` y `transposeCoalesced` del ejemplo 4 (`cuda_transposeMatrix`) con `cudaprof`.
  - `cd /usr/local/cuda/cudaprof/bin`
  - `./cudaprof`
    - File → Open… (`cuda_transposeMatrix.cpj`)

EJERCICIO 6

- Escribe un programa que calcule el producto escalar de dos vectores tomando como base los ejemplos.
  - Determina el número de registros y la cantidad de memoria compartida requerida por cada *thread block*, la configuración de ejecución óptima, el número de accesos *non-coalesced* a memoria global y el número de conflictos en el acceso a la memoria compartida durante su ejecución.

Contenidos

- Motivación
- Objetivos
- Introducción
- Arquitectura y programación de CUDA
- Optimización y depuración de código
- Librerías basadas en CUDA
- Alternativas a NVIDIA/CUDA
- Conclusiones
- Bibliografía

Librerías basadas en CUDA

- NVIDIA CUFFT
  - Implementación *Fast Fourier Transform* (FFT)
- NVIDIA CUBLAS
  - Implementación BLAS (*Basic Linear Algebra Subprograms*)
    - Representación FORTRAN para las matrices
- NVIDIA *Performance Primitives* (NPP)
  - Implementación de funciones de diversa naturaleza
- CUDA *Data Parallel Primitives Library* (CUDPP)
  - *Radix sort, (segmented) scan, compact array, SPMV*

Contenidos

- Motivación
- Objetivos
- Introducción
- Arquitectura y programación de CUDA
- Optimización y depuración de código
- Librerías basadas en CUDA
- Alternativas a NVIDIA/CUDA
- Conclusiones
- Bibliografía

Alternativas a NVIDIA/CUDA

ATI/AMD Stream con Brook+

```
kernel void saxpy(float a, float x<>, float y<>, out float res<>)
{
  res = a * x + y;
}

void main(void)
{
  float a; float X[100], Y[100], RES[100];
  float x<100>, y<100>, res<100>;
  . . . Initialize a, X, Y . . .
  streamRead(x, X);
  streamRead(y, Y);
  saxpy(a, x, y, res);
  streamWrite(res, RES);
}
```

Alternativas a NVIDIA/CUDA

- NVIDIA Fermi
  - Próxima generación de CUDA
    - Inclusión de caches, mejor rendimiento IEEE 754 DP, posibilidad de ejecutar varios *kernels* simultáneamente, etc.
- *Open Computing Language* (OpenCL)
  - Estándar abierto desarrollado por Khronos OpenCL *working group* (http://www.khronos.org) con el apoyo de múltiples empresas del sector como NVIDIA, ATI/AMD, Intel, IBM, etc.
  - Solución basada en una librería cuyo objetivo principal es conseguir la portabilidad entre plataformas
  - Disponibles las primeras implementaciones de NVIDIA y ATI/AMD
- ¿Intel Larrabee?

Contenidos

- Motivación
- Objetivos
- Introducción
- Arquitectura y programación de CUDA
- Optimización y depuración de código
- Librerías basadas en CUDA
- Alternativas a NVIDIA/CUDA
- Conclusiones
- Bibliografía

Conclusiones

- La paralelización de aplicaciones con CUDA difiere significativamente de la misma tarea con OpenMP sobre procesadores multinúcleo convencionales
  - Descomposición del problema en subproblemas INDEPENDIENTES
    - Paralelismo de datos que debe identificar el programador
  - *# threads*
  - Coste de creación y destrucción de los *threads*
  - Coste de los cambios de contexto
  - Posibilidades de comunicación y sincronización de *threads*
  - Tipos y gestión de la memoria disponible
  - Ausencia de bloqueos si se siguen unas reglas básicas

Conclusiones

- La mejora de rendimiento obtenida dependerá del porcentaje del tiempo de ejecución que supongan los *kernels* sobre el total de la aplicación (Ley de Amdahl)
- Las aplicaciones paralelizadas con CUDA escalan conforme aumentamos el número de SMs y/o GPUs
  - ¿Adaptación de la configuración de ejecución?

Conclusiones

- ¿Qué aplicaciones son susceptibles de ser paralelizadas con éxito utilizando CUDA?
  - Paralelismo de datos preferido sobre paralelismo de tareas
    - SIMT vs. SIMD
      - La divergencia de *threads* penaliza el rendimiento
  - Procesamiento de bloques de datos independientes
    - El flujo de control debe ser independiente de los datos
      - Algoritmos con muchas dependencias de datos serán difíciles de paralelizar de manera eficiente
  - Patrones de acceso a memoria regulares o susceptibles de ser adaptados usando la memoria compartida
  - Intensidad aritmética suficiente para amortizar las copias y mantener ocupados todos los recursos hardware de la GPU
  - IEEE 754 SP proporcionará una mayor mejora que DP

Bibliografía

- Manuales
  - [CPG2.3.1] *CUDA Programming Guide 2.3.1*
  - [CPBP2.3] *CUDA C Programming Best Practices Guide 2.3*
  - [CRM2.3] *CUDA Reference Manual 2.3*
- Ejemplos: `/opt/NVIDIA_GPU_Computing_SDK/sdk/C/src`
- *Websites*
  - *General-Purpose Computation on Graphics Hardware*: http:///gpgpu.org (Información general sobre GPGPU)
  - NVIDIA *Developer Zone*: http://developer.nvidia.com/object/gpucomputing.html (CUDA)
  - CUDPP: http://gpgpu.org/developer/cudpp

Bibliografía

Artículos

- Dinesh Manocha, General-Purpose Computations using Graphics Procesors, IEEE Computer, vol. 38 no. 8, pp. 85–88, August 2005
- Erik Lindholm et al., NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE Micro, vol. 28 no. 2, pp. 39–55, March 2008
- Tom R. Halfhill, Parallel Processing with CUDA, MicroProcessor Report online (http://www.mpronline.com), January 2008
- John Nickolls et al., Scalable Parallel Programming, ACM Queue, vol. 6 no. 2, pp. 40–53, March/April 2008
- Wen-mei Hwu et al., Compute Unified Device Architecture Application Suitability, Computing in Science and Engineering, vol. 11 no. 3, pp. 16–26, May/June 2009
- Michael Garland et al., Parallel Computing Experiences with CUDA, IEEE Micro, vol. 28 no. 4, pp. 13–27, July 2008

I Curso de Computación Científica en *Clusters*

Febrero/Marzo 2010

Programación de GPUs