UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

CUDA – Sesión 02 - Puzzles

Agustín Fernández

Departament d'Arquitectura de Computadors

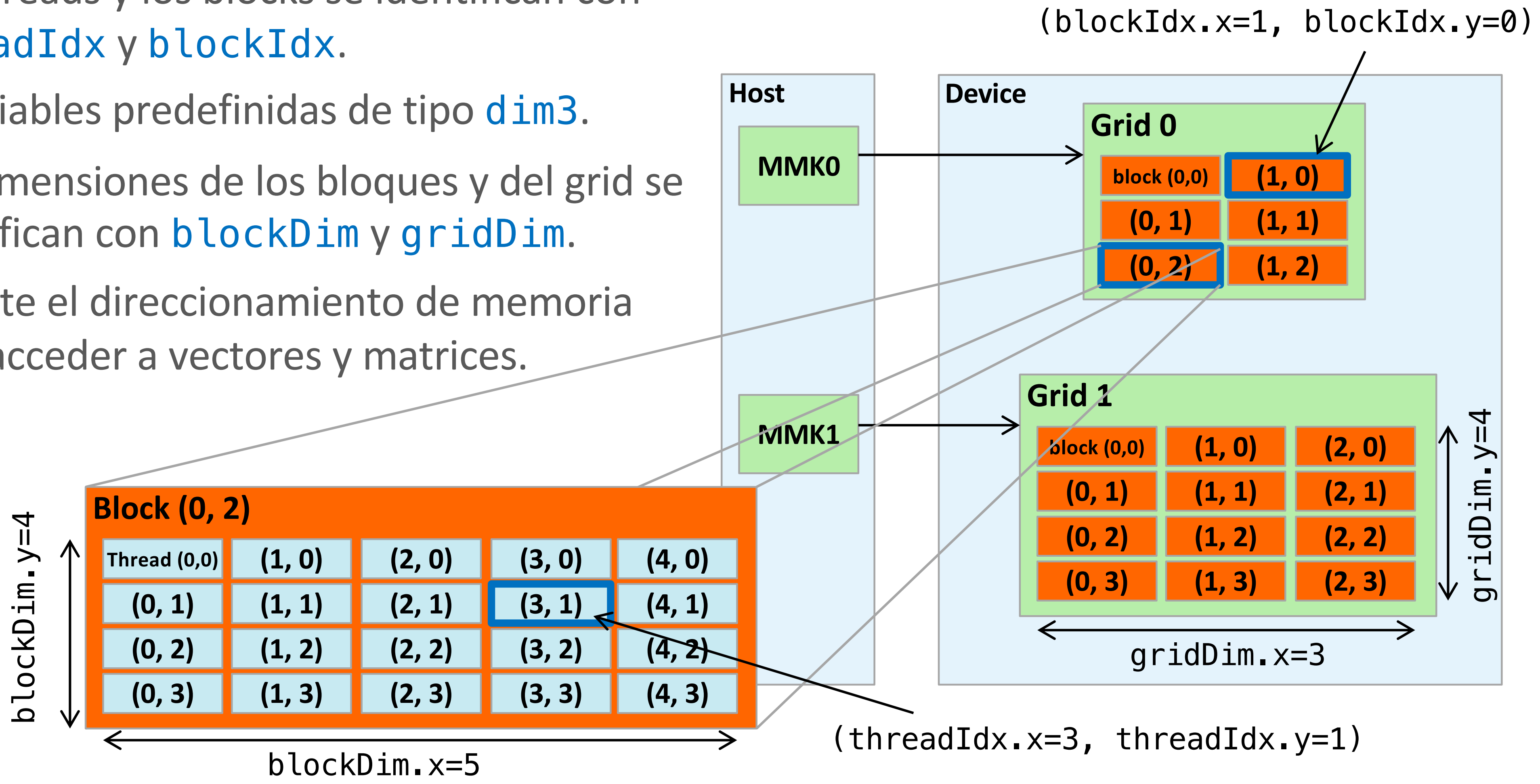
Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

Museu d'Arquitectura de Computadors

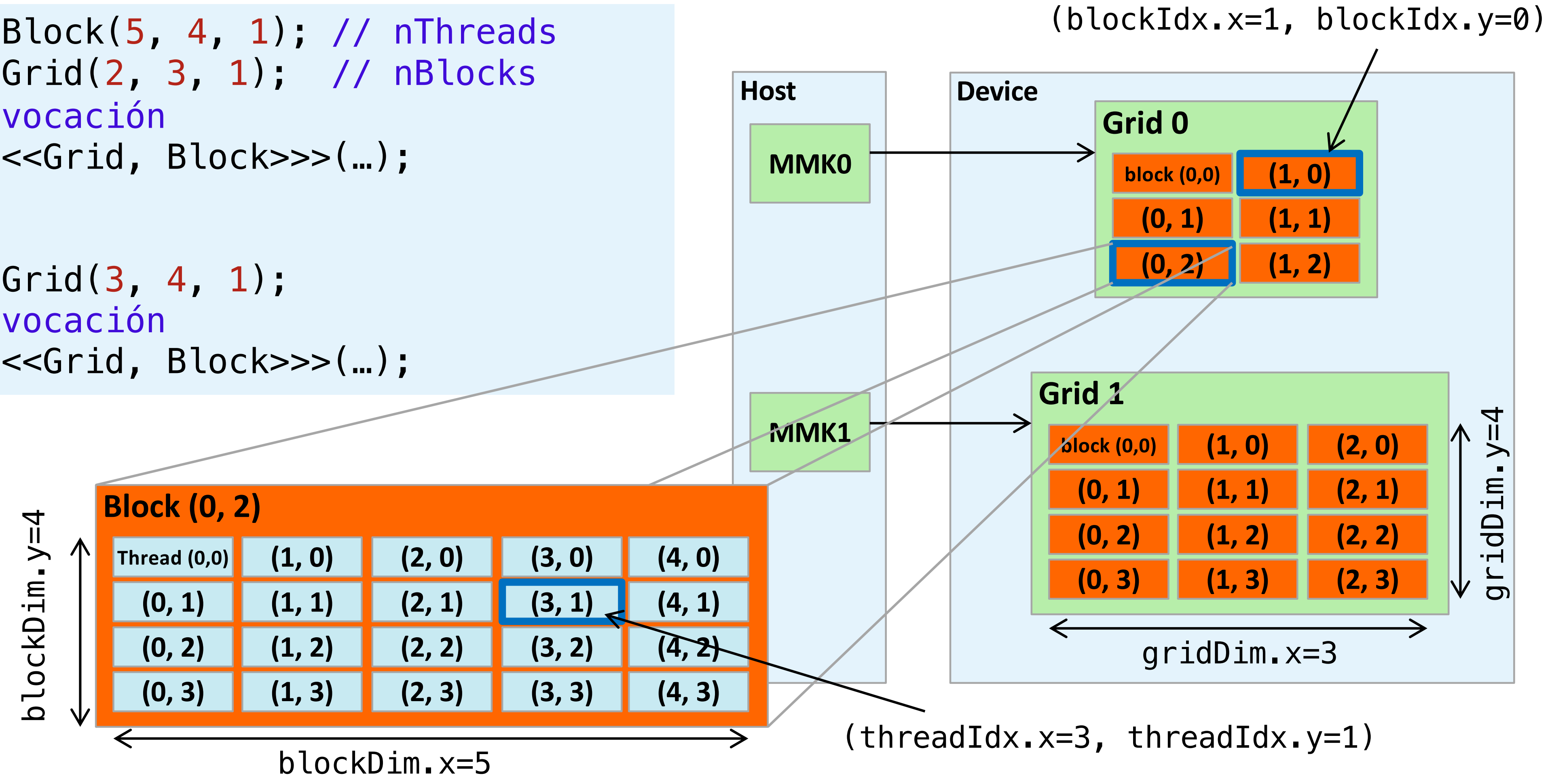
blockIdx & threadIdx

- Los threads y los blocks se identifican con `threadIdx` y `blockIdx`.
  - Variables predefinidas de tipo `dim3`.
- Las dimensiones de los bloques y del grid se identifican con `blockDim` y `gridDim`.
- Permite el direccionamiento de memoria para acceder a vectores y matrices.



blockIdx & threadIdx

```
dim3 Block(5, 4, 1); // nThreads
dim3 Grid(2, 3, 1);  // nBlocks
// Invocación
MMK0<<<Grid, Block>>>(…);


dim3 Grid(3, 4, 1);
// Invocación
MMK1<<<Grid, Block>>>(…);
```



Grid 0

block (0,0)	(1, 0)
(0, 1)	(1, 1)
(0, 2)	(1, 2)

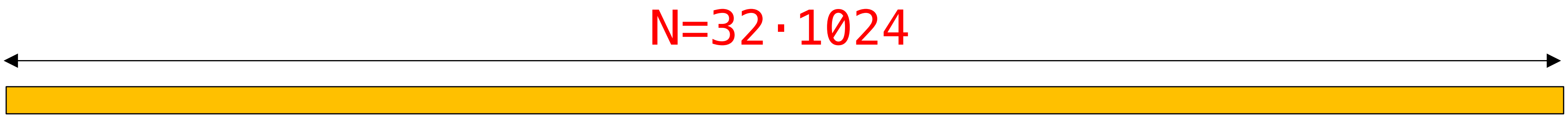
Grid 1

block (0,0)	(1, 0)	(2, 0)
(0, 1)	(1, 1)	(2, 1)
(0, 2)	(1, 2)	(2, 2)
(0, 3)	(1, 3)	(2, 3)

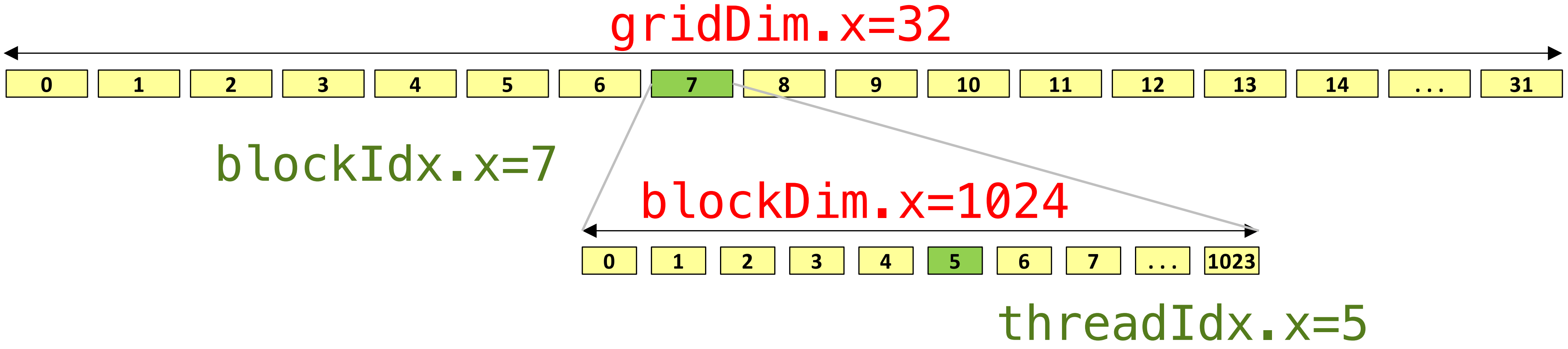
Block (0, 2)

Thread (0,0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)

Puzzle 1D



```
dim3 dimBlock(1024, 1, 1);
dim3 dimGrid((N+1023)/1024, 1, 1);
puzzle1DPAR<<<dimGrid, dimBlock>>>(N, . . .);
```



Id = blockIdx.x ∗ blockDim.x + threadIdx.x = 7·1024+5 = 7173

Puzzle1D

```
void puzzle1DSeq(int N, float *z, float *x, float *y) {
  int i;
  for (i=0; i<N; i++)
    z[i] = 0.5*x[i] + 0.75*y[i] + x[i]*y[i];
}
```

```
__global__ void puzzle1DPAR(int N, float *z, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  z[i] = 0.5*x[i] + 0.75*y[i] + x[i]*y[i];
}

nThreads = 1024;
nBlocks = N/nThreads;
dim3 dimGrid(nBlocks, 1, 1);
dim3 dimBlock(nThreads, 1, 1);
puzzle1DPAR<<<dimGrid, dimBlock>>>(N, dZ, dX, dY);
```

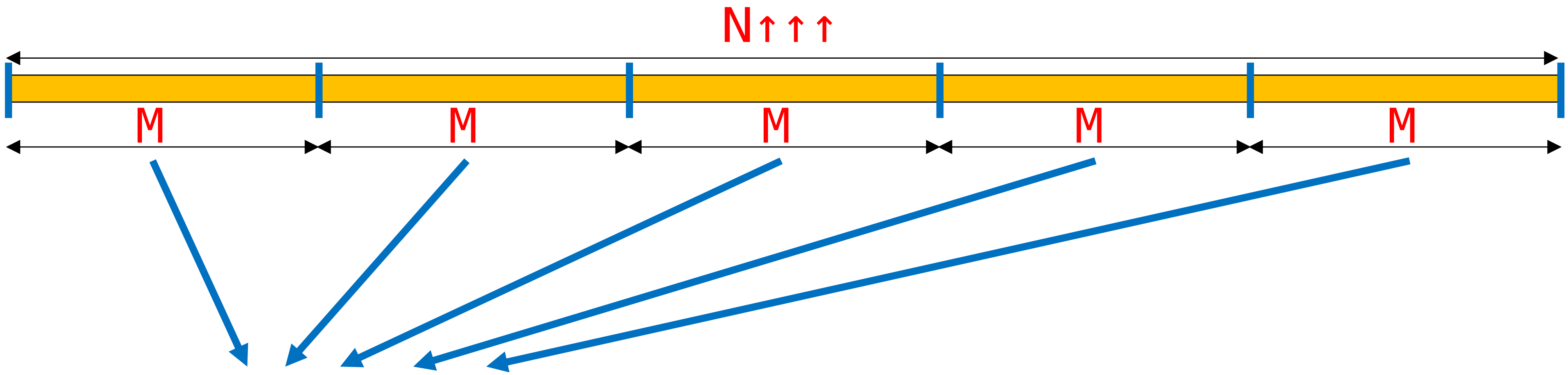
Puzzle1D. Cuestiones

2. Cambiad el tamaño para que **NO** sea múltiplo del número de threads. Modificad el código, dónde sea necesario, para hacer que funcione correctamente.

```
__global__ void puzzle1DPAR(int N, float *z, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
    z[i] = 0.5*x[i] + 0.75*y[i] + x[i]*y[i];
}

nThreads = 1024;
nBlocks = (N + nThreads - 1)/nThreads;
dim3 dimGrid(nBlocks, 1, 1);
dim3 dimBlock(nThreads, 1, 1);
puzzle1DPAR<<<dimGrid, dimBlock>>>(N, dZ, dX, dY);
```

Puzzle1D. Cuestiones

3. ¿Qué modificarías en la implementación para considerar ahora que el tamaño del problema **NO** cabe en la memoria de la GPU?

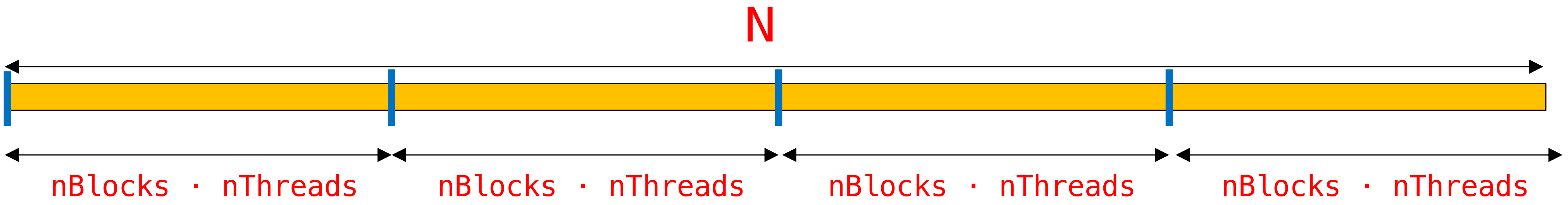


```
cudaMemcpy(…, M, HtD);
kernel<<<…>>>(M, …);
cudaMemcpy(…, M, DtH);
```

Hay que trocear el problema, para que los subproblemas quepan en memoria. Sólo hay que manipular punteros.

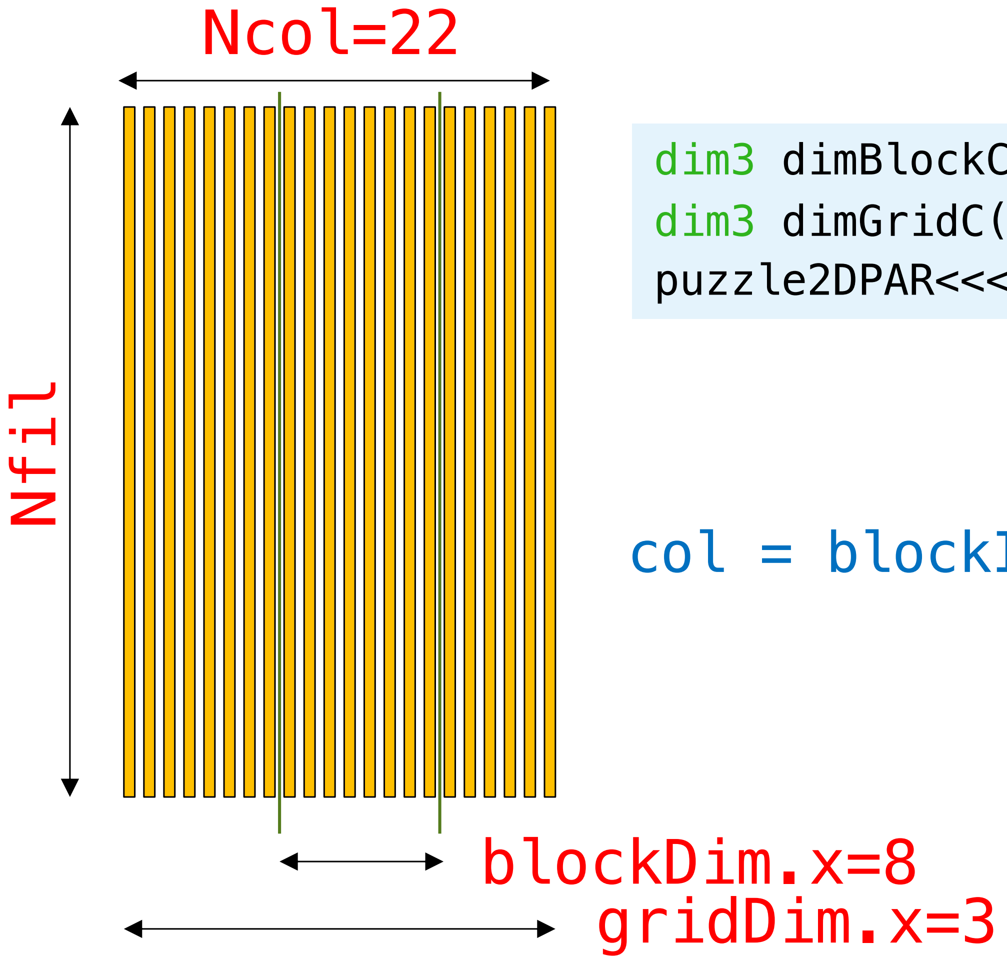
Puzzle1D. Cuestiones

4. ¿Qué modificarías en la implementación si el número de Blocks es fijo?



```
__global__ void puzzle1DPAR(int N, float *z, float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  while (i < N) {
    z[i] = 0.5*x[i] + 0.75*y[i] + x[i]*y[i];
    i = i + stride;
  }
}
puzzle1DPAR<<<10000, 1024>>>(N, dZ, dX, dY);
```

Puzzle 2D por columnas



```
dim3 dimBlockC(8, 1, 1);
dim3 dimGridC((22+7)/8, 1, 1);
puzzle2DPAR<<<dimGridC, dimBlockC>>>(N, . . .);
```

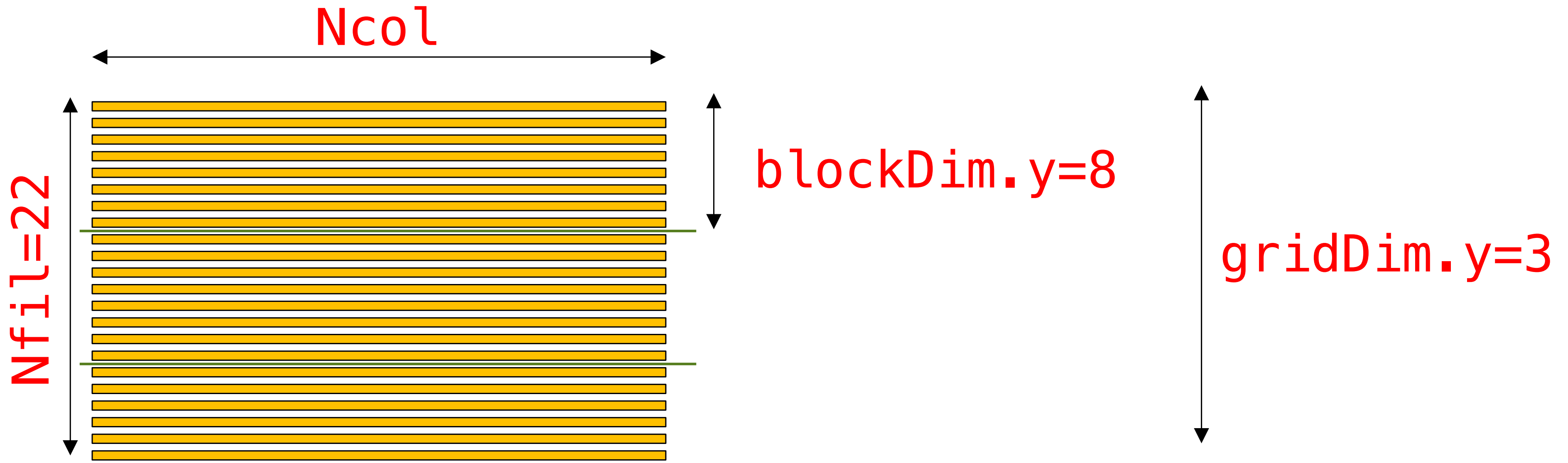
```
col = blockIdx.x * blockDim.x + threadIdx.x
```

Puzzle2D por columnas

```
void puzzle2DSeq(int Nfil, int Ncol, float *z, float *x, float *y) {
  int i, j, ind;
  for (i=0; i<Nfil; i++)
    for (j=0; j<Ncol; j++){
      ind = i * Ncol + j;
      z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
    }
}
```

```
__global__ void puzzle2DPARcol(int Nfil, int Ncol, float *z, float *x, float *y) {
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int ind = j;
  if (j < Ncol)
    for (int i=0; i<Nfil; i++, ind = ind + Ncol)
      z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
}

nThreads = 256;
nBlocks = (Ncol + nThreads - 1)/nThreads;
dim3 dimGridC(nBlocks, 1, 1);
dim3 dimBlockC(nThreads, 1, 1);
puzzle2DPARcol<<<dimGridC, dimBlockC>>>(Nfil, Ncol, dZ, dX, dY);
```

Puzzle 2D por filas



```
dim3 dimBlockF(1, 8, 1);
dim3 dimGridF(1, (22+7)/8, 1);
puzzle2DPAR<<<dimGridF, dimBlockF>>>(N, . . .);
```

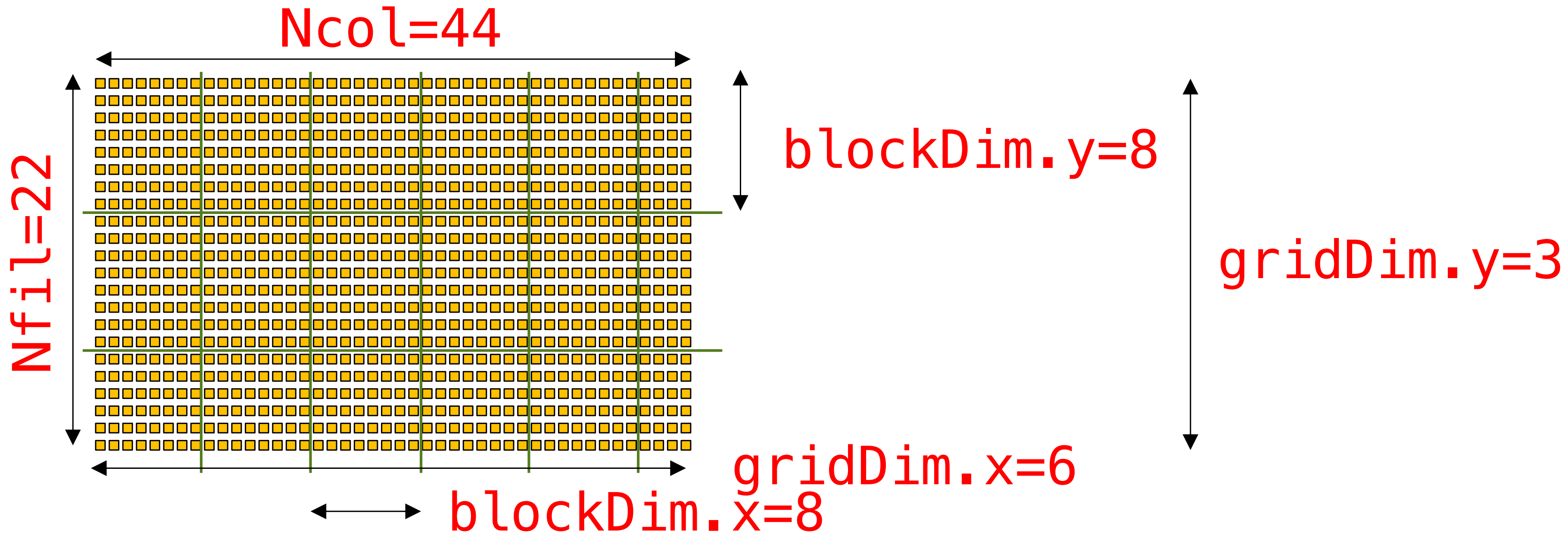
fil = blockIdx.y ∗ blockDim.y + threadIdx.y

Puzzle2D por filas

```
void puzzle2DSeq(int Nfil, int Ncol, float *z, float *x, float *y) {
  int i, j, ind;
  for (i=0; i<Nfil; i++)
    for (j=0; j<Ncol; j++){
      ind = i * Ncol + j;
      z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
    }
}
```

```
__global__ void puzzle2DPARfil(int Nfil, int Ncol, float *z, float *x, float *y) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int ind = i * Ncol;
  if (i < Nfil)
    for (int j=0; j<Ncol; j++, ind++)
      z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
}

nThreads = 256;
nBlocks = (Nfil + nThreads - 1)/nThreads;
dim3 dimGridF(1, nBlocks, 1);
dim3 dimBlockF(1, nThreads, 1);
puzzle2DPARfil<<<dimGridF, dimBlockF>>>(Nfil, Ncol, dZ, dX, dY);
```

Puzzle 2D elemento a elemento



Ncol=44

Nfil=22

blockDim.y=8

gridDim.y=3

gridDim.x=6

blockDim.x=8

```
dim3 dimBlockE(8, 8, 1);
dim3 dimGridE((44+7)/8, (22+7)/8, 1);
puzzle2DPAR<<<dimGridE, dimBlockE>>>(N, . . .);
```

```
fil = blockIdx.y * blockDim.y + threadIdx.y
col = blockIdx.x * blockDim.x + threadIdx.x
```

Puzzle2D elemento a elemento

```
void puzzle2DSeq(int Nfil, int Ncol, float *z, float *x, float *y) {
  int i, j, ind;
  for (i=0; i<Nfil; i++)
    for (j=0; j<Ncol; j++){
      ind = i * Ncol + j;
      z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
    }
}
```

```
__global__ void puzzle2DPAR1x1(int Nfil, int Ncol, float *z, float *x, float *y) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int ind = i * Ncol + j;
  if (i < Nfil && j < Ncol)
    z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
}
nThreads = 16;
nBlocksCol = (Ncol + nThreads - 1)/nThreads;
nBlocksFil = (Nfil + nThreads - 1)/nThreads;
dim3 dimGridE(nBlocksCol, nBlocksFil, 1);
dim3 dimBlockE(nThreads, nThreads, 1);
puzzle2DPAR1x1<<<dimGridE, dimBlockE>>>(Nfil, Ncol, dZ, dX, dY);
```

Puzzle2D columnas vs filas vs elementos

```
__global__ void puzzle2DPARcol(int Nfil, int Ncol, float *z, float *x, float *y) {
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int ind = j;
  if (j < Ncol)
    for (int i=0; i<Nfil; i++, ind = ind + Ncol)
      z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
}
```

```
nTh = 256;
nBlC = (Ncol + nTh-1)/nTh;
dim3 dimGridC(nBlC, 1, 1);
dim3 dimBlockC(nTh, 1, 1);
```

```
__global__ void puzzle2DPARfil(int Nfil, int Ncol, float *z, float *x, float *y) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int ind = i * Ncol;
  if (i < Nfil)
    for (int j=0; j<Ncol; j++, ind++)
      z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
}
```

```
nTh = 256;
nBlF = (Nfil + nTh-1)/nTh;
dim3 dimGridF(1, nBlF, 1);
dim3 dimBlockF(1, nTh, 1);
```

```
__global__ void puzzle2DPAR1x1(int Nfil, int Ncol, float *z, float *x, float *y) {
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int ind = i * Ncol + j;
  if (i < Nfil && j < Ncol)
    z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
}
```

```
nTh = 16;
nBlC = (Ncol + nTh-1)/nTh;
nBlF = (Nfil + nTh-1)/nTh;
dim3 dimGridE(nBlC, nBlF, 1);
dim3 dimBlockE(nTh, nTh, 1);
```

Puzzle2D columnas vs filas vs elementos

```
nThreads = 256;
nBlocks = (Ncol + nThreads – 1)/nThreads;
dim3 dimGridC(nBlocks, 1, 1);
dim3 dimBlockC(nThreads, 1, 1);
puzzle2DPARcol<<<dimGridC, dimBlockC>>>(Nfil, Ncol, dZ, dX, dY);
```

```
nThreads = 256;
nBlocks = (Nfil + nThreads – 1)/nThreads;
dim3 dimGridF(1, nBlocks, 1);
dim3 dimBlockF(1, nThreads, 1);
puzzle2DPARfil<<<dimGridF, dimBlockF>>>(Nfil, Ncol, dZ, dX, dY);
```

```
nThreads = 16;
nBlocksCol = (Ncol + nThreads – 1)/nThreads;
nBlocksFil = (Nfil + nThreads – 1)/nThreads;
dim3 dimGridE(nBlocksCol, nBlocksFil, 1);
dim3 dimBlockE(nThreads, nThreads, 1);
puzzle2DPAR1x1<<<dimGridE, dimBlockE>>>(Nfil, Ncol, dZ, dX, dY);
```

Puzzle2D Rendimiento

```
Kernel por Filas
Dimension problema: 1023 filas x 1023 columnas
Dimension Block: 1 x 256 x 1 (256) threads
Dimension Grid: 1 x 4 x 1 (4) blocks

Kernel por Columnas
Dimension problema: 1023 filas x 1023 columnas
Dimension Block: 256 x 1 x 1 (256) threads
Dimension Grid: 4 x 1 x 1 (4) blocks

Kernel Elemento a Elemento
Dimension problema: 1023 filas x 1023 columnas
Dimension Block: 16 x 16 x 1 (256) threads
Dimension Grid: 64 x 64 x 1 (4096) blocks

Resumen Rendimiento
Tiempo Paralelo Kernel filas: 0.670912 ms (7.80 GFLOPS)
Tiempo Paralelo Kernel columnas: 0.640416 ms (8.17 GFLOPS)
Tiempo Paralelo Kernel elemento a elemento: 0.052288 ms (100.07 GFLOPS)
Tiempo Secuencial: 1.088600 milseg (4.81 GFLOPS)
```

¿Explicación de los rendimientos?

Número de Bloques

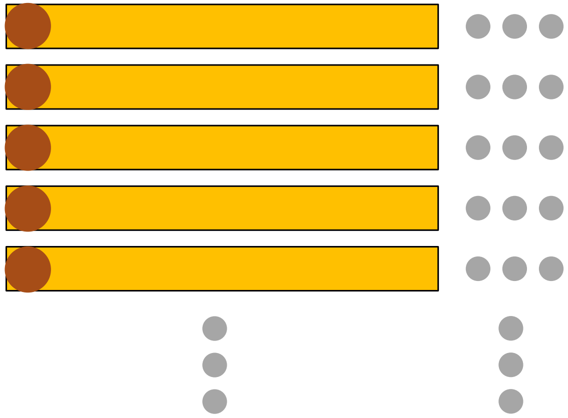
Patrones de acceso a memoria de los warps

RTX 3090

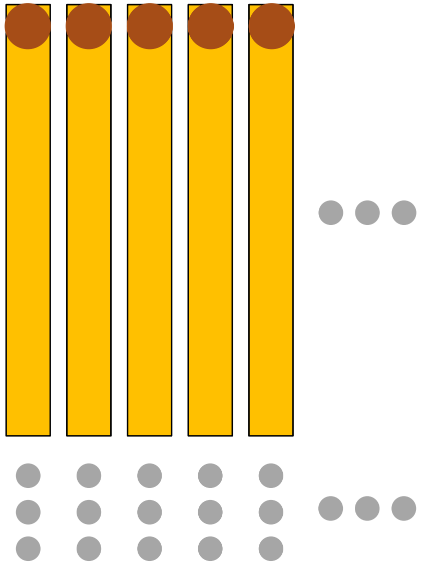
Puzzle2D Rendimiento

¿Explicación de los rendimientos?

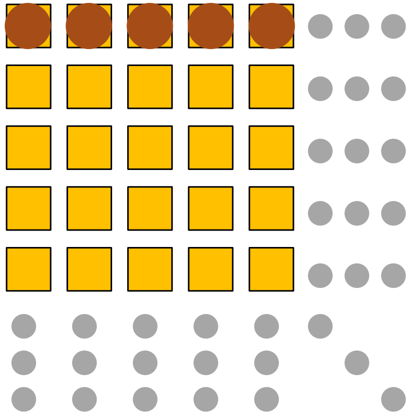
Patrones de acceso a memoria de los warps



Kernel por Filas
4 blocks (256 threads)
7.80 GFLOPS (RTX 3080)
1.92 GFLOPS (K40c)
6.97 GFLOPS (GTX 1080Ti)



Kernel por Columnas
4 blocks (256 threads)
8.17 GFLOPS (RTX 3080)
7.09 GFLOPS (K40c)
9.62 GFLOPS (GTX 1080Ti)



Kernel Elemento a Elemento
4096 blocks (16x16 threads)
100.07 GFLOPS (RTX 3080)
44.59 GFLOPS (K40c)
78.77 GFLOPS (GTX 1080Ti)

Puzzle3D elemento a elemento

```c
void puzzle3DSeq(int Ncar, int Nfil, int Ncol, float *z, float *x, float *y) {
  int i, j, t, ind;
  for (t=0; t<Ncar; t++)
    for (i=0; i<Nfil; i++)
      for (j=0; j<Ncol; j++) {
        ind = t*Nfil*Ncol + i*Ncol + j;
        z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
      }
}
```

```c
__global__ void puzzle3DPAR1x1x1(int Ncar, int Nfil, int Ncol, float *z, float *x, float *y){
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int t = blockIdx.z * blockDim.z + threadIdx.z;
  int ind = t*Nfil*Ncol + i*Ncol + j;
  if (t<Ncar && i<Nfil && j<Ncol)
    z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
}
```

Puzzle3D elemento a elemento

```
int nThreads = 8;
int nBlocksFil = (Nfil+nThreads-1)/(nThreads);
int nBlocksCol = (Ncol+nThreads-1)/nThreads;
int nBlocksCar = (Ncar+nThreads-1)/nThreads;

dim3 dimGridE(nBlocksCol, nBlocksFil, nBlocksCar);
dim3 dimBlockE(nThreads, nThreads, nThreads);
puzzle3DPAR1x1x1<<<dimGridE, dimBlockE>>>(Ncar, Nfil, Ncol, dZ, dX, dY);
```

```
__global__ void puzzle3DPAR1x1x1(int Ncar, int Nfil, int Ncol, float *z, float *x, float *y){
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int t = blockIdx.z * blockDim.z + threadIdx.z;
  int ind = t*Nfil*Ncol + i*Ncol + j;
  if (t<Ncar && i<Nfil && j<Ncol)
    z[ind] = 0.5*x[ind] + 0.75*y[ind] + x[ind]*y[ind];
}
```

Uso de nvprof

```
nsys nvprof --print-gpu-summary ./puzzle2D.exe 1024 1024 Y
```

```
CUDA Kernel Statistics:

 Time(%)  Total Time (ns)  Instances   Average   Minimum  Maximum                       Name
 -------  ---------------  ---------  ---------  -------  -------  ------------------------------------------------
    49.2          643,428          1  643,428.0  643,428  643,428  puzzle2DPARfil(int, int, float*, float*, float*)
    47.8          624,741          1  624,741.0  624,741  624,741  puzzle2DPARcol(int, int, float*, float*, float*)
     3.0           39,424          1   39,424.0   39,424   39,424  puzzle2DPAR1x1(int, int, float*, float*, float*)

CUDA Memory Operation Statistics (by time):

 Time(%)  Total Time (ns)  Operations   Average   Minimum  Maximum      Operation
 -------  ---------------  ----------  ---------  -------  -------  ------------------
    65.2          977,575           3  325,858.3  325,730  325,923  [CUDA memcpy DtoH]
    34.8          521,284           2  260,642.0  259,041  262,243  [CUDA memcpy HtoD]

CUDA Memory Operation Statistics (by size in KiB):

   Total     Operations   Average    Minimum    Maximum        Operation
 ----------  ----------  ---------  ---------  ---------  ------------------
  8,176.008           2  4,088.004  4,088.004  4,088.004  [CUDA memcpy HtoD]
 12,264.012           3  4,088.004  4,088.004  4,088.004  [CUDA memcpy DtoH]
```

También da una lista de todas las llamadas a la API de CUDA realizadas

Uso de nvprof

```
nsys nvprof --print-gpu-trace ./puzzle2D.exe 1024 1024 Y
```

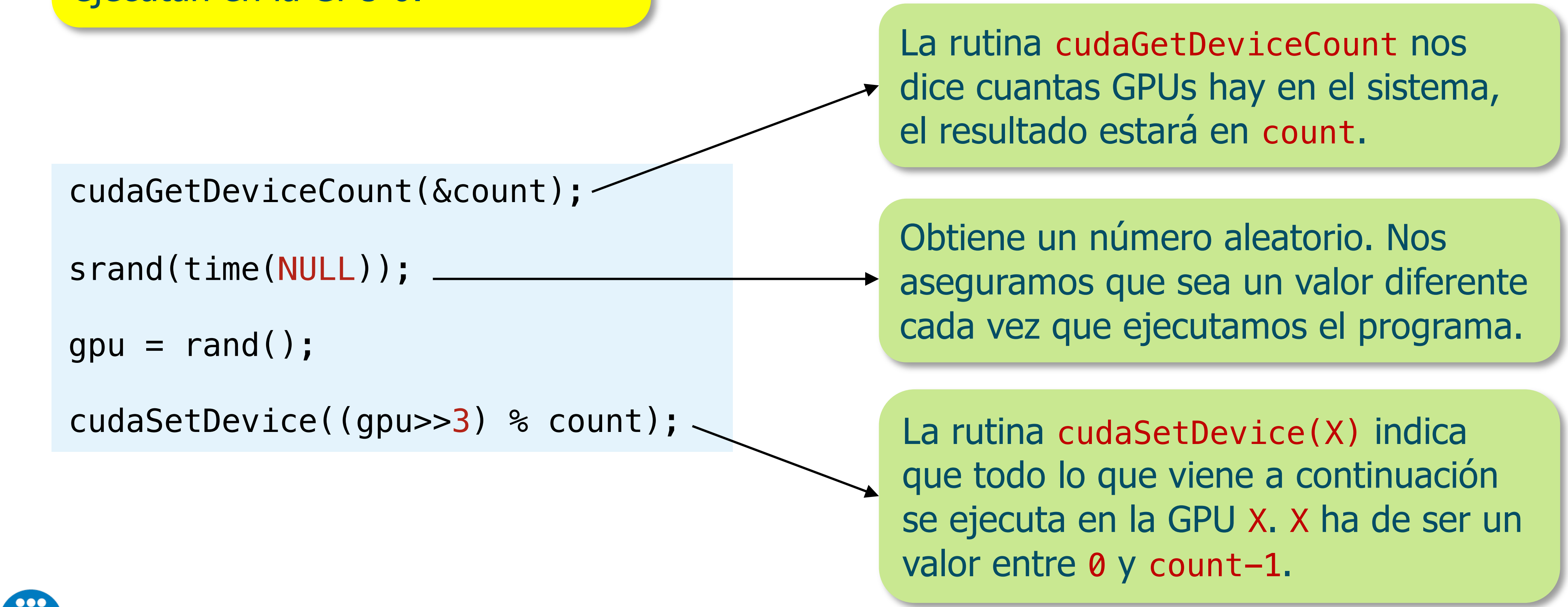
```
CUDA Kernel & Memory Operations Trace:


Start(sec)  Time(ns)  CorrId  Grid Size  Block Size  Reg/T  StcSMem  DymSMem    Bytes     (GB/s)  SrcMemKd  DstMemKd     Device      Str        Name
----------  --------  ------  ---------  ----------  -----  -------  -------  ---------  ------  --------  --------  ------------  ---  ------------------
  0.360799  262,243     426                                                   4,186,116  15.962  Pinned    Device    RTX 3080 (3)   7   [CUDA memcpy HtoD]
  0.361076  259,041     427                                                   4,186,116  16.160  Pinned    Device    RTX 3080 (3)   7   [CUDA memcpy HtoD]
  0.361421  643,428     431   (1 4 1)    (1 256 1)   24       0        0                                             RTX 3080 (3)   7   puzzle2DPARfil()
  0.362078  325,730     435                                                   4,186,116  12.851  Device    Pinned    RTX 3080 (3)   7   [CUDA memcpy DtoH]
  0.364149  624,741     439   (4 1 1)    (256 1 1)   24       0        0                                             RTX 3080 (3)   7   puzzle2DPARcol()
  0.364789  325,922     443                                                   4,186,116  12.843  Device    Pinned    RTX 3080 (3)   7   [CUDA memcpy DtoH]
  0.366808  39,424      447   (64 64 1)  (16 16 1)   18       0        0                                             RTX 3080 (3)   7   puzzle2DPAR1x1()
  0.366861  325,923     451                                                   4,186,116  12.843  Device    Pinned    RTX 3080 (3)   7   [CUDA memcpy DtoH]
```

Usando todas las GPUs

En Boada-10 hay 4 GPUs. Si no hacemos nada, TODOS los jobs se ejecutan en la GPU 0.

En el `job.sh`

```
...
#SBATCH --gres=gpu:4
...
```



```
cudaGetDeviceCount(&count);

srand(time(NULL));

gpu = rand();

cudaSetDevice((gpu>>3) % count);
```

La rutina `cudaGetDeviceCount` nos dice cuantas GPUs hay en el sistema, el resultado estará en `count`.

Obtiene un número aleatorio. Nos aseguramos que sea un valor diferente cada vez que ejecutamos el programa.

La rutina `cudaSetDevice(X)` indica que todo lo que viene a continuación se ejecuta en la GPU `X`. `X` ha de ser un valor entre `0` y `count-1`.

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

CUDA – Sesión 02 - Puzzles

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya