Programación de una GPU con CUDA

Programación de Arquitecturas Multinúcleo

Contenidos

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

Introducción

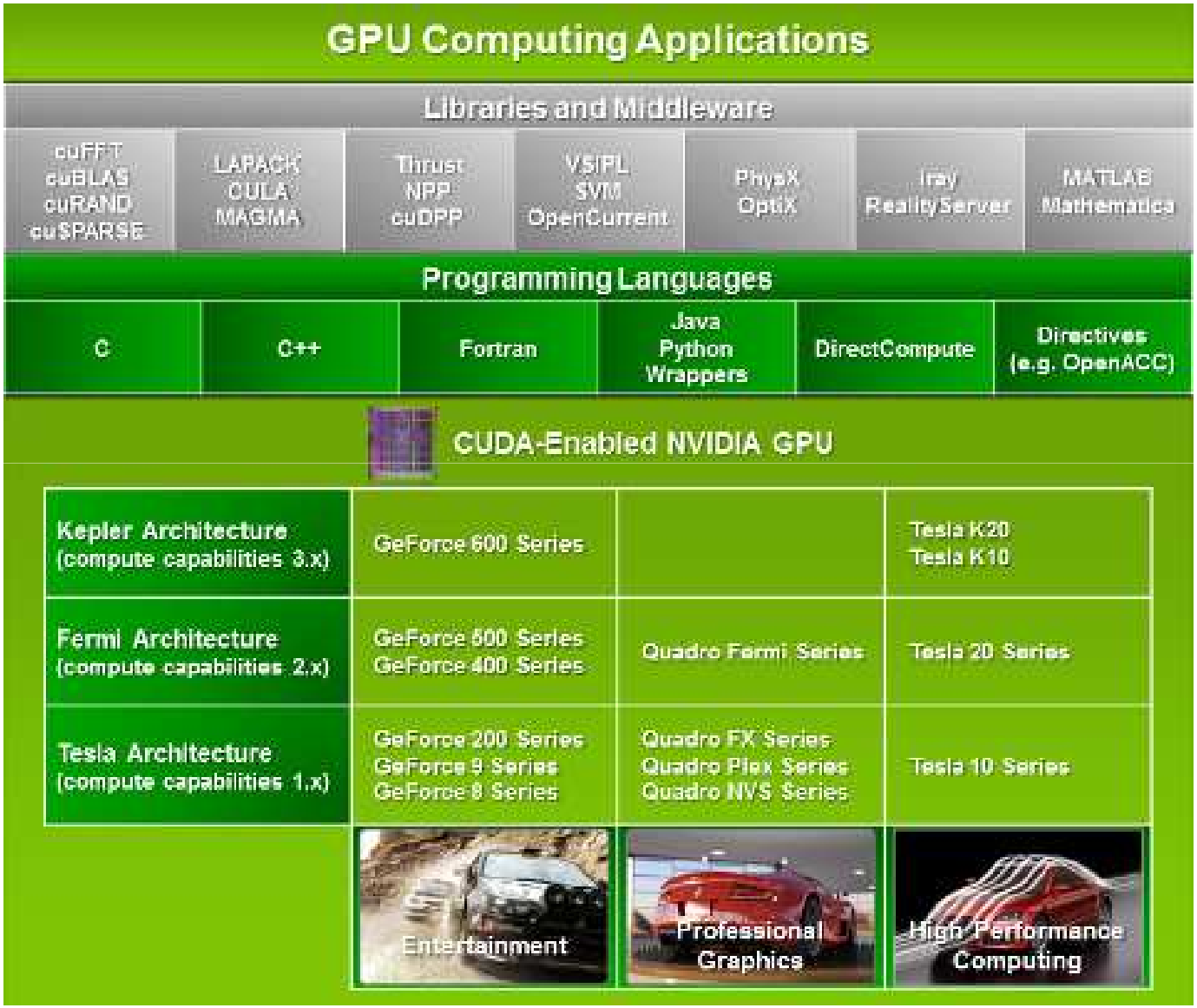
- En NOV'06 NVIDIA introduce la arquitectura unificada con la NVIDIA GeForce 8800

- GPUs de NVIDIA compatibles con CUDA*
  - Modelos:
    - GeForce Series 8, 9, 100, 200, 400 y 500 con > 256 MB
    - Tesla C870, D870, C1060, C2050, C2070, C2075 y K20
    - Quadro FX
    - Tesla K10, K20, K20x, K40

  - Diferencias:
    - Interfaz (ancho de banda) y memoria integrada (MB)
    - *Compute Capability* (1.x , 2.x, 3.x)
      - Recursos: # núcleos (# SMs y # SPs por SM), …
      - Funcionalidad: soporte IEEE 754 DP, …

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

Introducción



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

Contenidos

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

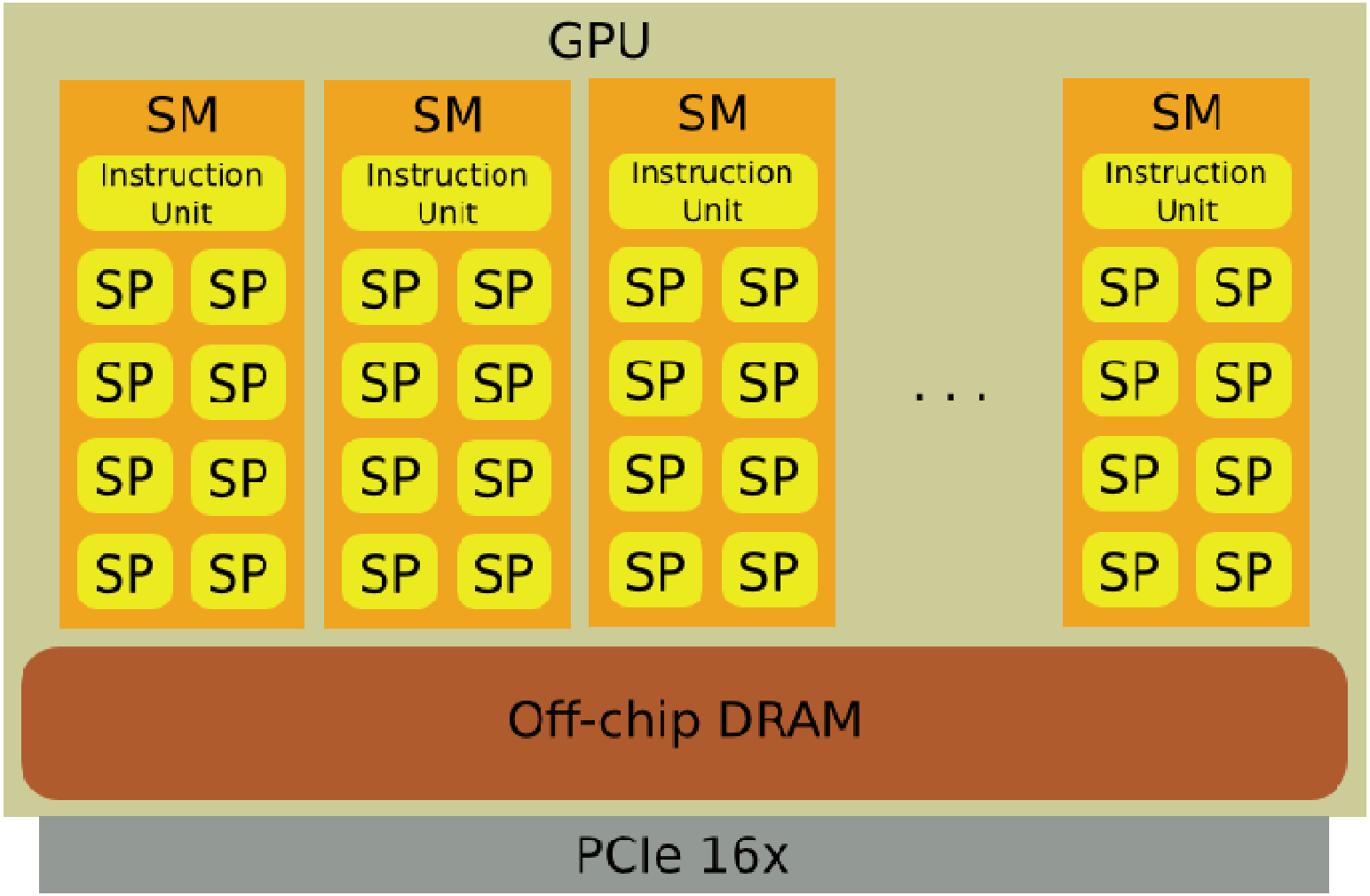
Contenidos

- Introducción
- **Arquitectura y programación de CUDA**
  - **Arquitectura hardware y software**
  - **Modelo de Memoria**
  - **Ejemplo 0: device_query**
  - **Modelo de Ejecución**
  - **Modelo de Programación**
    - **Ejemplo 1: suma de matrices**
    - **Ejemplo 2: template**
    - **Ejemplo 3: reducción**
- Optimización y depuración de código
- Librerías basadas en CUDA
- Alternativas a NVIDIA/CUDA
- Conclusiones
- Bibliografía

Arquitectura hardware y software

- GPU = vector: ***N* x *Streaming Multiprocessors*** (SMs)
  - SM = ***N* x *Streaming Processors*** (SPs)
    - Realizan operaciones escalares sobre enteros de 32 bits o reales SP (compatible con IEEE 754)
    - Ejecutan *threads* independientes pero…
    - …todos deberían ejecutar la instrucción leída por la *Instruction Unit* (IU) para optimizar el rendimiento
      - *Single Instruction Multiple Thread* (SIMT)
        - » Explotación paralelismo de datos y, en menor grado, de tareas
- Los *threads* gestionados por el hardware en cada SM
  - Creación/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 para ser ejecutadas por la GPU se denominan ***kernels***

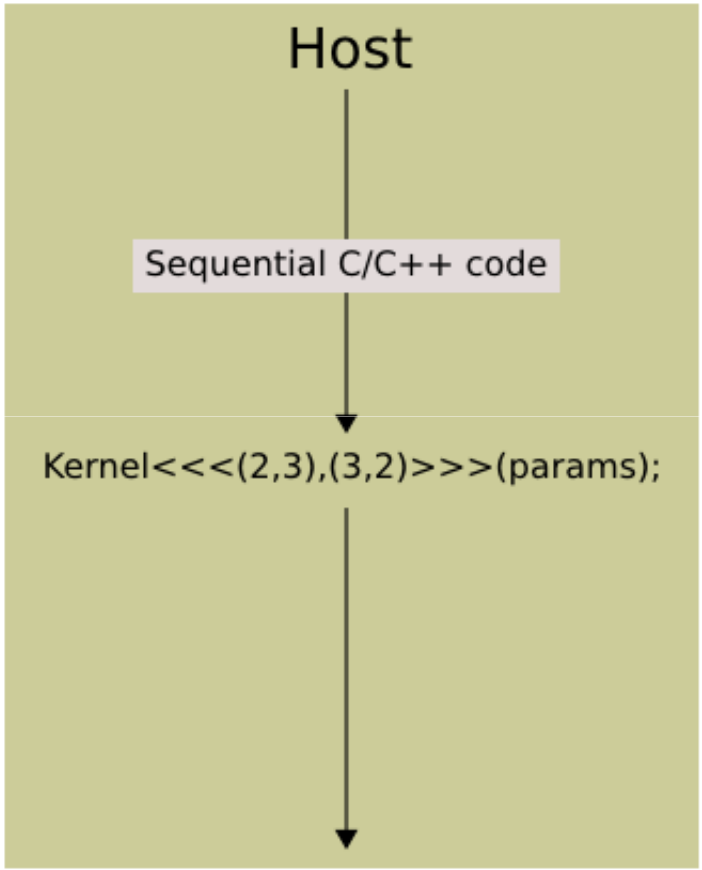
- Un *kernel* descompone un problema en un conjunto de subproblemas *independientes* y lo mapea sobre un *grid*

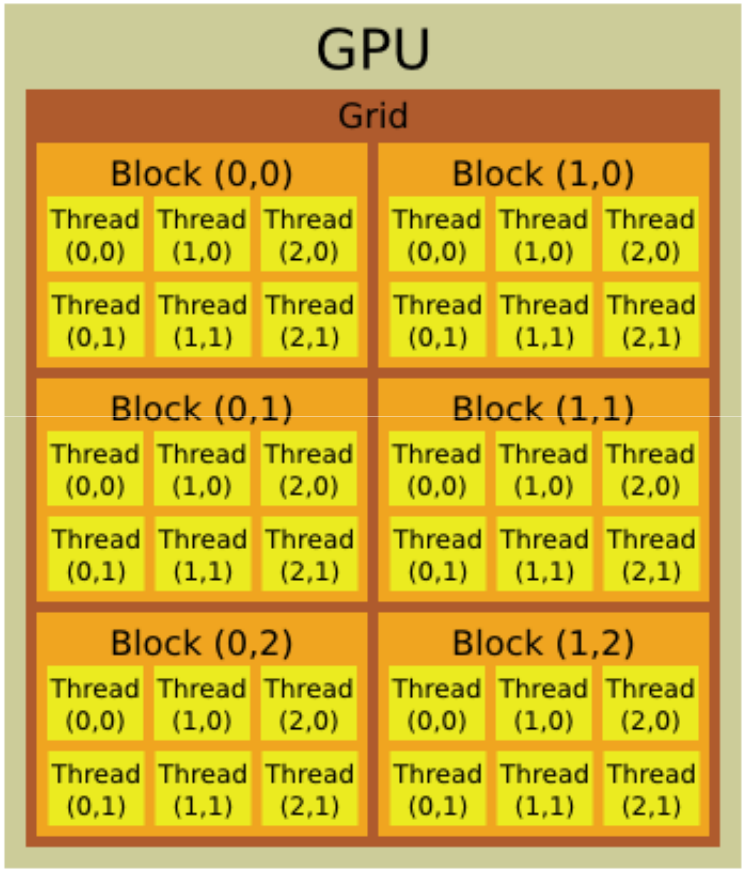
  - ***Grid***: vector 1D ó 2D de *thread blocks*
    - Cada *thread block* tiene su **BID (X,Y)** dentro del *grid*

  - ***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 realizar, como en OpenMP
  - *Single Program Multiple Data* (SPMD)

Arquitectura hardware y software





Arquitectura hardware y software

- Ejemplo: cálculo de **y=α·x+y**, donde **x** e **y** son vectores

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

```
/* Llamada código secuencial */
saxpy_serial(n, 2.0, x, y);
```

Arquitectura hardware y software

- Ejemplo: cálculo de **y=α·x+y**, donde **x** e **y** son vectores

```
__global__ /* Código GPU */
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
  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*):**
  - Repartidos entre todos los *thread blocks* en ejecución
  - Tiempo de acceso muy pequeño
- **Memoria compartida (*shared memory*):**
  - Repartida entre todos los *thread blocks* en ejecución
  - Compartida por todos los *threads* de cada *thread block*
    - Almacenamiento de datos temporales a modo de cache
  - Tiempo de acceso similar a los registros

Modelo de memoria

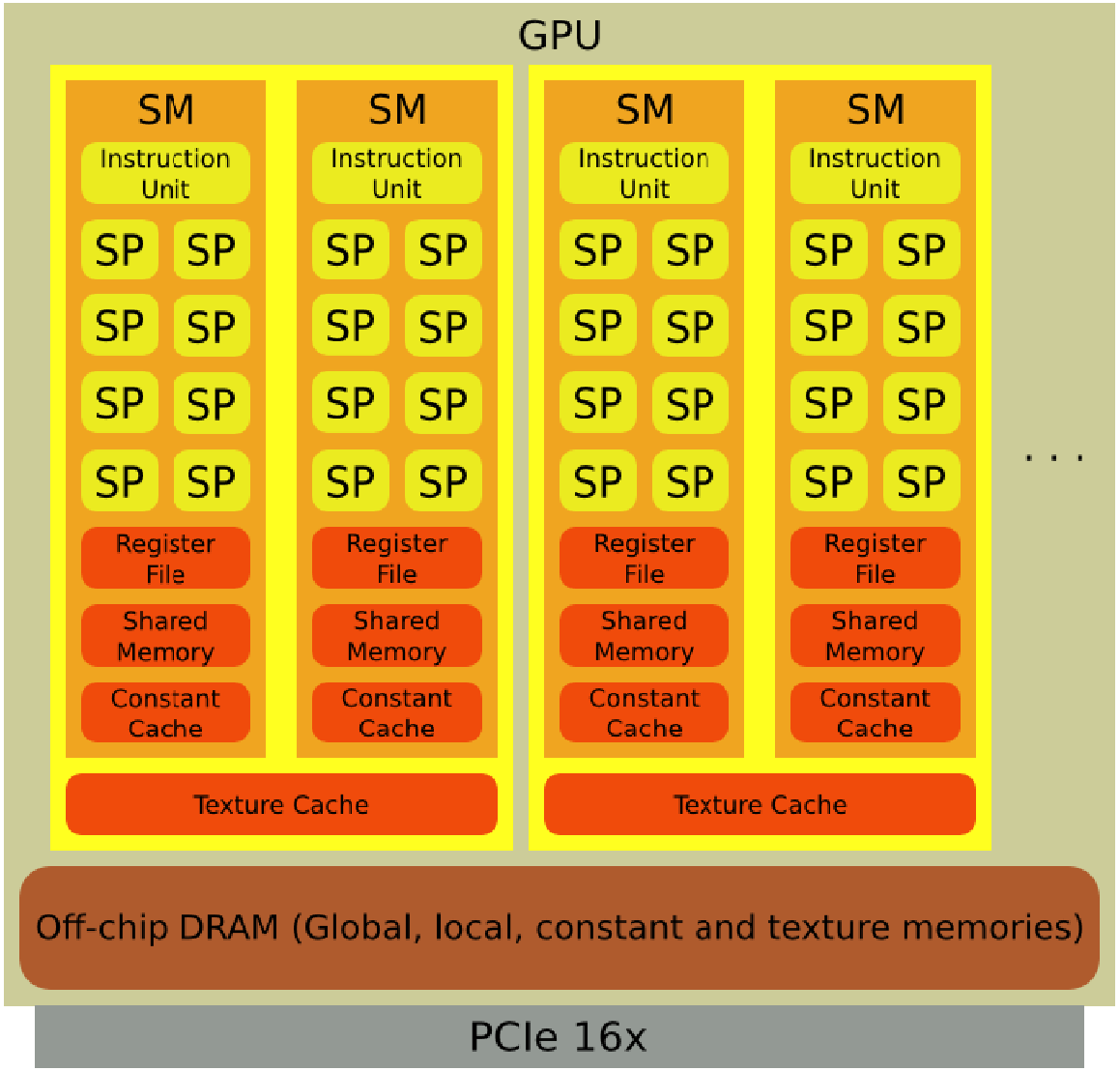
- **Memoria global (*global memory*):**
  - compartida por todos los *thread blocks*
  - Tiempo de acceso elevado (cientos de ciclos)
- **Memoria local (*local memory*)**
  - Memoria privada de cada *thread* para la pila y las variables locales con propiedades similares a la memoria global

Modelo de memoria

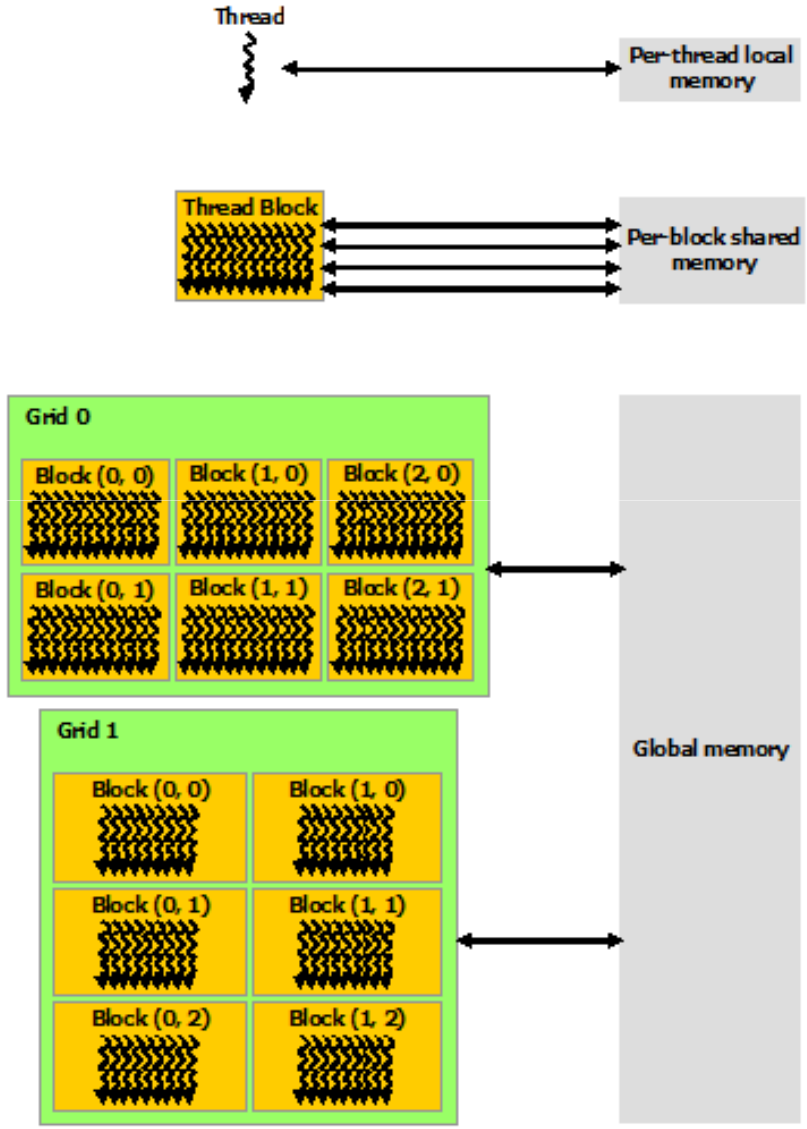
- **Memoria constante (*constant memory*):**
  - Todos los *threads* de un *warp* pueden leer el mismo valor de la memoria constante simultáneamente en un ciclo de reloj
  - Tiempo de acceso similar a los registros
  - Sólo admite operaciones de lectura

- **Memoria de texturas (*texture memory*):**
  - Explota localidad espacial con vectores 1D ó 2D
  - Tiempo de acceso elevado pero menor que memoria global
  - Sólo admite operaciones de lectura

Modelo de memoria



Modelo de memoria



Ejemplo 0

- Conectar a la máquina `tesla.inf.um.es` (SSH)
  - Con usuario: `pam2014-X` y clave: `pam2014-X.T1`
  - Donde X ∈ [1..4]
    - Francisco Herrera → pam2014-1
    - Luis Lorente → pam2014-2
    - Alberto Martínez → pam2014-3
    - Myriam Valera → pam2014-4
  - **`ssh pam2014-X@tesla.inf.um.es`**
- Cambiar clave:
  - **`pa`**
- Copia
  - **`mkdir cuda-ejemplos`**
  - **`cp /home/javiercm/alumnos-cuda-ejemplos/* cuda-ejemplos/. –R`**

OCULTA

Ejemplo 0

- Compilar ejemplo:

```
cd cuda_5.0_ejemplo0
make
```

- Ejecutar ejemplo:

```
./deviceQuery
```

- Editar ejemplo:

```
vim|joe 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)
```

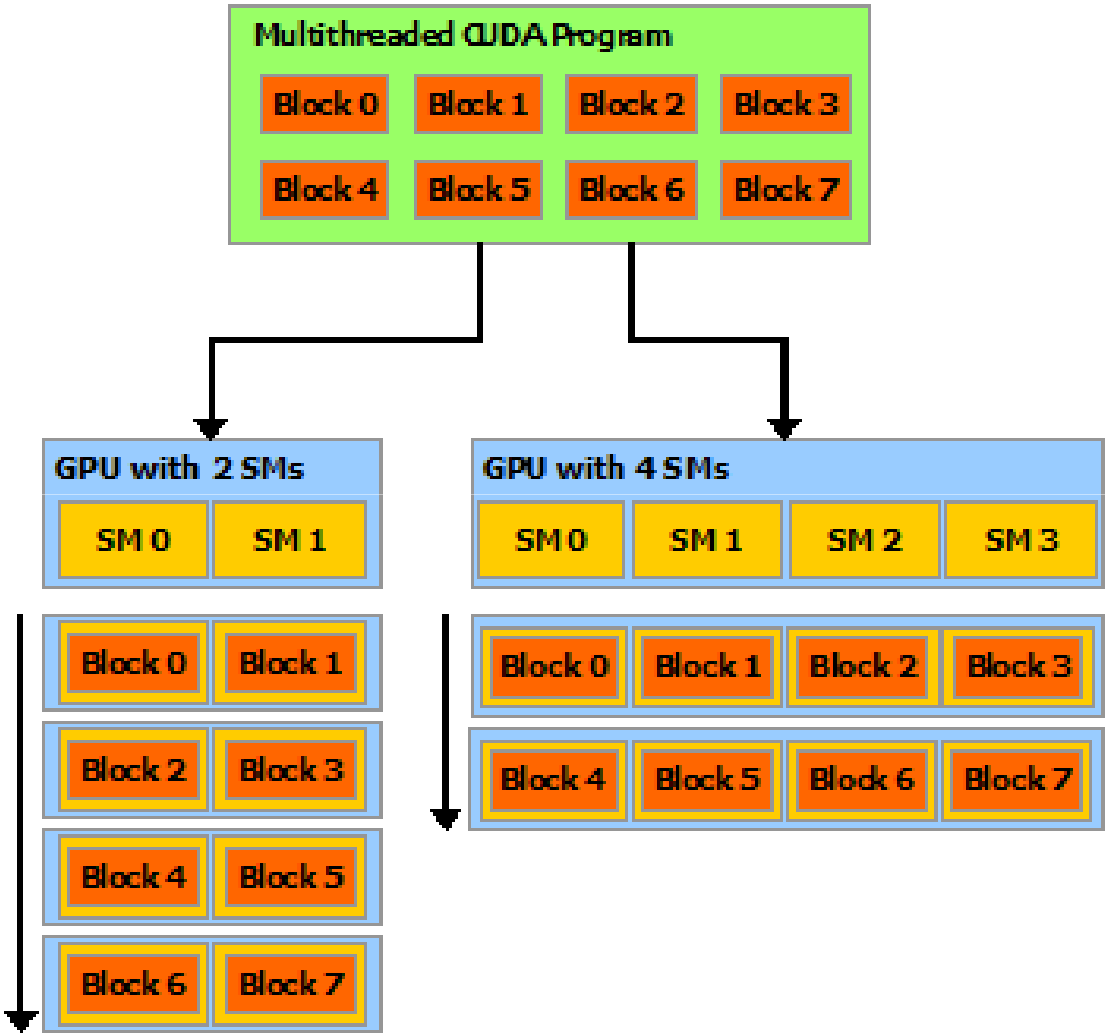
Ejemplo 0

```
Device 0: "GeForce GTX 480"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 1536 MBytes (1610285056 bytes)
  (15) Multiprocessors x ( 32) CUDA Cores/MP:    480 CUDA Cores
  GPU Clock rate:                                1401 MHz (1.40 GHz)
  Memory Clock rate:                             1848 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Max Texture Dimension Size (x,y,z)             1D=(65536), 2D=(65536,65535),
   3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(16384) x 2048, 2D=(16384,16384) x 2048
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
   Support host page-locked memory mapping:       Yes
   Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously)
```

Modelo de ejecución

- Cada *thread block* de un *grid* se asigna a un solo SM
- Cada SM asigna a cada *thread block* en ejecución (activo) todos los recursos necesarios
  - *Thread contexts*, registros, *shm*, etc.
- Un SM puede gestionar y ejecutar hasta 1536 *threads*
  - 12 *thread blocks* de 128 *threads*, 8 *thread blocks* de 192 *threads* , 6 *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



Modelo de ejecución

- Los *threads* de distintos *thread blocks* sólo se comunican vía memoria global y no se sincronizan
  - 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 en una GPU con CUDA
- Cada *thread block* se divide en grupos de 32 threads denominados ***warps***

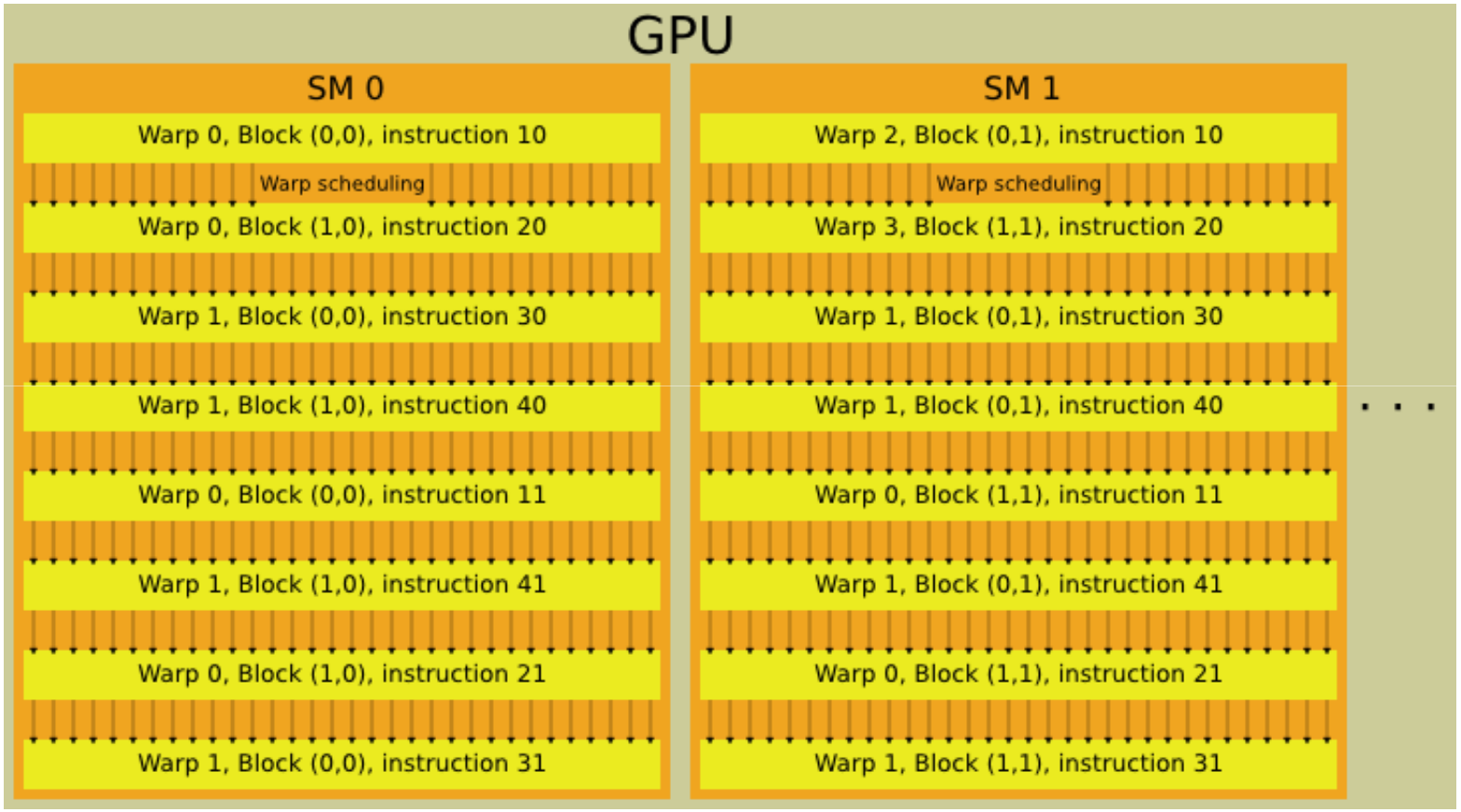
Modelo de ejecución

- Cada SM crea, planifica y ejecuta hasta 48 *warps* (32 threads por warp) de uno o más *thread blocks* (1536 *threads*)
- Cada *warp* ejecuta una instrucción en 1 ciclo reloj
- Cuando un *warp* se bloquea, el SM ejecuta otro *warp* perteneciente al mismo o a otro *thread block* activo
  - Ocultación de largas latencias de acceso a memoria

Modelo de ejecución

- Si los *threads* de un *warp* divergen (por ejemplo, con un 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

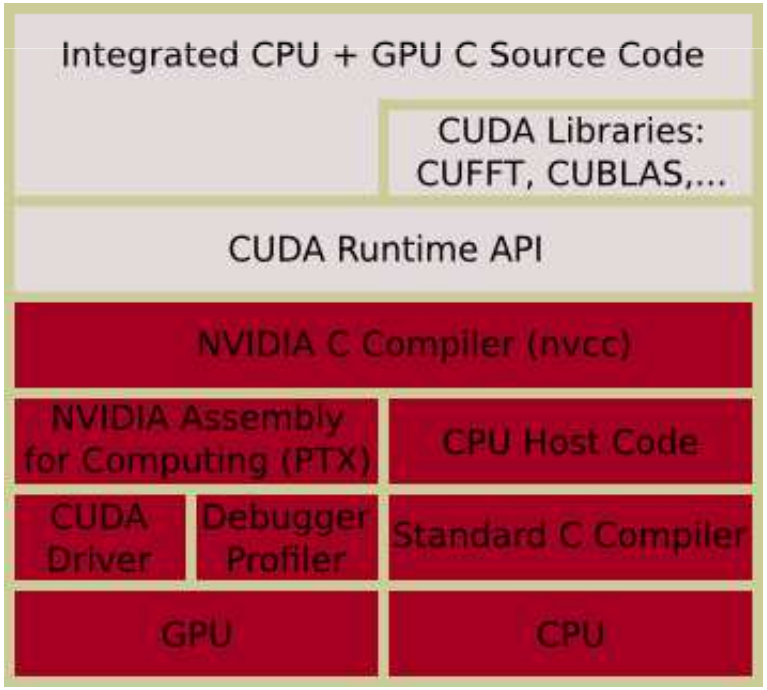
Modelo de ejecución



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 código CPU/GPU
  - Compilador convencional para el código de la CPU



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 → un kernel
    - `__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(1,2)`
      - `int2 var_int2 = make_int2(1,2);`
      - `float4 var_float4 = make_float4(1.0,2.0,3.0,4.0)`
    - **Acceso y modificación**: `variable.x|y`
      - `var_int2.x = 1; var_int2.y = 2`
    - En el código del *host*: Las variables vectoriales deben estar **alineadas** al tamaño de su tipo base
    - En el código del *device*: Las variables vectoriales deben estar **alineadas** al tamaño completo del vector
      - La dirección `int2 var_int2` debe ser múltiplo de 8
    - **`dim3`** equivale a `uint3` → 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 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.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.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 vectores

- Compilar ejemplo:

```
cd cuda_5.0_ejemplo1
make
```

- Ejecutar ejemplo:

```
./vectorAdd
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Done
```

- Editar ejemplo:

```
vim|joe vectorAdd.cu
```

Ejemplo 1: Suma de vectores

- Código

```
/* CUDA kernel device code */
__global__ void vectorAdd(const float *A, const float *B, float *C, int numElements)
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   if (i < numElements)
     {        C[i] = A[i] + B[i];    }
}


////////////////////////////////////////////////

/* Llamada código paralelo desde código CPU */

int numElements = 50000;
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
```

Modelo de programación

- Esquema general de una aplicación con CUDA:
  - Reservar memoria en la GPU

```
err = cudaMalloc((void **)&d_A, size);
```

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

```
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
```

  - Ejecutar uno o más *kernels*

```
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
```

Modelo de programación

- Esquema general de una aplicación con CUDA:
  - Realizar otras tareas

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

    ```
    err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    ```
  - Liberar memoria de la GPU

    ```
    err = cudaFree(d_A);
    ```
  - Resetear la GPU

    ```
    err = cudaDeviceReset();
    ```

Ejemplo 2: Esquema general

- Compilar ejemplo:

```
cd cuda_5.0_ejemplo2
make
```

- Ejecutar ejemplo:

```
./cuda_template -gsx=X -gsy=Y -bsx=X -bsy=Y
```

- Editar ejemplo:

```
vim|joe cuda_template.cu
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 los threads del mismo bloque (opcional)
    - `__syncthreads();`
  - Procesar los datos en memoria compartida
    - `shared_mem[tidb] = (tidg + constante_d[...]);`
  - Sincronizar los threads del mismo bloque (opcional)
    - `__syncthreads();`
  - Mover datos desde memoria compartida → memoria global
    - `gid_d[tidg] = shared_mem[tidb];`

Ejemplo 3: Reducción de un vector

- Compilar ejemplo:

```
cd cuda_5.0_ejemplo3
make
```

- Ejecutar ejemplo:

```
./cuda_vectorReduce -n=N -bsx=X
```

- Editar ejemplo:

```
vim|joe cuda_vectorReduce.cu
vim|joe cuda_vectorReduce_kernel.cu
```

Ejemplo 3: Reducción de un vector

- 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 de un vector

- 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();

   // write result for this block to global memory
   if (tidb == 0) {
     reduce_d[blockIdx.x] = sdata[0];
   }

 }
```

Ejemplo 3: Reducción de un vector

- 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
   }
}
```