Contenidos

2.4. Modelo de ejecución

- Cada *thread block* de un *grid* se asigna a un solo SM

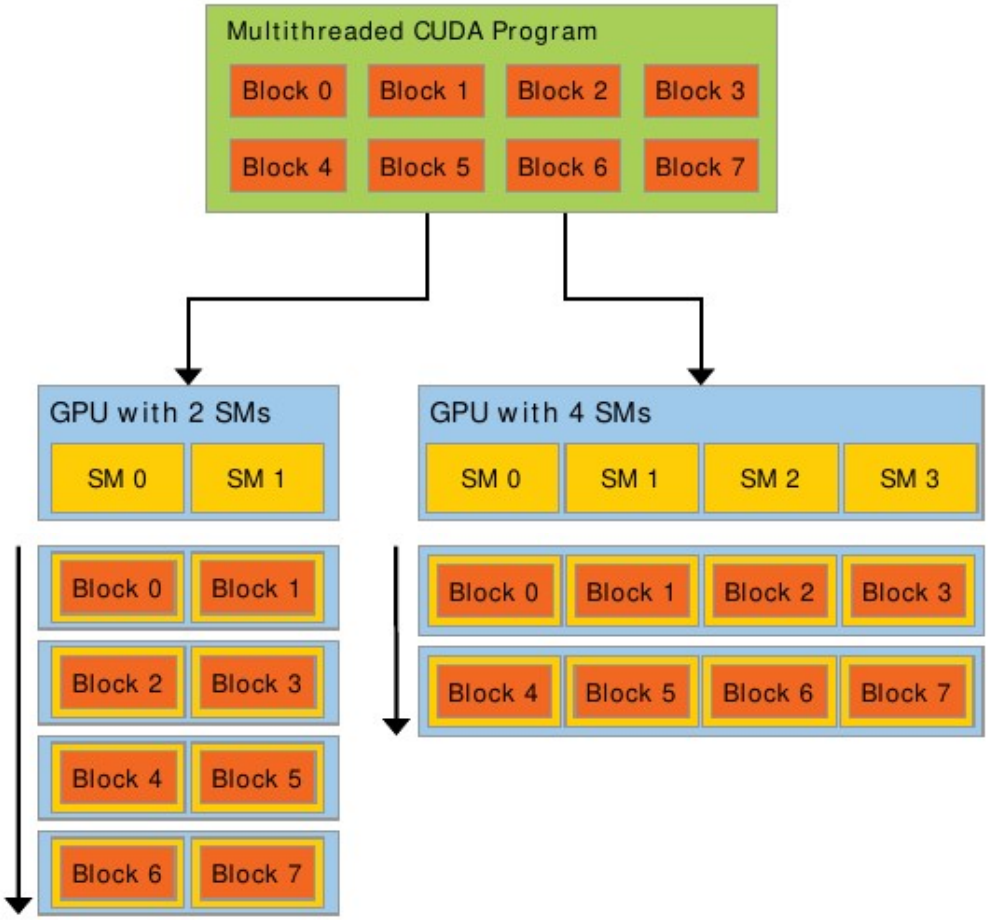
- Cada SM asigna a cada *thread block* en ejecución (activo) todos los recursos necesarios
  - *Thread contexts*, registros, *shared memory*, etc.

- Número de threads que puede gestionar un SM:
  - (2.x) → hasta 1536 threads = 48 warps
  - (3.x en adelante) → hasta 2048 threads = 64 warps

- 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();`

2.4. Modelo de ejecución



A GPU is built around an array of Streaming Multiprocessors (SMs) (see Hardware Implementation for more details). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

2.4. 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
    - favorece la escalabilidad
    - limita el rango de aplicaciones para paralelizar con éxito en GPU con CUDA
- Cada *thread block* se divide en grupos de 32 threads → **1 *warp***

2.4. Modelo de ejecución

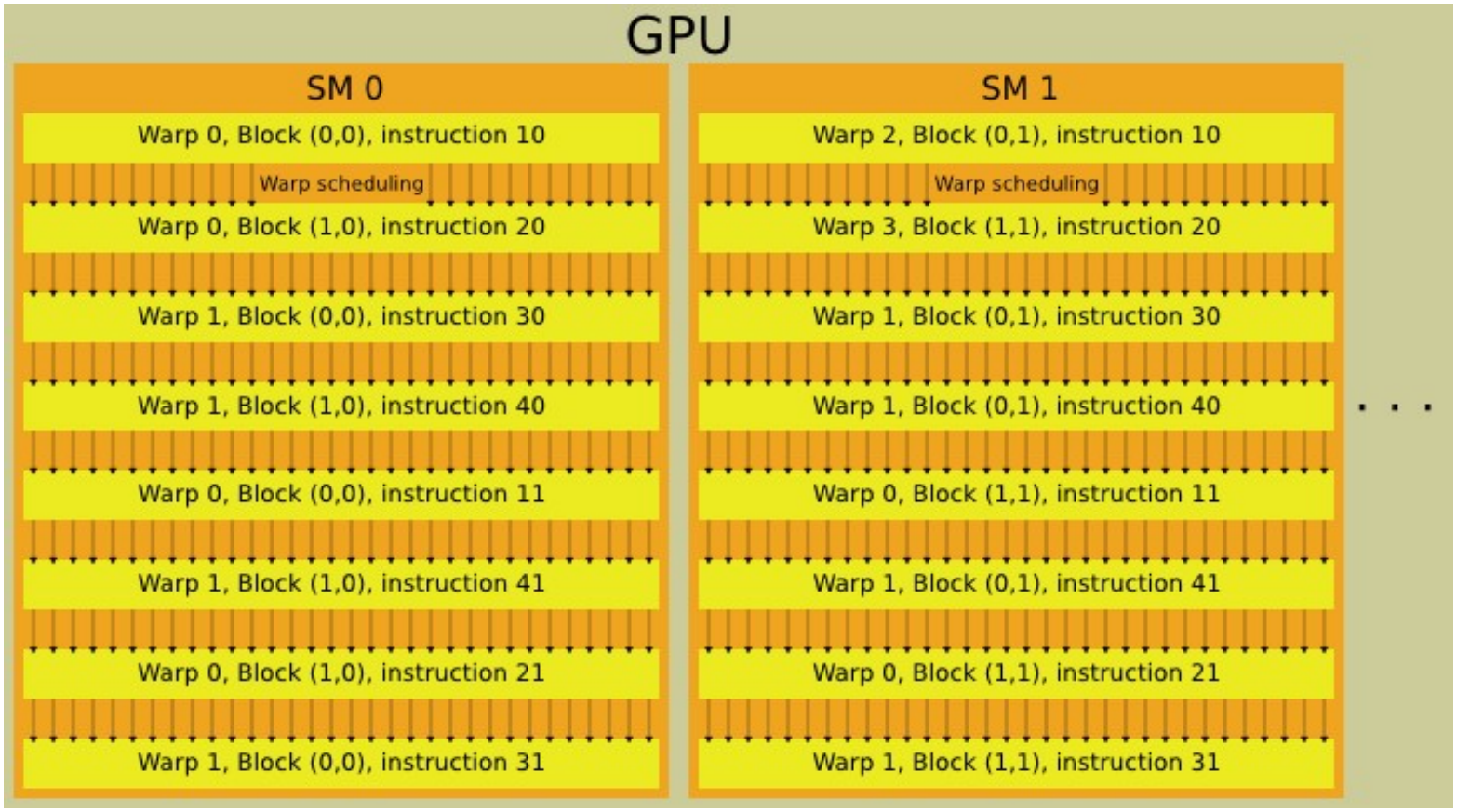
- **(2.x) -> Cada SM:**
  - Crea, planifica y ejecuta hasta 48 *warps* (pertenecientes a uno o más *thread blocks*)
  - TOTAL: 1536 *threads activos en un momento dado*

- **(3.x en adelante) -> Cada SM:**
  - crea, planifica y ejecuta hasta 64 *warps* (pertenecientes a uno o más *thread blocks*)
  - TOTAL: 2048 *threads activos en un momento dado*

- Los 32 threads de un *warp* ejecutan misma instrucción en 1 ciclo reloj

- Cuando un *warp* se bloquea:
  → el correspondiente warp scheduler del SM ejecuta otro *warp* de cualquier *thread block* activo
  → Ocultación de largas latencias de acceso a memoria

2.4. Modelo de ejecución



GPUs actuales: más de un *scheduler* por SM. Por lo que cada SM:

1. Distribuye estáticamente sus *warps* entre sus *schedulers*
2. En cada ciclo, cada *scheduler* lanza 1 instrucción para uno de sus *warps* que esté preparado

2.4. Modelo de ejecución

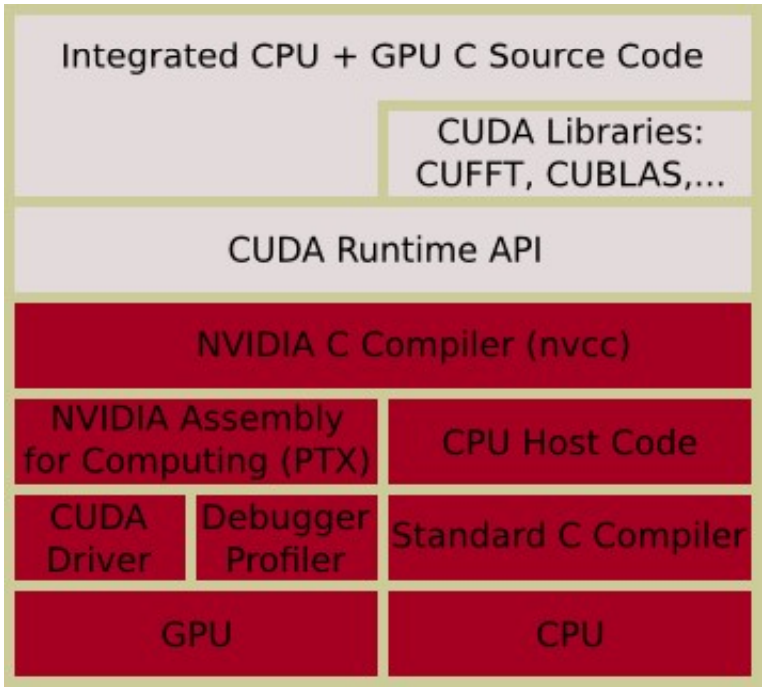
- Si los *threads* de un *warp* divergen (por ejemplo, con un salto condicional), su ejecución se serializa de forma que en cada rama…
  - Todos los *threads* (SPs) ejecutan la instrucción leída por la IU inhabilitando los *threads* que siguieron otra rama distinta
  - Con degradación de rendimiento en muchos casos
- …hasta que todos convergen
- ***Independent Thread Scheduling :***
  - A partir de Volta (7.0) los *threads* de *warp* pueden diverger, manteniendo ejecución simultánea

Contenidos

2.5. Modelo de programación

- Código fuente integrado para CPU/GPU
  - Extensiones del lenguaje C/C++ (sólo C para GPU)
  - CUDA *Runtime* API (librería de funciones)

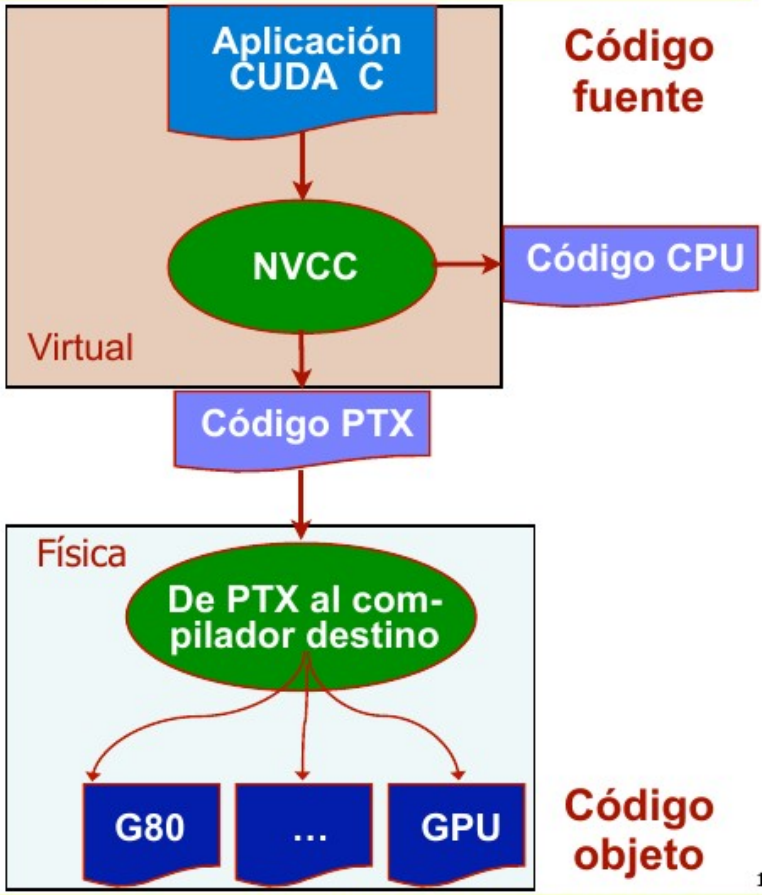
- NVIDIA C Compiler (`nvcc`) separa código CPU/GPU
  - Compilador convencional para el código de la CPU



2.5. Modelo de programación

Los diferentes módulos de compilación

- El código fuente CUDA se compila con NVCC.
  - NVCC separa el código que se ejecuta en CPU del que lo hace en GPU.
- La compilación se realiza en dos etapas:
  - Virtual: Genera código PTX (Parallel Thread eXecution).
  - Física: Genera el binario para una GPU específica (o incluso para una CPU multicore).

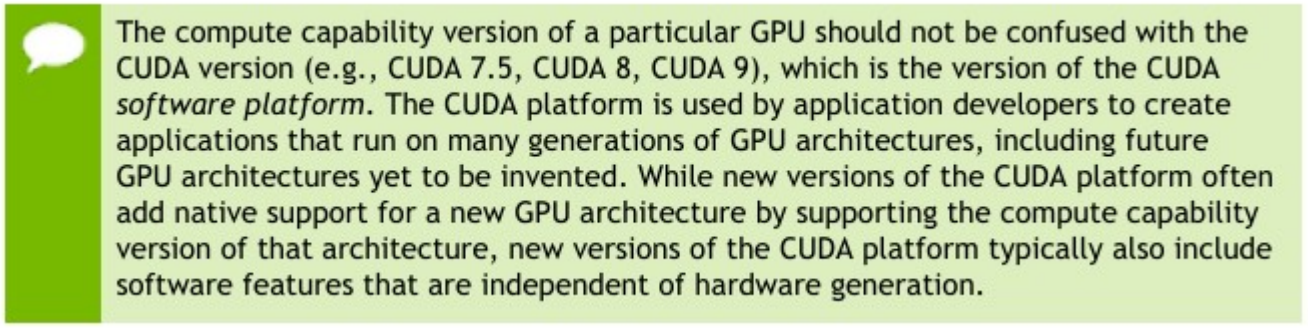


Manuel Ujaldon - Nvidia CUDA Fellow

2.5. Modelo de programación

Supported CUDA level of GPU and card. See also at Nvidia:

- CUDA SDK 6.0 support for compute capability 1.0 – 3.5. (Tesla, Fermi, Kepler)
- CUDA SDK 6.5 support for compute capability 1.1 – 5.x (Tesla, Fermi, Kepler, Maxwell). Last version with support for compute capability 1.x (Tesla)
- CUDA SDK 7.5 support for compute capability 2.0 – 5.x (Fermi, Kepler, Maxwell)
- CUDA SDK 8.0 support for compute capability 2.0 – 6.x (Fermi, Kepler, Maxwell, Pascal). Last version with support for compute capability 2.x (Fermi)
- CUDA SDK 9.0/9.1/9.2 support for compute capability 3.0 – 7.2 (Kepler, Maxwell, Pascal, Volta)
- CUDA SDK 10.0 support for compute capability 3.0 – 7.5 (Kepler, Maxwell, Pascal, Volta, Turing)



2.5. Modelo de programación

- **Extensiones del lenguaje C/C++**

  Declaración de funciones

  `__host__`

  Función ejecutada por CPU y llamada desde código secuencial

  Por defecto si no se especifican los anteriores

  `__global__`

  Función ejecutada por GPU y llamada desde código secuencial (un ***kernel***)

  `__device__`

  Función ejecutada por GPU y llamada desde *kernels* (funciones auxiliares)

2.5. Modelo de programación

- **Extensiones del lenguaje C/C++**

Declaración de variables

`__device__`

Variable residente en memoria global accesible por todos los *threads* de cualquier *thread block* durante el tiempo de vida de aplicación

`__constant__`

Variable que reside en memoria constante accesible por todos *threads* de cualquier *thread block* 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*.

Definición de tamaño:

- Estáticamente: en tiempo de compilación.
- Dinámicamente: en tiempo de ejecución.

2.5. Modelo de programación

- **Extensiones del lenguaje C/C++**

  Declaración de variables

  `__shared__`

  Estáticamente: en tiempo de compilación. Por ejemplo:

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

  Dinámicamente: en tiempo de ejecución. Por ejemplo:

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

  el tamaño (en bytes requeridos) se pasará como parámetro de configuración al kernel en su llamada desde CPU:

```
a_size=256;
kernel<<< gridDim, blockDim, a_size >>>();
```

2.5. 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 (2 enteros de 4 bytes)
    - El tipo **`dim3`** equivale a `uint3`
      - --> se usar para especificar las dimensiones de *grids* y *thread blocks*

2.5. 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`**: cantidad de *threads* en un *warp*

2.5. Modelo de programación

- **Extensiones del lenguaje C/C++**
  - ***Intrinsics***

`__syncthreads()`

Sincronización de los *threads* de un *thread block*

`__threadfence_block()`

Todas las escrituras a memoria realizadas por T, el thread que la invoca, antes de esta llamada se observan por todos los restantes threads del bloque tal como que han ocurrido antes de todas las escrituras que T hace tras la llamada.

Todas las lecturas de memoria realizadas por T antes de la llamada se ordenan antes de todas las lecturas realizadas por T después de la llamada.

`__threadfence()`

Similar a `__threadfence_block()` pero además, en relación a la memoria global, afecta a todos los *threads* del dispositivo

2.5. Modelo de programación

Example: thread 1 executes writeXY(), thread 2 executes readXY().

```
__device__ int X = 1, Y = 2;


__device__ void writeXY()
{
          X = 10;
          __threadfence();
          Y = 20;
}


__device__ void readXY()
{
          int B = Y;
          __threadfence();
          int A = X;
}
```

For this code, the following outcomes can be observed:

- **A equal to 1 and B equal to 2,**
- **A equal to 10 and B equal to 2,**
- **A equal to 10 and B equal to 20.**

The fourth outcome (▸ **A equal to 1 and B equal to 20**) is not possible, because the first write must be visible before the second write.

2.5. 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);
}
```

2.5. 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

2.5. 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)

Contenidos

2.5. Modelo de programación
Ejemplo 1: Suma de vectores

- Compilar ejemplo:

```
make
```

- Ejecutar ejemplo:

```
./vectorAdd
[Vector addition of 4096 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 16 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

- Editar ejemplo:

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

2.5. Modelo de programación
Ejemplo 1: Suma de vectores

```
/* 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 = 4096;
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;

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

2.5. Modelo de programación
Ejemplo 1: Suma de vectores

Esquema general de una aplicación con CUDA (vista desde CPU):

- 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);
```

- La CPU se pone a 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();
```

Contenidos

2.5. Modelo de programación
Ejemplo 2

- Compilar ejemplo:

```
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
```

2.5. Modelo de programación
Ejemplo 2

- Código del kernel

```
__constant__ int constante_d[CM_SIZE];

__global__ 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];
}
```

- Código secuencial en la CPU

```
foo<<<grid, block, shared_mem_size>>>(gid_d);
```

2.5. Modelo de programación
Ejemplo 2

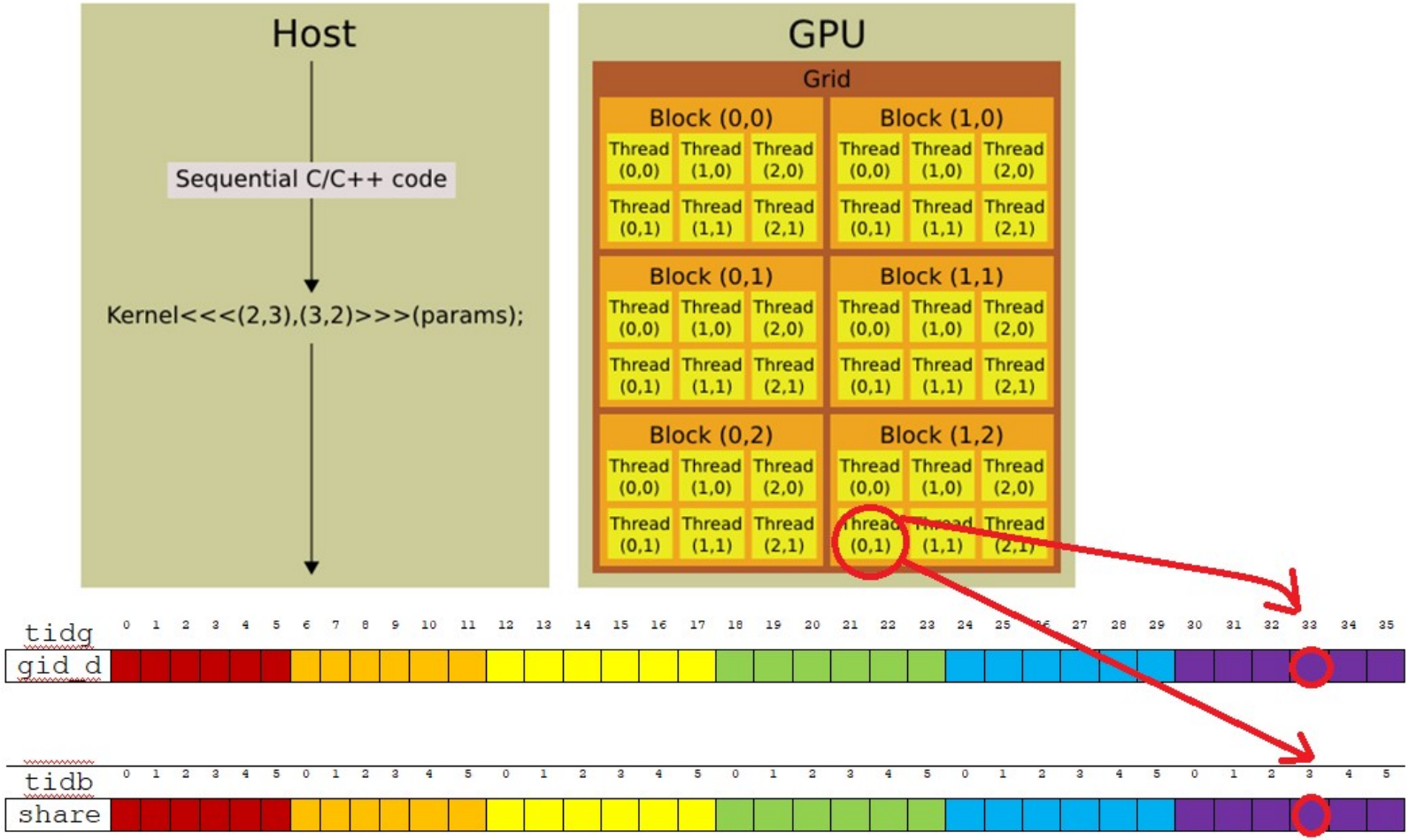
- **Esquema general de un *kernel* (1/2):**
  - Calcular GID a partir de BID y TID

```
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);
```

2.5. Modelo de programación
Ejemplo 2

Calcular GID a partir de BID y TID

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



2.5. Modelo de programación
Ejemplo 2

- **Esquema general de un *kernel* (2/2):**
  - 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];`

Contenidos

2.5. Modelo de programación
Ejemplo 3: Reducción de un vector

- Compilar ejemplo:

```
make
```

- Ejecutar ejemplo:

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

- Editar ejemplo:

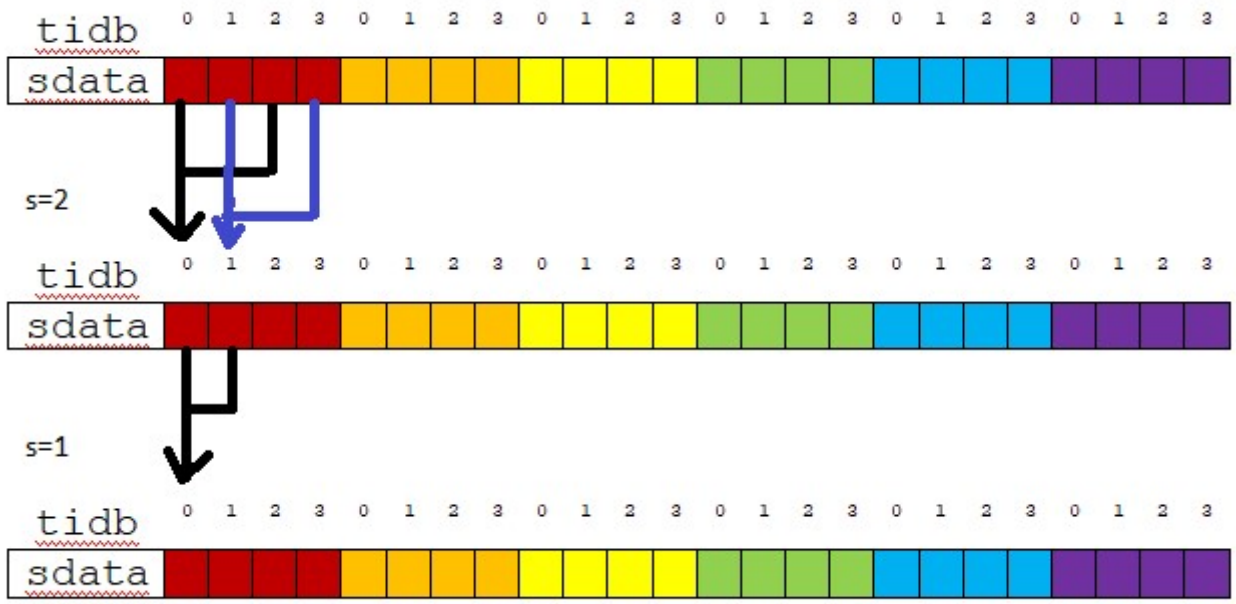
```
vim|joe cuda_vectorReduce.cu
vim|joe cuda_vectorReduce_kernel.cu
```

2.5. Modelo de programación
Ejemplo 3: Reducción de un vector

```
__global__ void vectorReduce(float *vector_d, float *reduce_d, int n)
{
   extern __shared__ int sdata[];
   unsigned int tidb = threadIdx.x; // thread ID in thread block
   unsigned int tidg = blockIdx.x * blockDim.x + threadIdx.x; // thread ID in grid

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

   // 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];
}
```

2.5. Modelo de programación
Ejemplo 3: Reducción de un vector

```
// 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();
}
```



2.5. Modelo de programación
Ejemplo 3: Reducción de un vector

```
. . .
 // execute the kernel
  vectorReduce<<<grid, block, block.x * sizeof(float)>>>(vector_d, reduce_d, n);

. . .
 cudaMemcpy(reduce_h, reduce_d, grid.x * sizeof(float), cudaMemcpyDeviceToHost);

. . .
```

```
 //compute final stage
 for(int i = 1; i < grid.x; i++)
     reduce_h[0] += reduce_h[i];
```

2.5. Modelo de programación
Ejemplo 3: Reducción de un vector

- Código del kernel

(una posible alternativa reducción final atómica: evita etapa final de reducción en CPU)

```
    . . .

    // write result for this block to global memory + reduction

    if (tidb == 0) {
       atomicAdd(reduce_d,sdata[0]); // Compute Capability ≥ 1.1
    }
}
```