



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

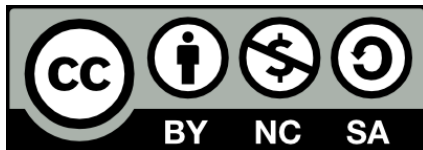
CUDA – Sesión 01

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

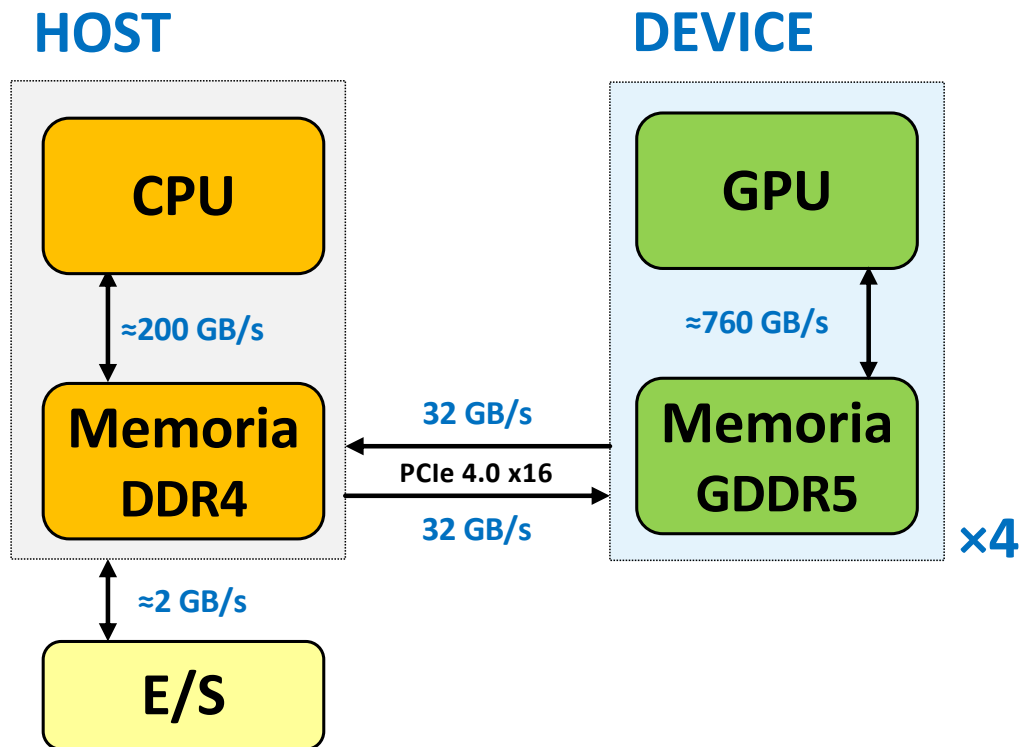
Universitat Politècnica de Catalunya



Entorno de Programación

- ❑ Usamos el servidor boada.ac.upc.edu (múltiples nodos).
- ❑ En [boada-10](#) están instaladas las 4 GPUs (RTX 3080).
- ❑ Cuando entramos en boada, lo hacemos en boada.ac.upc.edu.
- ❑ El sistema decide en que nodo nos ubica para editar / compilar / etc.
- ❑ Los programas han de tener acceso exclusivo a las 4 GPUs.
- ❑ No se puede ejecutar de forma interactiva en boada-10
- ❑ Para acceder a las GPUS usaremos la cola [cuda](#):
 - [sbatch job.sh](#)
- ❑ Algunos comandos útiles:
 - [scancel <job_id>](#)
 - [squeue](#)

Servidor (boada-10) and Devices (RTX 3080 ×4)



Servidor

- ❑ 2 CPUs Xeon 4314
- ❑ 16 cores por CPU
- ❑ 8 canales memoria por CPU
- ❑ 128 GB DDR4
- ❑ $\approx 200 \text{ GB/s}$ por CPU

GPUs (×4)

- ❑ GeForce RTX 3080
- ❑ 10 GB por GPU
- ❑ 760 GB/s

ScanDevices

- ❑ Cuando instalamos un CUDA o una nueva GPU siempre hemos de correr una aplicación simular a ésta, para obtener las características de las tarjetas instaladas. Esa información la genera la rutina `cudaGetDeviceProperties()`.

```
Device 0: "NVIDIA GeForce RTX 3080"
Major revision number:      8
Minor revision number:     6
Total amount of global memory: 10495655936 bytes
Number of multiprocessors:  68
Total CUDA Cores            8704
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size:                  32
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: 2147483647 x 65535 x 65535
Maximum memory pitch:       2147483647 bytes
Texture alignment:          512 bytes
Clock rate:                  1.71 GHz
Memory Clock rate:          9.50 GHz
Memory Bus Width:           320 bits
Number of asynchronous engines: 2
It can execute multiple kernels concurrently: Yes
Concurrent copy and execution: Yes
```

Computing Capability: 8.6

8
6

Computing Capability 8.6

- ❑ Hemos de usar esta información para compilar correctamente
 - Uno de los flags de compilación sirve para indicar la arquitectura de la GPU dónde se va a ejecutar nuestro programa:

```
-gencode arch=compute_86,code=[sm_86,compute_86]
```

- Si no sabemos que GPU tenemos instalada, o queremos que el mismo programa utilice varias GPUs diferentes, es posible generar código para varias GPUs:

```
ARCH= -gencode arch=compute_86,code=[sm_86,compute_86] \  
      -gencode arch=compute_35,code=[sm_35,compute_35] \  
      -gencode arch=compute_61,code=[sm_61,compute_61]  
  
nvcc $(ARCH) [...]
```

Computing Capability 8.6

- ❑ Cores per Multiprocesor: **128**
- ❑ Total amount of shared memory per block: **49.152 bytes**
- ❑ Total shared memory per multiprocessor: **102.400 bytes**
- ❑ Total number of 32 bit registers available per block: **65.536**
- ❑ Total number of 32-bit registers per thread: **255**
- ❑ Warp size: **32**
- ❑ Maximum number of threads per multiprocessor: **1.536**
- ❑ Maximum number of threads per block: **1.024**
- ❑ Max dimension size of a thread block (x,y,z): (**1.024, 1.024, 64**)
- ❑ Max dimension size of a grid size (x,y,z): (**2.147.483.647, 65.535, 65.535**)
- ❑ Maximum number of resident grids per device: **128**
- ❑ Maximum number of resident blocks per SM: **16**
- ❑ Maximum number of resident warps per SM: **48**

Información relevante de la GeForce RTX 3080

- ❑ Multiprocessors: **68**

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>

Detalles de programación

```
// Obtener Memoria en el host  
h_x = (float*) malloc(numBytes);  
h_y = (float*) malloc(numBytes);  
H_y= (float*) malloc(numBytes);
```

```
// // Obtiene Memoria [pinned] en el host  
cudaMallocHost((float**)&h_x, numBytes);  
cudaMallocHost((float**)&h_y, numBytes);  
cudaMallocHost((float**)&H_y, numBytes);
```

Es de buen programador comprobar que los malloc han funcionado bien

```
// Obtener Memoria en el host  
h_x = (float*) malloc(numBytes);  
if (h_x == NULL) ERROR&EXIT;
```

Gestión de errores en CUDA

```
// Obtener Memoria en el device
cudaMalloc((float**)&d_x, numBytes);
cudaMalloc((float**)&d_y, numBytes);
CheckCudaError((char *) "Obtener Memoria en el device", __LINE__);
```

Ponemos CheckCudaError después de cada llamada CUDA

```
void CheckCudaError(char sms[], int line) {
    cudaError_t error;
    error = cudaGetLastError();
    if (error) {
        printf("(ERROR) %s - %s in %s at line %d\n", sms,
            cudaGetErrorString(error), __FILE__, line);
        exit(EXIT_FAILURE);
    }
}
```


Gestión de errores en CUDA

```
...
// Obtener Memoria en el device
cudaMalloc((float**)&d_x, numBytes);
cudaMalloc((float**)&d_y, numBytes);
CheckCudaError((char *) "Obtener Memoria en el device", __LINE__);

// Copiar datos desde el host en el device
cudaMemcpy(d_x, h_x, numBytes, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, numBytes, cudaMemcpyHostToDevice);
CheckCudaError((char *) "Copiar Datos Host --> Device", __LINE__);

// Ejecutar el kernel
saxpyP<<<nBlocks, nThreads>>>(N, 3.5, d_x, d_y);
CheckCudaError((char *) "Invocar Kernel", __LINE__);

// Guardamos el resultado en H_y para poder comprobar el resultado
cudaMemcpy(H_y, d_y, numBytes, cudaMemcpyDeviceToHost);
CheckCudaError((char *) "Copiar Datos Device --> Host", __LINE__);
...
```

¡No funciona!

Los ejecución
de un kernel es
asíncrona.

```
saxpyP<<<...>>>(...);
if (debug) {
    cudaDeviceSynchronize();
    CheckCudaError(...);
}
```

Estructura de un programa CUDA

```
// Obtener Memoria en el host  
  
// Inicializar datos en el host  
  
// Obtener memoria en el device  
  
// Copiar datos del host en el device  
// Ejecutar el kernel  
// Copiar el resultado del device en el host  
  
// Liberar espacio en el host y en el device
```

Código a
Evaluar

Código a
Evaluar

¿Cómo evaluamos un código?: Tiempo de ejecución

```
• • •  
    cudaEventRecord(E1, 0);
```

```
// Copiar datos del host en el device  
// Ejecutar el kernel  
// Copiar el resultado del device en el host
```

```
    cudaEventRecord(E4, 0);  
    cudaEventSynchronize(E4);
```

```
• • •  
    cudaEventElapsedTime(&elapsedTime, E1, E4);
```

Tiempo Global: 42.880577 milseg
Tiempo Kernel: 0.311104 milseg
Tiempo HtD: 12.117184 milseg
Tiempo DtH: 30.452288 milseg

Los tiempos no son
siempre coherentes

Código a
Evaluar

elapsedTime (float) nos da el tiempo de ejecución medido en milisegundos

Otra forma de evaluar: nvprof

- ❑ Ejecutar el comando con:

```
nsys nvprof --print-gpu-summary ./SaxpyP.exe
```

CUDA Kernel Statistics:

Time(%)	Total Time (ns)	Instances	Average	Minimum	Maximum	Name
100.0	295,713	1	295,713.0	295,713	295,713	saxpyP(unsigned int, float, float*, float*)

CUDA Memory Operation Statistics (by time):

Time(%)	Total Time (ns)	Operations	Average	Minimum	Maximum	Operation
71.4	56,979,316	1	56,979,316.0	56,979,316	56,979,316	[CUDA memcpy DtoH]
28.6	22,872,687	2	11,436,343.5	10,914,008	11,958,679	[CUDA memcpy HtoD]

- ❑ Tiempos obtenidos por programa en la misma ejecución:

```
Tiempo Global: 59.554367 milseg  
Tiempo Kernel: 0.348704 milseg  
Tiempo HtD: 27.825697 milseg  
Tiempo DtH: 31.379969 milseg
```

Por ahora utilizaremos los tiempos que obtenemos por programa.

Ancho de Banda entre CPU y GPU

- ❑ Nos centramos en las transferencias CPU – GPU (PCIe)

```
cudaMemcpy(d_x, h_x, numBytes, cudaMemcpyHostToDevice);
```

```
numBytes = 4*N = 226 bytes = 67.108.864 bytes (64 MB)
```

Tiempo Global: 42.9 milseg

Tiempo Kernel: 0.3 milseg

Tiempo HtD: 12.1 milseg

Tiempo DtH: 30.5 milseg

N = 16.777.216

AB_{\max}
32+32 GB/s

AnchoBanda (HtoD) = 2^{27} bytes/12,1 ms = 11,09 GB/s

AnchoBanda (DtoH) = 2^{26} bytes/30,5 ms = 2,20 GB/s

AnchoBanda = $3 \cdot 2^{26}$ bytes/42,6 ms = 4,73 GB/s

GFLOPS y Ancho de Banda con Memoria Global

- ❑ Nos centramos en el kernel

```
__global__ void saxpyP (int N, float a, float *x, float *y) {  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    y[i] = a * x[i] + y[i];  
}
```

$N = 16.777.216$

flops = $2*N = 2^{25}$ Ops CF = 33.554.432 Ops en CF

Accesos a memoria = $2*N$ reads + N writes = $3*4*N$ bytes = $12 \cdot 2^{24}$ bytes

Tiempo Kernel: 0,3 ms

Tiempo Total: 42,9 ms

GFLOPS(Kernel) = $2^{25} \text{ flops} / (0,3\text{ms} \cdot 10^6) = 111,85 \text{ GFLOPS}$

GFLOPS(Global) = $2^{25} \text{ flops} / (42,9\text{ms} \cdot 10^6) = 782,2 \text{ MFLOPS}$

AB Memoria Global = $12 \cdot 2^{24} \text{ bytes} / 0,3\text{ms} = 671,1 \text{ GB/s}$

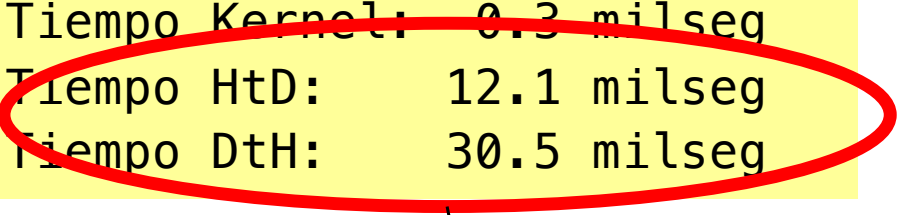
AB_{RTX3080} = 760GB/s

Pinned vs No-pinned Memory

```
// Obtener Memoria en el host  
h_x = (float*) malloc(numBytes);
```

```
// Obtiene Memoria [pinned] en el host  
cudaMallocHost((float**)&h_y, numBytes);
```

```
Tiempo Global: 42.9 milseg  
Tiempo Kernel: 0.3 milseg  
Tiempo HtD: 12.1 milseg  
Tiempo DtH: 30.5 milseg
```



```
Tiempo Global: 8.18 milseg  
Tiempo Kernel: 0.31 milseg  
Tiempo HtD: 5.32 milseg  
Tiempo DtH: 2.55 milseg
```

¿?

Tiene que ver con la Memoria Virtual

Ancho de Banda entre CPU y GPU con Pinned Memory

- ❑ Nos centramos en las transferencias CPU – GPU (PCIe)

```
cudaMemcpy(d_x, h_x, numBytes, cudaMemcpyHostToDevice);
```

```
numBytes = 4*N = 226 bytes = 67.108.864 bytes (64 MB)
```

Tiempo Global: 8.18 milseg

Tiempo Kernel: 0.31 milseg

Tiempo HtD: 5.32 milseg

Tiempo DtH: 2.55 milseg

N = 16.777.216

AB_{\max}
32+32 GB/s

AnchoBanda (HtoD) = 2^{27} bytes / 5,32 ms = 25,23 GB/s

AnchoBanda (DtoH) = 2^{26} bytes / 2,55 ms = 26,32 GB/s

AnchoBanda = $3 \cdot 2^{26}$ bytes / 7,85 ms = 25,65 GB/s

Jugando con el número de Threads

```
nThreads = ...; // 32, 64, 128, 256, 512, 1024
nBlocks = N/nThreads;
saxpyP<<nBlocks, nThreads>>>(N, 3.5, d_x, d_y);
```

nThreads	nBlocks	Tiempo Kernel
32	524288	0,408 ms
64	262144	0,308-0337 ms
128	131072	
256	65536	
512	32768	
1024	16384	

Tiempo de Kernel,
calculado con
eventos

Entre 1024 – 64, no hay grandes diferencias, con 32 si las hay

¿Qué pasa cuando N no es múltiplo de nThreads?

Jugando con el número de Threads

```
nThreads = ...; // 50, 100, 200, 500, 1500
nBlocks = N/nThreads;
saxpyP<<nBlocks, nThreads>>>(N, 3.5, d_x, d_y);
```

nThreads	
50	TEST FAIL
100	
200	
500	
1500	(ERROR) Invocar Kernel – invalid configuration argument in main.cu at line 117

```
__global__ void saxpyP (int N, ...) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    y[i] = a * x[i] + y[i];
}
```

nBlocks está
mal calculado

Jugando con el número de Threads

```
nThreads = ...; // 50, 100, 200, 500, 1500
nBlocks = (N+nThreads-1)/nThreads;
saxpyP<<nBlocks, nThreads>>>(N, 3.5, d_x, d_y);
```

nThreads	
50	TEST FAIL
100	
200	
500	
1500	(ERROR) Invocar Kernel – invalid configuration argument in main.cu at line 117

```
__global__ void saxpyP (int N, ...) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    y[i] = a * x[i] + y[i];
}
```

Estamos
usando $i > N$

Jugando con el número de Threads

```
nThreads = ...; // 50, 100, 200, 500, 1500
nBlocks = (N+nThreads-1)/nThreads;
saxpyP<<nBlocks, nThreads>>>(N, 3.5, d_x, d_y);
```

nThreads	
50	TEST PASS
100	
200	
500	
1500	(ERROR) Invocar Kernel – invalid configuration argument in main.cu at line 117

```
__global__ void saxpyP (int N, ...) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i<N)
        y[i] = a * x[i] + y[i];
}
```

Funciona, pero
con peores
rendimientos

Jugando con el número de Blocks

```
N = 1024 * 1024 * 128;  
nThreads = 8;  
nBlocks = N/nThreads; // nBlocks = 16777216  
saxpyP<<<nBlocks, nThreads>>>(N, 3.5, d_x, d_y);
```

Versión CUDA : 8.0.61

(ERROR) Invocar Kernel – invalid argument in main.cu at line 77

Versión CUDA: 11.2.1

```
nThreads: 8  
nBlocks: 16777216  
Tiempo Global: 403.896484 milseg  
Tiempo Kernel: 11.771200 milseg  
...  
TEST PASS
```

Maximum sizes of each dimension of a grid: 2147483647 x 65535 x 65535

Otra forma de hacer las cosas

```
N = 1024 * 1024 * 128;
nBlocks = 10000;
nThreads = 1024;
saxpyP<<<nBlocks, nThreads>>>(N, 3.5, d_x, d_y);
```

Fijamos el número de bloques y el número de threads.

```
__global__ void saxpyP (int N, ...) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    while (i<N) {
        y[i] = a * x[i] + y[i];
        i = i + blockDim.x * blockDim.x;
    }
}
```

Hay que cambiar el kernel

El rendimiento es similar.

```
nThreads: 1024
nBlocks: 10000
Tiempo Global: 384.137817 milseg
Tiempo Kernel: 2.371616 milseg
...
TEST PASS
```

```
nThreads: 1024
nBlocks: 16777216
Tiempo Global: 386.870850 milseg
Tiempo Kernel: 2.357504 milseg
...
TEST PASS
```

Liberar espacio

Una buena práctica es liberar todo el espacio que hemos pedido, cuando ya no sea necesario.

```
// Obtener Memoria en el host
h_x = (float*) malloc(numBytes);

// Obtiene Memoria [pinned] en el host
cudaMallocHost((float**)&h_y, numBytes);
```

```
// Liberar Memoria
free(h_x);
h_x = NULL;

// Liberar Memoria [pinned]
cudaFreeHost(h_y);
h_y = NULL;
```

```
// Obtener Memoria en el device
cudaMalloc((float**)&d_x, numBytes);
```

```
// Liberar Memoria en el device
cudaFree(d_x);
d_x = NULL;
```

Errores de precisión

```
int error(float a, float b) {  
    if (abs (a - b) / a > 0.000001) return 1;  
    else return 0;  
}
```

```
int error(float a, float b) {  
    return a != b;  
}
```

```
int error(double a, double b) {  
    return a != b;  
}
```

Con double no hay error de precisión

```
nThreads: 1024  
nBlocks: 16384  
Tiempo Global: 81.003555 milseg  
Tiempo Kernel: 0.321600 milseg  
...  
TEST PASS
```

```
nThreads: 1024  
nBlocks: 16384  
Tiempo Global: 81.136963 milseg  
Tiempo Kernel: 0.321664 milseg  
...  
TEST FAIL
```

```
nThreads: 1024  
nBlocks: 16384  
Tiempo Global: 185.503235 milseg  
Tiempo Kernel: 0.623680 milseg  
...  
TEST PASS
```




UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

CUDA – Sesión 01

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

