

CUDA

(Compute Unified Device Architecture)

Alvaro Cuno

23/01/2010

CUDA

- Arquitectura de computación paralela de propósito general
- La programación para la arquitectura CUDA puede hacerse usando lenguaje C
- Incluye una biblioteca con rutinas para acceso a la GPU
- Permite que partes del código de una aplicación corra en la GPU y otras en la CPU
- Presenta el hardware de forma transparente
 - Una GPU puede tener cientos de núcleos que soportan miles de hilos

CUDA

- Requerimientos
 - Hardware
 - GPU que soporte CUDA
 - Software
 - Distribución/versión de Linux/Windows que soporte CUDA
 - Herramientas CUDA

Hardware

- Fabricantes
 - NVidia
 - AMD (antiguamente ATI)
 - IBM
 - Desarrolló el Cellprocessor para el PlayStation con Sony y Toshiba
 - Intel
 - Desarrolla el GPU “Larrabee” a lanzarse a inicios del 2010

Hardware



GeForce 9800 GX2 (series 8,9)



GeForce GTX 295 (series 200)



Quadro FX 5800



Tesla S1070

Hardware



GeForce 9500 GT

- Precio: 50 dolares
- No necesita fuente de energía especial

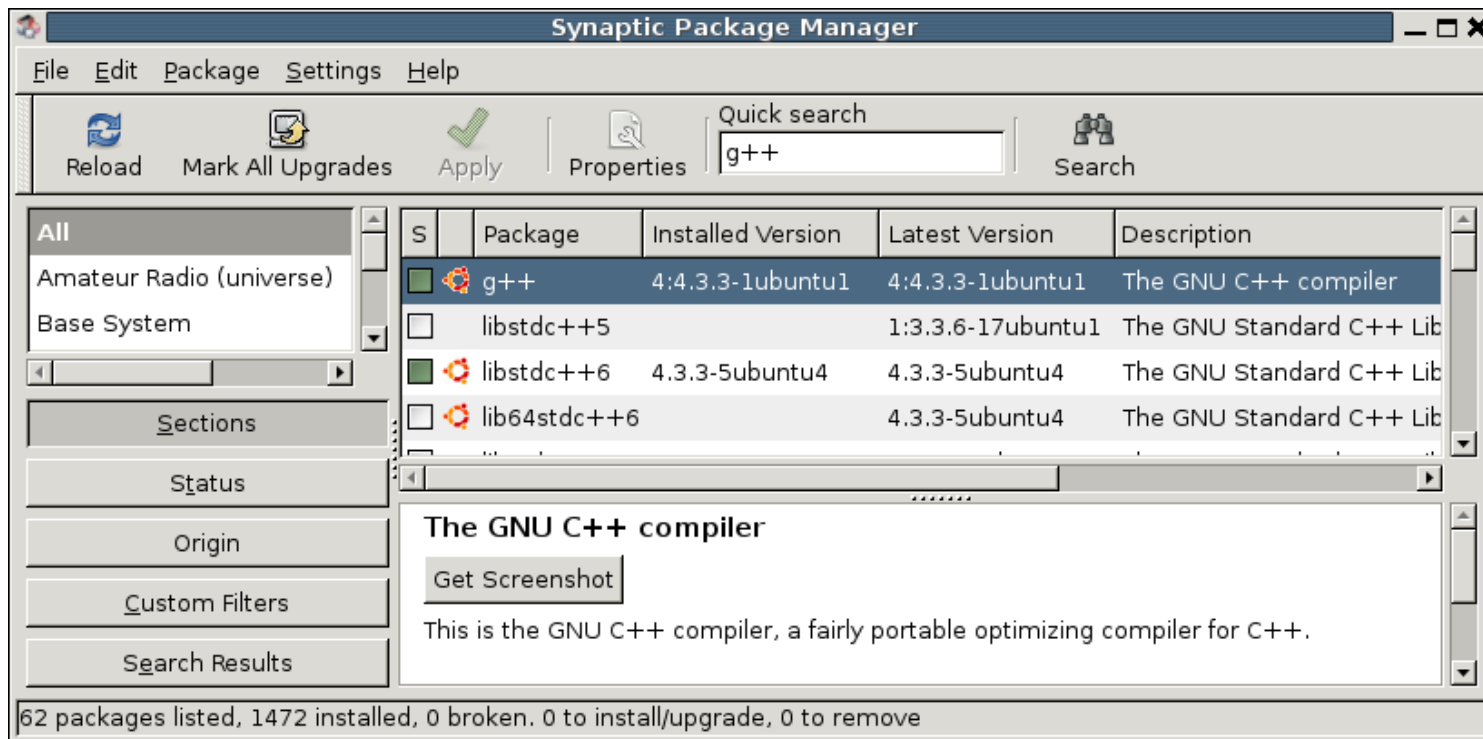
Software

Software

- Instalación
 - SO
 - Red Hat, Suse, Fedora, Ubuntu
 - Driver
 - Toolkit
 - SDK

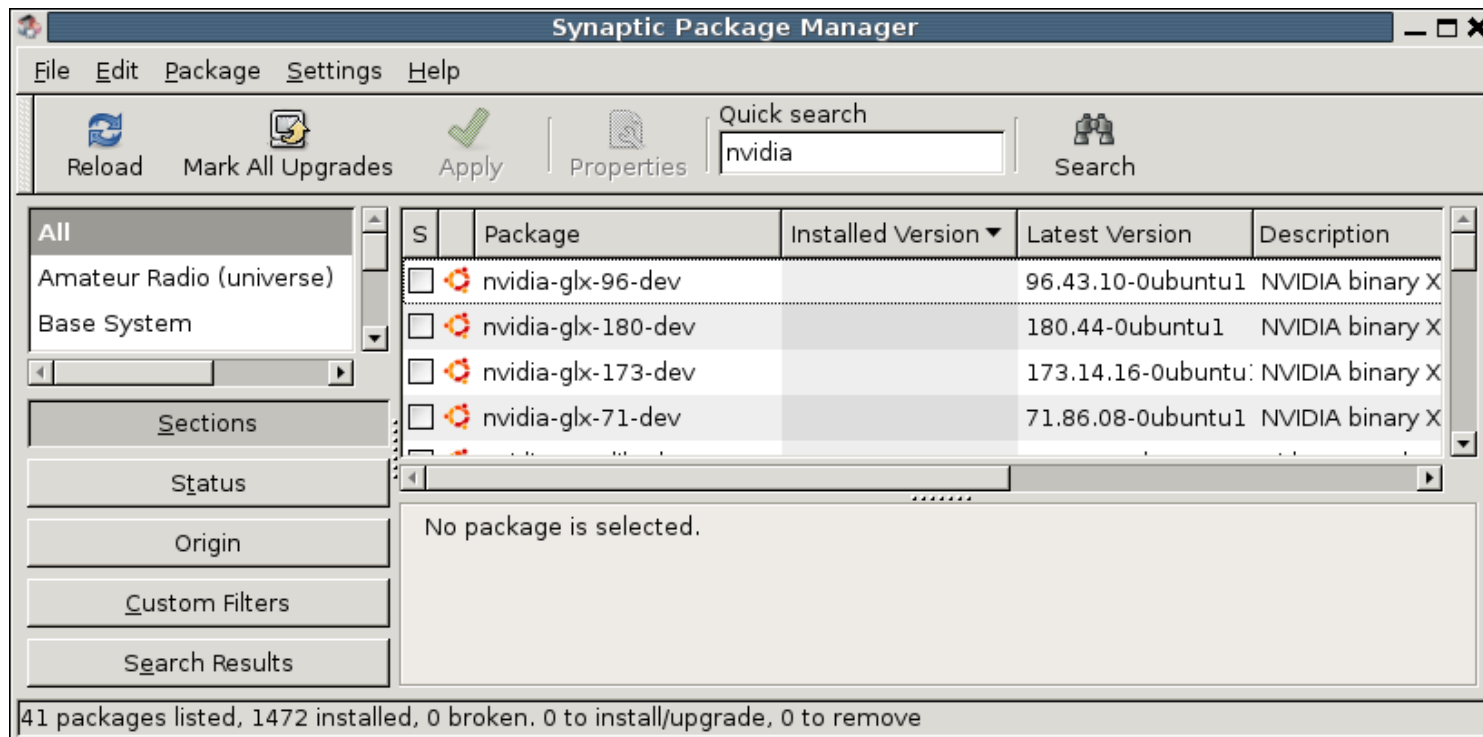
Software

- Pre-requisitos (Ubuntu 9.04)
 - Instalar el compilador g++, gcc
 - Instalar freeglut, freeglut3-dev, libglui-dev, libglew
 - Desinstalar cualquier otro driver Nvidia



Software

- Pre-requisitos (Ubuntu 9.04)
 - Instalar el compilador g++, gcc
 - Instalar freeglut, freeglut3-dev, libglui-dev, libglew
 - **Desinstalar cualquier otro driver Nvidia**



Software

- **Instalación** (http://www.nvidia.com/object/cuda_get.html)
 - **Driver** (NVIDIA Driver 190.18 Beta for Linux (Ubuntu 9.04) with CUDA Support)
 - Driver para la placa gráfica
 - **Toolkit** (CUDA Toolkit 2.3 for Linux (Ubuntu 9.04))
 - Herramientas para construir y compilar una aplicación CUDA
 - **SDK** (CUDA SDK 2.3 code samples for Linux (Ubuntu 9.04))
 - Códigos de ejemplos

Driver Nvidia

1) Salir del ambiente gráfico

- `sudo /sbin/init 3`

2) Correr el instalador del driver

3) Volver al ambiente gráfico

- `sudo startx`

Consultar: [CUDA_Getting_Started_2.3_Linux.pdf](#)

Driver Nvidia

1) Salir del ambiente gráfico

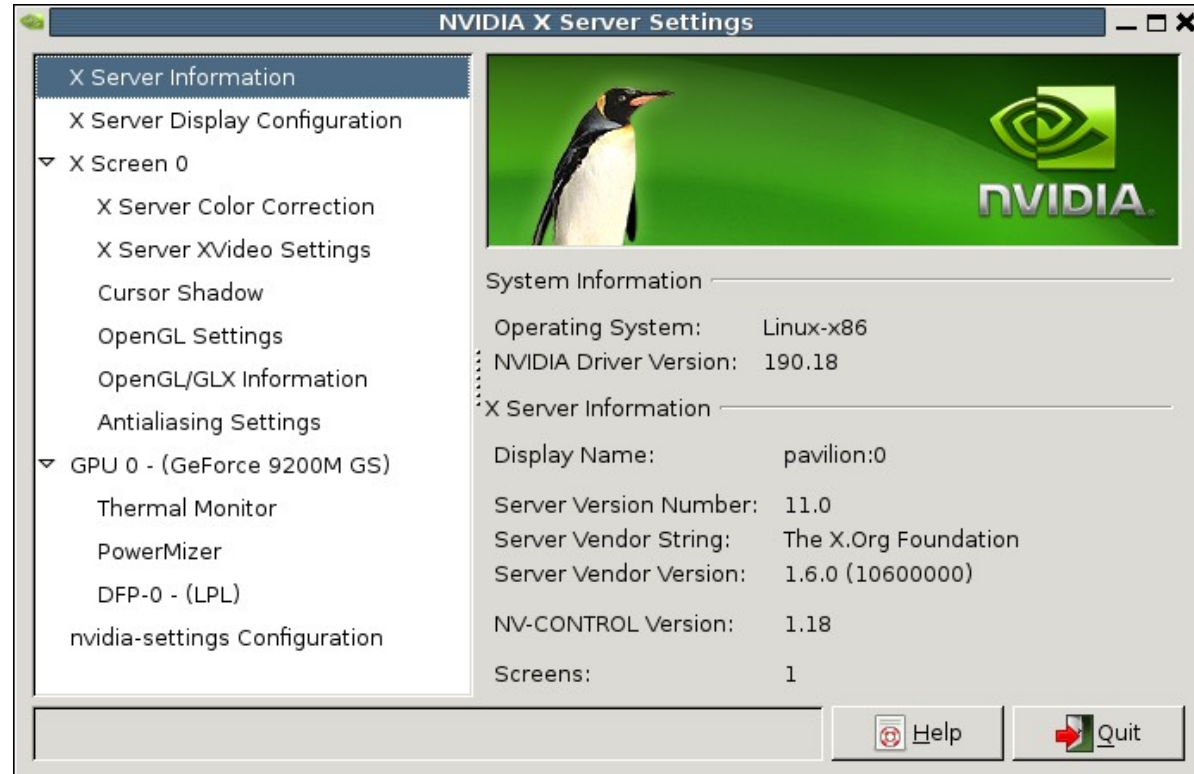
- `sudo /sbin/init 3`

2) Correr el instalador del driver

3) Volver al ambiente gráfico

- `sudo startx`

Si todo fue ok, verificar:
System>Preferences>NVidiaXServerSettings



CUDA Toolkit

1) Correr el archivo: `./cudatoolkit_2.3_linux_32_ubuntu9.04.run`

- Puede ser como usuario normal o como `sudo`
- Dependiendo del usuario la ubicación de la instalación será diferente

2) En el archivo `~.bashrc` incluir las siguientes lineas:

```
export PATH=~ /CUDA/cuda/bin:"${PATH}"
```

```
export LD_LIBRARY_PATH=~ /CUDA/cuda/lib:"${LD_LIBRARY_PATH}"
```

```
export CUDA_INSTALL_PATH=~ /CUDA/cuda
```

CUDA SDK

- **Correr el archivo:** `cudasdk_2.3_linux.run`

Verificar la instalación

Verificar la instalación

- Ejemplos del CUDA SDK

```
alvaro@pavilion: ~/CUDA/SDK/C/bin/linux/release
File Edit View Terminal Help
alvaro@pavilion:~/CUDA/SDK/C/bin/linux/release$ ls
3dfd                               deviceQuery                       nbody                               simpleStreams
alignedTypes                       deviceQueryDrv                   oceanFFT                             simpleTemplates
asyncAPI                           dwtHaar1D                        params.txt                           simpleTexture
bandwidthTest                      dxtc                             particles                           simpleTexture3D
barbara_cuda1.bmp                 eigenvalues                       postProcessGL                       simpleTextureDrv
barbara_cuda2.bmp                 fastWalshTransform              ptxjit                              simpleVoteIntrinsics
barbara_cuda_short.bmp            fluidsGL                          quasirandomGenerator               simpleZeroCopy
barbara_gold1.bmp                 histogram                        radixSort                          smokeParticles
barbara_gold2.bmp                 imageDenoising                  recursiveGaussian                   SobelFilter
bicubicTexture                    lineOfSight                     reduction                           SobolQRNG
binomialOptions                   Mandelbrot                      scalarProd                          sortingNetworks
BlackScholes                      marchingCubes                    scan                                 template
boxFilter                          matrixMul                        scanLargeArray                    threadFenceReduction
clock                              matrixMulDrv                    simpleAtomicIntrinsics             threadMigration
convolutionFFT2D                  matrixMulDynlinkJIT             simpleCUBLAS                       transpose
convolutionSeparable              MersenneTwister                 simpleCUFFT                        transposeNew
convolutionTexture                 mlsGlut                         simpleGL                            volumeRender
cppIntegration                    MonteCarlo                       simpleMultiGPU                     simplePitchLinearTexture
dct8x8                             MonteCarloMultiGPU              simplePitchLinearTexture
alvaro@pavilion:~/CUDA/SDK/C/bin/linux/release$
```

Ejemplos del CUDA SDK

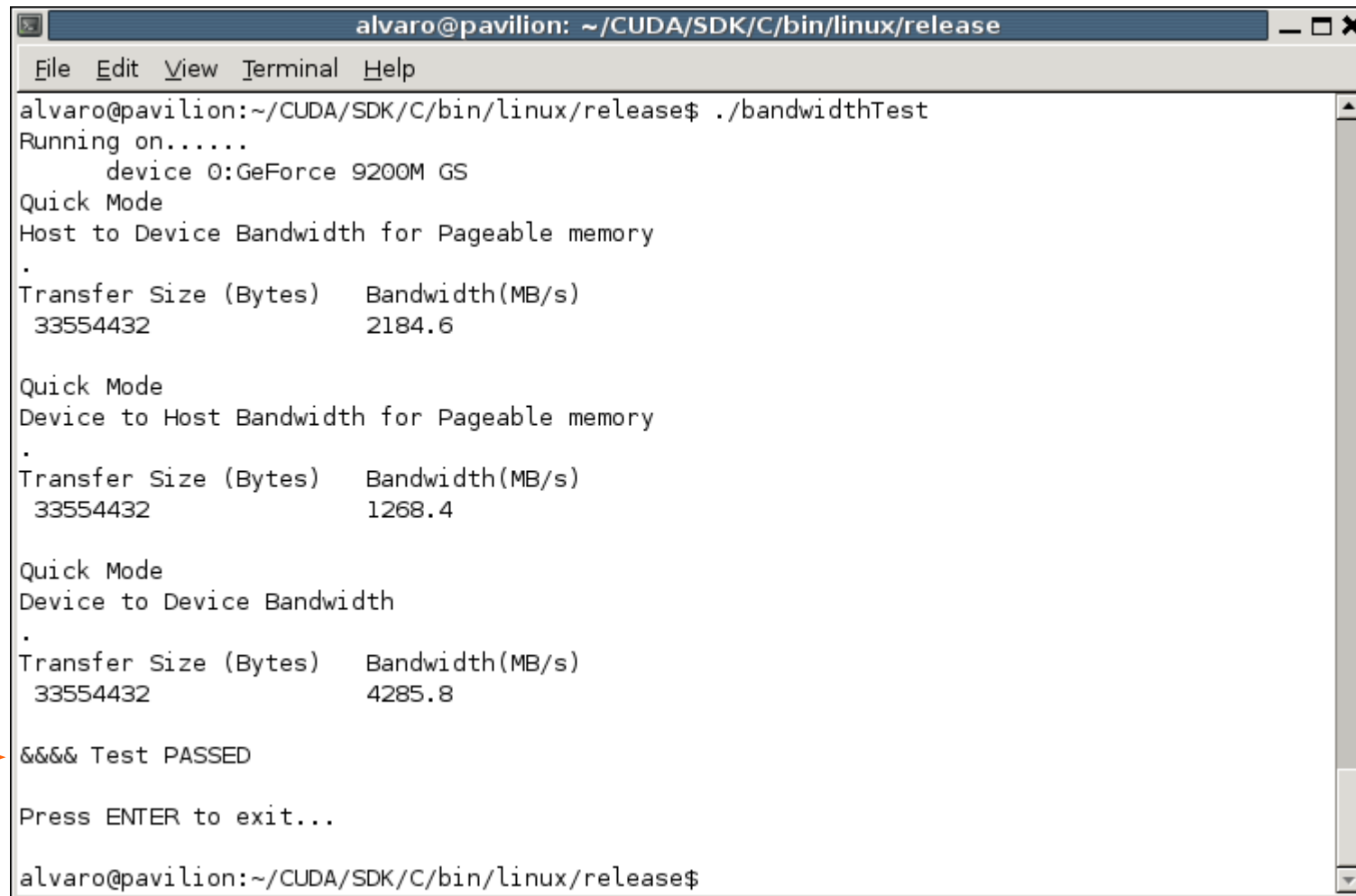
```
alvaro@pavilion: ~/CUDA/SDK/C/bin/linux/release
File Edit View Terminal Help
alvaro@pavilion:~/CUDA/SDK/C/bin/linux/release$
alvaro@pavilion:~/CUDA/SDK/C/bin/linux/release$ ./deviceQuery
CUDA Device Query (Runtime API) version (CUDA static linking)
There is 1 device supporting CUDA

Device 0: "GeForce 9200M GS"
  CUDA Driver Version:            2.30
  CUDA Runtime Version:          2.30
  CUDA Capability Major revision number: 1
  CUDA Capability Minor revision number: 1
  Total amount of global memory:  267714560 bytes
  Number of multiprocessors:      1
  Number of cores:                8
  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:                     1.30 GHz
  Concurrent copy and execution:  No
  Run time limit on kernels:      Yes
  Integrated:                     No
  Support host page-locked memory mapping: No
  Compute mode:                   Default (multiple host threads can use this device simultaneously)
```

Ejemplos del CUDA SDK

```
alvaro@pavilion: ~/CUDA/SDK/C/bin/linux/release
File Edit View Terminal Help
alvaro@pavilion:~/CUDA/SDK/C/bin/linux/release$
alvaro@pavilion:~/CUDA/SDK/C/bin/linux/release$ ./deviceQuery
CUDA Device Query (Runtime API) version (CUDA static linking)
There is 1 device supporting CUDA
Device 0: "GeForce 9200M GS"
  CUDA Driver Version:          2.30
  CUDA Runtime Version:        2.30
  CUDA Capability Major revision number: 1
  CUDA Capability Minor revision number: 1
  Total amount of global memory: 267714560 bytes
  Number of multiprocessors:    1
  Number of cores:              8
  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:                   1.30 GHz
  Concurrent copy and execution: No
  Run time limit on kernels:    Yes
  Integrated:                   No
  Support host page-locked memory mapping: No
  Compute mode:                 Default (multiple host threads can use this device simultaneously)
```

Ejemplos del CUDA SDK



```
alvaro@pavilion: ~/CUDA/SDK/C/bin/linux/release
File Edit View Terminal Help
alvaro@pavilion:~/CUDA/SDK/C/bin/linux/release$ ./bandwidthTest
Running on.....
    device 0:GeForce 9200M GS
Quick Mode
Host to Device Bandwidth for Pageable memory
.
Transfer Size (Bytes)    Bandwidth(MB/s)
33554432                 2184.6

Quick Mode
Device to Host Bandwidth for Pageable memory
.
Transfer Size (Bytes)    Bandwidth(MB/s)
33554432                 1268.4

Quick Mode
Device to Device Bandwidth
.
Transfer Size (Bytes)    Bandwidth(MB/s)
33554432                 4285.8

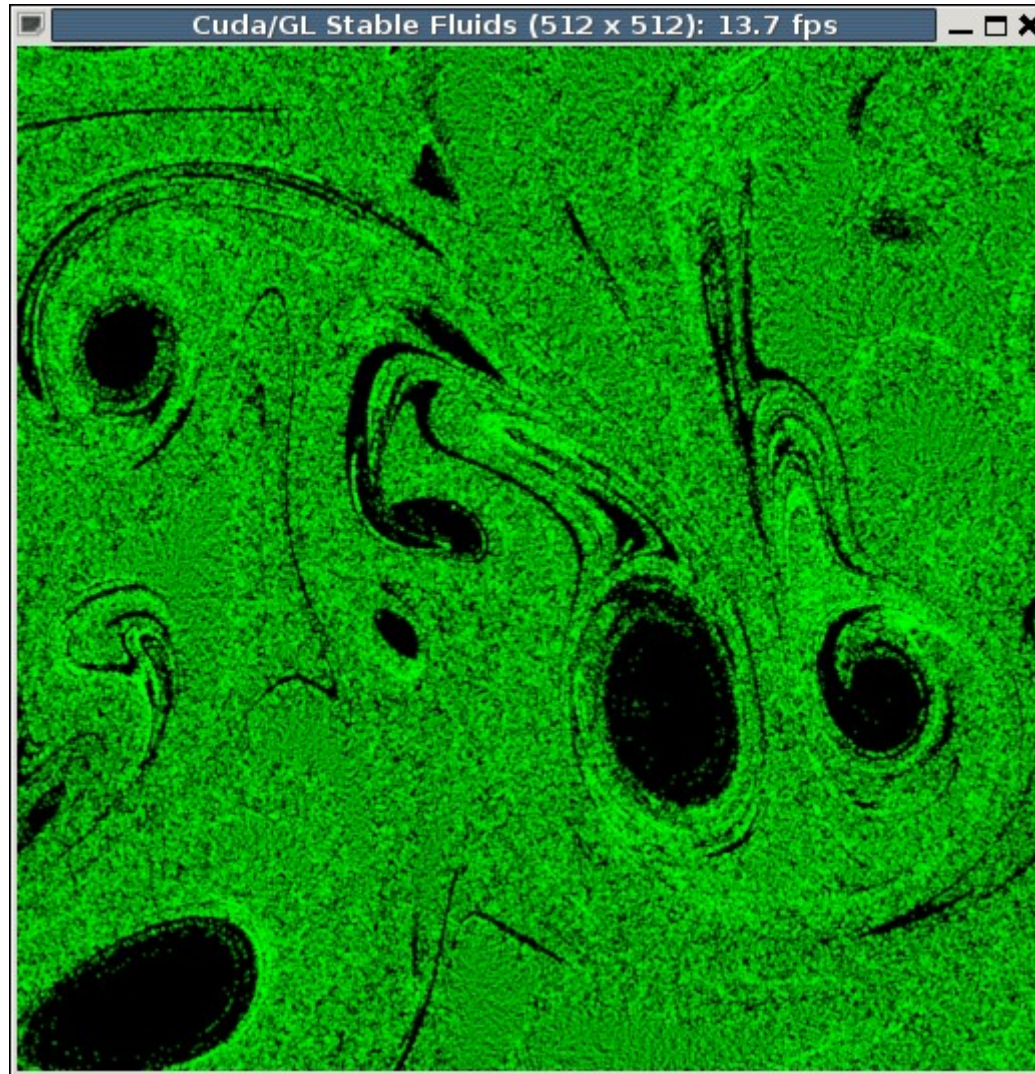
&&&& Test PASSED

Press ENTER to exit...

alvaro@pavilion:~/CUDA/SDK/C/bin/linux/release$
```

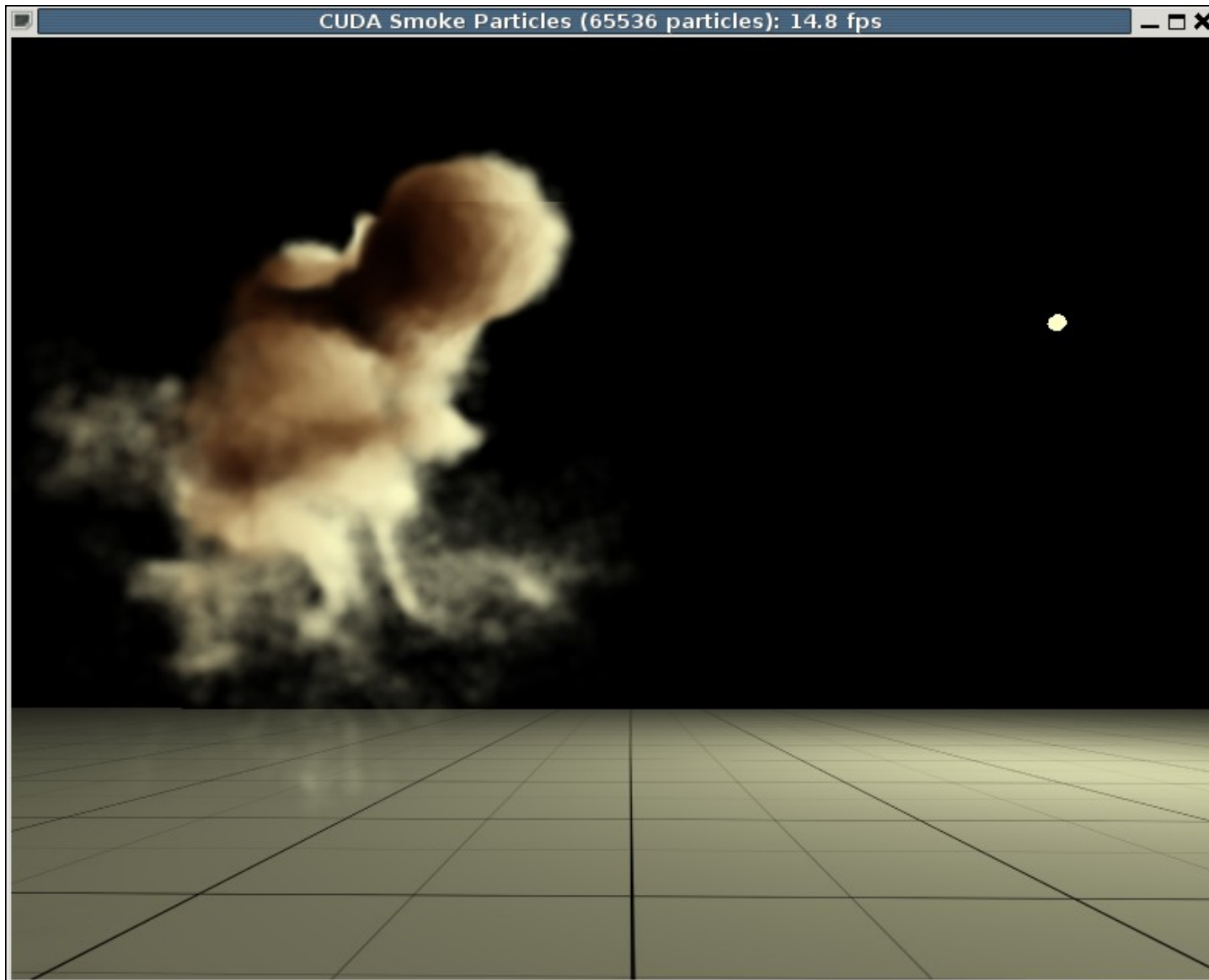
Verificando la comunicación

Ejemplos del CUDA SDK



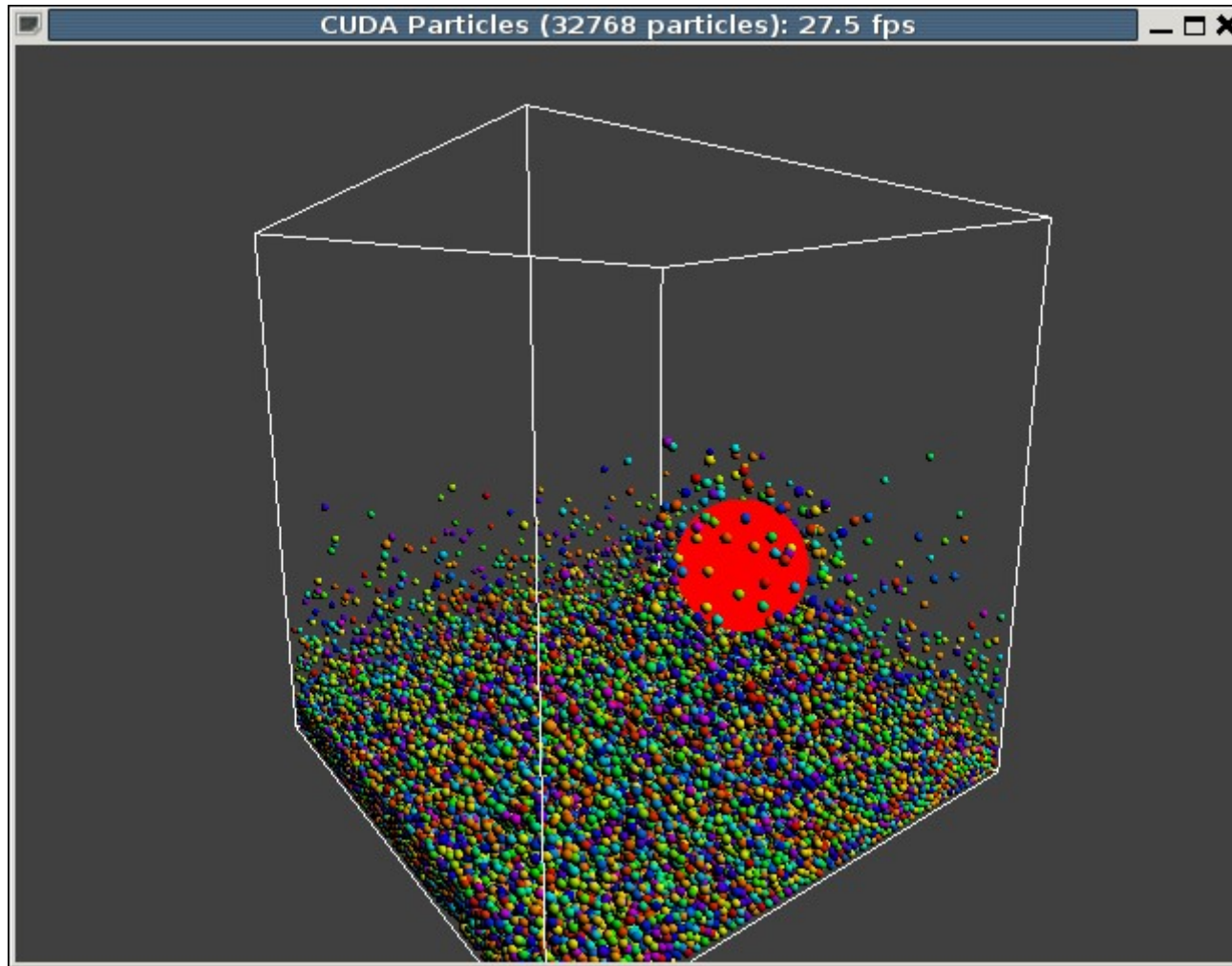
`./fluidsGL`

Ejemplos del CUDA SDK



./smokeParticles

Ejemplos del CUDA SDK

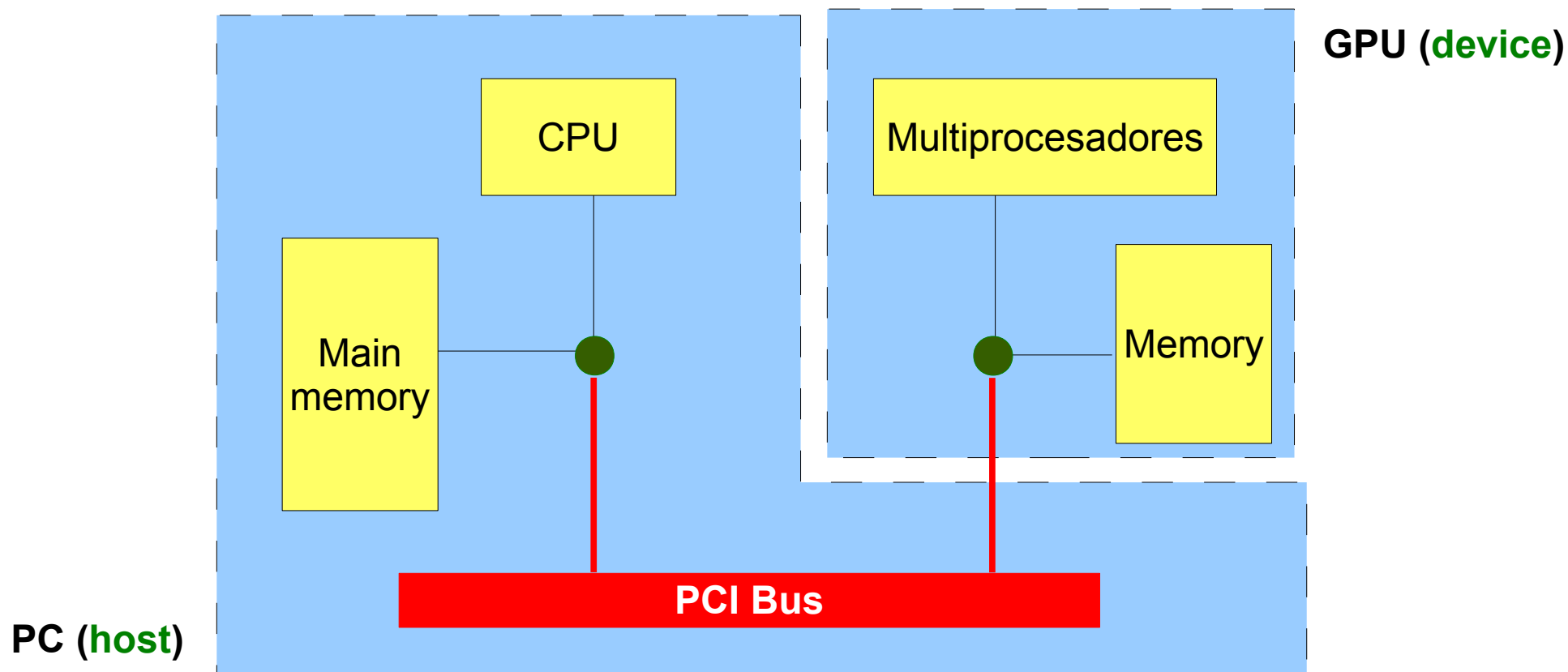


./particles

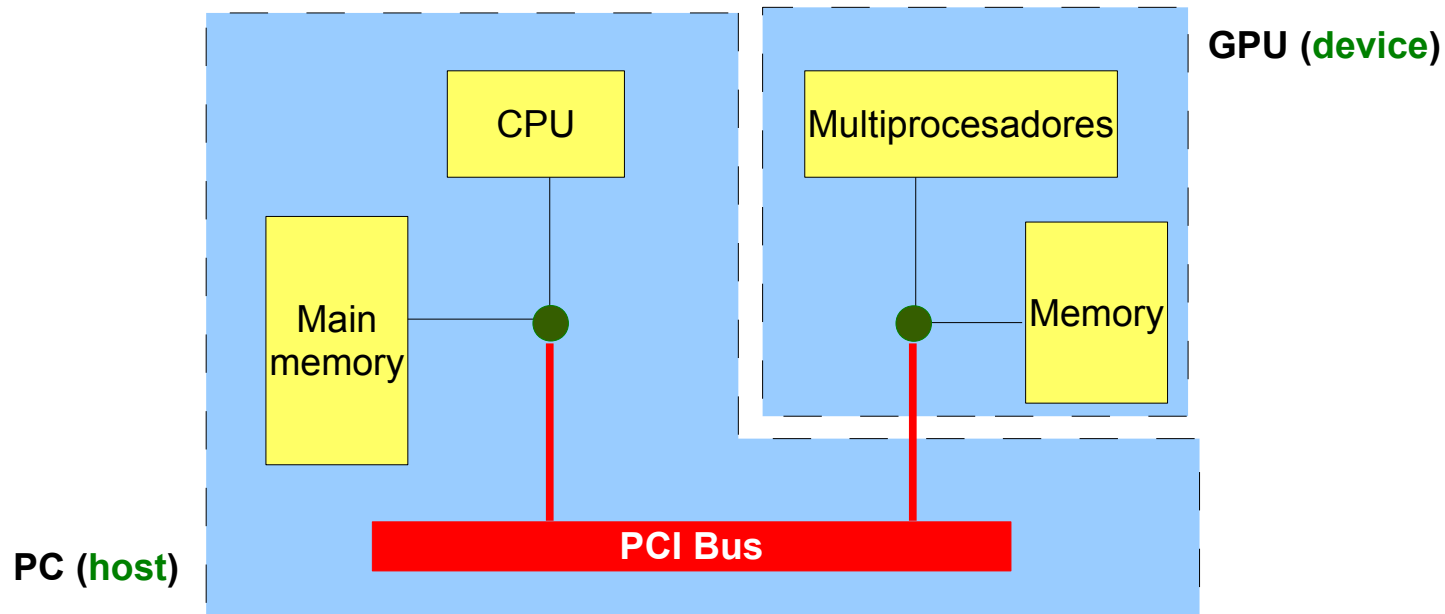
Modelo CUDA

Modelo CUDA

- La GPU y la CPU son tratados como dispositivos separados con su propio espacio de memoria



Modelo CUDA



- La GPU no puede acceder directamente a la memoria principal
- La CPU no puede acceder directamente a la memoria de la GPU
- La transferencia de datos necesita ser explícita
- No es posible hacer printf en la GPU

Modelo CUDA

- Abstracciones
 - Jerarquía de hilos
 - Jerarquía de memoria
 - Barrier synchronization
- Escalabilidad
 - Un programa CUDA puede ejecutarse sobre cualquier número de procesadores
 - No es necesario recompilar el código
 - Solo en tiempo de ejecución es necesario conocer el #
 - Lo cual permite homogeneidad de conceptos para usuarios: GeForce, Quadro y Tesla

Suma de vectores

Suma de vectores

- Sumar dos arreglos lineales usando la GPU
 - $s[i] = a[i] + b[i]$
- Operaciones
 1. Escribir kernels (`__global__`)
 2. Reservar memoria de la GPU (`cudaMalloc()`)
 3. Copiar datos de/hacia la GPU (`cudaMemcpy()`)
 4. Ejecutar/invocar kernels (`<<<>>>`)
 5. Compilación y ejecutar (`nvcc`)

1. El kernel

```
__global__  
void sumaVectores_kernel(int N, float *a, float *b, float *s) {  
    for (int i=0; i<N; i++)  
        s[i] = a[i] + b[i];  
}
```

2. Reservar memoria

```
int main(int argc, char** argv) {  
  
    int N=16; // longitud de los arreglos  
  
    // separar memoria en el host  
    float *a_h = (float *)malloc(sizeof(float)*N);  
    float *b_h = (float *)malloc(sizeof(float)*N);  
    float *s_h = (float *)malloc(sizeof(float)*N);  
  
    // punteros a arreglos en el device  
    float *a_d, *b_d, *s_d;  
    // separar memoria en el device  
    cudaMalloc((void **)&a_d, sizeof(float)*N);  
    cudaMalloc((void **)&b_d, sizeof(float)*N);  
    cudaMalloc((void **)&s_d, sizeof(float)*N);  
}
```

3. Copiar datos hacia la GPU

```
// inicializar a y b en el host
```

```
for (int i=0; i < N; i++) {  
    a_h[i] = (float)i;  
    b_h[i] = (float)i;  
}
```

```
// transferir a y b desde el host hacia el device
```

```
cudaMemcpy((void *)a_d, (void *)a_h, sizeof(float)*N, cudaMemcpyHostToDevice);  
cudaMemcpy((void *)b_d, (void *)b_h, sizeof(float)*N, cudaMemcpyHostToDevice);
```

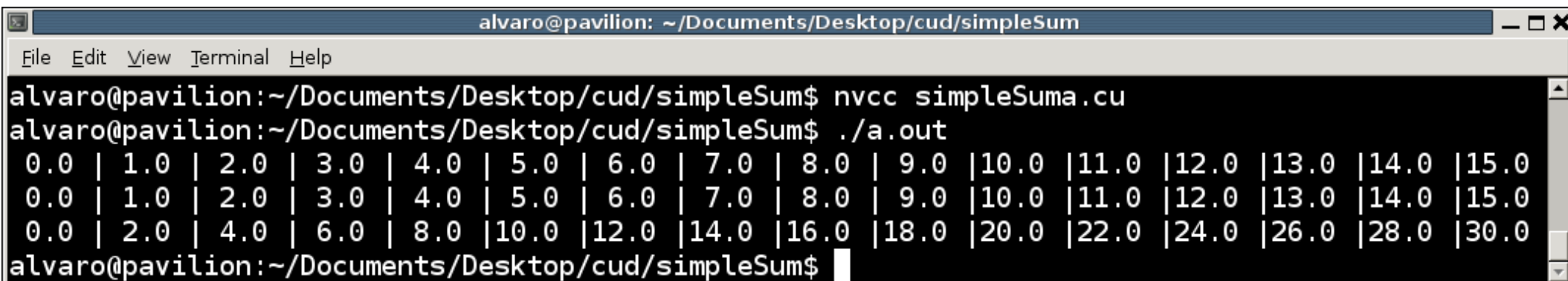

4. Invocar el kernel y obtener el resultado

```
// invocar el kernel (un solo hilo/thread)  
sumaVectores_kernel<<<1,1>>>(N, a_d, b_d, s_d);  
  
// copiar el resultado del device hacia host  
cudaMemcpy((void *)s_h, (void *)s_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
```

5. Compilar y ejecutar

```
// imprimir datos
for (int i=0; i<N; i++)      printf("%4.1f |", a_h[i]); printf("\n");
for (int i=0; i<N; i++)      printf("%4.1f |", b_h[i]); printf("\n");
for (int i=0; i<N; i++)      printf("%4.1f |", s_h[i]); printf("\n");

// liberar la memoria
free(a_h); free(b_h); free(s_h);
cudaFree(a_d); cudaFree(b_d); cudaFree(s_d);
return 0;
}
```



The screenshot shows a terminal window with the following content:

```
alvaro@pavilion: ~/Documents/Desktop/cud/simpleSum
File Edit View Terminal Help
alvaro@pavilion:~/Documents/Desktop/cud/simpleSum$ nvcc simpleSuma.cu
alvaro@pavilion:~/Documents/Desktop/cud/simpleSum$ ./a.out
0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 | 7.0 | 8.0 | 9.0 | 10.0 | 11.0 | 12.0 | 13.0 | 14.0 | 15.0
0.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 6.0 | 7.0 | 8.0 | 9.0 | 10.0 | 11.0 | 12.0 | 13.0 | 14.0 | 15.0
0.0 | 2.0 | 4.0 | 6.0 | 8.0 | 10.0 | 12.0 | 14.0 | 16.0 | 18.0 | 20.0 | 22.0 | 24.0 | 26.0 | 28.0 | 30.0
alvaro@pavilion:~/Documents/Desktop/cud/simpleSum$
```

Kernels

- Rutinas ejecutadas paralelamente N veces por N diferentes hilos
- Definidos por `__global__`

Kernels

- Rutinas ejecutadas paralelamente N veces por N diferentes hilos
- Definidos por `__global__`

```
__global__  
void sumaVectores_kernel(int N, float *a, float *b, float *s) {  
  
    for (int i=0; i<N; i++)  
        s[i] = a[i] + b[i];  
}
```

Kernels

- Rutinas ejecutadas paralelamente N veces por N diferentes hilos
- Definidos por `__global__`

```
__global__  
void sumaVectores_kernel(int N, float *a, float *b, float *s) {  
  
    for (int i=0; i<N; i++)  
        s[i] = a[i] + b[i];  
}  
  
// invocar el kernel (un solo hilo/thread)  
sumaVectores_kernel<<<1,1>>>(N, a_d, b_d, s_d);
```

Tipos de funciones

- CUDA ofrece “tipos” de funciones
 - Permite definir donde corre una función
- `__host__`: el código debe correr en el host (valor por default)
- `__device__`: el código debe correr en la GPU, y la función debe ser llamada por código corriendo en la GPU
- `__global__`: el código debe correr en la GPU pero invocado desde el host. Es el **punto de acceso**

Tipos de funciones

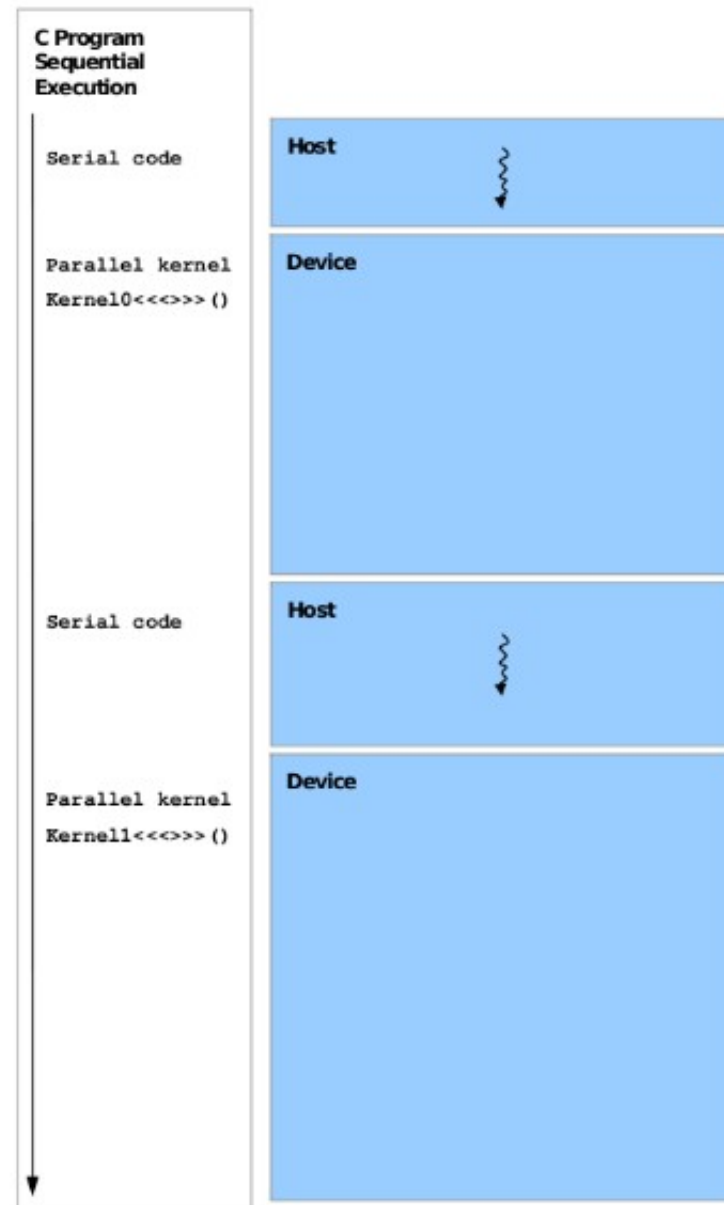
- Restricciones
 - Código de DEVICE debe ser escrito solamente en C
 - Código de HOST puede ser escrito en C++
 - Es posible usar STL
 - Código de DEVICE no puede ser llamado recursivamente

Ejercicio

- Re-escribir en C++ el código que suma dos vectores
 - Usar STL

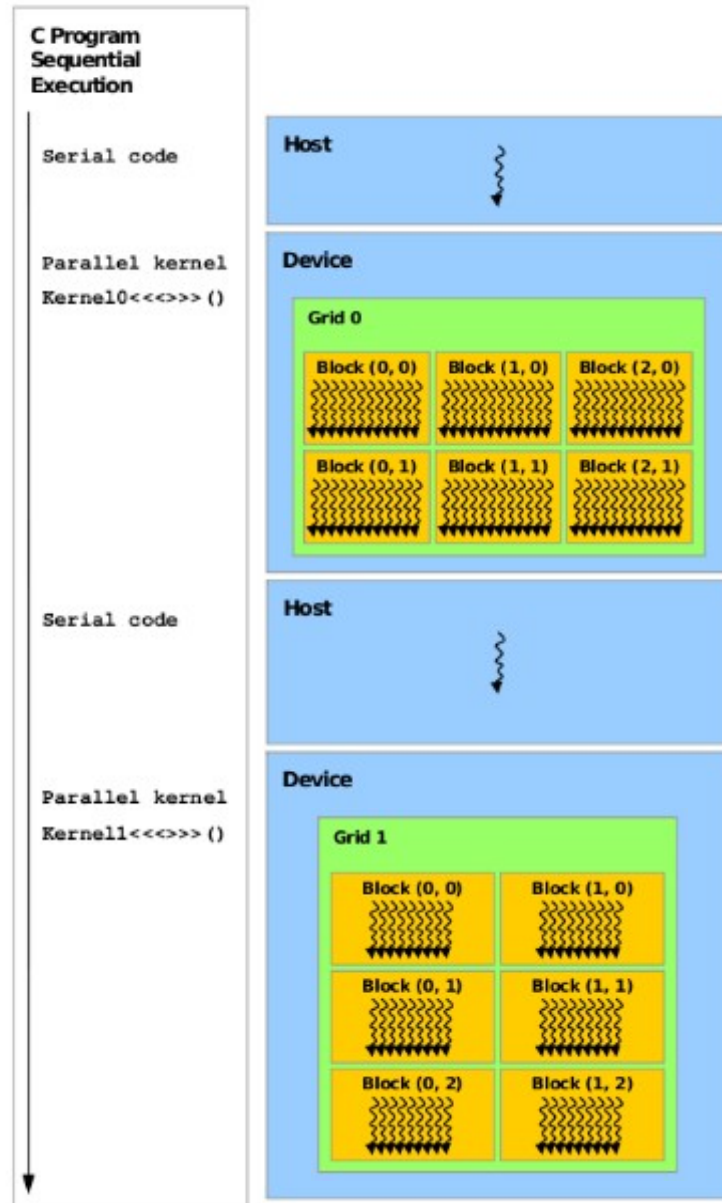
Threads

Ejecución del código



Serial code executes on the host while parallel code executes on the device.

Ejecución del código



Serial code executes on the host while parallel code executes on the device.

Ejecución del código

- Ejercicio
 - Modificar el programa de suma de dos vectores:
 $A+B=C$ para, posterior a la suma, multiplicar: $C*K$

Threads

- Todas las llamadas a una función `__global__` deben especificar el número de hilos a ser instanciados
 - Sintaxis: `<<< >>>`

Múltiples hilos

```
// invocar el kernel (un solo hilo/thread)
sumaVectores_kernel<<<1,1>>>(N, a_d, b_d, s_d);

__global__
void sumaVectores_kernel(int N, float *a, float *b, float *s) {

    for (int i=0; i<N; i++)
        s[i] = a[i] + b[i];
}
```

Múltiples hilos

```
// invocar el kernel (un solo hilo/thread)
sumaVectores_kernel<<<1,1>>>(N, a_d, b_d, s_d);

__global__
void sumaVectores_kernel(int N, float *a, float *b, float *s) {

    for (int i=0; i<N; i++)
        s[i] = a[i] + b[i];
}

// invocar el kernel (N hilos)
sumaVectores_kernel<<<1,N>>>(a_d, b_d, s_d);
```

- Cada hilo que ejecuta un kernel tiene un identificador único
- El identificador es accesible a través de la variable **threadIdx**

Múltiples hilos

```
// invocar el kernel (un solo hilo/thread)
sumaVectores_kernel<<<1,1>>>(N, a_d, b_d, s_d);

__global__
void sumaVectores_kernel(int N, float *a, float *b, float *s) {

    for (int i=0; i<N; i++)
        s[i] = a[i] + b[i];
}

// invocar el kernel (N hilos)
sumaVectores_kernel<<<1,N>>>(a_d, b_d, s_d);
```

- Cada hilo que ejecuta un kernel tiene un identificador único
- El identificador es accesible a través de la variable **threadIdx**

```
__global__
void sumaVectores_kernel(float *a, float *b, float *s) {

    int i = threadIdx.x;
    s[i] = a[i] + b[i];
}
```


Ejercicio

- Modificar el código del archivo **simpleSuma.cu** para ejecutar la suma usando 4 hilos

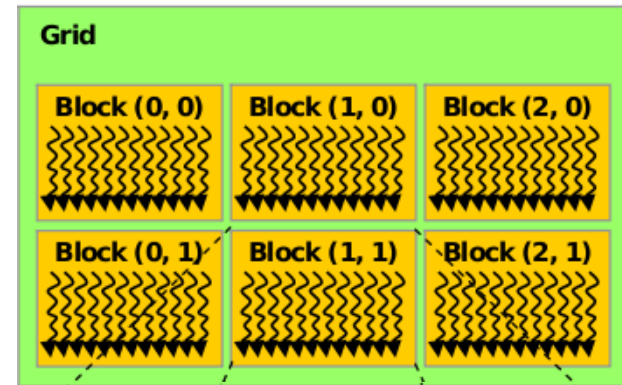
Jerarquía de hilos

- **threads** son agrupados en **blocks** y estos en una **grid**
 - Esto define una jerarquía
 - Grid->Blocks->Threads



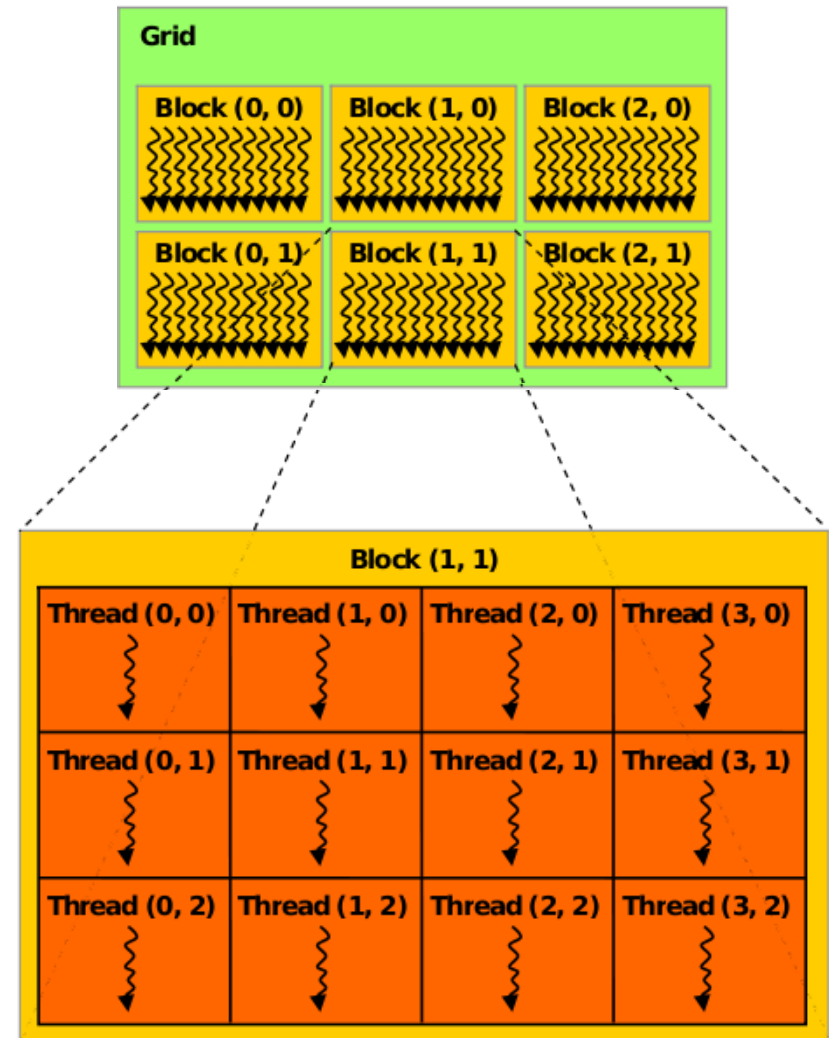
Jerarquía de hilos

- **threads** son agrupados en **blocks** y estos en una **grid**
 - Esto define una jerarquía
 - Grid->Blocks->Threads



Jerarquía de hilos

- **threads** son agrupados en **blocks** y estos en una **grid**
 - Esto define una jerarquía
 - Grid->Blocks->Threads



Jerarquía de hilos

- Dentro de `<<<>>>` se necesitan dos argumentos
 - Pueden ser dos mas (tienen valores default)
- **Ejemplo:** `myKernel<<<g, b>>>(agr1, arg2)`
- `g` especifica las dimensiones de la `grid` mientras que `b` define la dimensión de cada `block`
- `g` y `b` son de tipo `dim3`
 - Tres `unsigned int` (nuevo tipo de dato CUDA)
 - `dim3 g(2, 2)` define `g.x=2, g.y=2, g.z=1`
- Es permitida la sintaxis 1D: `myKernel<<<5, 6>>>`

Jerarquía de hilos

- Ids
 - Threads:
 - 3D Ids, únicas en un bloque
 - Blocks:
 - 2D Ids, únicas en una grid

Jerarquía de hilos

- Para código corriendo en la GPU (`__device__` y `__global__`) hay variables predefinidas (pueden ser accedidas)
 - `dim3 gridDim`: Dimensions of the grid.
 - `uint3 blockIdx`: location of this block in the grid.
 - `dim3 blockDim`: Dimensions of the blocks
 - `uint3 threadIdx`: location of this thread in the block

Ejercicio

- Codificar un programa que sume dos matrices NxN haciendo uso de hilos

$A_{0,0}$	$A_{1,0}$	$A_{2,0}$	$A_{3,0}$
$A_{0,1}$	$A_{1,1}$	$A_{2,1}$	$A_{3,1}$
$A_{0,2}$	$A_{1,2}$	$A_{2,2}$	$A_{3,2}$
$A_{0,3}$	$A_{1,3}$	$A_{2,3}$	$A_{3,3}$

+

$B_{0,0}$	$B_{1,0}$	$B_{2,0}$	$B_{3,0}$
$B_{0,1}$	$B_{1,1}$	$B_{2,1}$	$B_{3,1}$
$B_{0,2}$	$B_{1,2}$	$B_{2,2}$	$B_{3,2}$
$B_{0,3}$	$B_{1,3}$	$B_{2,3}$	$B_{3,3}$

=

$P_{0,0}$	$P_{1,0}$	$P_{2,0}$	$P_{3,0}$
$P_{0,1}$	$P_{1,1}$	$P_{2,1}$	$P_{3,1}$
$P_{0,2}$	$P_{1,2}$	$P_{2,2}$	$P_{3,2}$
$P_{0,3}$	$P_{1,3}$	$P_{2,3}$	$P_{3,3}$

Multiplicación de Matrices

Multiplicación de Matrices

$A_{0,0}$	$A_{1,0}$	$A_{2,0}$	$A_{3,0}$
$A_{0,1}$	$A_{1,1}$	$A_{2,1}$	$A_{3,1}$
$A_{0,2}$	$A_{1,2}$	$A_{2,2}$	$A_{3,2}$
$A_{0,3}$	$A_{1,3}$	$A_{2,3}$	$A_{3,3}$

\times

$B_{0,0}$	$B_{1,0}$	$B_{2,0}$	$B_{3,0}$
$B_{0,1}$	$B_{1,1}$	$B_{2,1}$	$B_{3,1}$
$B_{0,2}$	$B_{1,2}$	$B_{2,2}$	$B_{3,2}$
$B_{0,3}$	$B_{1,3}$	$B_{2,3}$	$B_{3,3}$

$=$

$P_{0,0}$	$P_{1,0}$	$P_{2,0}$	$P_{3,0}$
$P_{0,1}$	$P_{1,1}$	$P_{2,1}$	$P_{3,1}$
$P_{0,2}$	$P_{1,2}$	$P_{2,2}$	$P_{3,2}$
$P_{0,3}$	$P_{1,3}$	$P_{2,3}$	$P_{3,3}$

Multiplicación de Matrices

$A_{0,0}$	$A_{1,0}$	$A_{2,0}$	$A_{3,0}$
$A_{0,1}$	$A_{1,1}$	$A_{2,1}$	$A_{3,1}$
$A_{0,2}$	$A_{1,2}$	$A_{2,2}$	$A_{3,2}$
$A_{0,3}$	$A_{1,3}$	$A_{2,3}$	$A_{3,3}$

\times

$B_{0,0}$	$B_{1,0}$	$B_{2,0}$	$B_{3,0}$
$B_{0,1}$	$B_{1,1}$	$B_{2,1}$	$B_{3,1}$
$B_{0,2}$	$B_{1,2}$	$B_{2,2}$	$B_{3,2}$
$B_{0,3}$	$B_{1,3}$	$B_{2,3}$	$B_{3,3}$

$=$

$P_{0,0}$	$P_{1,0}$	$P_{2,0}$	$P_{3,0}$
$P_{0,1}$	$P_{1,1}$	$P_{2,1}$	$P_{3,1}$
$P_{0,2}$	$P_{1,2}$	$P_{2,2}$	$P_{3,2}$
$P_{0,3}$	$P_{1,3}$	$P_{2,3}$	$P_{3,3}$

Multiplicación de Matrices

$A_{0,0}$	$A_{1,0}$	$A_{2,0}$	$A_{3,0}$
$A_{0,1}$	$A_{1,1}$	$A_{2,1}$	$A_{3,1}$
$A_{0,2}$	$A_{1,2}$	$A_{2,2}$	$A_{3,2}$
$A_{0,3}$	$A_{1,3}$	$A_{2,3}$	$A_{3,3}$

\times

$B_{0,0}$	$B_{1,0}$	$B_{2,0}$	$B_{3,0}$
$B_{0,1}$	$B_{1,1}$	$B_{2,1}$	$B_{3,1}$
$B_{0,2}$	$B_{1,2}$	$B_{2,2}$	$B_{3,2}$
$B_{0,3}$	$B_{1,3}$	$B_{2,3}$	$B_{3,3}$

$=$

$P_{0,0}$	$P_{1,0}$	$P_{2,0}$	$P_{3,0}$
$P_{0,1}$	$P_{1,1}$	$P_{2,1}$	$P_{3,1}$
$P_{0,2}$	$P_{1,2}$	$P_{2,2}$	$P_{3,2}$
$P_{0,3}$	$P_{1,3}$	$P_{2,3}$	$P_{3,3}$

Multiplicación de Matrices

$A_{0,0}$	$A_{1,0}$	$A_{2,0}$	$A_{3,0}$
$A_{0,1}$	$A_{1,1}$	$A_{2,1}$	$A_{3,1}$
$A_{0,2}$	$A_{1,2}$	$A_{2,2}$	$A_{3,2}$
$A_{0,3}$	$A_{1,3}$	$A_{2,3}$	$A_{3,3}$

\times

$B_{0,0}$	$B_{1,0}$	$B_{2,0}$	$B_{3,0}$
$B_{0,1}$	$B_{1,1}$	$B_{2,1}$	$B_{3,1}$
$B_{0,2}$	$B_{1,2}$	$B_{2,2}$	$B_{3,2}$
$B_{0,3}$	$B_{1,3}$	$B_{2,3}$	$B_{3,3}$

$=$

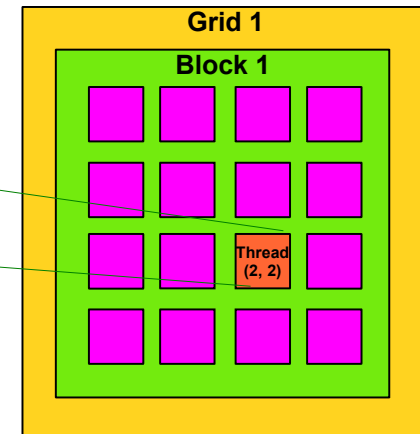
$P_{0,0}$	$P_{1,0}$	$P_{2,0}$	$P_{3,0}$
$P_{0,1}$	$P_{1,1}$	$P_{2,1}$	$P_{3,1}$
$P_{0,2}$	$P_{1,2}$	$P_{2,2}$	$P_{3,2}$
$P_{0,3}$	$P_{1,3}$	$P_{2,3}$	$P_{3,3}$

Multiplicación de Matrices

$B_{0,0}$	$B_{1,0}$	$B_{2,0}$	$B_{3,0}$
$B_{0,1}$	$B_{1,1}$	$B_{2,1}$	$B_{3,1}$
$B_{0,2}$	$B_{1,2}$	$B_{2,2}$	$B_{3,2}$
$B_{0,3}$	$B_{1,3}$	$B_{2,3}$	$B_{3,3}$

$A_{0,0}$	$A_{1,0}$	$A_{2,0}$	$A_{3,0}$
$A_{0,1}$	$A_{1,1}$	$A_{2,1}$	$A_{3,1}$
$A_{0,2}$	$A_{1,2}$	$A_{2,2}$	$A_{3,2}$
$A_{0,3}$	$A_{1,3}$	$A_{2,3}$	$A_{3,3}$

$P_{0,0}$	$P_{1,0}$	$P_{2,0}$	$P_{3,0}$
$P_{0,1}$	$P_{1,1}$	$P_{2,1}$	$P_{3,1}$
$P_{0,2}$	$P_{1,2}$	$P_{2,2}$	$P_{3,2}$
$P_{0,3}$	$P_{1,3}$	$P_{2,3}$	$P_{3,3}$



Ejercicio

- Con el raciocinio anterior, implementar un kernel para la multiplicación de dos matrices 4x4

Ejercicio

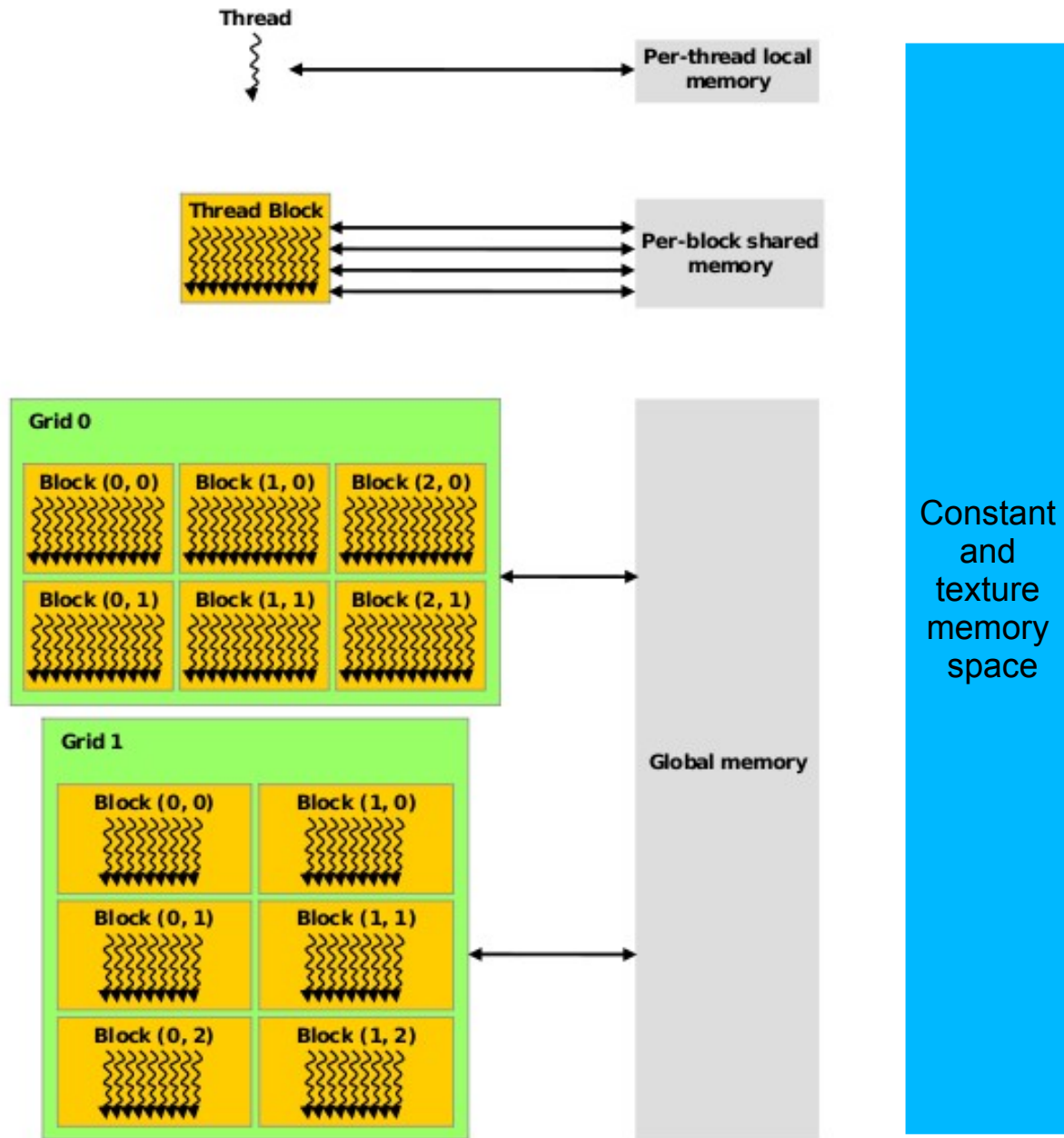
- Con el raciocinio anterior, implementar un kernel para la multiplicación de dos matrices 4x4
 - Desventajas
 - Matriz < 512 elementos

Ejercicio

- Con el raciocinio anterior, implementar un kernel para la multiplicación de dos matrices 4x4
 - Desventajas
 - Matriz < 512 elementos
- Implementar un programa para multiplicar matrices de tamaño arbitrario

Jerarquía de memoria

Jerarquía de memoria



Jerarquía de memoria

Table 3.1 Salient features of device memory

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

Jerarquía de memoria

- Para código corriendo en la GPU (`__device__` y `__global__`) la memoria usada para almacenar una variable puede ser especificada como:
 - `__device__`: the variable resides in the GPU's global memory and is defined while the code runs.
 - `__constant__`: the variable resides in the constant memory space of the GPU and is defined while the code runs
 - `__shared__`: the variable resides in the shared memory of the thread block and has the same lifespan as the block