UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

CUDA – Producto de Matrices

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

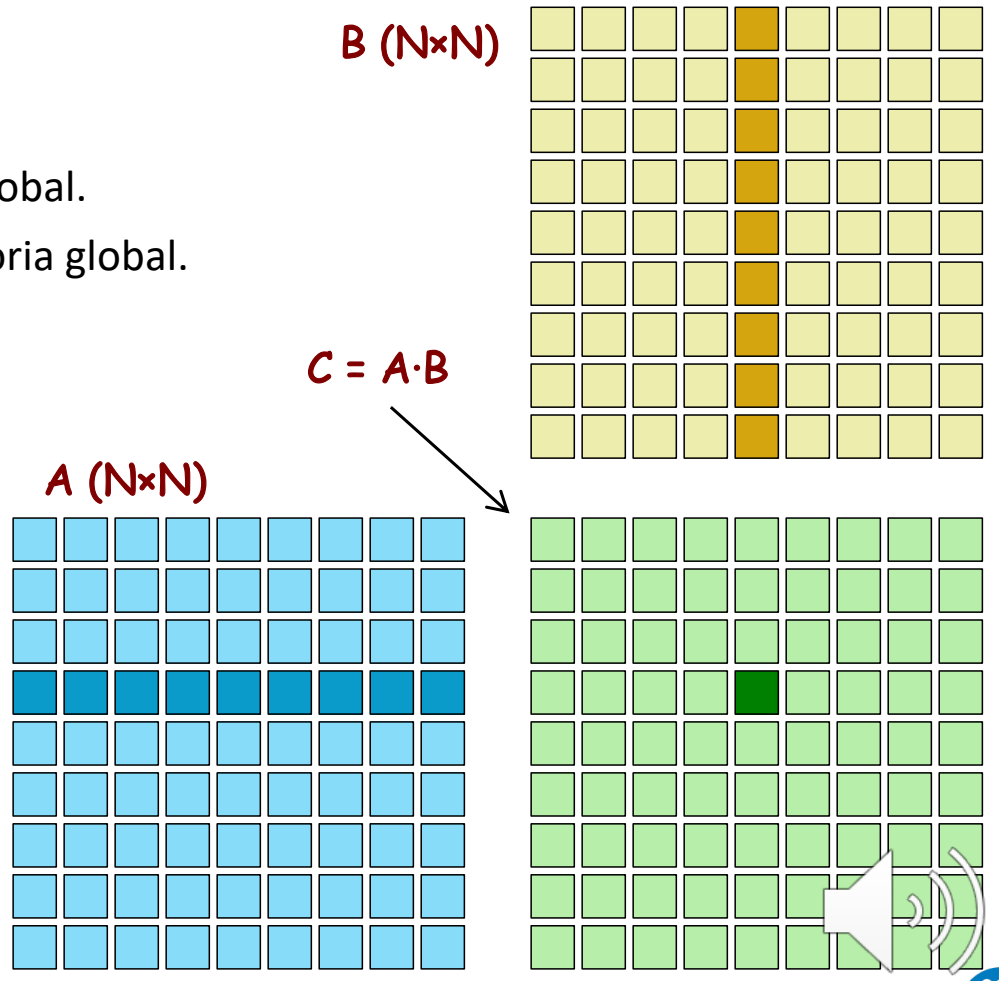
Universitat Politècnica de Catalunya

C=A·B : elemento a elemento (kernel00)

C = A · B (tamaño N × N)

- Cada thread calcula un elemento de la matriz C.
  - Cada fila de A es leída N veces desde la memoria global.
  - Cada columna de B es leída N veces desde la memoria global.
- Se desperdicia mucho ancho de banda.
- Mal equilibrio entre cálculo y ancho de banda

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++) {
    tmp = 0.0;
    for (k=0; k<N; k++)
      tmp += A[i][k] * B[k][j];
    C[i][j] = tmp;
  }
```



C=A·B : elemento a elemento (kernel00)

```
__global__ void MMkernel(float *dA, float *dB, float *dC, int N) {

  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  float tmp = 0.0;
  for (int k=0; k<N; k++)
    tmp += dA[row*N+k] * dB[k*N+col];
  dC[row*N+col] = tmp;
}
```

Quién soy yo

Qué hace cada thread

```
dim3 dimGrid(N/size, N/size, 1);
dim3 dimBlock(size, size, 1);

// Invocación
MMkernel<<<dimGrid, dimBlock>>>(dA, dB, dC, N);
```

¡Atención con estos valores!

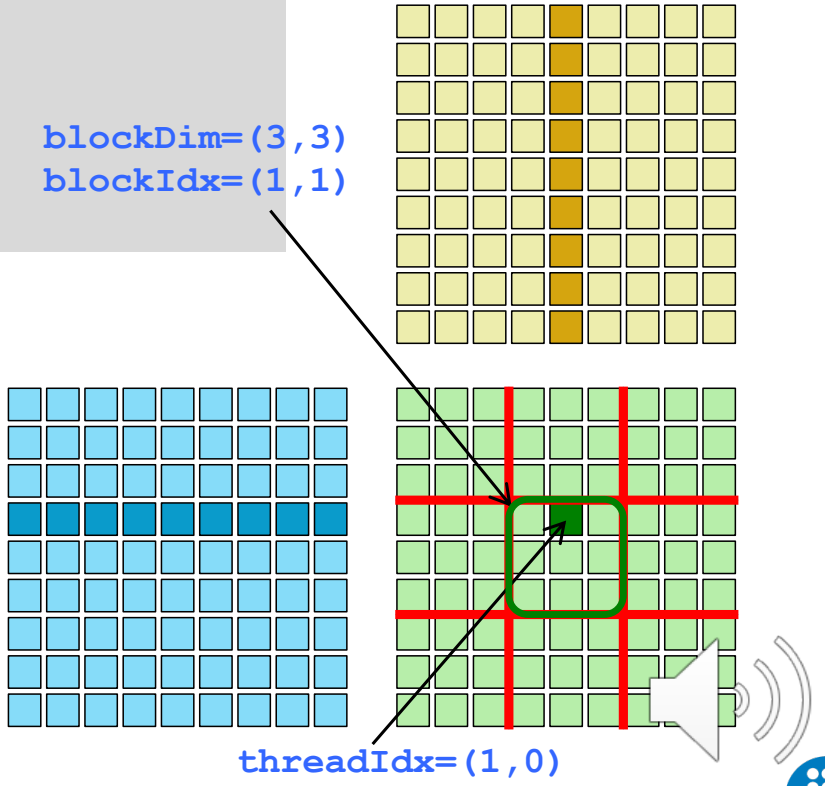
C=A·B : elemento a elemento (kernel00)

```
__global__ void MMkernel(float *dA, float *dB, float *dC, int N) {

  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  float tmp = 0.0;
  for (int k=0; k<N; k++)
    tmp += dA[row*N+k] * dB[k*N+col];
  dC[row*N+col] = tmp;
}
```

```
row = blockIdx.y*blockDim.y+threadIdx.y = 1*3+0 = 3
col = blockIdx.x*blockDim.x+threadIdx.x = 1*3+1 = 4
```

```
@M[r][c] = @M+(r*Ncol+c)*tam
```

```
A[row][k]:   dA[row*N+k]
B[k][col]:   dB[k*N+col]
C[row][col]: dC[row*N+col]
```



C=A·B : elemento a elemento (kernel00)

```
KERNEL 00
Dimensiones: 640x640
nThreads: 32x32 (1024)
nBlocks: 20x20 (400)
NO usa Pinned Memory
Tiempo Global: 8.699456 milseg
Tiempo Kernel: 4.098496 milseg
Rendimiento Global: 60.27 GFLOPS
Rendimiento Kernel: 127.92 GFLOPS
TEST PASS
```

```
KERNEL 00
Dimensiones: 640x640
nThreads: 32x32 (1024)
nBlocks: 20x20 (400)
Usando Pinned Memory
Tiempo Global: 6.147008 milseg
Tiempo Kernel: 4.131744 milseg
Rendimiento Global: 85.29 GFLOPS
Rendimiento Kernel: 126.89 GFLOPS
TEST PASS
```

```
==172540== NVPROF is profiling process 172540, command: ./kernel00.exe 640 Y
…
Time(%)      Time     Calls       Avg       Min       Max  Name
 84.88%  4.0574ms         1  4.0574ms  4.0574ms  4.0574ms  Kernel00(…)
 11.78%  563.05us         2  281.52us  279.27us  283.78us  [CUDA memcpy HtoD]
  3.35%  159.91us         1  159.91us  159.91us  159.91us  [CUDA memcpy DtoH]

 89.11%  4.1486ms         1  4.1486ms  4.1486ms  4.1486ms  Kernel00(…)
  7.21%  335.49us         2  167.75us  167.55us  167.94us  [CUDA memcpy HtoD]
  3.69%  171.75us         1  171.75us  171.75us  171.75us  [CUDA memcpy DtoH]
```

C=A·B : elemento a elemento (kernel00)

```
./kernel00.exe 641 Y


KERNEL 00
Dimensiones: 641x641
nThreads: 32x32 (1024)
nBlocks: 20x20 (400)
Usando Pinned Memory
Tiempo Global: 7.088832 milseg
Tiempo Kernel: 5.120288 milseg
Rendimiento Global: 74.31 GFLOPS
Rendimiento Kernel: 102.87 GFLOPS
0:640: 167.605087 - 0.000000 = 167.605087
TEST FAIL
```

nBlocks es incorrecto.

Debería ser 21x21.

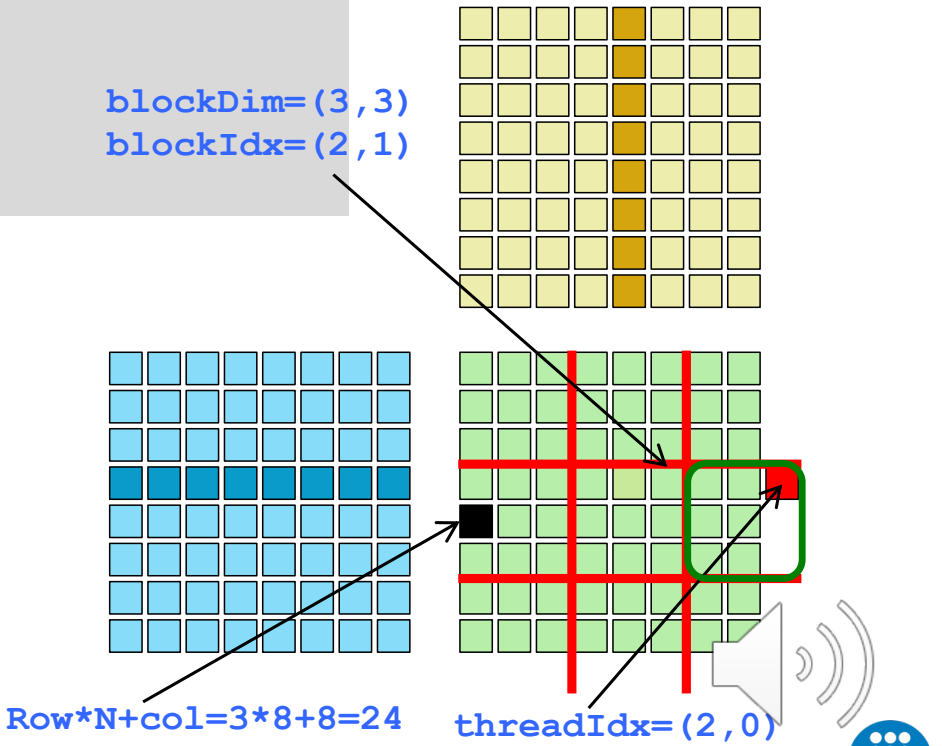
C=A·B : elemento a elemento (kernel00)

```
__global__ void MMkernel(float *dA, float *dB, float *dC, int N) {

  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  float tmp = 0.0;
  for (int k=0; k<N; k++)
    tmp += dA[row*N+k] * dB[k*N+col];
  dC[row*N+col] = tmp;
}
```

```
row = blockIdx.y*blockDim.y+threadIdx.y = 1*3+0 = 3
col = blockIdx.x*blockDim.x+threadIdx.x = 2*3+2 = 8
```

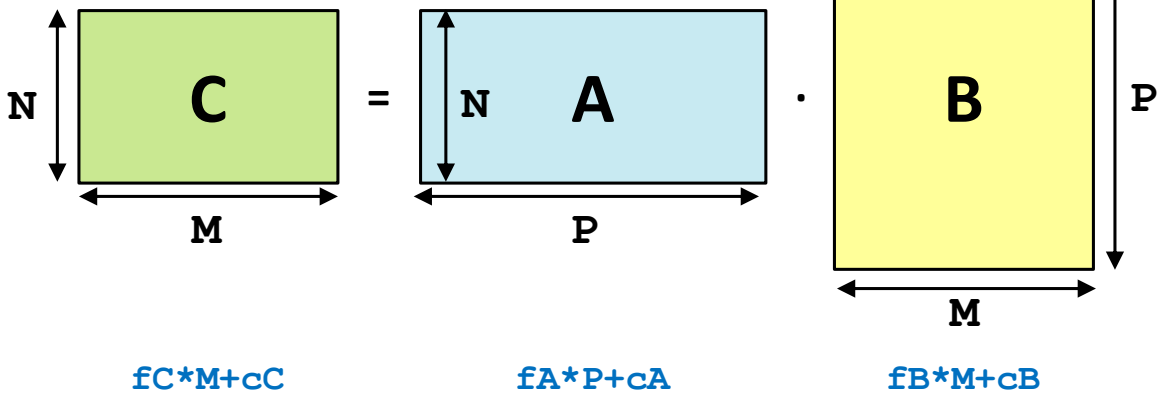
```
@M[r][c] = @M+(r*Ncol+c)*tam
```

```
A[row][k]:   dA[row*N+k]
B[k][col]:   dB[k*N+col]
C[row][col]: dC[row*N+col]
```

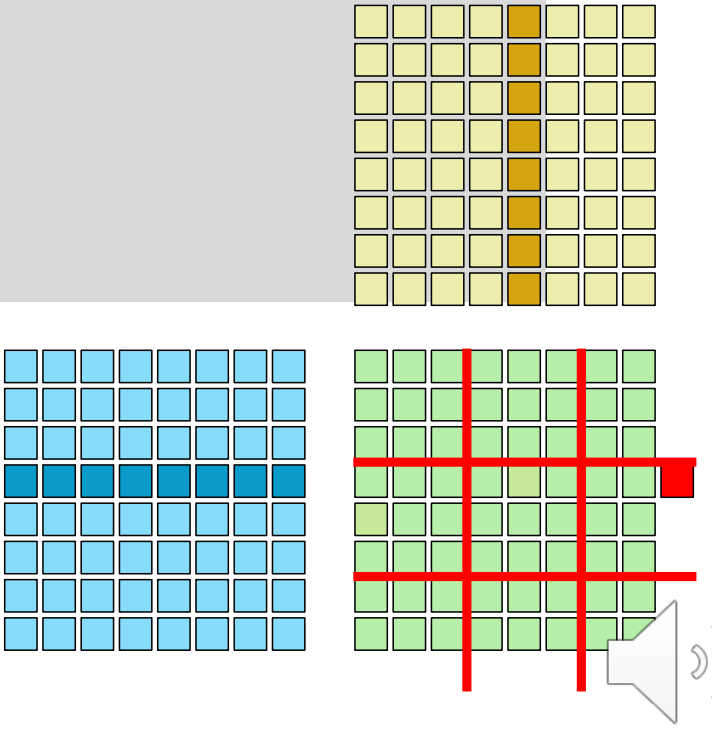


C=A·B : elemento a elemento (kernel01)

```
__global__ void MMkernel(float *dA, float *dB, float *dC, int N, int M, int P) {

  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;

  if (row < N && col < M) {
    float tmp = 0.0;
    for (int k=0; k<P; k++)
      tmp += dA[row*P+k] * dB[k*M+col];
    dC[row*M+col] = tmp;
 }
}
```



fC*M+cC fA*P+cA fB*M+cB



C=A·B : elemento a elemento (kernel01)

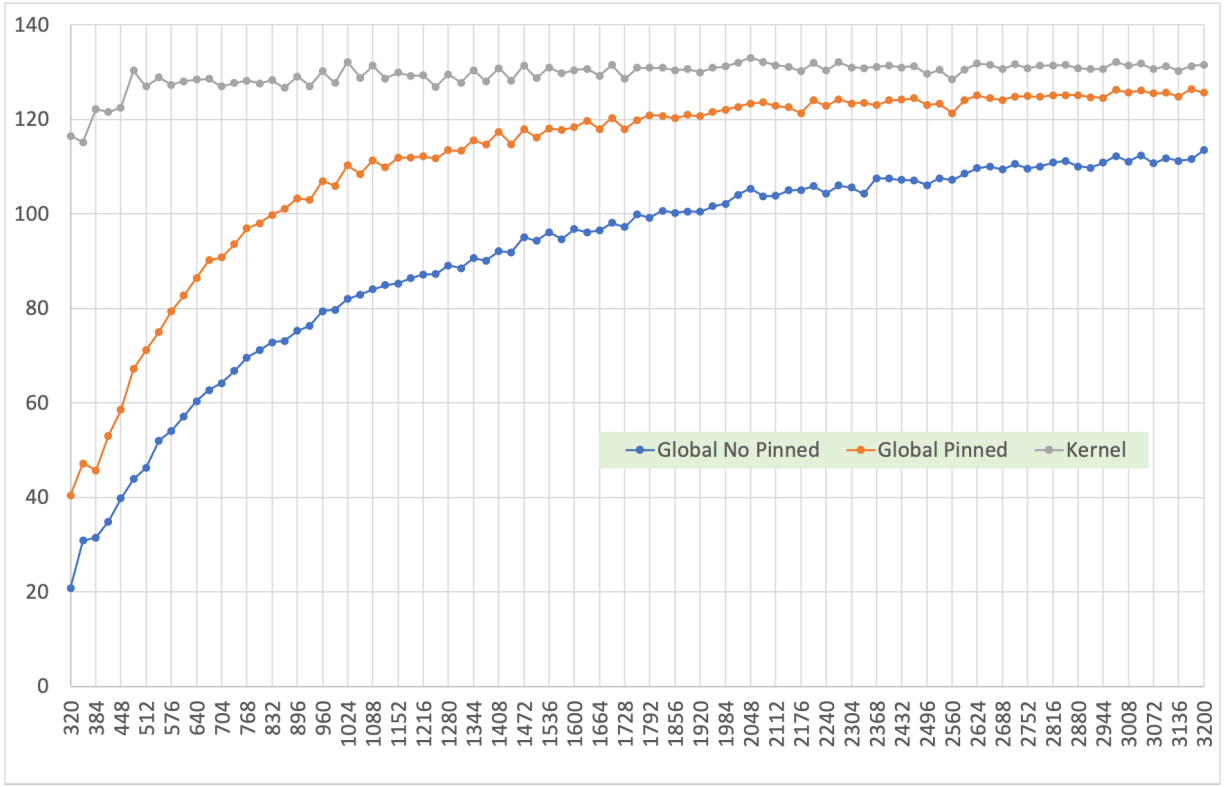
```
KERNEL 01
Dimensiones: 640x640
nThreads: 32x32 (1024)
nBlocks: 20x20 (400)
Usando Pinned Memory
Tiempo Global: 6.101248 milseg
Tiempo Kernel: 4.091168 milseg
Rendimiento Global: 85.93 GFLOPS
Rendimiento Kernel: 128.15 GFLOPS
TEST PASS
```

```
KERNEL 01
Dimensiones: 641x641
nThreads: 32x32 (1024)
nBlocks: 21x21 (441)
Usando Pinned Memory
Tiempo Global: 7.129984 milseg
Tiempo Kernel: 5.168544 milseg
Rendimiento Global: 73.88 GFLOPS
Rendimiento Kernel: 101.91 GFLOPS
TEST PASS
```

La influencia de los restos no es despreciable.

C=A·B : elemento a elemento (kernel01)

GFLOPS



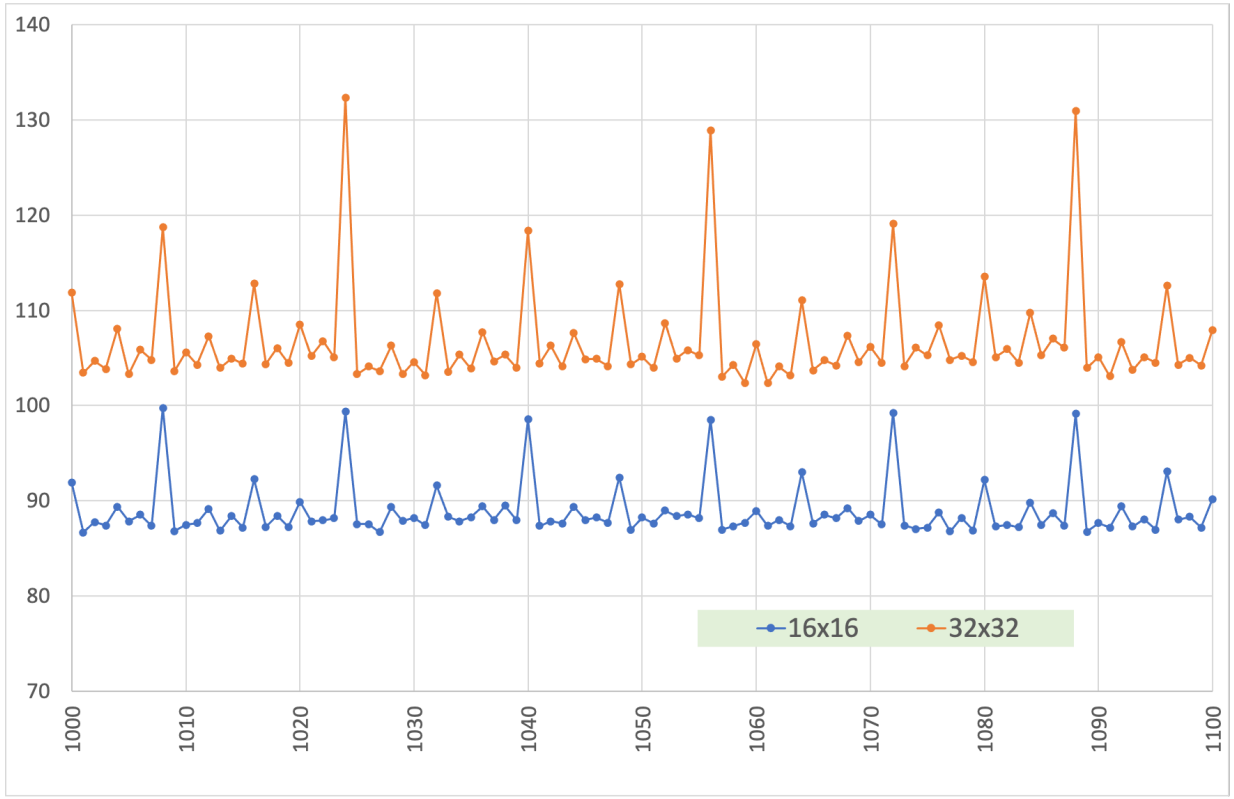
N

El coste de la transferencia se atenúa al aumentar el tamaño del problema.

Matrices cuadradas, N múltiplo de 32.

Producto de Matrices: kernel01

GFLOPS



N

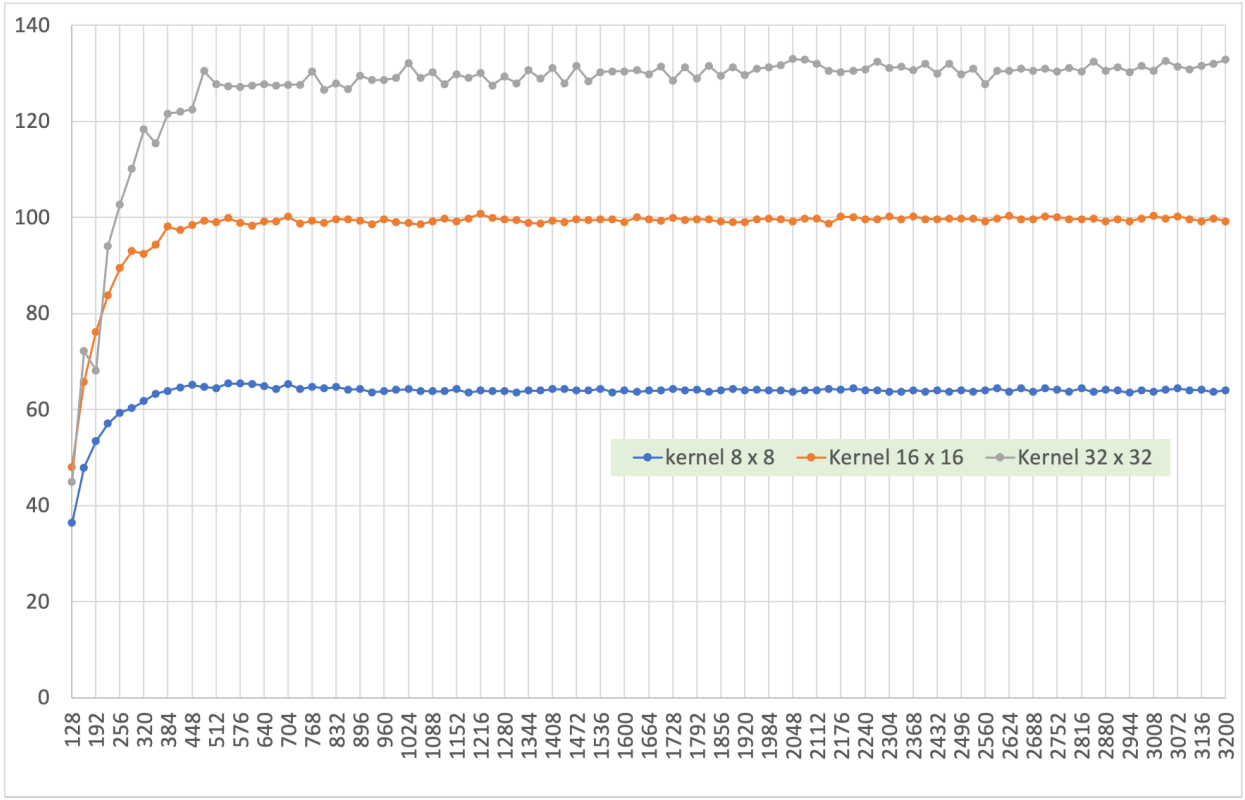
N de 1000 a 1100, N++
matrices cuadradas

La influencia de los restos no es despreciable.

Los picos de rendimiento están en N, múltiplo del tamaño de bloque (32 y 16).

C=A·B : elemento a elemento (kernel01)

GFLOPS



N

32×32

16×16

8 × 8

GTX 1080Ti
N=3.200
358 GFLOPS

¡El tamaño de bloque IMPORTA!

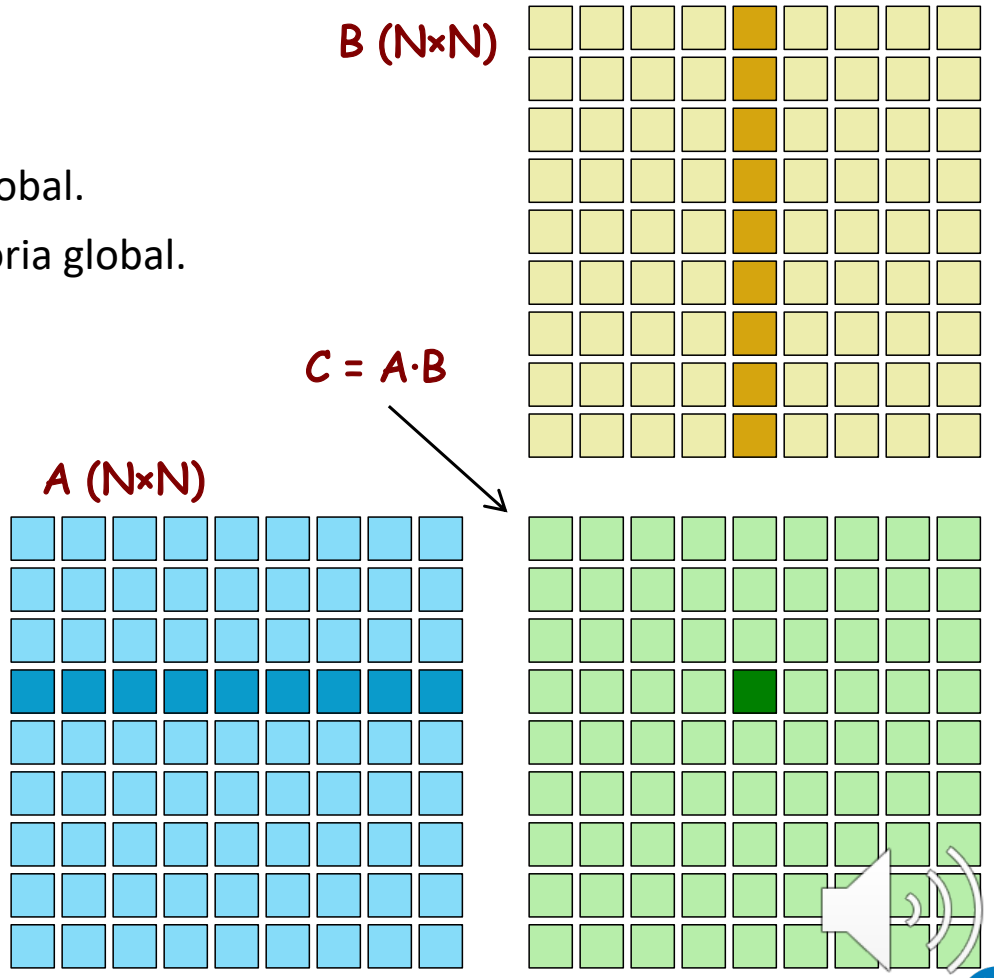
N: de 128 a 3200 (N+=32), matrices cuadradas, N múltiplo de 32.

C=A·B : elemento a elemento (kernel00 y 01)

C = A · B (tamaño N × N)

- Cada thread calcula un elemento de la matriz C.
  - Cada fila de A es leída N veces desde la memoria global.
  - Cada columna de B es leída N veces desde la memoria global.
- Se desperdicia mucho ancho de banda.
- Mal equilibrio entre cálculo y ancho de banda

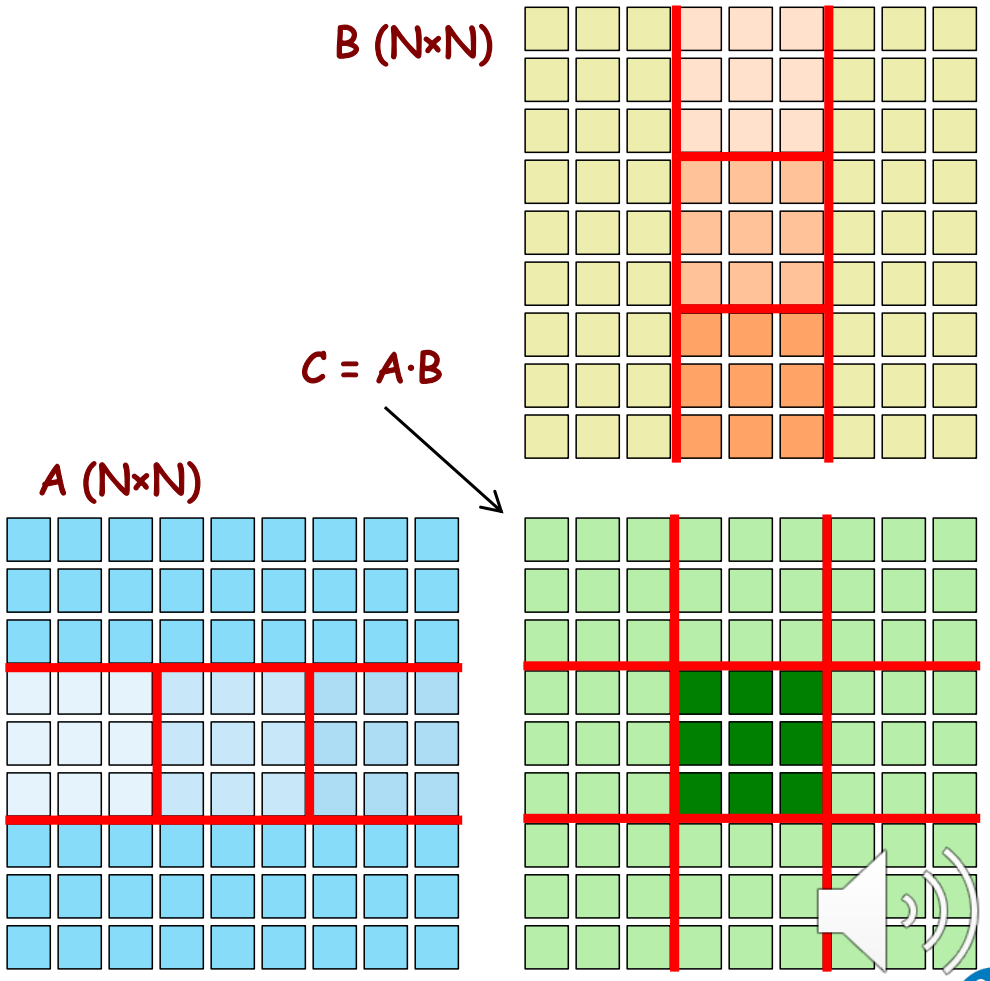
- En cada thread:
  - Operaciones: 2N
  - Accesos a **Memoria Global**: 2N + 1
  - **¡Patrón de acceso!**



C=A·B : a bloques (kernel10)

C = A · B (tamaño N × N)

- Cada block thread calcula un bloque de datos de SZ×SZ elementos de la matriz C.
- El algoritmo itera N/SZ veces. En cada iteración
  - Leemos un bloque SZ×SZ de A de la memoria global. La matriz A se lee N/SZ veces.
  - Leemos un bloque SZ×SZ de B de la memoria global. La matriz B se lee N/SZ veces.
- El valor de SZ es importante:
  - Las submatrices han de caber en la memoria compartida.
  - Necesitamos SZ×SZ threads por bloque.
- Utilizamos menos ancho de banda. El equilibrio entre cálculo y ancho de banda mejorará.



C=A·B : a bloques (kernel10)

```
__global__ void MMkernel2(float *dA, float *dB, float *dC, int N, int M, int P) {
  __shared__ float sA[SZ][SZ]; __shared__ float sB[SZ][SZ];
  int thx = threadIdx.x; int thy = threadIdx.y;
  int row = blockIdx.y * SZ + thy;
  int col = blockIdx.x * SZ + thx;
  float tmp = 0.0;
  for (s=0; s<(P/SZ); s++) {
    sA[thy][thx] = dA[row*P + s*SZ + thx];
    sB[thy][thx] = dB[(s*SZ + thy)*M + col];
    __syncthreads();
    for (int k=0; k<SZ; k++)
      tmp += sA[thy][k] * sB[k][thx];
    __syncthreads();
  }
  dC[row*M+col] = tmp;
}
```

Submatrices en la Memoria Compartida

Todos los threads del bloque cooperan para cargar las submatrices A y B en la memoria compartida

Antes de empezar a calcular hay que asegurarse que todos los threads hayan acabado de cargar las submatrices A y B.

Antes de continuar hay que asegurarse que todos los threads hayan acabado los cálculos que le corresponden.

C=A·B : a bloques (kernel10)

```
KERNEL 10
Dimensiones: 640x640
nThreads: 32x32 (1024)
nBlocks: 20x20 (400)
NO usa Pinned Memory
Tiempo Global: 6.185728 milseg
Tiempo Kernel: 1.458368 milseg
Rendimiento Global: 84.76 GFLOPS
Rendimiento Kernel: 359.50 GFLOPS
TEST PASS
```

```
KERNEL 10
Dimensiones: 640x640
nThreads: 32x32 (1024)
nBlocks: 20x20 (400)
Usando Pinned Memory
Tiempo Global: 3.476960 milseg
Tiempo Kernel: 1.492032 milseg
Rendimiento Global: 150.79 GFLOPS
Rendimiento Kernel: 351.39 GFLOPS
TEST PASS
```

```
====172623== NVPROF is profiling process 172623, command: ./kernel10.exe 640 Y
...
Time(%)      Time     Calls       Avg       Min       Max  Name
 65.97%  1.4595ms         1  1.4595ms  1.4595ms  1.4595ms  Kernel10(...)
 26.36%  583.20us         2  291.60us  284.42us  298.79us  [CUDA memcpy HtoD]
  7.67%  169.60us         1  169.60us  169.60us  169.60us  [CUDA memcpy DtoH]

 73.73%  1.4264ms         1  1.4264ms  1.4264ms  1.4264ms  Kernel10(...)
 17.25%  333.64us         2  166.82us  166.31us  167.33us  [CUDA memcpy HtoD]
  9.02%  174.53us         1  174.53us  174.53us  174.53us  [CUDA memcpy DtoH]
```

Nvprof: kernel00 y kernel10

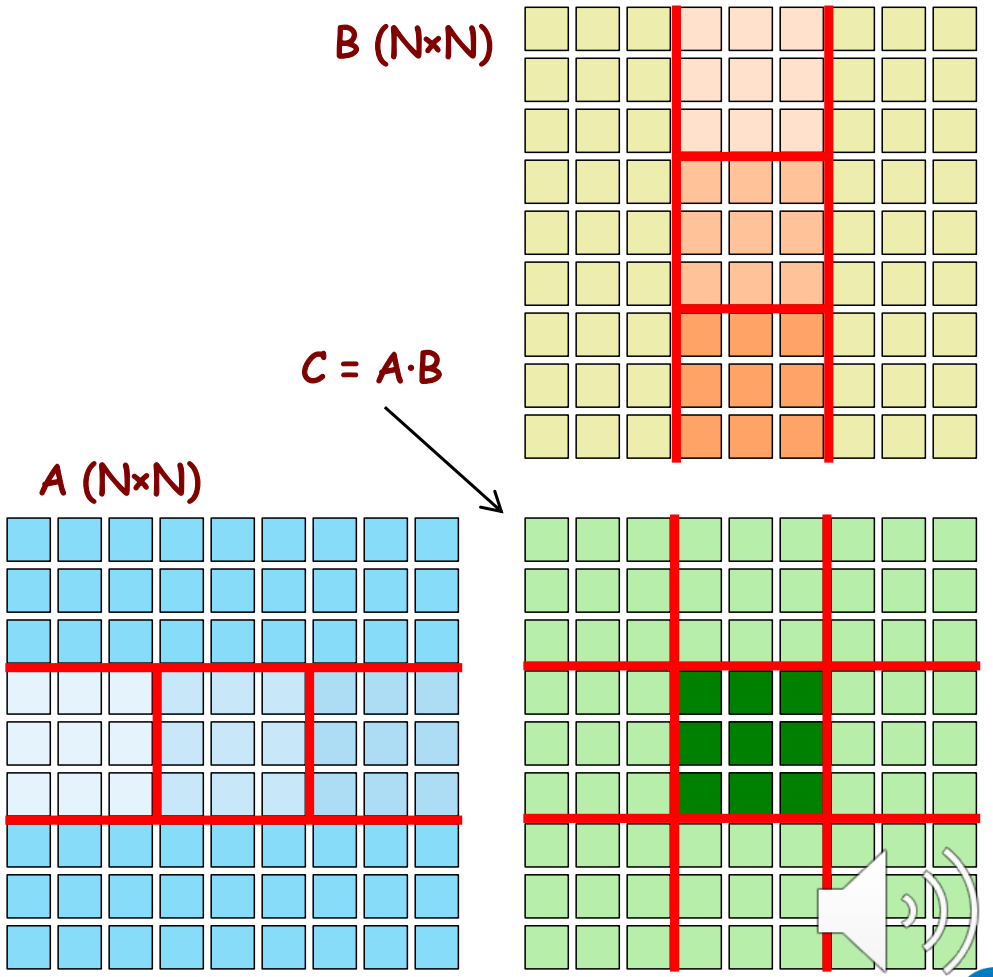
	kernel00	kernel10
Tamaño	2048	2048
Grid Size	(64, 64, 1)	(64, 64, 1)
Block Size	(32, 32, 1)	(32, 32, 1)
Reg	24	23
Shared Memory	-	8 KB
sm_efficiency	99.68%	99,78%
Achieved_occupancy	0,996	0,998
gld_request_throughput	260,77 GB/s	44,32 GB/s
gst_request_throughput	126,44 MB/s	354,55 MB/s
Dram_utilization	Low (1)	Low (1)
Shared_load_throughput	-	2127,3 GB/s

C=A·B : a bloques (kernel10)

C = A · B (tamaño N × N)

- Cada block thread calcula un bloque de datos de SZ×SZ elementos de la matriz C.
  - A es leída N/SZ veces desde la memoria global.
  - B es leída N/SZ veces desde la memoria global.
- Utilizamos mucho menos ancho de banda.
- Mejor equilibrio entre cálculo y ancho de banda

- En cada thread:
  - Operaciones: 2N
  - Accesos a **Memoria Global**: 2N/SZ + 1
  - Accesos a **Memoria Compartida**: 2N
  - **¡Patrones de Acceso!**



C=A·B : a bloques (kernel10)

```
./kernel10.exe 641 641 641 Y


KERNEL 10
Dimensiones: 641x641 <- 641x641 * 641x641
nThreads: 32x32 (1024)
nBlocks: 21x21 (441)
Usando Pinned Memory
Tiempo Global: 0.000000 milseg
Tiempo Kernel: -110526464.000000 milseg
Rendimiento Global:  inf GFLOPS
Rendimiento Kernel: -0.00 GFLOPS
0:0: 159.012283 - 0.000000 = 159.012283
TEST FAIL
```

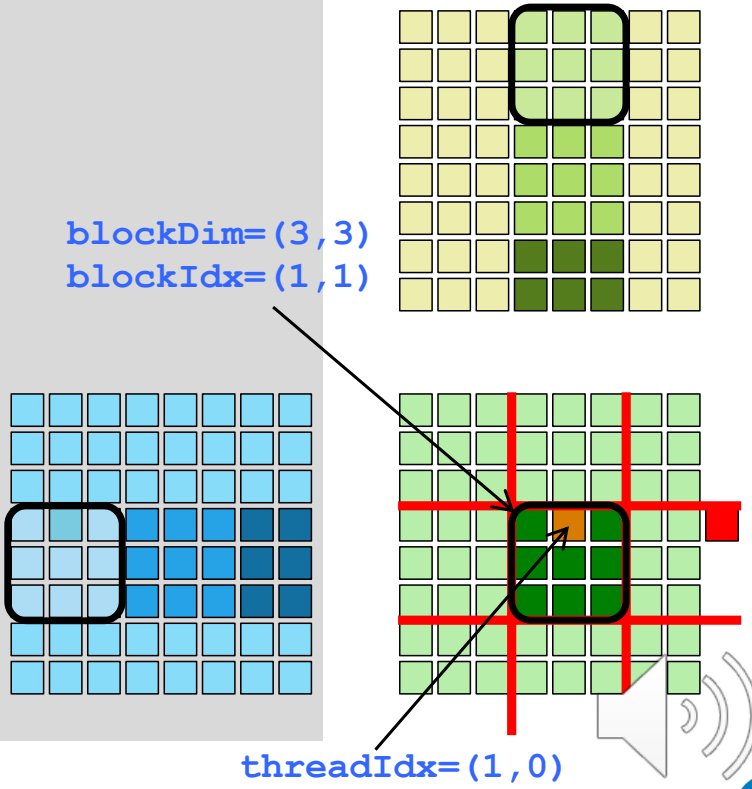
nBlocks es correcto.

Estamos accediendo mal a Memoria.

C=A·B : a bloques (kernel11)

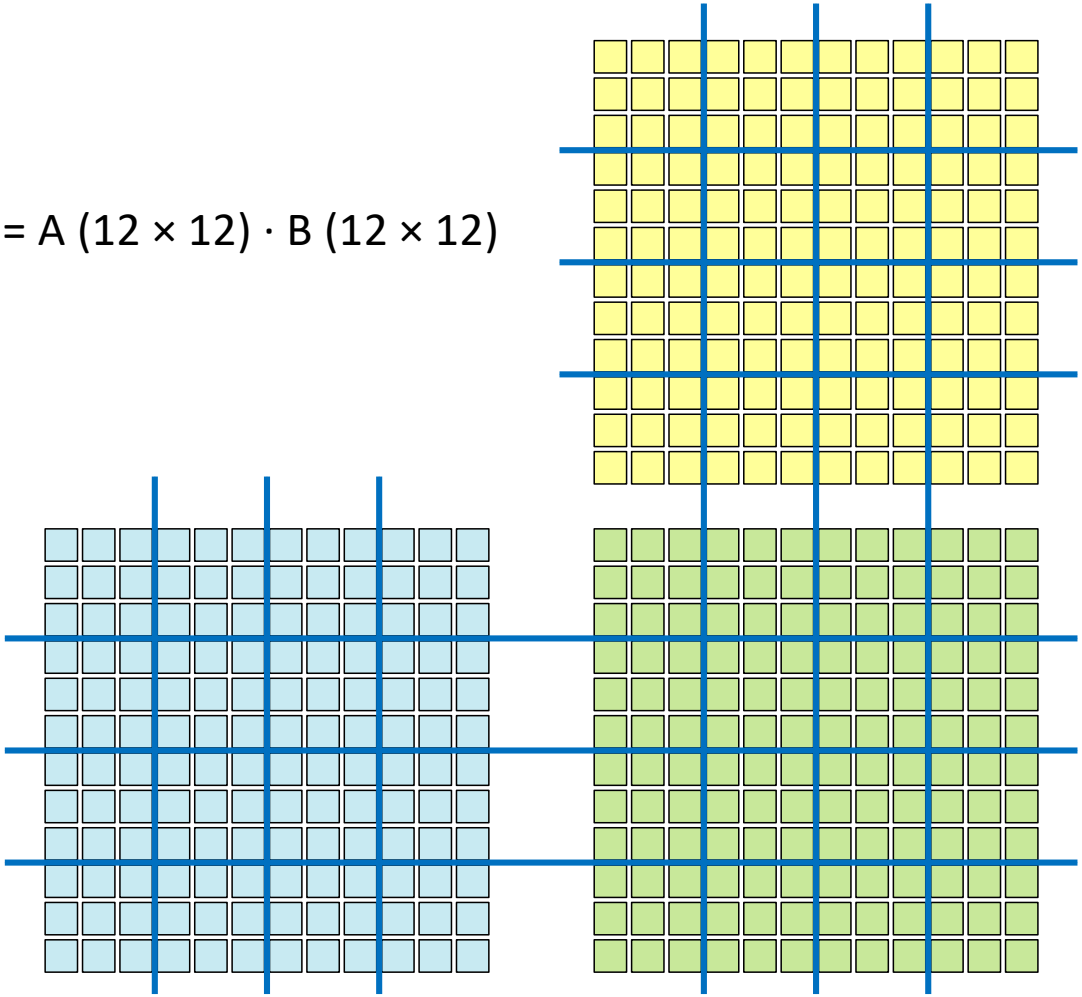
```
__global__ void MMkernel(float *dA, float *dB, float *dC, int N,M,P) {
  __shared__ float sA[SZ][SZ];
  __shared__ float sB[SZ][SZ];
  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
  int row = by * SZ + ty;
  int col = bx * SZ + tx;

  if (row < N && col < M) {
    float tmp = 0.0;
    for (s=0; s<(P/SZ); s++) {
      sA[ty][tx] = dA[row*P + s*SZ + tx];
      sB[ty][tx] = dB[(s*SZ + ty)*M + col];
      __syncthreads();
      for (int k=0; k<SZ; k++)
        tmp += sA[ty][k] * sB[k][tx];
      __syncthreads();
      }
      dC[row*M+col] = tmp;
    }
}
```

NO es SUFICIENTE



C=A·B : a bloques (kernel11)

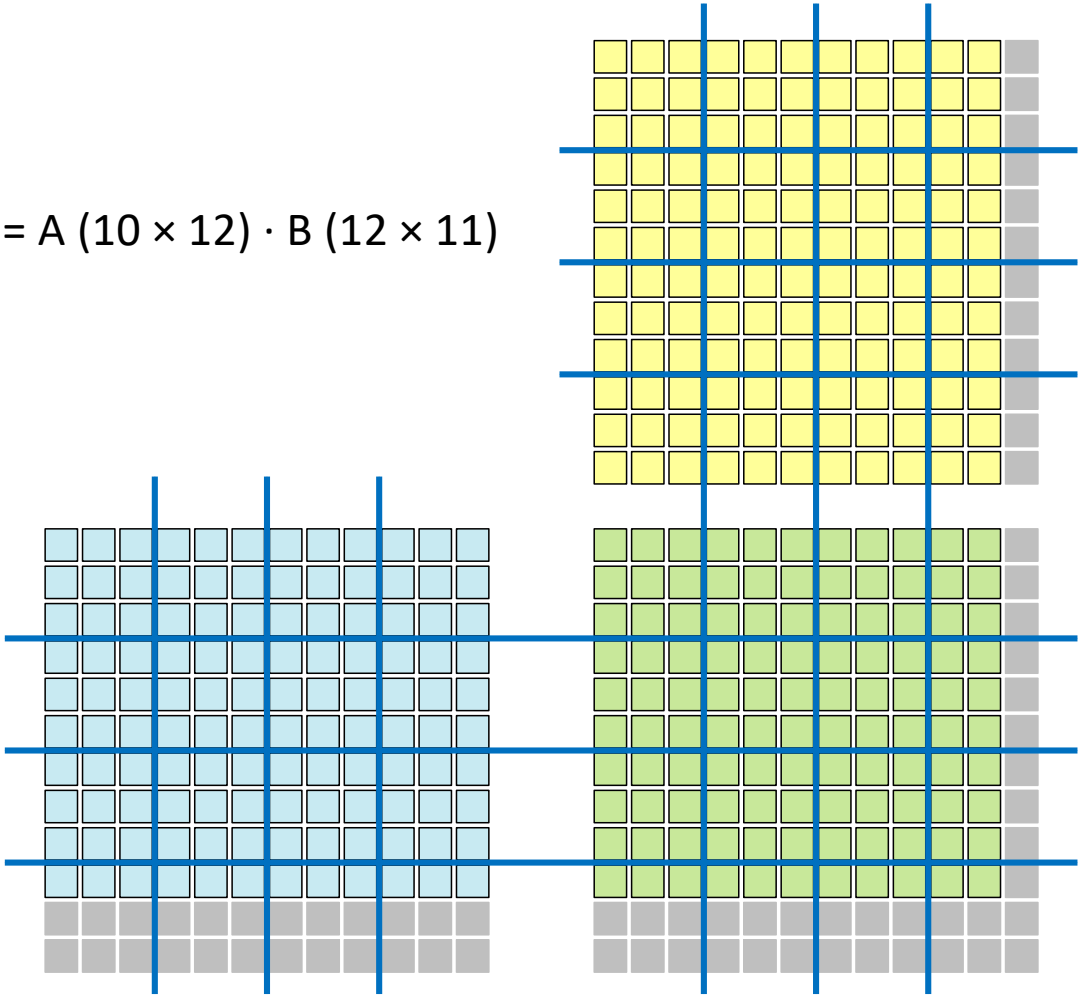
C (12 × 12) = A (12 × 12) · B (12 × 12)



Si el tamaño del problema es múltiplo del número de Threads, todo funciona bien.

C=A·B : a bloques (kernel11)

C (10 × 11) = A (10 × 12) · B (12 × 11)



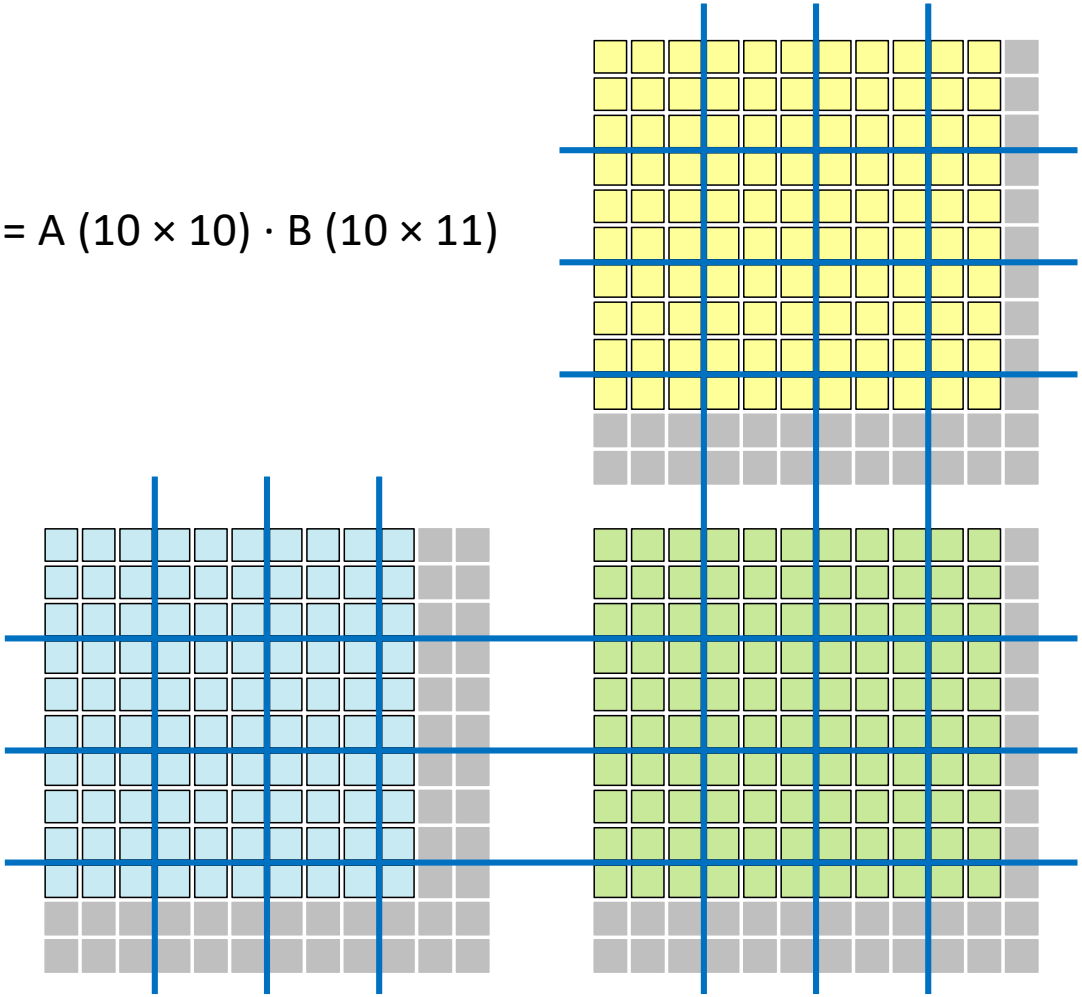
Si P es múltiplo del número de Threads, es simple.

Sólo hay que asegurar que no se escriba el resultado.

```
if (row<N && col<M)
  C[row*M+col] = tmp;
```

C=A·B : a bloques (kernel11)

$C\ (10 \times 11) = A\ (10 \times 10) \cdot B\ (10 \times 11)$



Si P NO es múltiplo del número de Threads, …

NO es suficiente con asegurar que no se escriba el resultado.

Hay que asegurar que NO utilizamos los elementos (■).

C=A·B : a bloques (kernel11)

```
__global__ void MMkernel(float *dA, float *dB, float *dC, int N, int M, int P) {
  __shared__ float sA[SIZE][SIZE];
  __shared__ float sB[SIZE][SIZE];
  int bx = blockIdx.x;  int tx = threadIdx.x;  int col = bx * SIZE + tx;
  int by = blockIdx.y;  int ty = threadIdx.y;  int row = by * SIZE + ty;

  for (m=0; m < P-SZ; m=m+SZ) {
    sA[ty][tx] = A[row*P + m + tx];
    sB[ty][tx] = B[col + (m + ty)*M];
    __syncthreads();
    for (k=0; k<SZ; k++)
      tmp += sA[ty][k] * sB[k][tx];
    __syncthreads();
  }
  sA[ty][tx] = A[row*P + m + tx];
  sB[ty][tx] = B[col + (m + ty)*M];
  __syncthreads();
  for (k=0; m<P; k++, m++)
    tmp += sA[ty][k] * sB[k][tx];
  if (row<N && col<M) C[row*M+col] = tmp;
}
```

C=A·B : a bloques (kernel11)

```
KERNEL 11
Dimensiones: 640x640<-640x640*640x640
nThreads: 32x32 (1024)
nBlocks: 20x20 (400)
Usando Pinned Memory
Tiempo Global: 3.529024 milseg
Tiempo Kernel: 1.469920 milseg
Rendimiento Global: 148.56 GFLOPS
Rendimiento Kernel: 356.68 GFLOPS
TEST PASS
```

32×32 threads

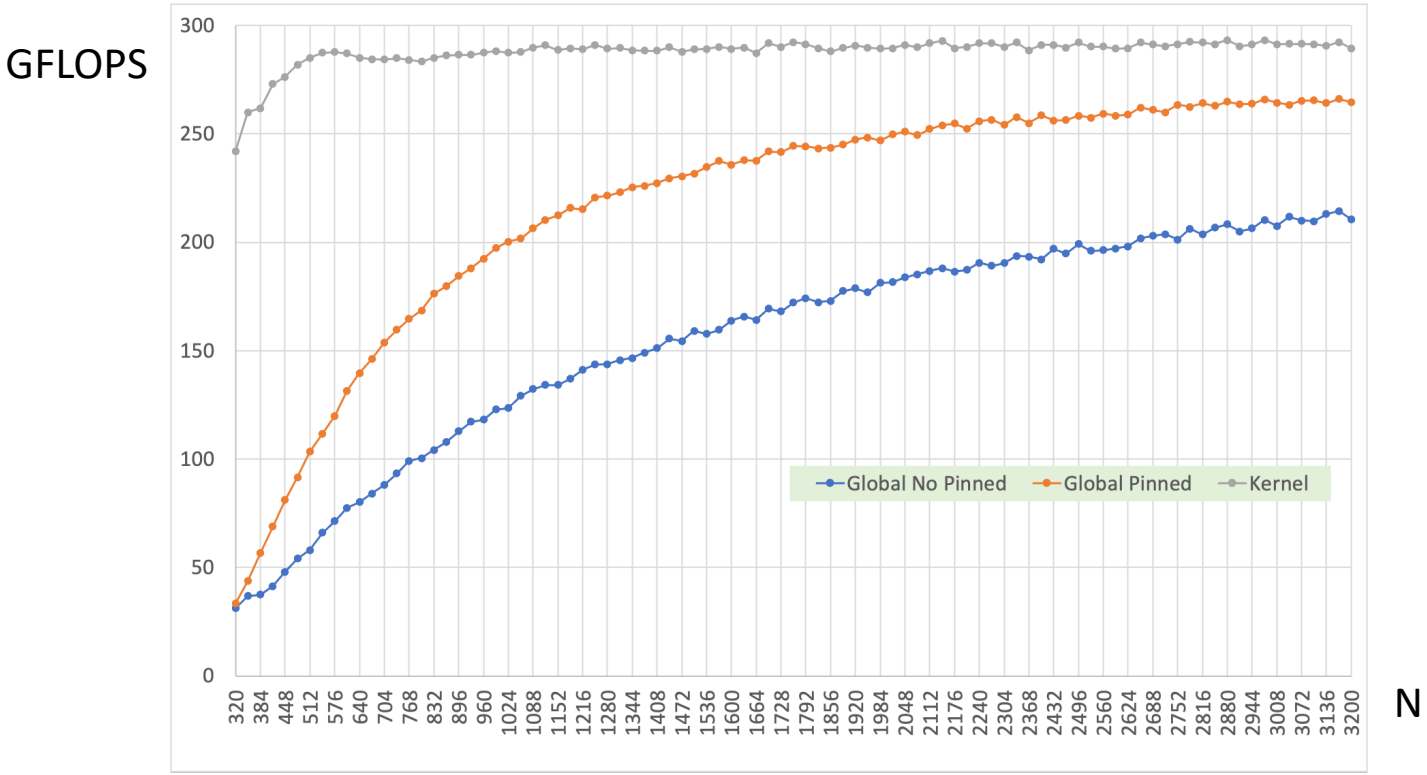
```
KERNEL 11
Dimensiones: 641x641<-641x641*641x641
nThreads: 32x32 (1024)
nBlocks: 21x21 (441)
Usando Pinned Memory
Tiempo Global: 3.581152 milseg
Tiempo Kernel: 1.680640 milseg
Rendimiento Global: 147.09 GFLOPS
Rendimiento Kernel: 313.42 GFLOPS
TEST PASS
```

```
KERNEL 11
Dimensiones: 640x640<-640x640*640x640
nThreads: 16x16 (256)
nBlocks: 40x40 (1600)
Usando Pinned Memory
Tiempo Global: 3.830560 milseg
Tiempo Kernel: 1.838912 milseg
Rendimiento Global: 136.87 GFLOPS
Rendimiento Kernel: 285.11 GFLOPS
TEST PASS
```

16×16 threads

```
KERNEL 11
Dimensiones: 641x641<-641x641*641x641
nThreads: 16x16 (256)
nBlocks: 41x41 (1681)
Usando Pinned Memory
Tiempo Global: 4.125248 milseg
Tiempo Kernel: 2.152992 milseg
Rendimiento Global: 127.69 GFLOPS
Rendimiento Kernel: 244.66 GFLOPS
TEST PASS
```

Producto de Matrices: kernel11

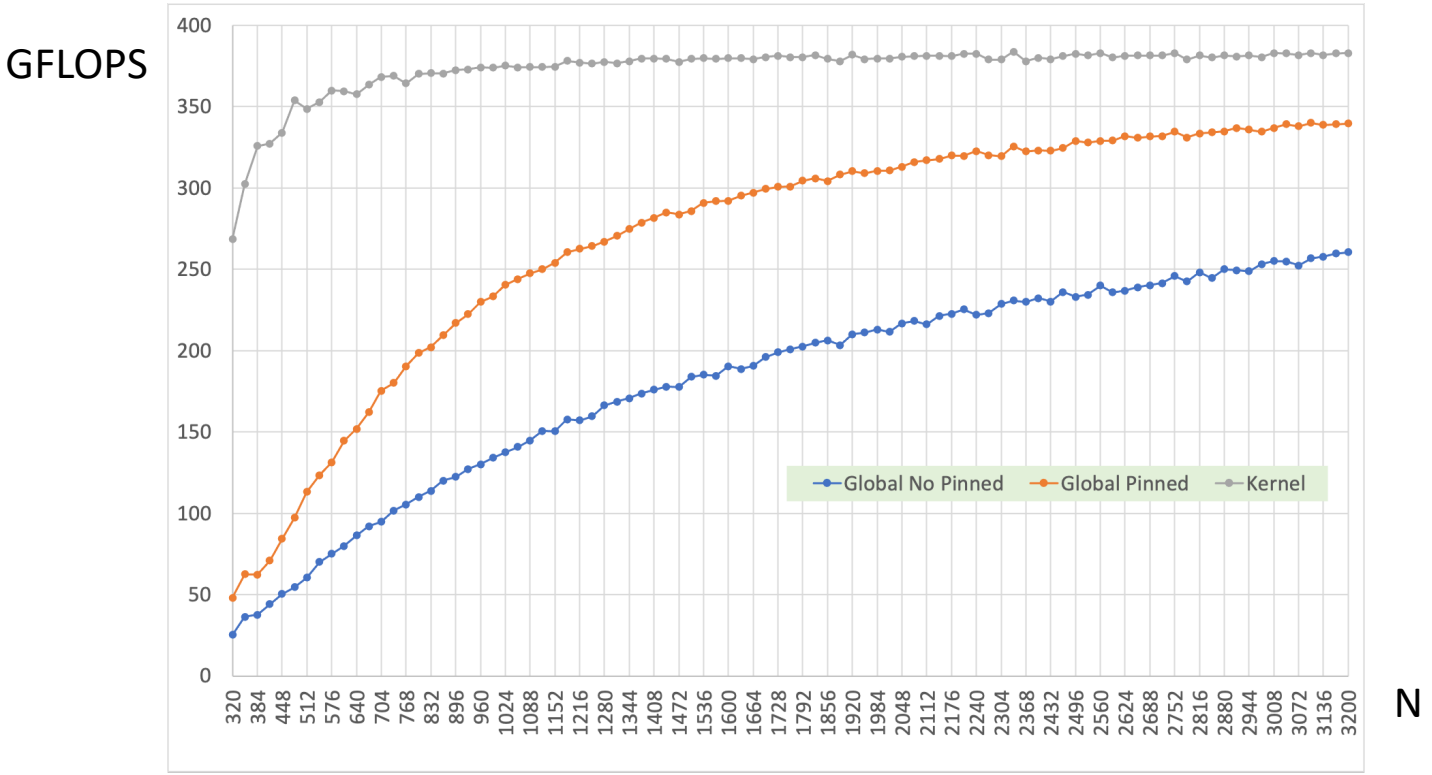


El coste de la transferencia se atenúa al aumentar el tamaño del problema.

N múltiplo de 32. Matrices cuadradas.

16×16 threads

Producto de Matrices: kernel11



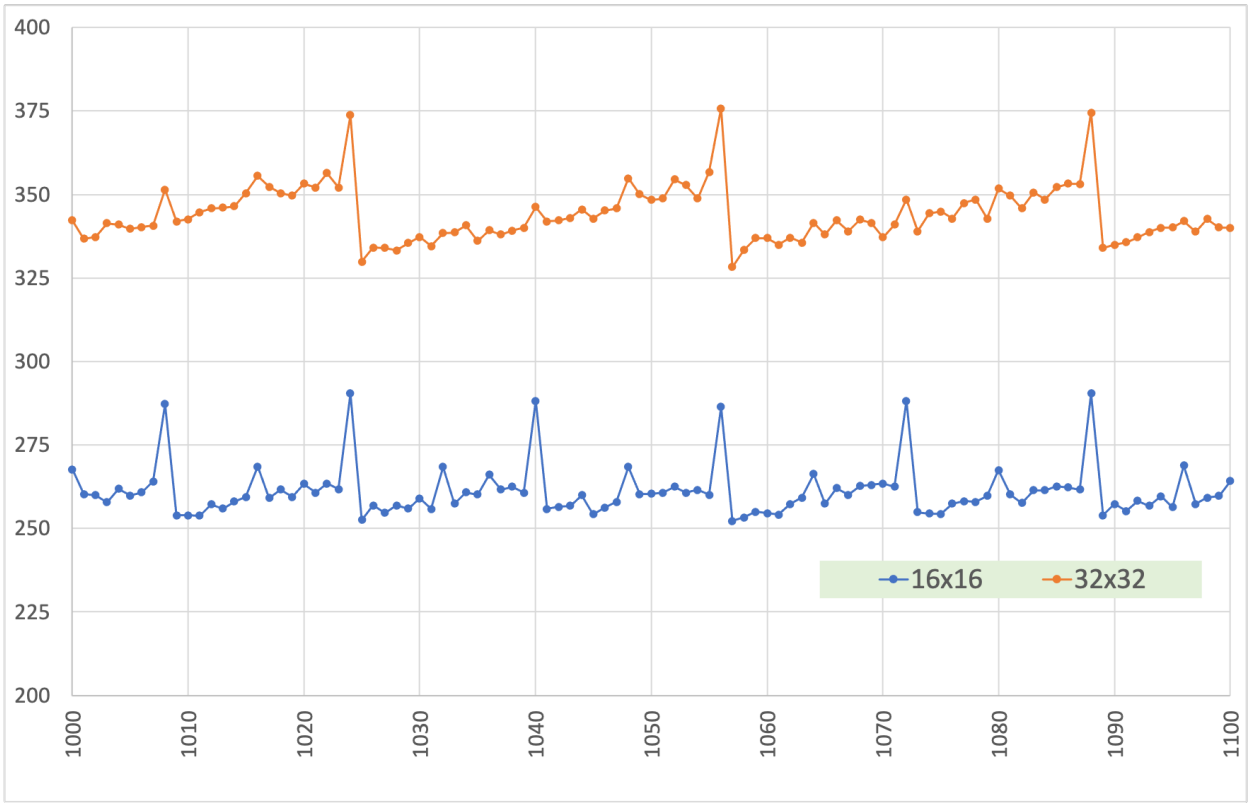
El coste de la transferencia se atenúa al aumentar el tamaño del problema.

N múltiplo de 32. Matrices cuadradas.

32×32 threads

Producto de Matrices: kernel11

GFLOPS



N de 1000 a 1100, N++
matrices cuadradas

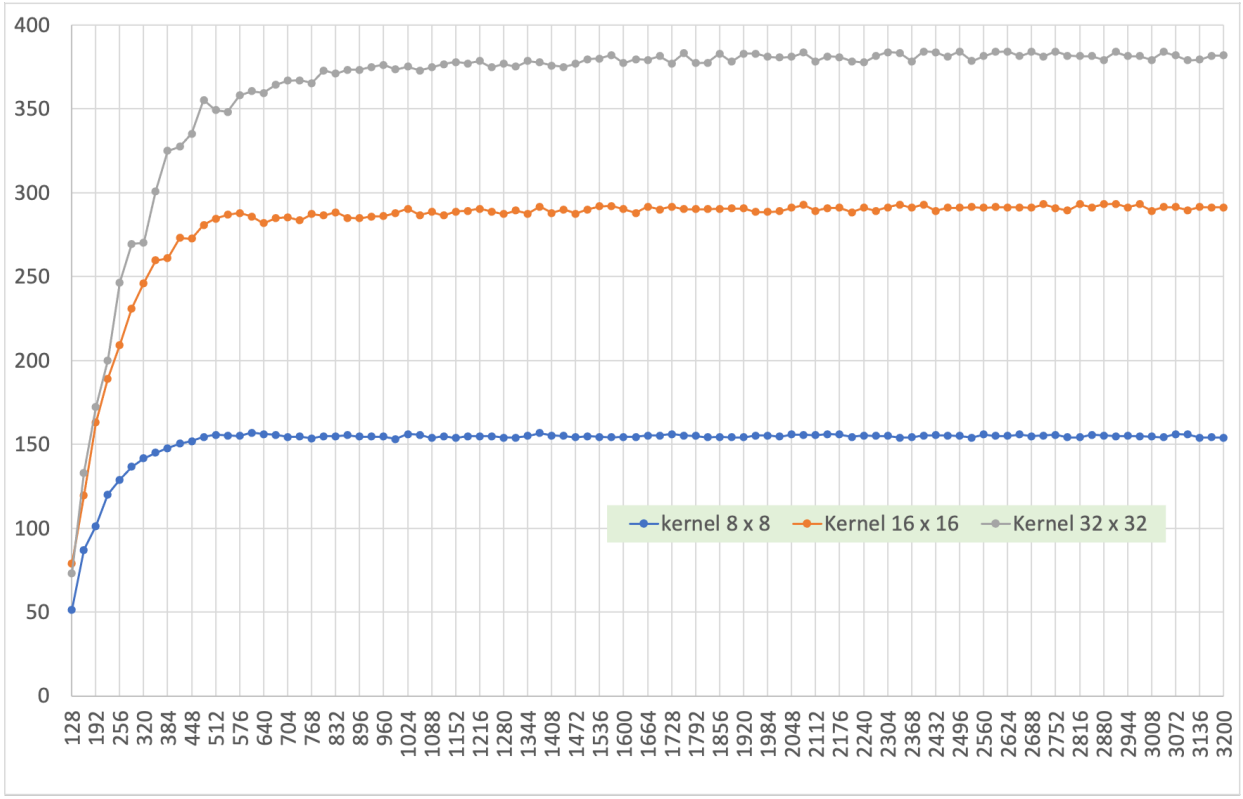
N

La influencia de los restos no es despreciable.

Los picos de rendimiento están en N, múltiplo del tamaño de bloque (32 y 16).

C=A·B : elemento a elemento (kernel11)

GFLOPS



N

32×32

16×16

8 × 8

GTX 1080Ti
N=3.200
1.188 GFLOPS

¡El tamaño de bloque IMPORTA!

N: de 128 a 3200 (N+=32), matrices cuadradas, N múltiplo de 32.

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

CUDA – Producto de Matrices

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya