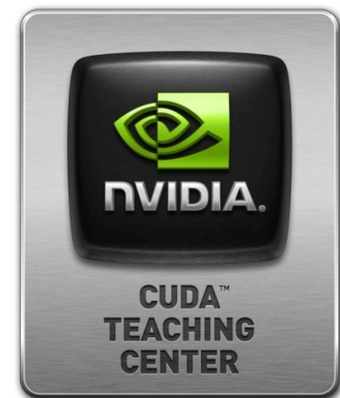
Programación masivamente paralela en procesadores gráficos (GPUs)

E. Dufrechou , P. Ezzatti y M. Pedemonte



Clase 5
Programación CUDA

Contenido

- **Modelo de programación CUDA**
  - **Introducción**
- **Programación en CUDA**
  - **Conceptos básicos**

Modelo de programación CUDA

Introducción

El device:

- Es un coprocesador de la CPU (host).
- Posee su propia memoria DRAM (memoria del device) –lógicamente ya no, hay dispositivos que comparten la memoria físicamente).
- Ejecuta "muchos" threads en paralelo.
- Generalmente es una GPU, pude ser otro tipo de procesador.

Una porción de los cálculos con paralelismo de datos se computan con kernels en el device que corren en "muchos" threads.

Introducción

Diferencias entre los threads de la GPU y CPU

- **Los threads en GPU son extremadamente livianos :**

  Muy poco overhead para crearlos.

- **La GPU necesita 1000s de threads para alcanzar la eficiencia:**

  Generalmente las CPU (multi-core) necesitan unos pocos.

Introducción

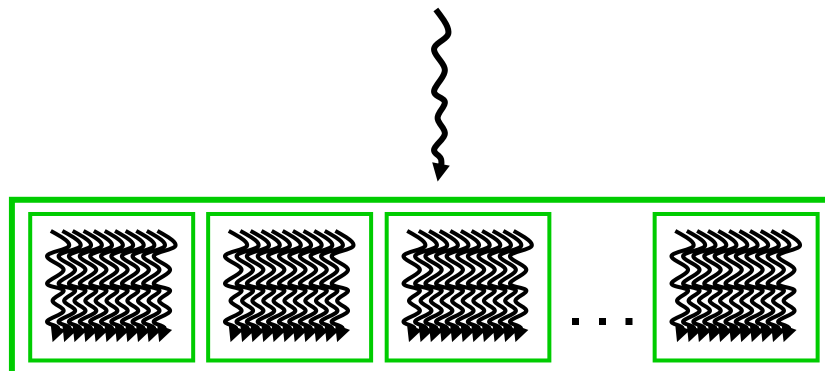
Se integra código en el host con código en la tarjeta:

- **El código del host puede ser secuencial o paralelo.**
- **En la tarjeta el código es altamente paralelo (SPMT).**

Código en el host (sequencial)

Kernel device (paralelo)

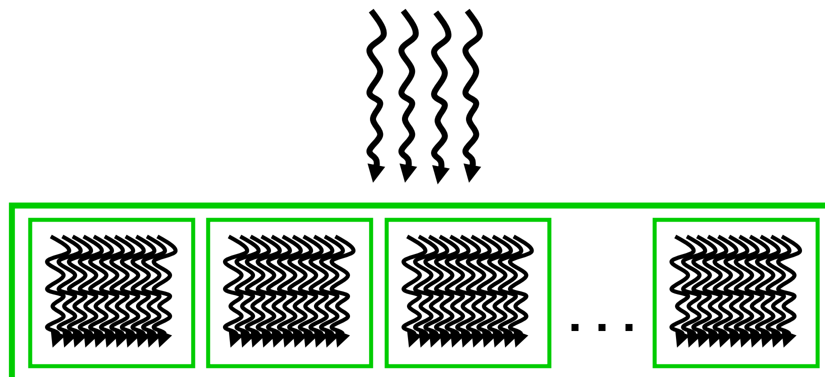
KernelA<<< nBlk, nTid >>>(args);



Código en el host (paralelo)

Kernel device (paralelo)

KernelB<<< nBlk, nTid >>>(args);



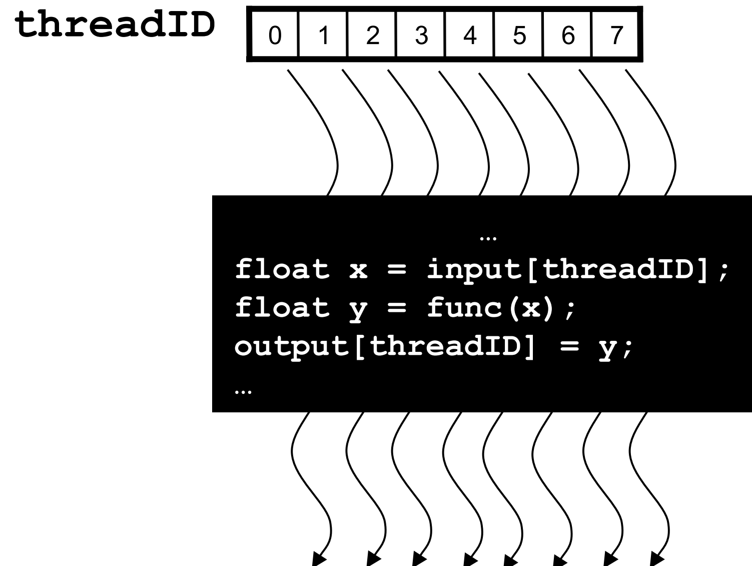
Introducción

Algoritmo básico

1. Instrucciones en el host
2. Se envían los datos del host a la GPU
3. Se procesa en GPU
4. Se recuperan los datos
5. Continúan instrucciones en el host

Array de threads

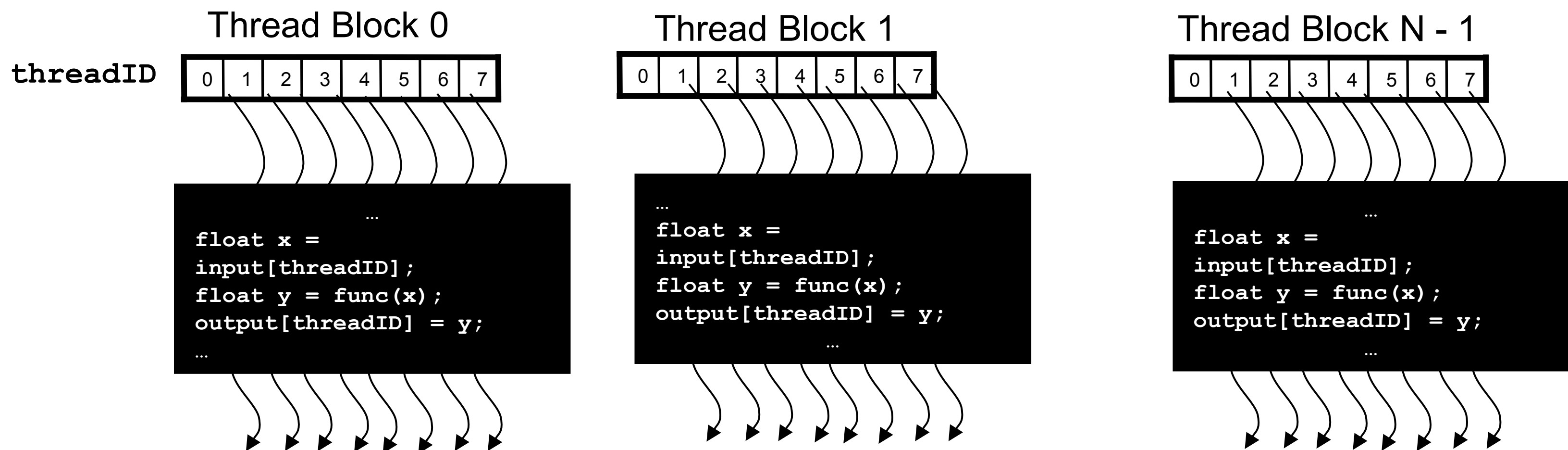
- Un kernel es ejecutado por un array de threads
  - Todos los threads corren el mismo código (SPMT)
  - Cada thread tiene un ID que se puede utilizar para computar direcciones de memoria y tomar decisiones de control



Introducción

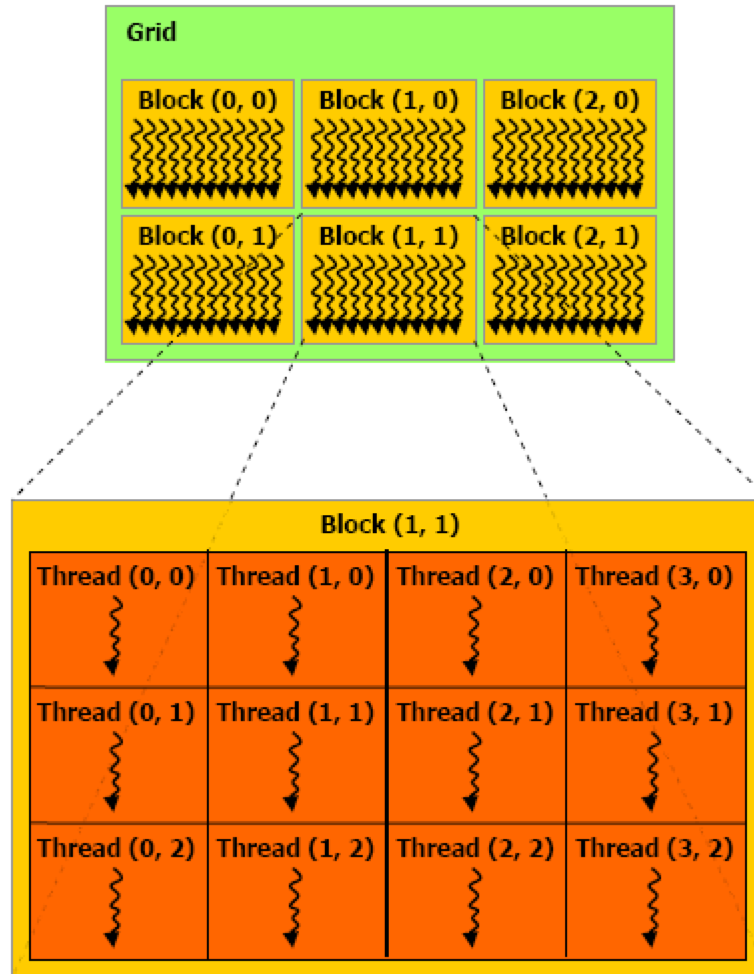
Bloques de threads

- Facilita la escalabilidad.
- Permite dividir un array de threads en múltiples bloques:
  - Threads en un mismo bloque cooperan via **memoria shared, operaciones atomic** y **barrier de sincronización.**
  - Threads en diferentes bloques no pueden cooperar directamente (operaciones atomic en mem global).



Introducción

Jerarquía de threads

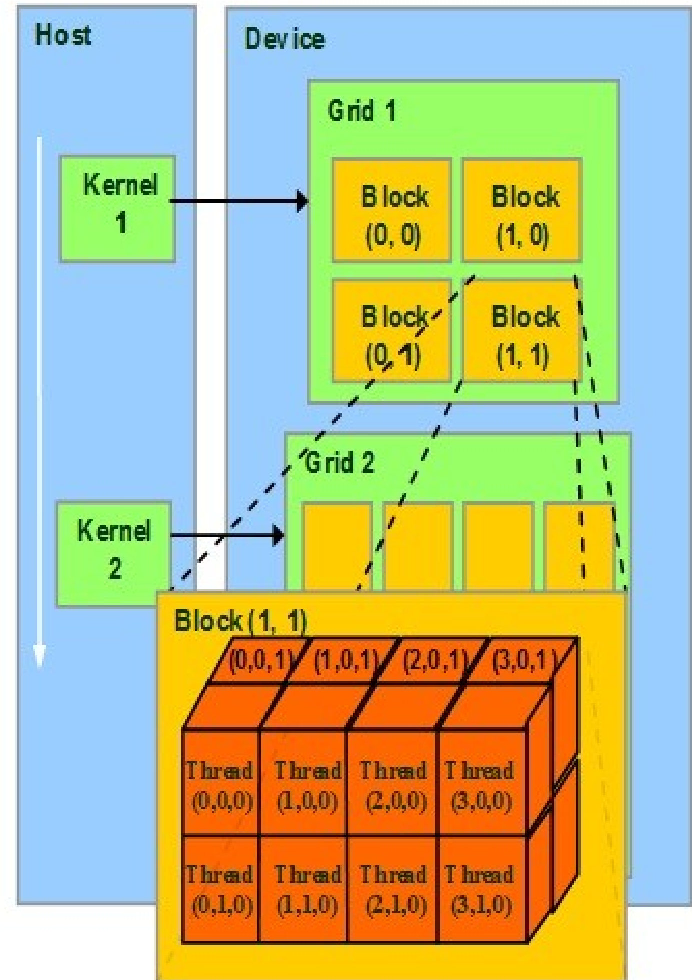


Introducción

IDs bloque e IDs de thread

- Cada thread usa el IDs para decidir sobre que datos trabajar
  - Block ID: 1D, 2D o 3D (3D desde Fermi)
  - Thread ID: 1D, 2D o 3D

- Puede simplificar el direccionamiento de memoria
  - Procesamiento de imágenes.
  - Resolución de PDEs (regulares) en 3D.



Introducción

Jerarquía de threads

- **Los bloques dentro de un grid tienen que ser independientes.**
- **Los bloques de threads son ejecutados en cualquier orden y en diferentes multiprocesadores.**
- **En las tarjetas nuevas hay ejecución concurrente de kernels (en el curso no lo vamos a tener en cuenta !!!).**

Programación en CUDA

Programación en CUDA

Las funciones

	Ejecutable en:	Invocable desde:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` define funciones "kernel"
- `__device__` y `__host__` pueden ser utilizados juntas (con compilación condicional).
- `__host__` puede no ponerse.

Programación en CUDA

- Para invocar un kernel es necesario determinar “una configuración de ejecución”.

```
__global__ void KernelFunc(...);

dim3   DimGrid(100, 50);     // 5000 bloques
dim3   DimBlock(4, 8, 8);    // 256 threads por block
size_t SharedMemBytes = 64; // 64 bytes de memoria shared
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

- Las llamadas a kernels son asincrónicas !!

  Es necesario utilizar sincronizaciones explícitas.

Programación en CUDA

Espacios de declaración de variables

Se dispone de distintos espacios (algunos unificados) de declaración:

- global
- _device_ residen en la memoria global del device.
- _shared_ reside en la memoria shared del device.

  extern _shared_ para allocamiento dinámico.
- _constant_ reside en memoria constante del device.

```
__device__ float filter[N];
__global__ void convolve (float *image)
__shared__ float region[M];
extern __shared__ float region[];
```

Programación en CUDA

CUDA posee diversas keywords

- `threadIdx (uint3) se accede con .x/.y/.z`
- `blockIdx (uint3) se accede con .x/.y/.z`
- `blockDim (dim3) se accede con .x/.y/.z`
- `gridDim (dim3) se accede con .x/.y/.z`

```
region[threadIdx.x] = image[i];
```

Programación en CUDA

Funciones intrínsecas

- `cudaDeviceSynchronize()`: sincroniza todos los threads. Ejecuta en el host.
- **`__syncthreads`**: permite sincronizar threads del mismo bloque.

  Se invoca dentro de un kernel.
- `clock_t clock()`: permite medir tiempos en el device.

Funciones matemáticas (más rápidas pero con posible diferencia en el valor):

`__sinf(x), __cosf(x), __log2f(x) …. __powf(x,y)`

Funciones atómicas:

`atomicAdd``(), … atomicMin(), …`

Programación en CUDA

La API ofrece diversas funciones

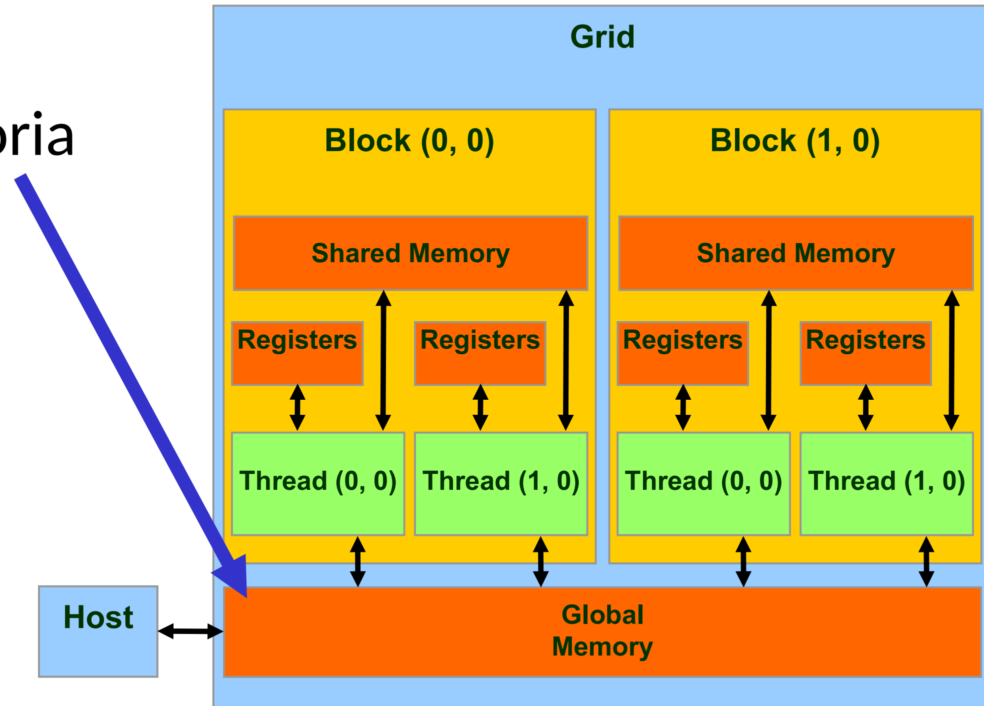
- Para manejo de memoria
- Para transferencias
- Para manejo de las ejecuciones (lanzamiento)

```
// Allocate memoria en GPU
cudaMalloc((void**)& SDEVPTR, bytes)
```

Programación en CUDA

CUDA Device Memory Allocation

- `cudaMalloc()`
  - Reserva espacio en la memoria global del device
  - Dos parámetros
    - Puntero
    - Tamaño de la memoria reservada



- `cudaFree()`
  - Libera la memoria asociada a un puntero en la mem. global del device

La reserva de memoria global se realiza en el host !!!

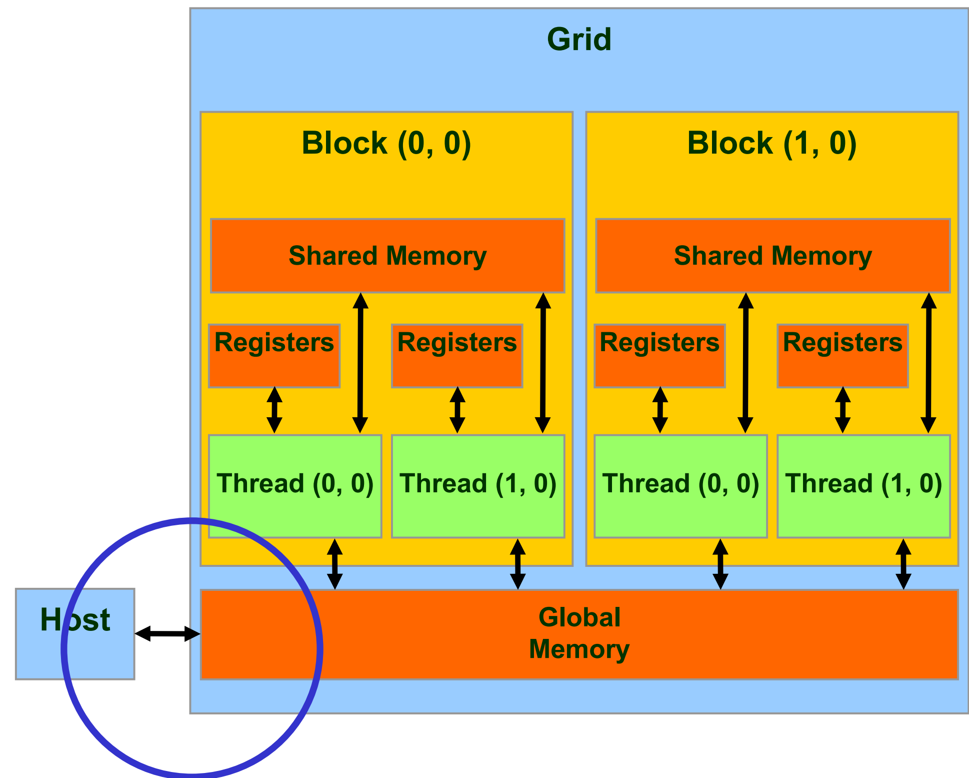
Programación en CUDA

- Código de ejemplo
  - Reservar una matriz de 64 * 64 floats (precisión simple).

```
TILE_WIDTH = 64;
float* SDEVPTR
int size = TILE_WIDTH *TILE_WIDTH* sizeof(float);

cudaMalloc((void**)& SDEVPTR, size);

…..

cudaFree(SDEVPTR);
```

Transferencias de datos Host-Device

- `cudaMemcpy()`
  - Transferencia de datos
  - 4 parámetros
    - Puntero destino
    - Puntero origen
    - Número de bytes a copiar
    - Tipo de transferencia
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device



- **Transferencias asincrónicas**

Programación en CUDA

- Código de ejemplo
  - Transferir una matriz de 64 x 64 floats (precisión simple)
    - **cudaMemcpyHostToDevice** y **cudaMemcpyDeviceToHost** son constantes simbólicas.

```
cudaMemcpy(SDEVPTR, SHPTR, size, cudaMemcpyHostToDevice);

cudaMemcpy(SHPTR, SDEVPTR, size, cudaMemcpyDeviceToHost);
```

Lanzamiento

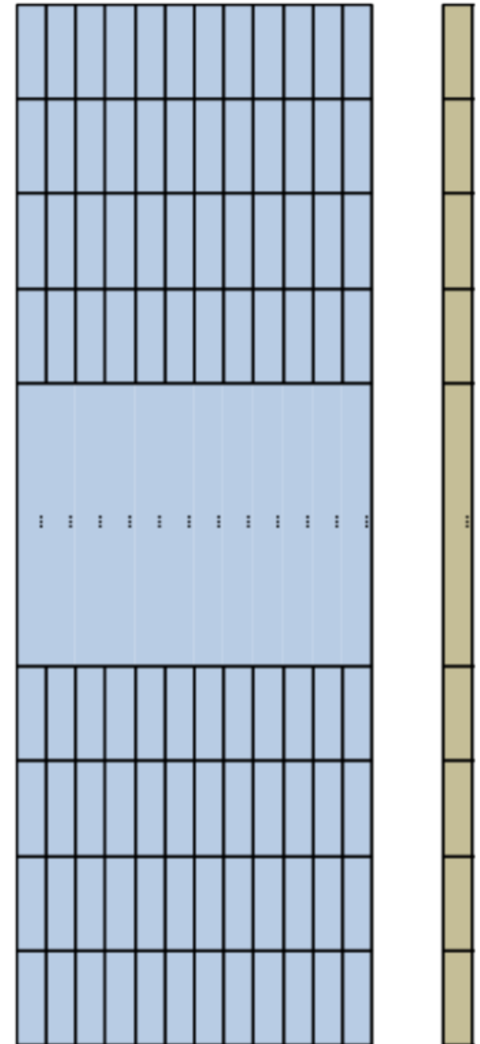
- Dimensiones de la grilla (3D): variable de tipo dim3
- Dimensiones de los bloques de threads (3D): variable de tipo dim3
- Opcionales:
  - cantidad de memoria compartida por bloque
  - identificador de stream

```
dim3 gridSize( 16, 16 );
dim3 blockSize( 32, 32 );
kernel<<<gridSize, blockSize, 0, 0>>>( ... );
kernel<<<32, 512>>>( ... );
```

Programación en CUDA

Retomando los ejemplos de programación paralela básica :

- **Consideramos que se tiene una matriz con número grande de filas y se quiere obtener la suma por filas de la matriz (Almacenado por filas).**



Programación en CUDA

- **Ejemplo, suma por fila de los elementos de una matriz.**

```
void SumaFilaMatriz(int M, int N, float * Mh, float* Xh){
int size = M * N * sizeof(float), size2 = M*sizeof(float);
float* Md, *Xd;
…
// Allocate en device
cudaMalloc((void**)&Md, size);
cudaMalloc((void**)&Xd, size2);

// Inicializo matrices en el device
cudaMemcpy(Md, Mh, size, cudaMemcpyHostToDevice);
cudaMemset(Xd,0, size2);

// Invocar el kernel que suma en GPU

// Traer resultado;
cudaMemcpy(Xh, Xd, size2, cudaMemcpyDeviceToHost);

// Free matrices en device
cudaFree(Md); cudaFree(Xd);
}
```

Programación en CUDA

Kernel

```
// Suma por filas de una matriz
__global__ void SumaFilaMatrizKernel(int N, float* Md, float* Xd){
// Pvalue es usado para el valor intermedio
  float Pvalue = 0;
  int aux = threadIdx.x*N;
  for (int k = 0; k < N; ++k) {
     Pvalue = Pvalue + Md[aux+k];
  }
  Xd[threadIdx.x] = Pvalue;
}
```

Lanzamiento del kernel

```
// configuración de la ejecución
dim3 tamGrid(1, 1); //Grid dimensión
dim3 tamBlock(M,1,1); //Block dimensión
// lanzamiento del kernel
SumaFilaMatrizKernel<<<tamGrid, tamBlock>>>(N, Md, Xd);
```

Restricciones:

- Solo se pueden procesar matrices de tantas filas como threads en un bloque !!!!
- Un solo bloque, un solo multiprocesador computando.

Lanzamiento del kernel (2)

```
// configuración de la ejecución
int bloques = M / 128;
dim3 tamGrid(bloques, 1); //Grid dimensión
dim3 tamBlock(128, 1, 1); //Block dimensión
// lanzamiento del kernel
SumaFilaMatrizKernel<<<tamGrid, tamBlock>>>(N, Md, Xd);
```

Hay que hacer cambios en el Kernel !!!!

Kernel (2)

```
 // Suma por filas de una matriz
  __global__ void SumaFilaMatrizKernel(int N, float* Md, float* Xd){

// Pvalue es usado para el valor intermedio
   float Pvalue = 0;
   int aux = blockIdx.x*blockDim.x + threadIdx.x;
   int aux2 = aux*N;
   for (int k = 0; k < N; ++k) {
      Pvalue = Pvalue + Md[aux2+k];
   }
   Xd[aux] = Pvalue;
}
```

Más threads …

- Que pasa si la matriz tiene “muchas” columnas y “pocas” filas ?!

Lanzamiento del kernel (3)

```
// configuración de la ejecución
int chunk = 32; // Se asume N múltiplo de 32

dim3 tamGrid(M, 1); //Grid dimensión
dim3 tamBlock(N/chunk,1,1); //Block dimensión
// lanzamiento del kernel
SumaFilaMatrizKernel<<<tamGrid, tamBlock>>>(N, Md, Xd);
```

Programación en CUDA

Kernel (3)

```
// Suma por filas de una matriz
__global__ void SumaFilaMatrizKernel(int N, float* Md, float* Xd){
// Pvalue es usado para el valor intermedio
  float Pvalue = 0;

  int aux = blockIdx.x*N + threadIdx.x*(N/blockDim.x);
  int aux2 = aux + (N/blockDim.x);

  for (int k = aux; k < aux2; ++k) {
     Pvalue = Pvalue + Md[k];
  }

  Xd[blockIdx.x] = Xd[blockIdx.x] + Pvalue;
}
```

Programación en CUDA

Recordemos ….

```
__ atomicAdd(float* address, float val);
```

`atomicAdd()` funciona a partir de compute capabilities 2.0 !!!

Kernel (3)

```
// Suma por filas de una matriz
__global__ void SumaFilaMatrizKernel(int N, float* Md, float* Xd){
// Pvalue es usado para el valor intermedio
  float Pvalue = 0;

  int aux = blockIdx.x*N + threadIdx.x*(N/blockDim.x);
  int aux2 = aux + (N/blockDim.x);

  for (int k = aux; k < aux2; ++k) {
     Pvalue = Pvalue + Md[k];
  }

  atomicAdd(&(Xd[blockIdx.x]), Pvalue);
}
```

Lanzamiento del kernel (4)

Si quiero que un bloque procese más de una fila:

```
// configuración de la ejecución
int chunk = 32; // Se asume M y N múltiplos de 32
dim3 tamGrid(M/chunk,1); //Grid dimensión
dim3 tamBlock(chunk, N/chunk, 1); //Block dimensión
// lanzamiento del kernel
SumaFilaMatrizKernel<<<tamGrid, tamBlock>>>(M, N, Md, Xd);
```

Programación en CUDA

Kernel (4)

```
// Suma por filas de una matriz
__global__void SumaFilaMatrizKernel (int M,int N,float*Md,float* Xd){
// Pvalue es usado para el valor intermedio
  float Pvalue = 0;

  int fila = blockIdx.x*(M/gridDim.x)+threadIdx.x;
  int pasos = N/ blockDim.y;
  int posIni = fila * N + threadIdx.y * pasos;

  for (int k = 0; k < pasos; ++k) {
     Pvalue = Pvalue + Md[posIni + k];
  }

  atomicAdd(&(Xd[fila]), Pvalue);

}
```