INTRODUCCIÓN A LA PROGRAMACIÓN DE GPUS CON CUDA

Francisco Igual Peña

Valencia, 12 de noviembre de 2009

ÍNDICE DE CONTENIDOS

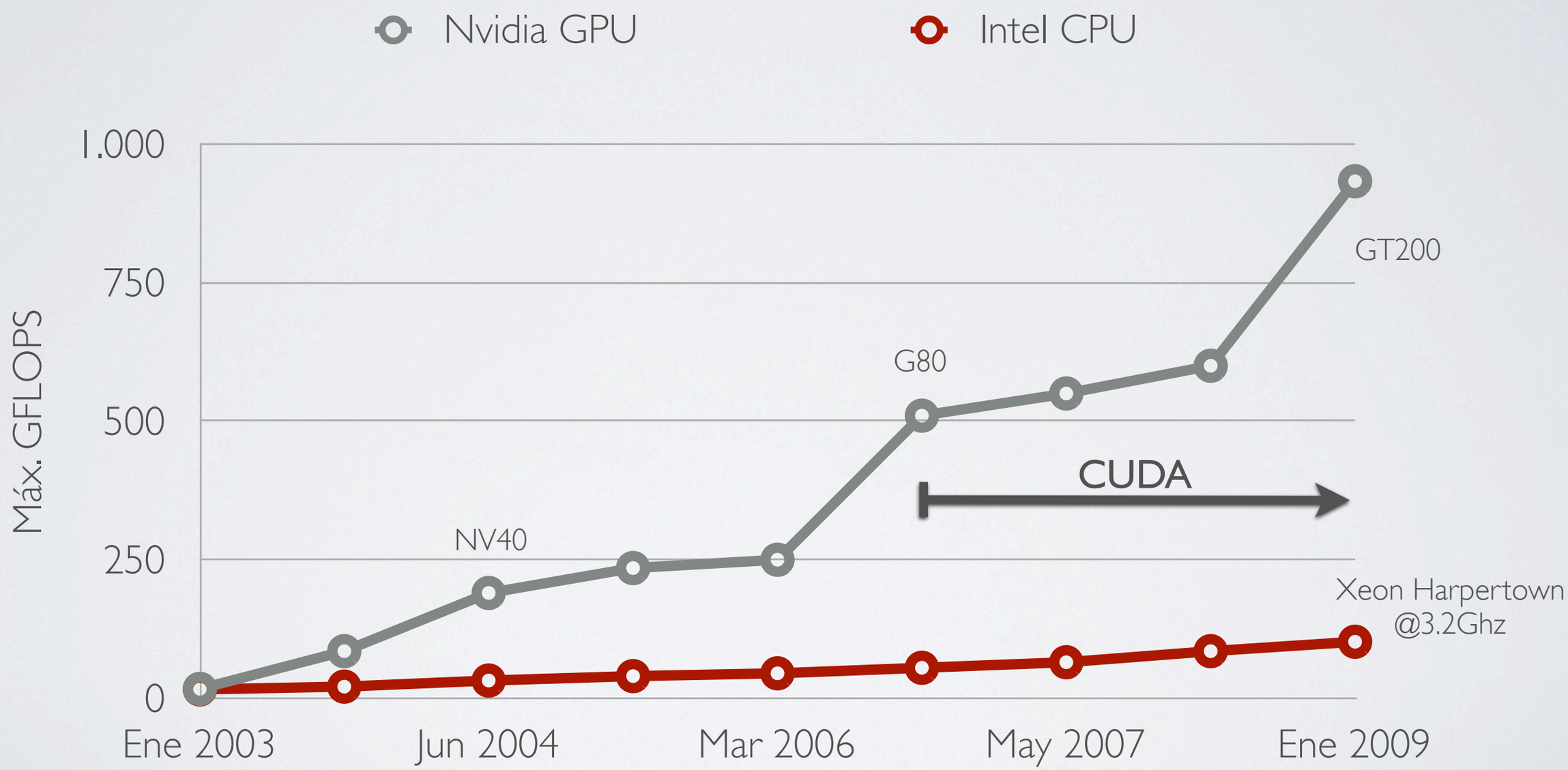
1. Introducción / Motivación
2. La arquitectura CUDA
3. Espacios de memoria
4. Modelo de ejecución CUDA
5. Programación de kernels
6. Conceptos avanzados

ÍNDICE DE CONTENIDOS

1. Introducción / Motivación
2. La arquitectura CUDA
3. Espacios de memoria
4. Modelo de ejecución CUDA
5. Programación de kernels
6. Conceptos avanzados

¿POR QUÉ GPUS?

Rendimiento



¿POR QUÉ GPUS?

Facilidad de programación

Ensamblador

OpenGL/DirectX

Cg → Los inicios de GPGPU

Brook++ / RapidMind

2006

CUDA

CUBLAS
CUFFT
CUDPP
...

→ Popularización de GPGPU

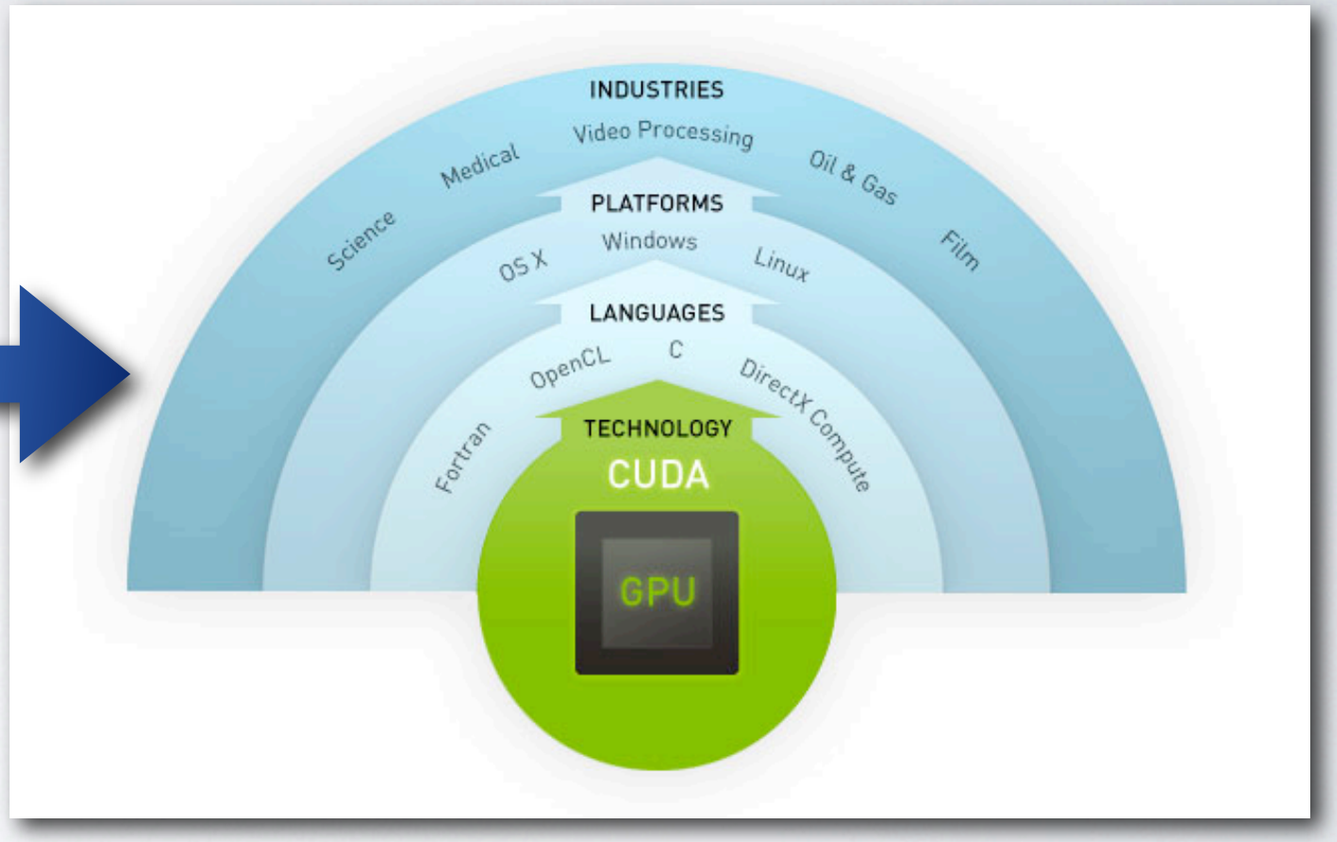
2009

OpenCL → ¿El futuro de GPGPU?

¿POR QUÉ GPUS?

Versatilidad





Las GPUs ya no son únicamente dispositivos lúdicos
... pero lo siguen siendo

¿POR QUÉ GPUS?

Precio y disponibilidad

- Más de 100 millones de GPUs CUDA en el mundo (2008)





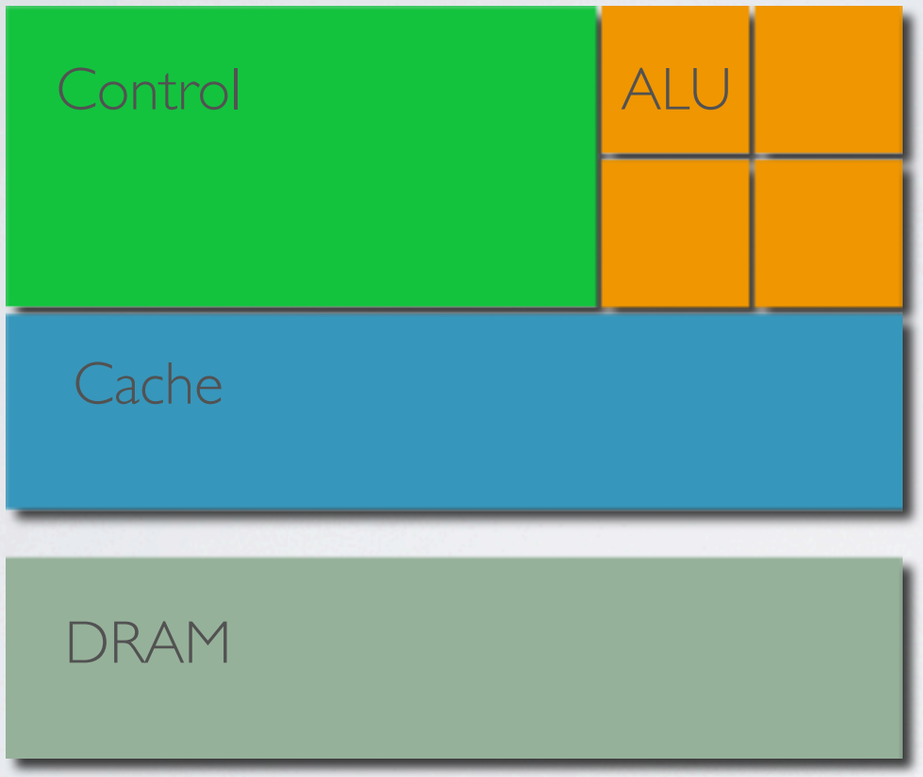
	2 x Xeon QuadCore	2 x Xeon QuadCore + Tesla s1070
Cores	8	8 + 240*4 = 968
Rendimiento pico (simple)	0,17 TFlops	4,14 TFlops
Rendimiento pico (doble)	0,08 TFlops	0,35 TFlops
Precio	$ 3000	$ 11000
Consumo	700 W	1500 W

GPUS: CARACTERÍSTICAS

Distribución de transistores

GPUS: CARACTERÍSTICAS

Distribución de transistores

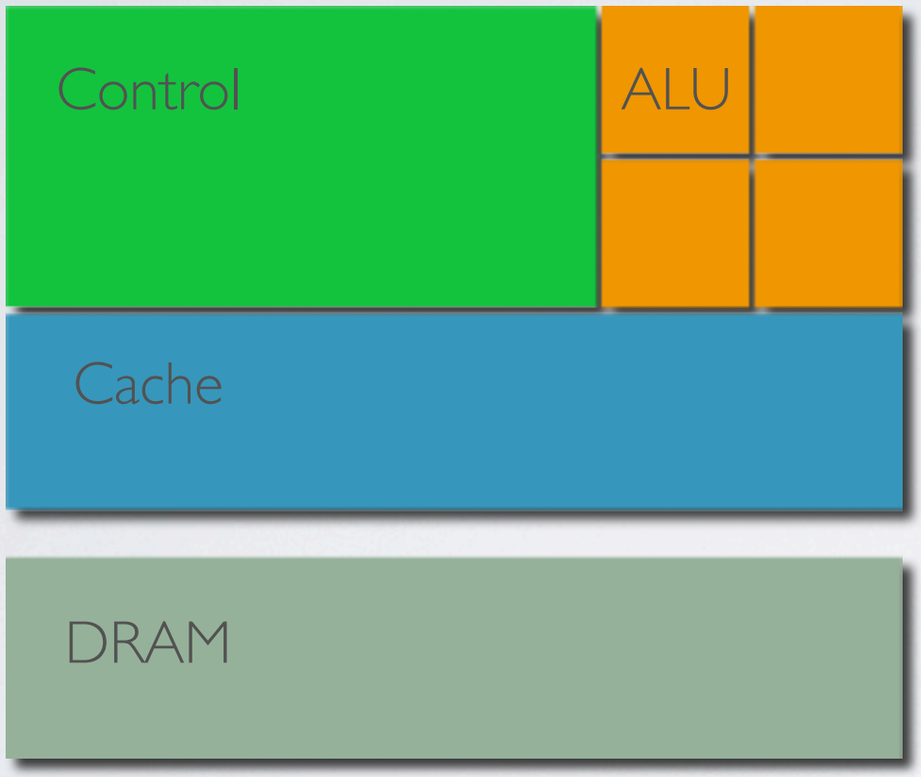


CPU

- Grandes caches
- Pocos elementos de proceso
- Optimizada para localidad espacial y temporal
- Sofisticados sistemas de control

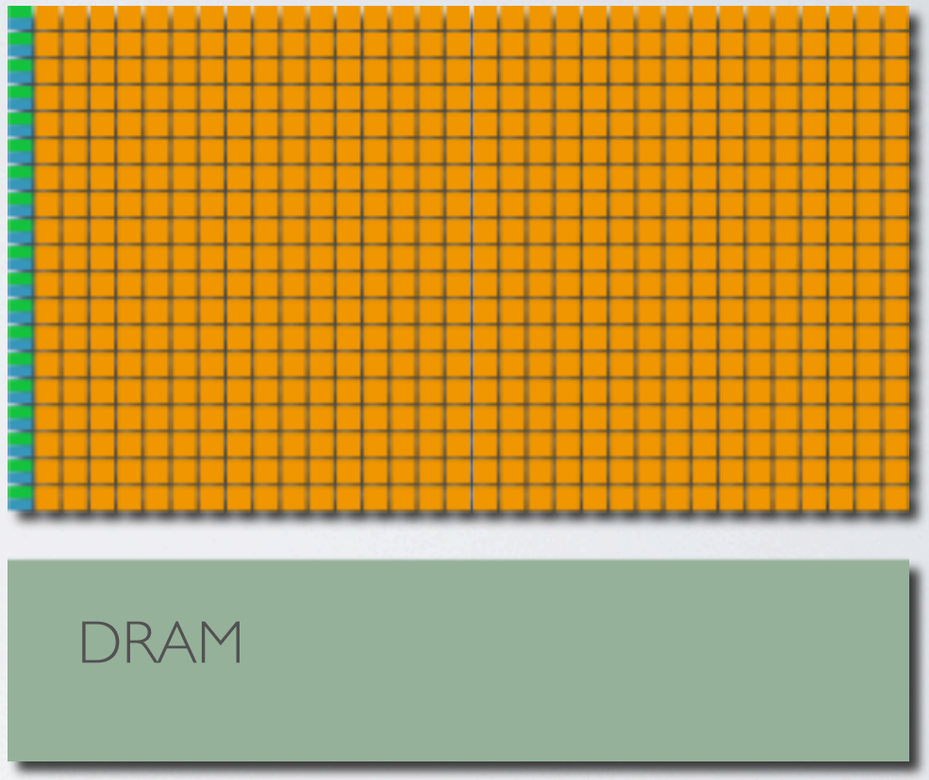
GPUS: CARACTERÍSTICAS

Distribución de transistores



CPU

- Grandes caches
- Pocos elementos de proceso
- Optimizada para localidad espacial y temporal
- Sofisticados sistemas de control

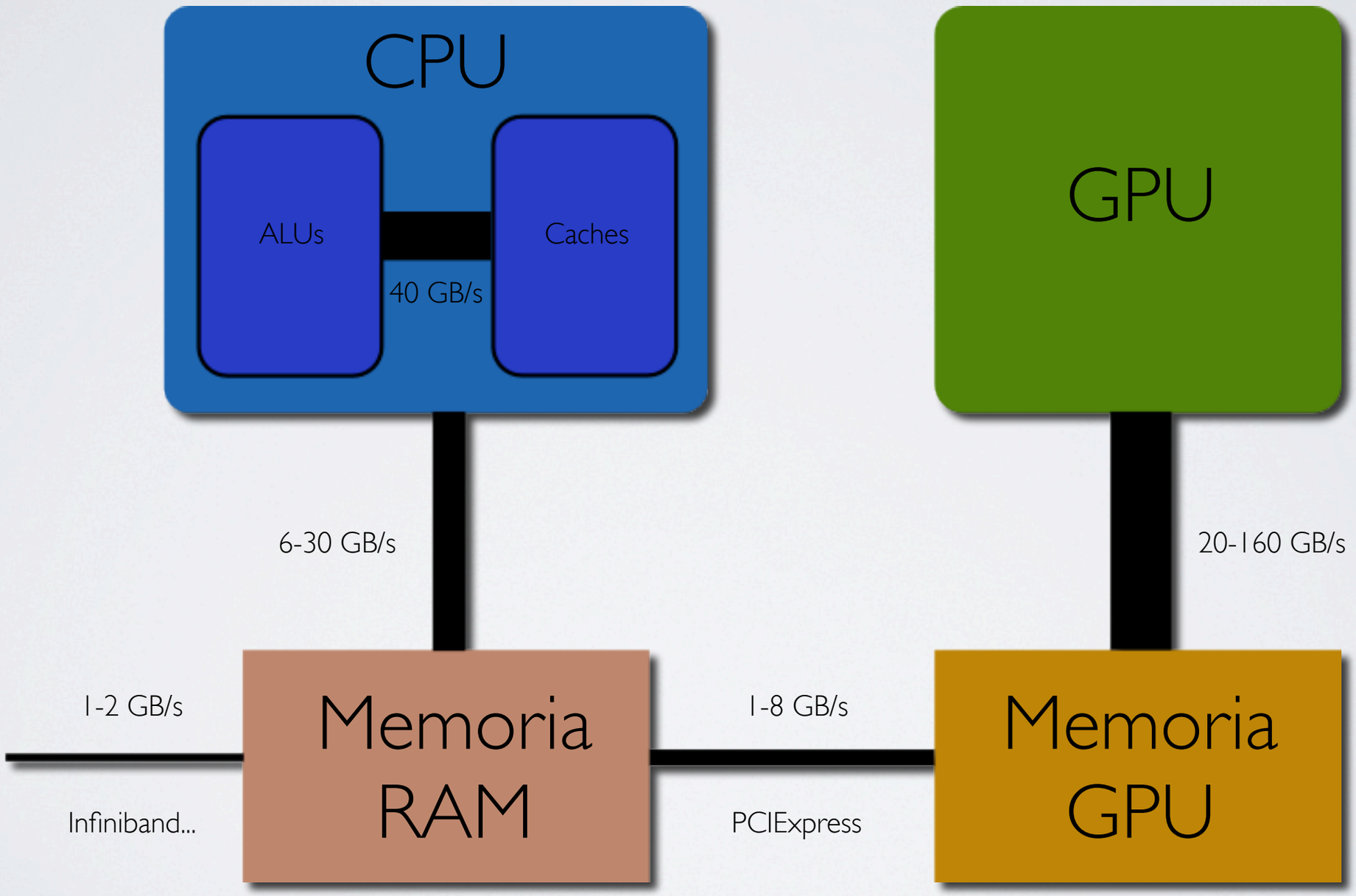


GPU

- Caches reducidas
- Elevado número de elementos de proceso
- Optimizada para acceso secuencial a datos (*streaming*)

GPUS: CARACTERÍSTICAS

Ancho de banda



ÍNDICE DE CONTENIDOS

1. Introducción / Motivación

2. La arquitectura CUDA

3. Espacios de memoria

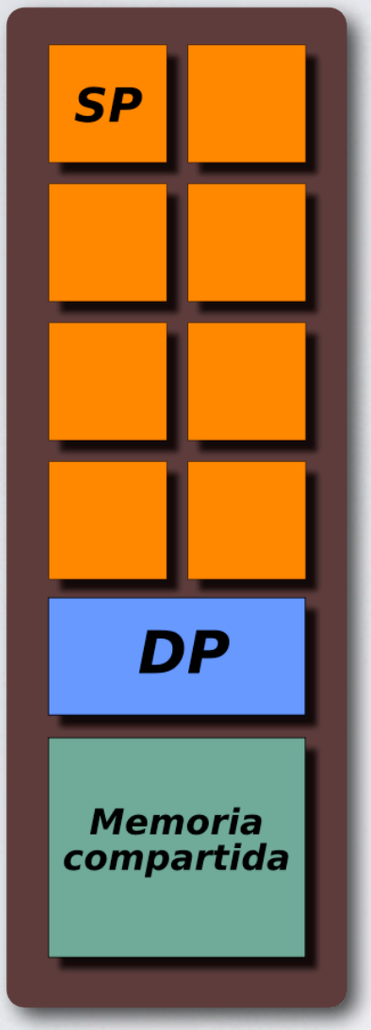
4. Modelo de ejecución CUDA

5. Programación de kernels

6. Conceptos avanzados

ARQUITECTURA CUDA

Arquitectura CUDA: conjunto de *Streaming Multiprocessors* (SM)

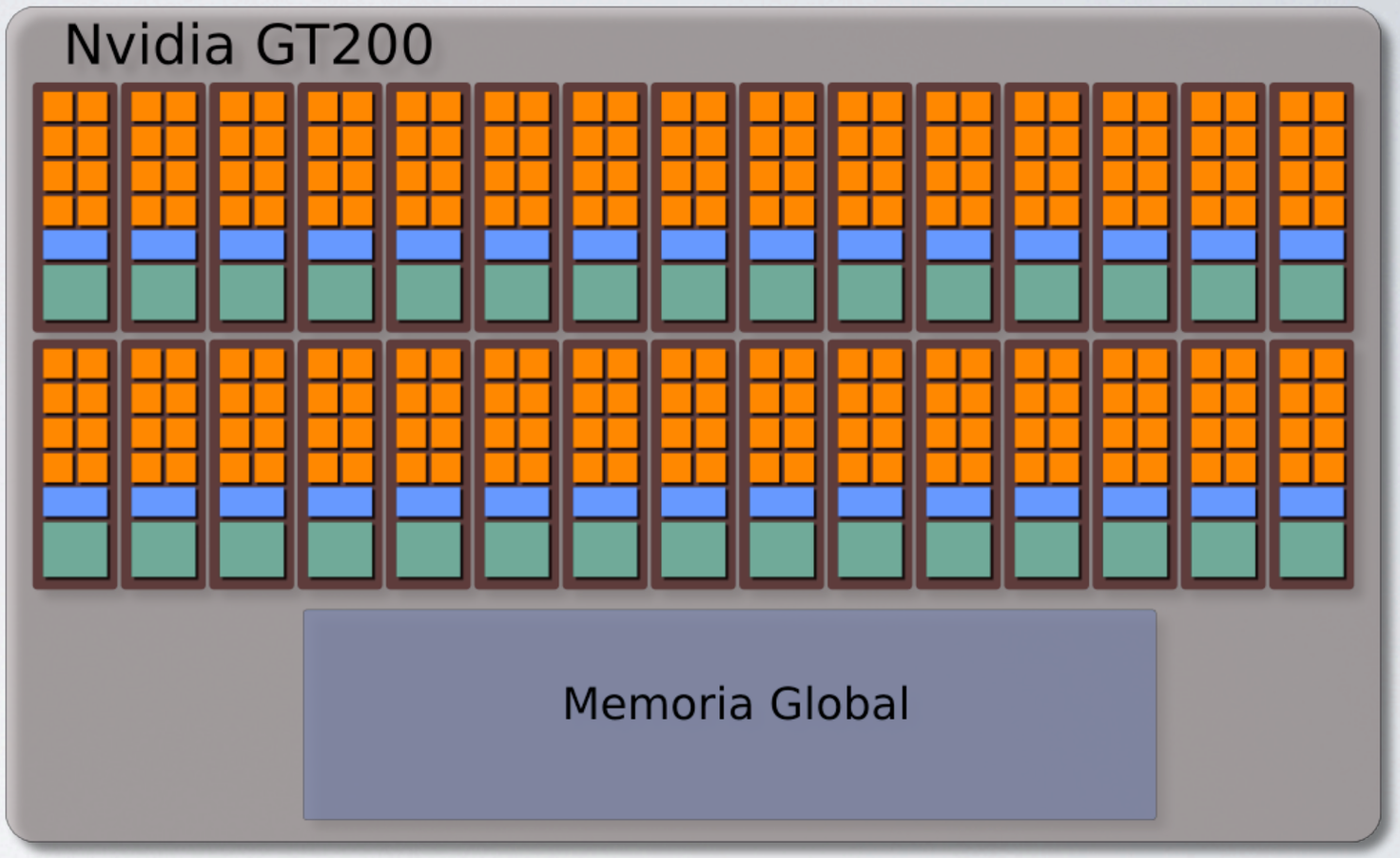


Arquitectura de un SM:

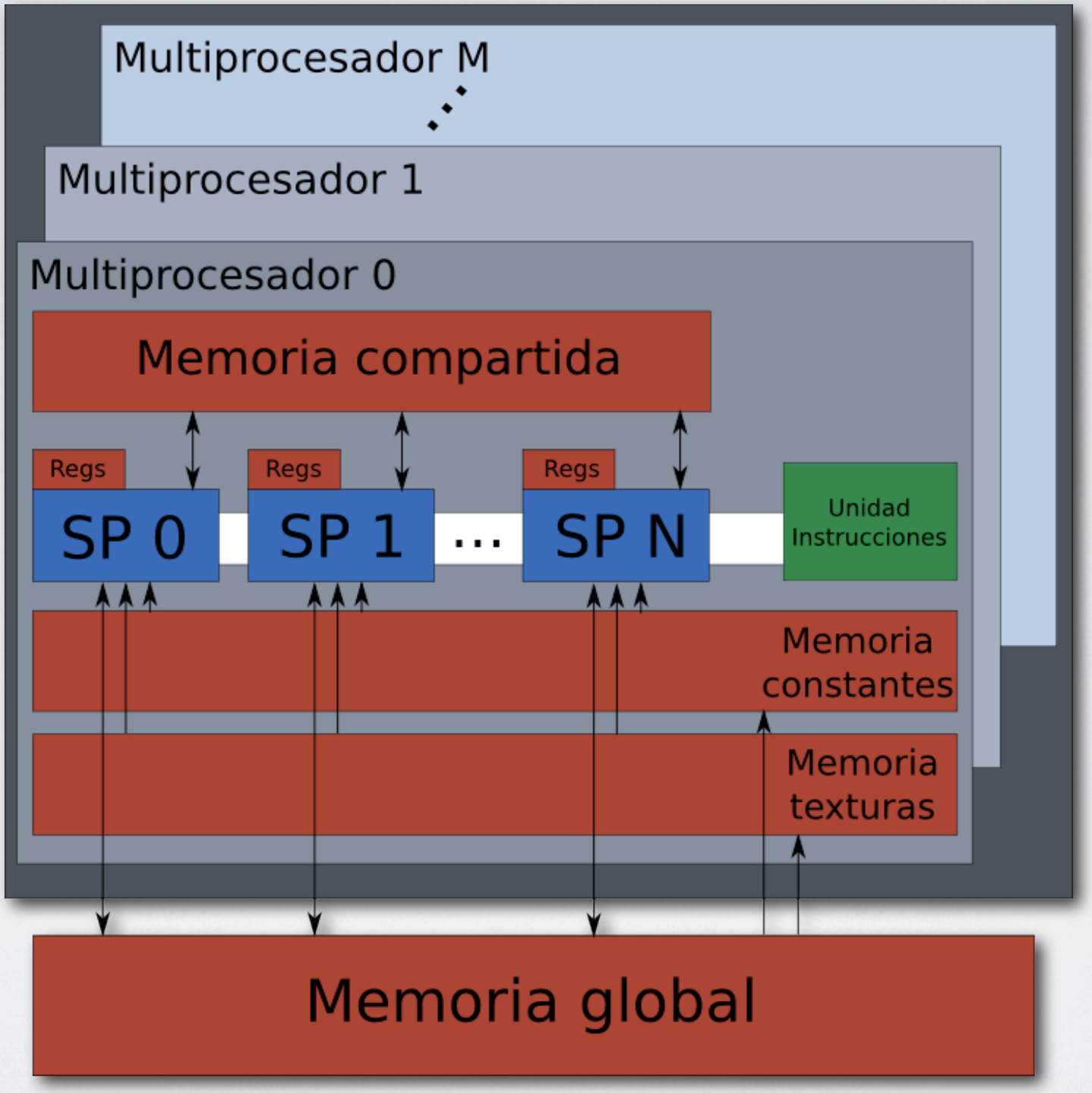
- 8 *Scalar Processors* (SP)
- 1 Unidad de Doble Precisión (DP)
- 16 Kbytes de Memoria Compartida
- Banco de registros (16384, 64 Kbytes)

LA ARQUITECTURA CUDA

- Nvidia GT200
  - 30 SM
  - 240 SP (“*cores*”)
- Arquitectura escalable
  - Aumentando el número de SM



ARQUITECTURA DE MEMORIA



ARQUITECTURA DE MEMORIA

Memoria	Localización	Cache	Acceso	Ámbito	Existencia
Registros	On-chip	N/A	R/W	Un thread	Thread
Compartida	On-chip	N/A	R/W	Threads dentro de un mismo bloque	Bloque
Global	Off-chip	No	R/W	Todos los threads + Host	Aplicación
Constantes	Off-chip	Sí	R	Todos los threads + Host	Aplicación
Texturas	Off-chip	Sí	R	Todos los threads + Host	Aplicación

ÍNDICE DE CONTENIDOS

1. Introducción / Motivación
2. La arquitectura CUDA
3. Espacios de memoria
4. Modelo de ejecución CUDA
5. Programación de kernels
6. Conceptos avanzados

ESPACIOS DE MEMORIA

- CPU y GPU poseen espacios de memoria independientes
  - Los datos se mueven a través del bus PCIExpress
  - Reserva/copia/liberación explícitas
  - Cualquier operación de reserva/copia/liberación es realizada por el host
- Los punteros son sólo direcciones
  - No posee información de situación (CPU/GPU)
  - Precauciones:
    - NO acceder a una dirección en GPU desde CPU

GESTIÓN DE MEMORIA

- El host (CPU): reserva y liberación de memoria
  - cudaMalloc(void ** ptr, size_t nbytes)
  - cudaMemset(void * ptr, int value, size_t count)
  - cudaFree(void * ptr)

```
int main( void ){
  int n        = 1024 * sizeof( int );
  int * d_A = NULL;

  cudaMalloc( (void **)&d_A, n );

  cudaMemset( d_A, 0, n );

  cudaFree( d_A );
}
```

GESTIÓN DE MEMORIA

- El host (CPU): reserva y liberación de memoria
  - cudaMalloc(void ** ptr, size_t nbytes)
  - cudaMemset(void * ptr, int value, size_t count)
  - cudaFree(void * ptr)

```
int main( void ){
  int n       = 1024 * sizeof( int );
  int * d_A = NULL;

  cudaMalloc( (void **)&d_A, n );

  cudaMemset( d_A, 0, n );

  cudaFree( d_A );
}
```

Reserva

GESTIÓN DE MEMORIA

- El host (CPU): reserva y liberación de memoria
  - cudaMalloc(void ** ptr, size_t nbytes)
  - cudaMemset(void * ptr, int value, size_t count)
  - cudaFree(void * ptr)

```
int main( void ){
  int n       = 1024 * sizeof( int );
  int * d_A = NULL;

  cudaMalloc( (void **)&d_A, n );

  cudaMemset( d_A, 0, n );

  cudaFree( d_A );
}
```

Reserva

Inicialización

GESTIÓN DE MEMORIA

- El host (CPU): reserva y liberación de memoria
  - cudaMalloc(void ** ptr, size_t nbytes)
  - cudaMemset(void * ptr, int value, size_t count)
  - cudaFree(void * ptr)

```
int main( void ){
  int n       = 1024 * sizeof( int );
  int * d_A = NULL;

  cudaMalloc( (void **)&d_A, n );

  cudaMemset( d_A, 0, n );

  cudaFree( d_A );
}
```

Reserva

Inicialización

Liberación

GESTIÓN DE MEMORIA

- El host (CPU): transferencias de datos
  - cudaMemcpy(void * dst, void * src, size_t nbytes, enum cudaMemcpyKind direction)
    - No comienza hasta la finalización de las llamadas CUDA anteriores
    - Llamadas bloqueantes (también existen no bloqueantes)
  - enum cudaMemcpyKind
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice

EJEMPLO 1

1. Reserva de memoria en CPU para *n* flotantes
2. Reserva de memoria en GPU para *n* flotantes
3. Inicialización de la memoria en GPU a 0s
4. Transferencia entre memoria de vídeo y RAM
5. Liberación de memoria

EJEMPLO

```
1  int main( void )
2  {
3     int n = 16;
4     int num_bytes = n * sizeof( float );
5
6     float * h_A = NULL;
7     float * d_A = NULL;
8
9     h_A = (float *)malloc( num_bytes );
10    cudaMalloc( (void **)&d_A, num_bytes );
11
12    cudaMemset( d_A, 0, num_bytes );
13    cudaMemcpy( h_A, d_A, num_bytes, cudaMemcpyDeviceToHost );
14
15    // Mostrar valores
16    // ...
17
18    free( h_A );
19    cudaFree( d_A );
20
21    return 0;
22
23 }
```

EJEMPLO

```
1  int main( void )
2  {
3     int n = 16;
4     int num_bytes = n * sizeof( float );
5
6     float * h_A = NULL;
7     float * d_A = NULL;
8
9     h_A = (float *)malloc( num_bytes );
10    cudaMalloc( (void **)&d_A, num_bytes );
11
12    cudaMemset( d_A, 0, num_bytes );
13    cudaMemcpy( h_A, d_A, num_bytes, cudaMemcpyDeviceToHost );
14
15    // Mostrar valores
16    // ...
17
18    free( h_A );
19    cudaFree( d_A );
20
21    return 0;
22
23 }
```

Reserva de memoria en GPU

EJEMPLO

```
1  int main( void )
2  {
3     int n = 16;
4     int num_bytes = n * sizeof( float );
5
6     float * h_A = NULL;
7     float * d_A = NULL;
8
9     h_A = (float *)malloc( num_bytes );
10    cudaMalloc( (void **)&d_A, num_bytes );
11
12    cudaMemset( d_A, 0, num_bytes );
13    cudaMemcpy( h_A, d_A, num_bytes, cudaMemcpyDeviceToHost );
14
15    // Mostrar valores
16    // ...
17
18    free( h_A );
19    cudaFree( d_A );
20
21    return 0;
22
23 }
```

Reserva de memoria en GPU

Inicialización de memoria en GPU

EJEMPLO

```
1  int main( void )
2  {
3     int n = 16;
4     int num_bytes = n * sizeof( float );
5
6     float * h_A = NULL;
7     float * d_A = NULL;
8
9     h_A = (float *)malloc( num_bytes );
10    cudaMalloc( (void **)&d_A, num_bytes );
11
12    cudaMemset( d_A, 0, num_bytes );
13    cudaMemcpy( h_A, d_A, num_bytes, cudaMemcpyDeviceToHost );
14
15    // Mostrar valores
16    // ...
17
18    free( h_A );
19    cudaFree( d_A );
20
21    return 0;
22
23 }
```

Reserva de memoria en GPU

Inicialización de memoria en GPU

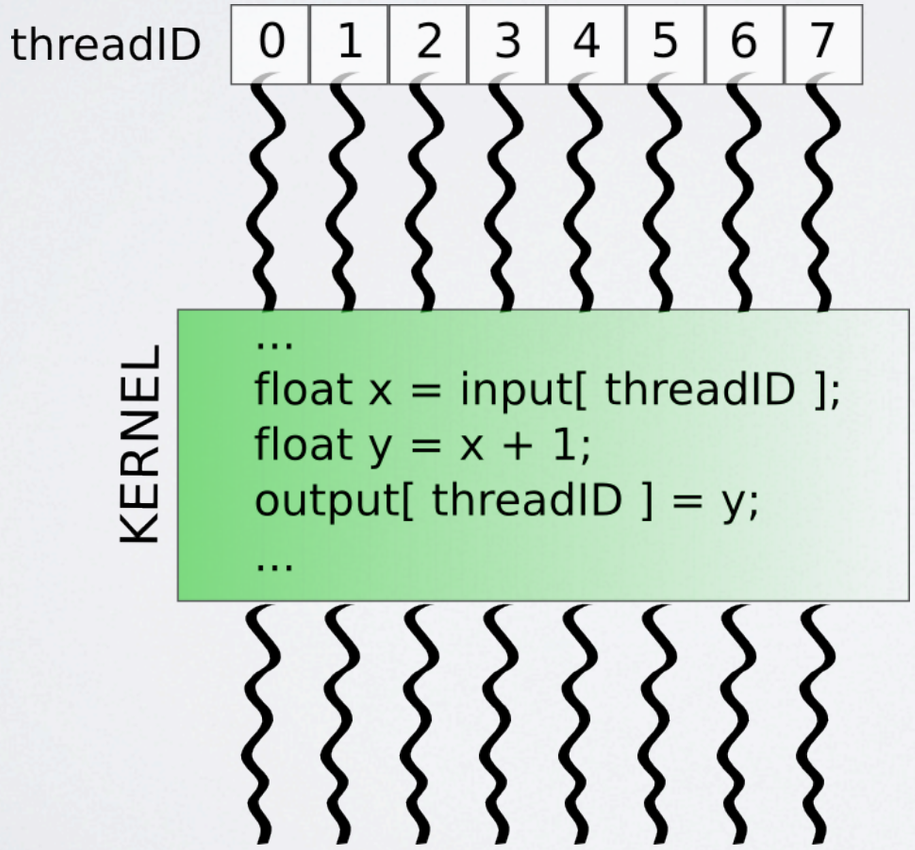
Transferencia de datos entre CPU y GPU

ÍNDICE DE CONTENIDOS

1. Introducción / Motivación
2. La arquitectura CUDA
3. Espacios de memoria
4. Modelo de ejecución CUDA
5. Programación de kernels
6. Conceptos avanzados

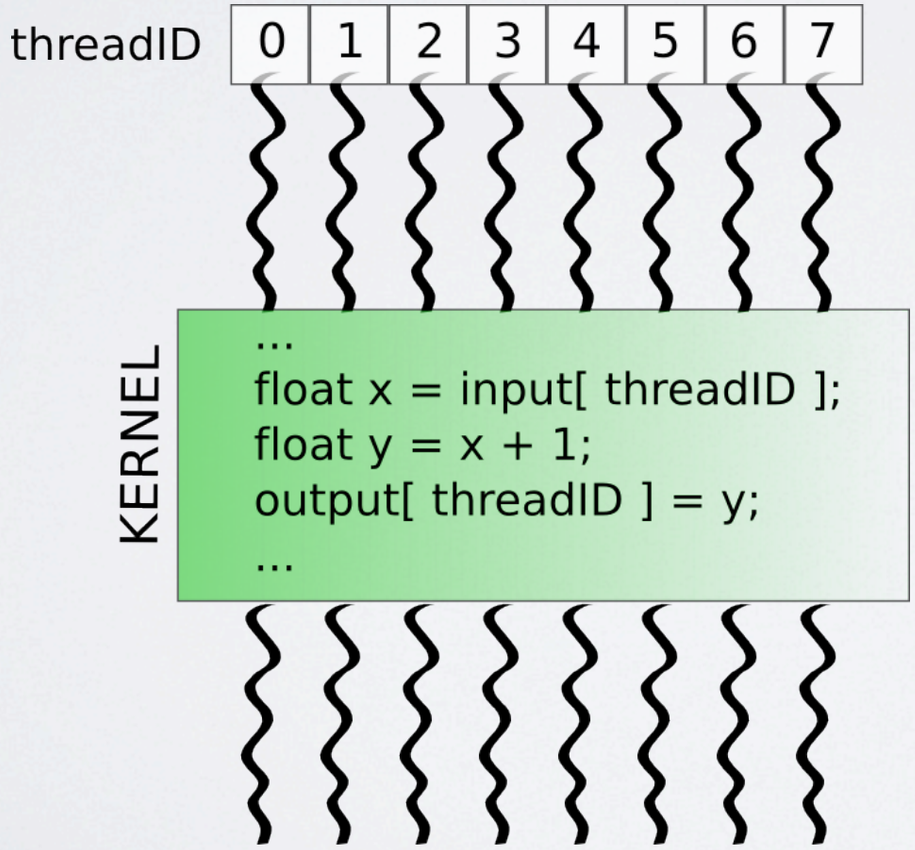
MODELO DE EJECUCIÓN

- Un *kernel* CUDA es ejecutado por un array de *threads*
  - Todos los *threads* ejecutan el mismo código
  - Cada *thread* recibe un identificador único
  - En función de su ID:
    - Tomará decisiones de control
    - Operará sobre unos datos u otros (controlado por el programador)



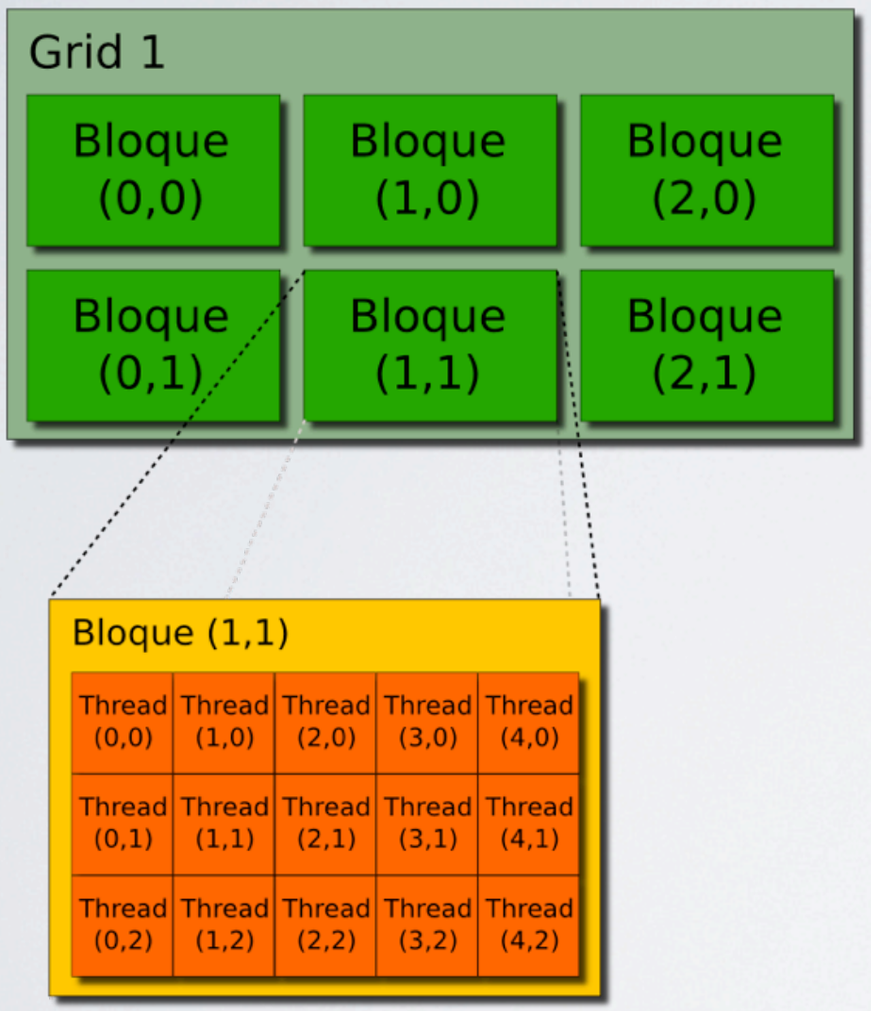
MODELO DE EJECUCIÓN

- Un *kernel* CUDA es ejecutado por un array de *threads*
  - Todos los *threads* ejecutan el mismo código
  - Cada *thread* recibe un identificador único
  - En función de su ID:
    - Tomará decisiones de control
    - Operará sobre unos datos u otros (controlado por el programador)



Paradigma **SIMD** (Simple Instruction - Multiple Data)

MODELO DE EJECUCIÓN



JERARQUÍA DE THREADS

- Los threads se particionan en **bloques de *threads***
  - ***Grid***: conjunto de bloques de threads lanzados a ejecución
- **Bloque de *threads***: grupo de *threads* que pueden
  - Sincronizar su ejecución
  - Comunicarse a través de la memoria compartida

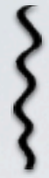
- Número y disposición de *threads* se deciden en tiempo de ejecución

MODELO DE EJECUCIÓN

Software Hardware

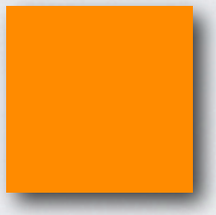
MODELO DE EJECUCIÓN

Software



Thread

Hardware

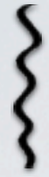


Streaming Processor

- Cada thread se ejecuta en un procesador escalar

MODELO DE EJECUCIÓN

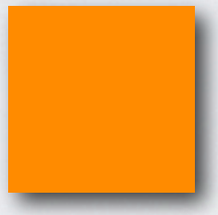
Software



Thread

Bloque de threads

Hardware



Streaming Processor



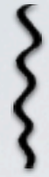
Multiprocesador

- Cada thread se ejecuta en un procesador escalar

- Los bloques de threads se ejecutan en los multiprocesadores
- Los bloques de threads NO migran entre MP
- Ejecución concurrente de múltiples bloques de threads en un MP -- Limitado por la memoria compartida y número de registros

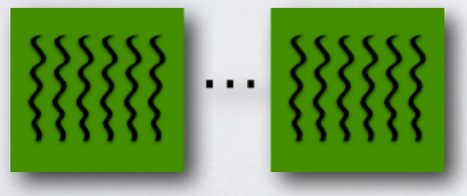
MODELO DE EJECUCIÓN

Software



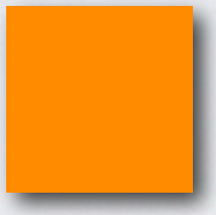
Thread

Bloque de threads



Grid

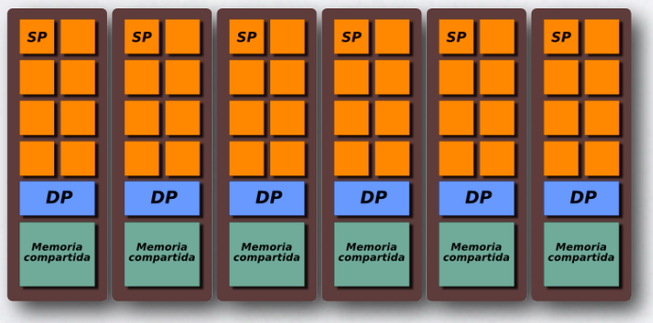
Hardware



Streaming Processor



Multiprocesador



GPU

- Cada thread se ejecuta en un procesador escalar

- Los bloques de threads se ejecutan en los multiprocesadores
- Los bloques de threads NO migran entre MP
- Ejecución concurrente de múltiples bloques de threads en un MP -- Limitado por la memoria compartida y número de registros

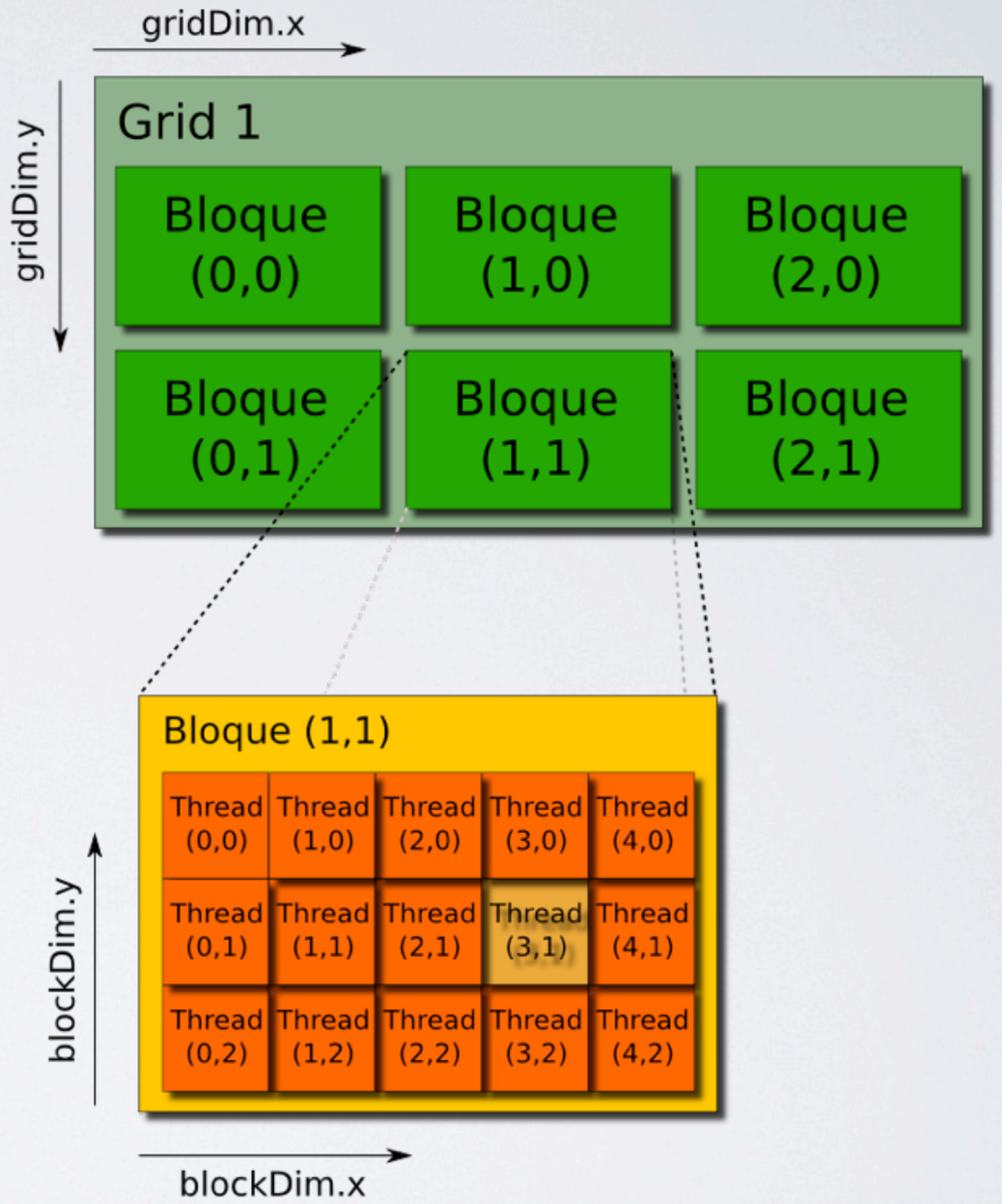
- Los *kernels* se ejecutan como *grids* de bloques
- UN único *kernel* concurrente*

MODELO DE PROGRAMACIÓN

- Kernel ejecutado por un conjunto de threads
- Los threads se agrupan en bloques de threads
  - Para sincronizar su ejecución
  - Para comunicarse a través de la memoria compartida
- El código del kernel es ejecutado por cada thread
  - Cada thread *puede* ejecutar un camino distinto dentro del código
  - Cada thread lleva asociados:
    - Identificador propio dentro del bloque
    - Identificador del bloque al que pertece dentro del grid

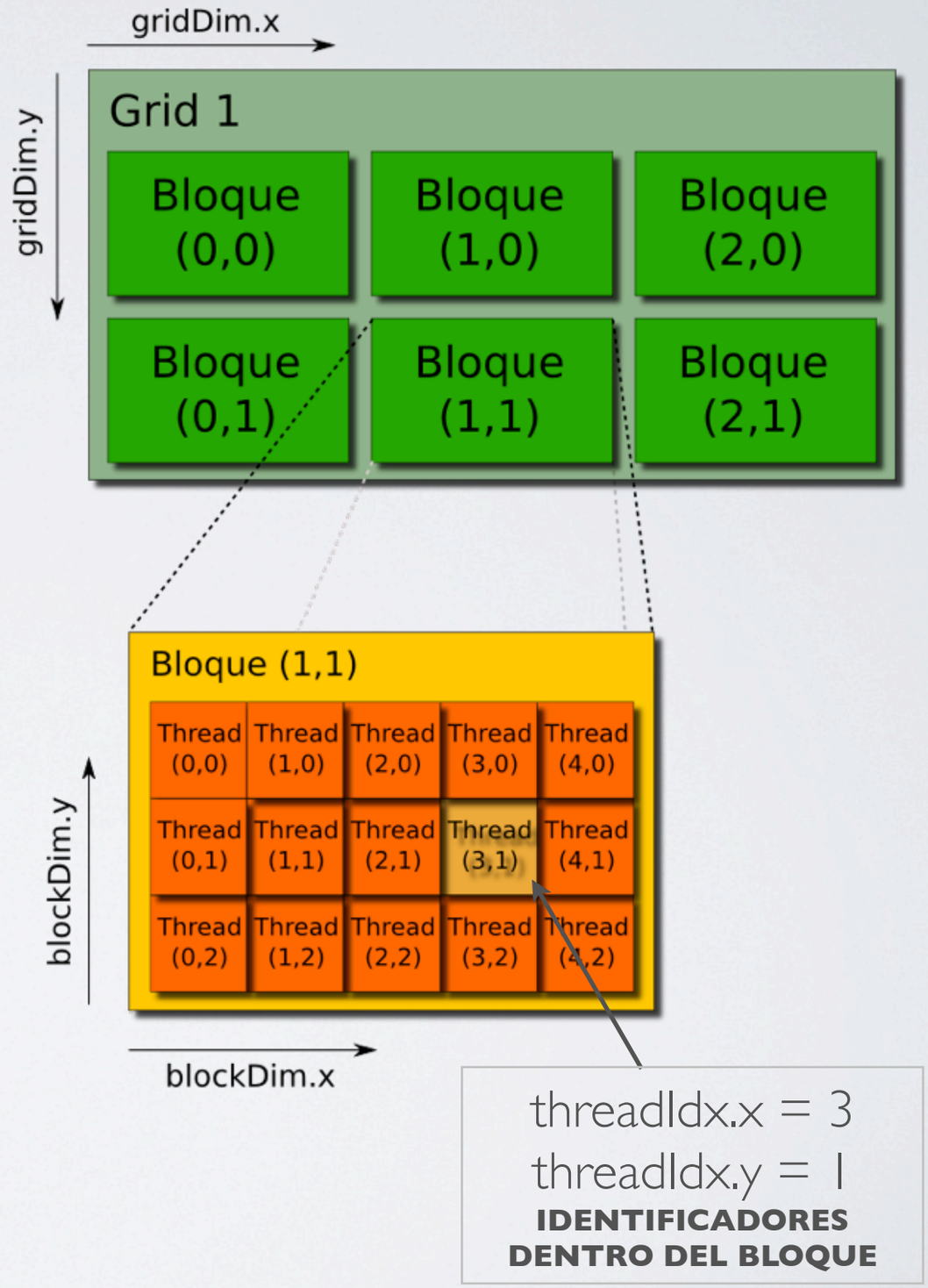
IDENTIFICADORES Y DIMENSIONES

- Especificados por el programador en tiempo de ejecución:
  - Tamaño de bloque
  - Número de bloques
- Variables intrínsecas
  - **threadIdx, blockIdx**
  - **blockDim, gridDim**



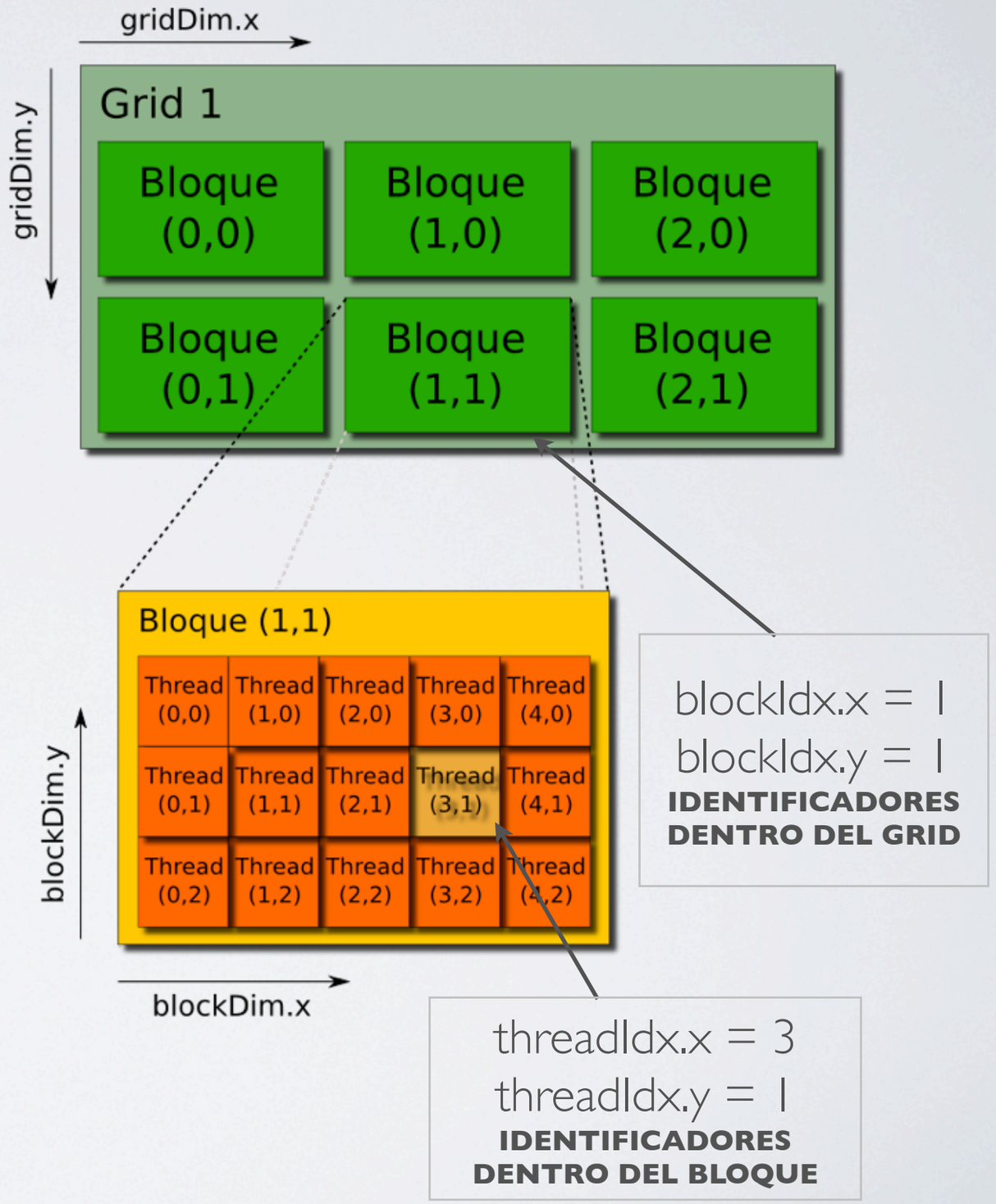
IDENTIFICADORES Y DIMENSIONES

- Especificados por el programador en tiempo de ejecución:
  - Tamaño de bloque
  - Número de bloques
- Variables intrínsecas
  - **threadIdx, blockIdx**
  - **blockDim, gridDim**



IDENTIFICADORES Y DIMENSIONES

- Especificados por el programador en tiempo de ejecución:
  - Tamaño de bloque
  - Número de bloques
- Variables intrínsecas
  - **threadIdx, blockIdx**
  - **blockDim, gridDim**

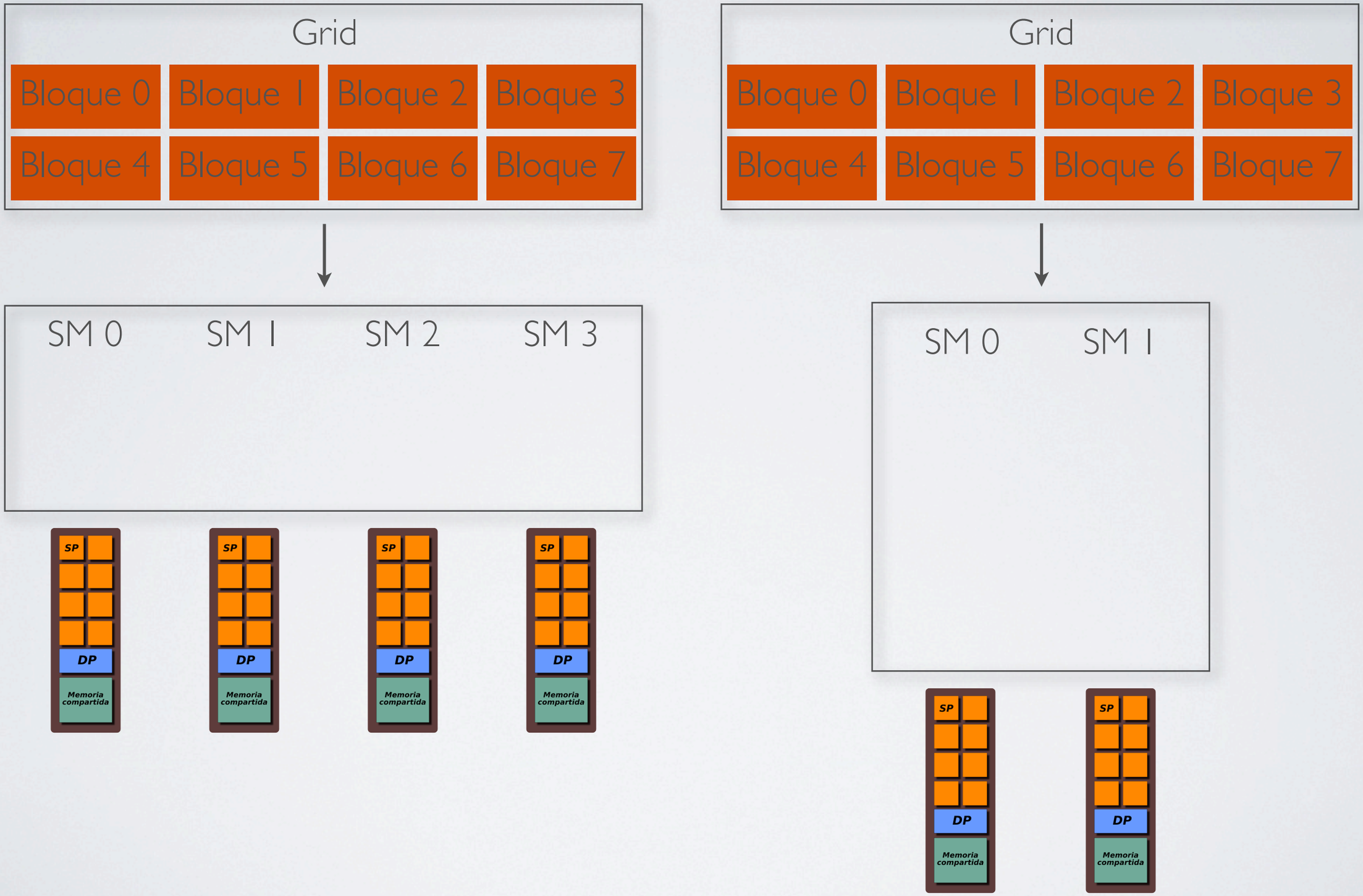


PLANIFICACIÓN Y EJECUCIÓN DE THREADS

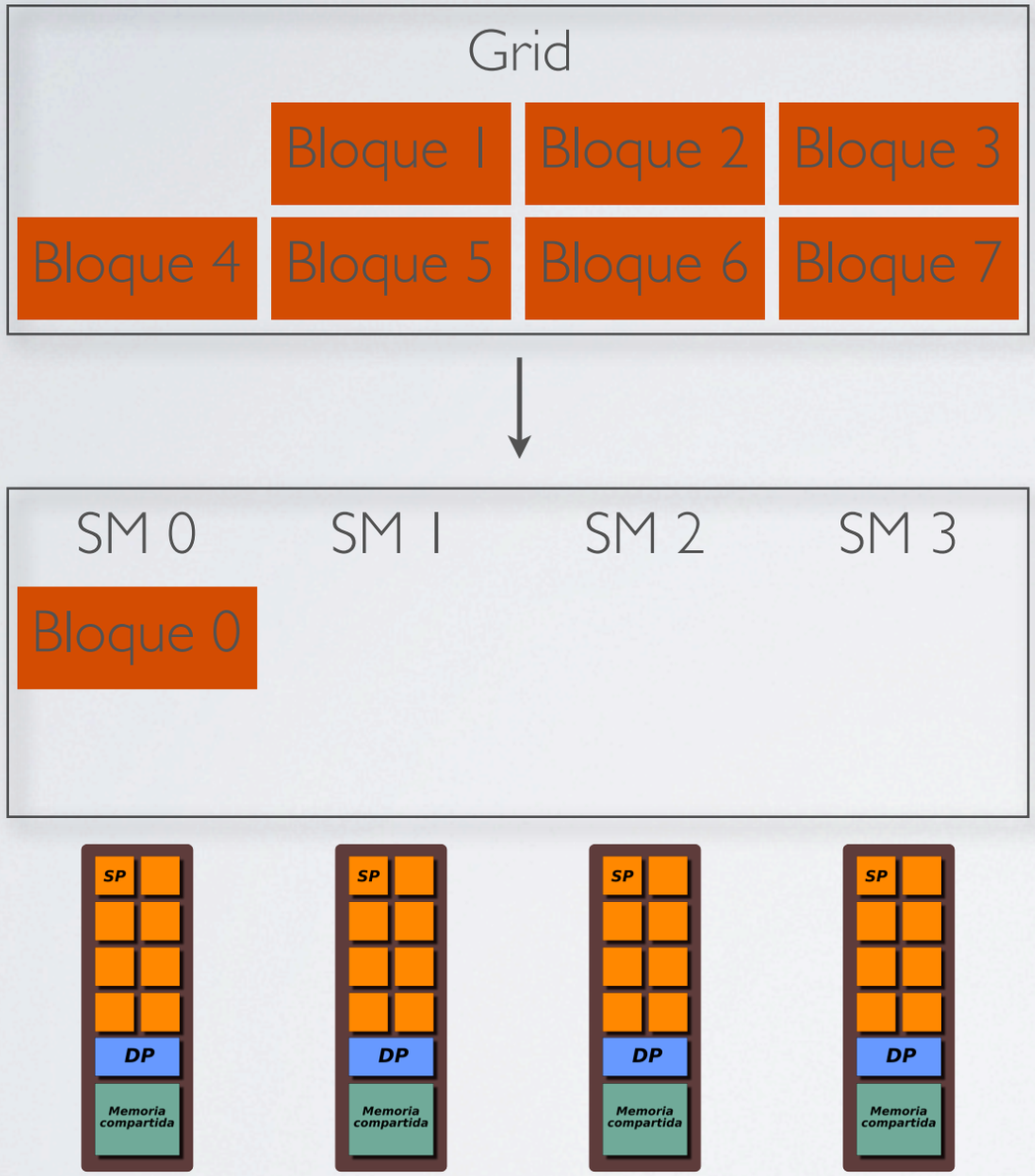
- Threads ligeros: bajo sobrecoste en la creación y planificación
- Miles de threads: paralelismo de grano muy fino
- Ejecución de un kernel:

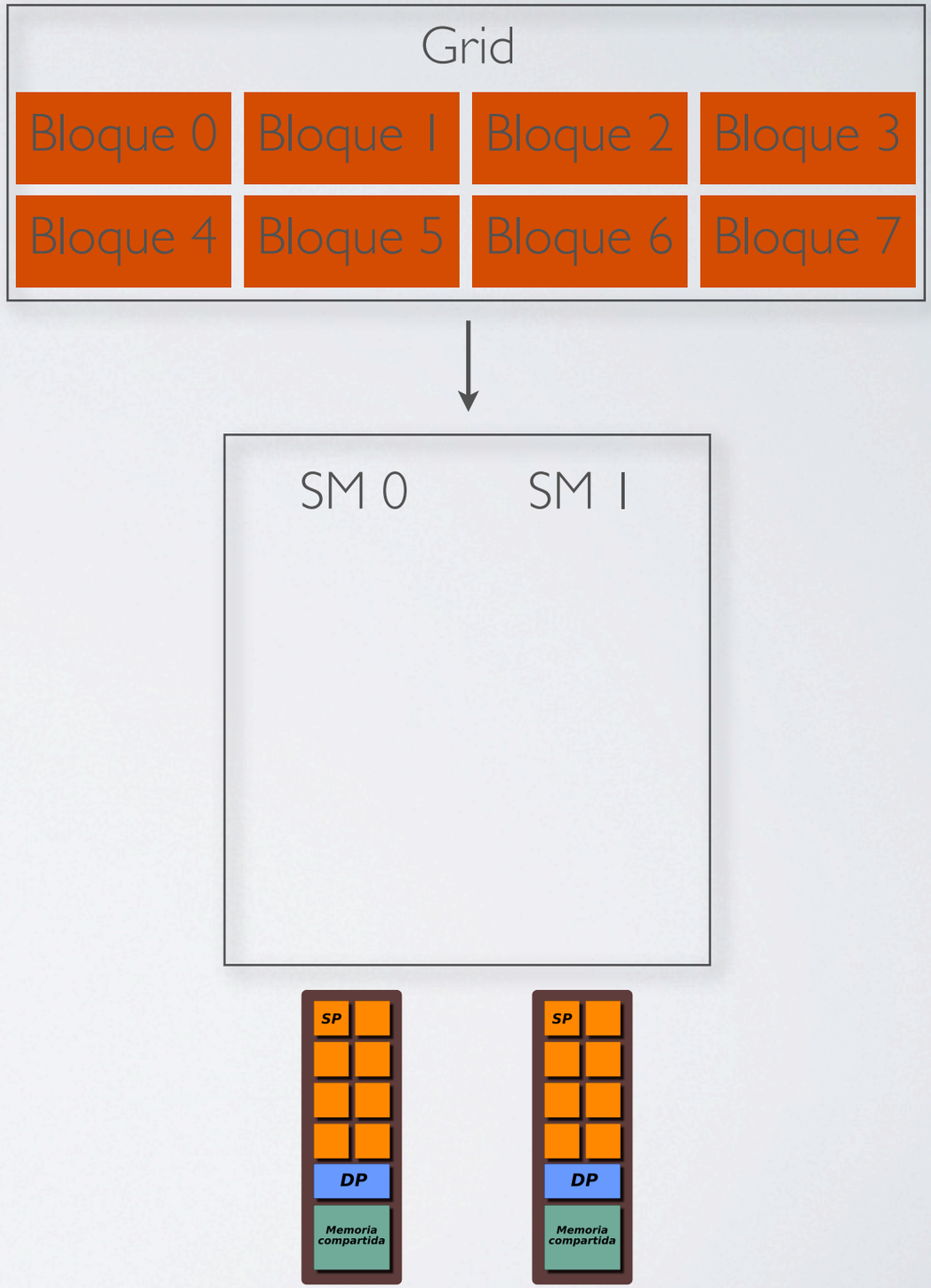
1. Se enumeran los bloques de threads
2. Se distribuyen los bloques de threads a los MP disponibles
3. Los threads dentro de un bloque se ejecutan **concurrentemente** en un MP
4. Al final de la ejecución de un bloque, se lanzan a ejecución nuevos bloques sobre el mismo MP

PLANIFICACIÓN

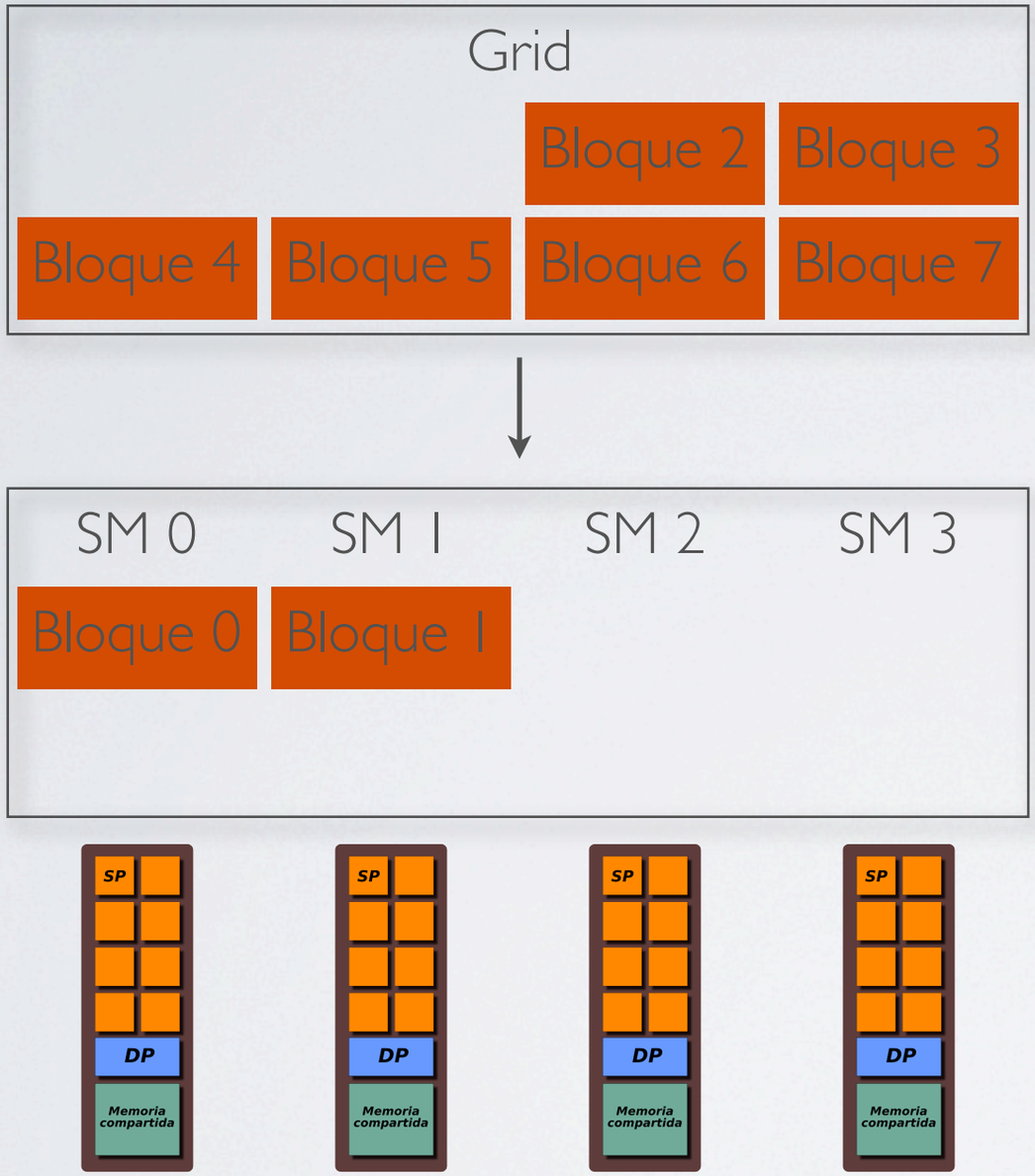


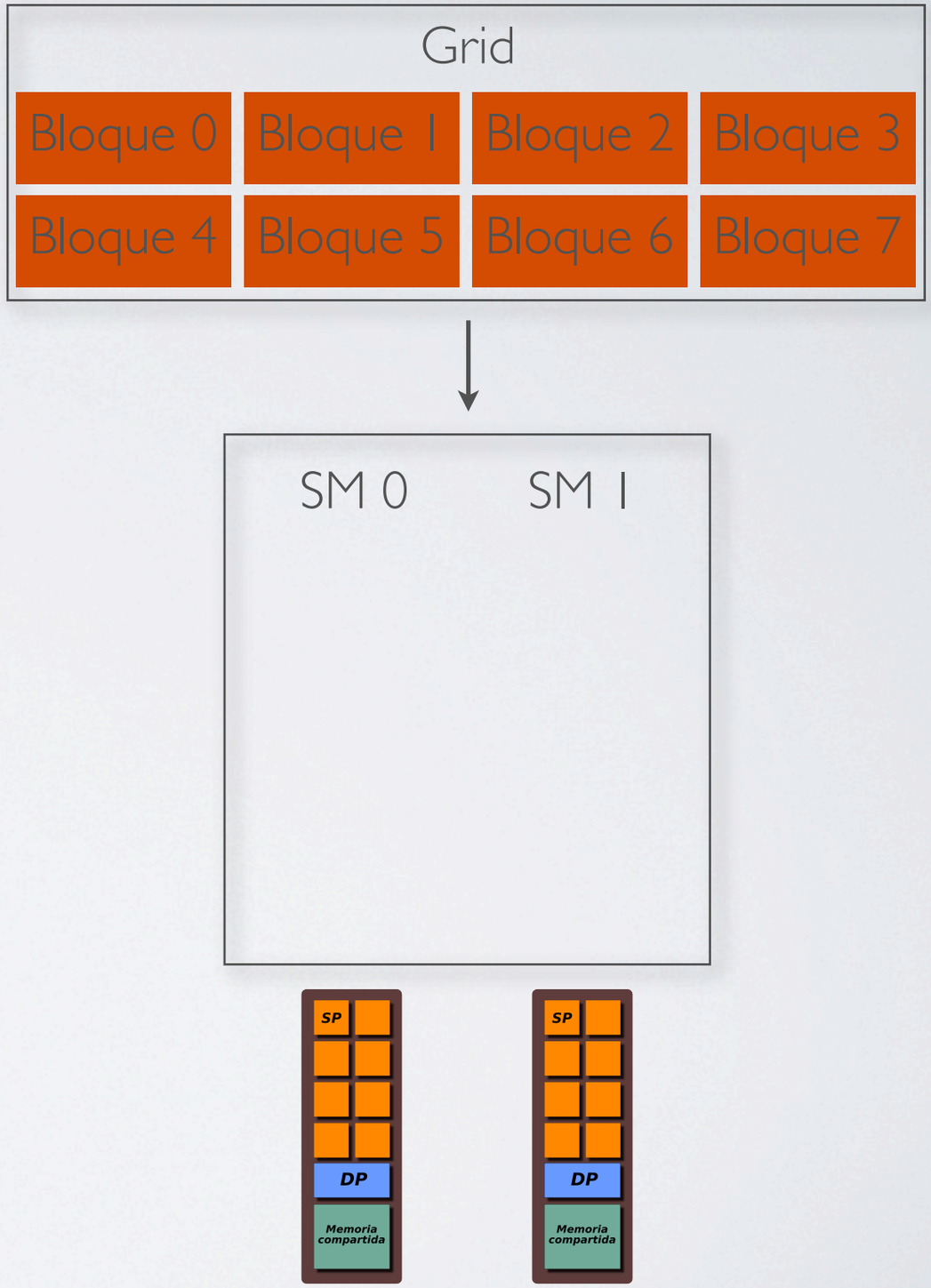
PLANIFICACIÓN



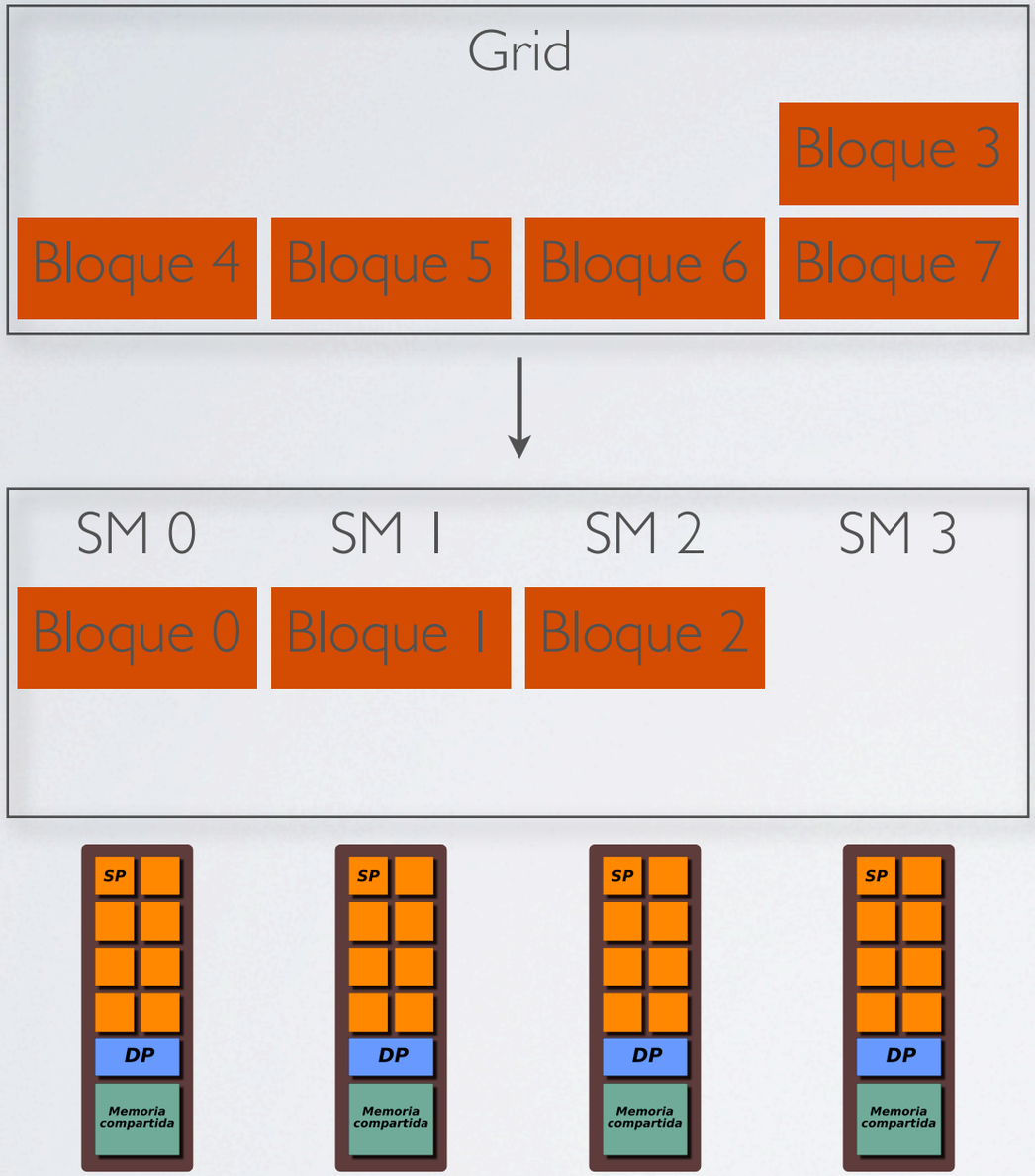


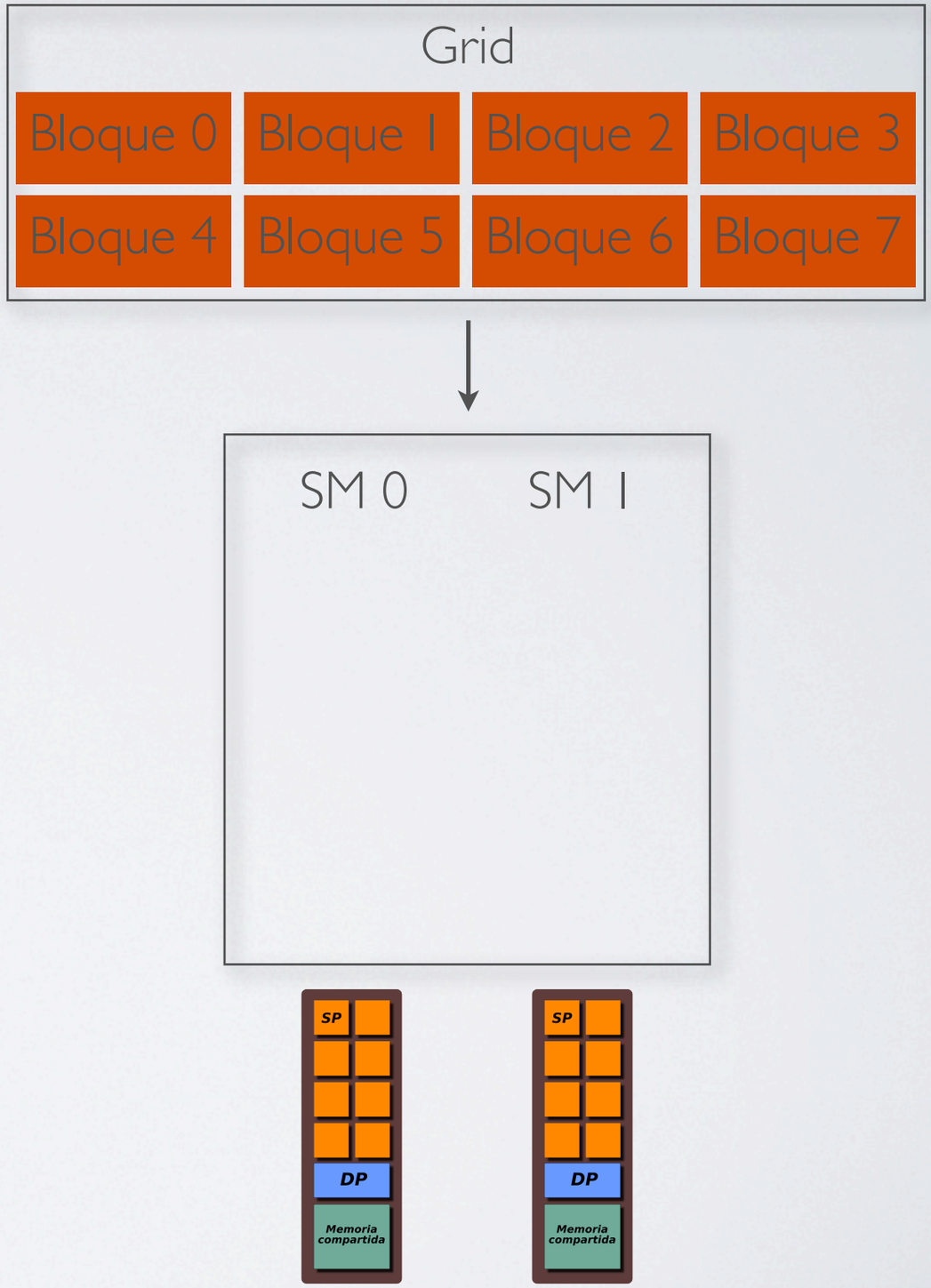
PLANIFICACIÓN



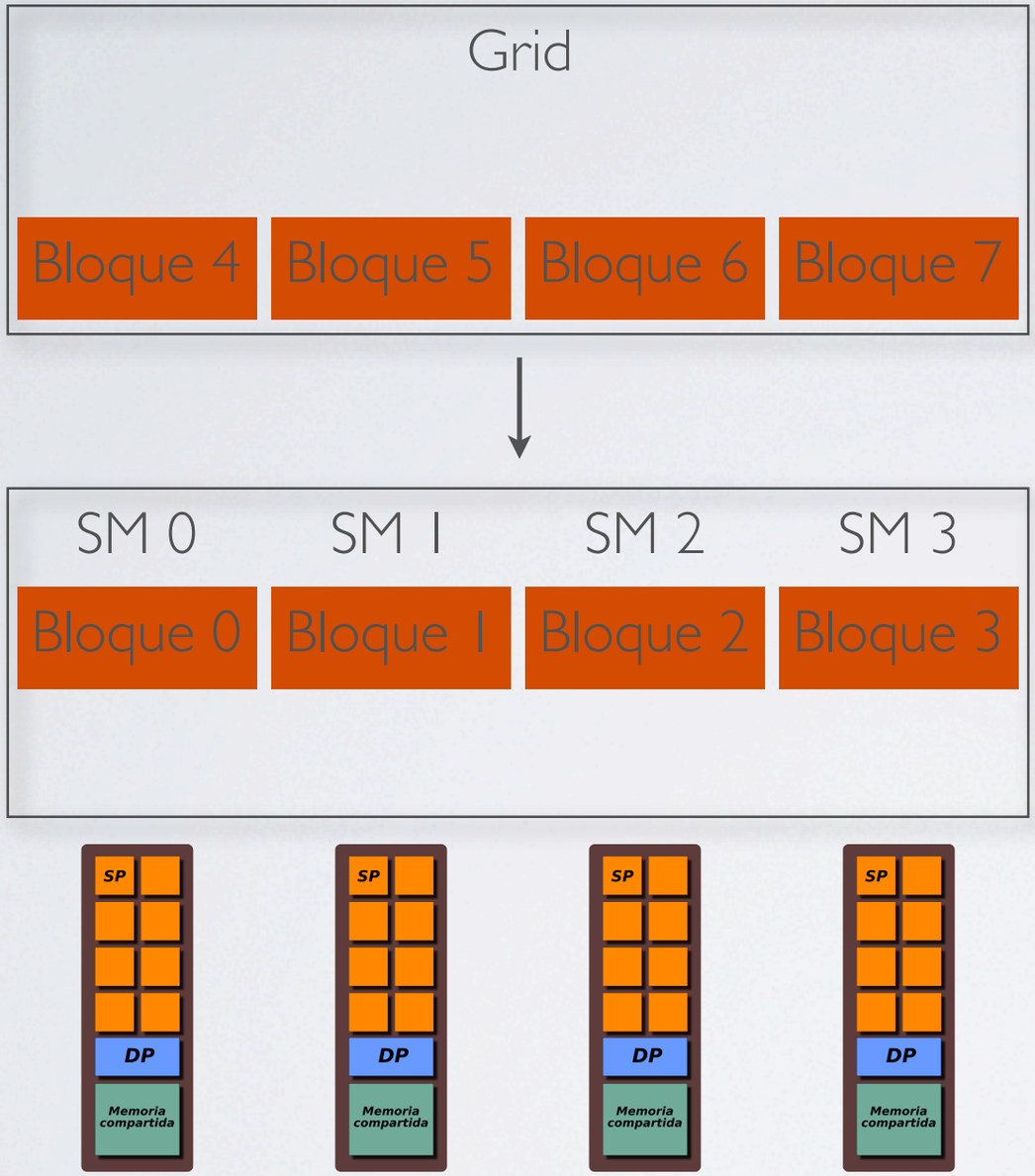


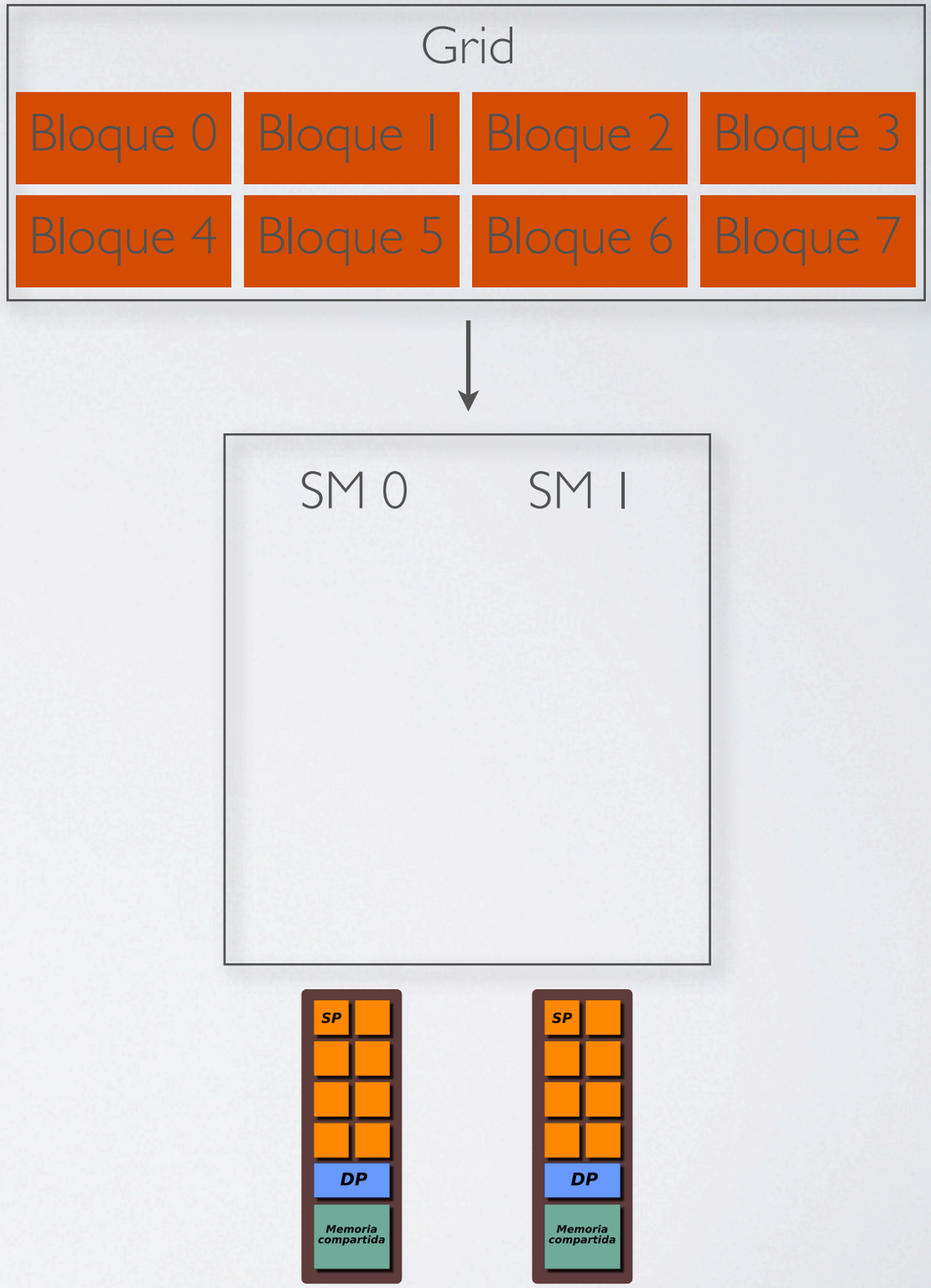
PLANIFICACIÓN



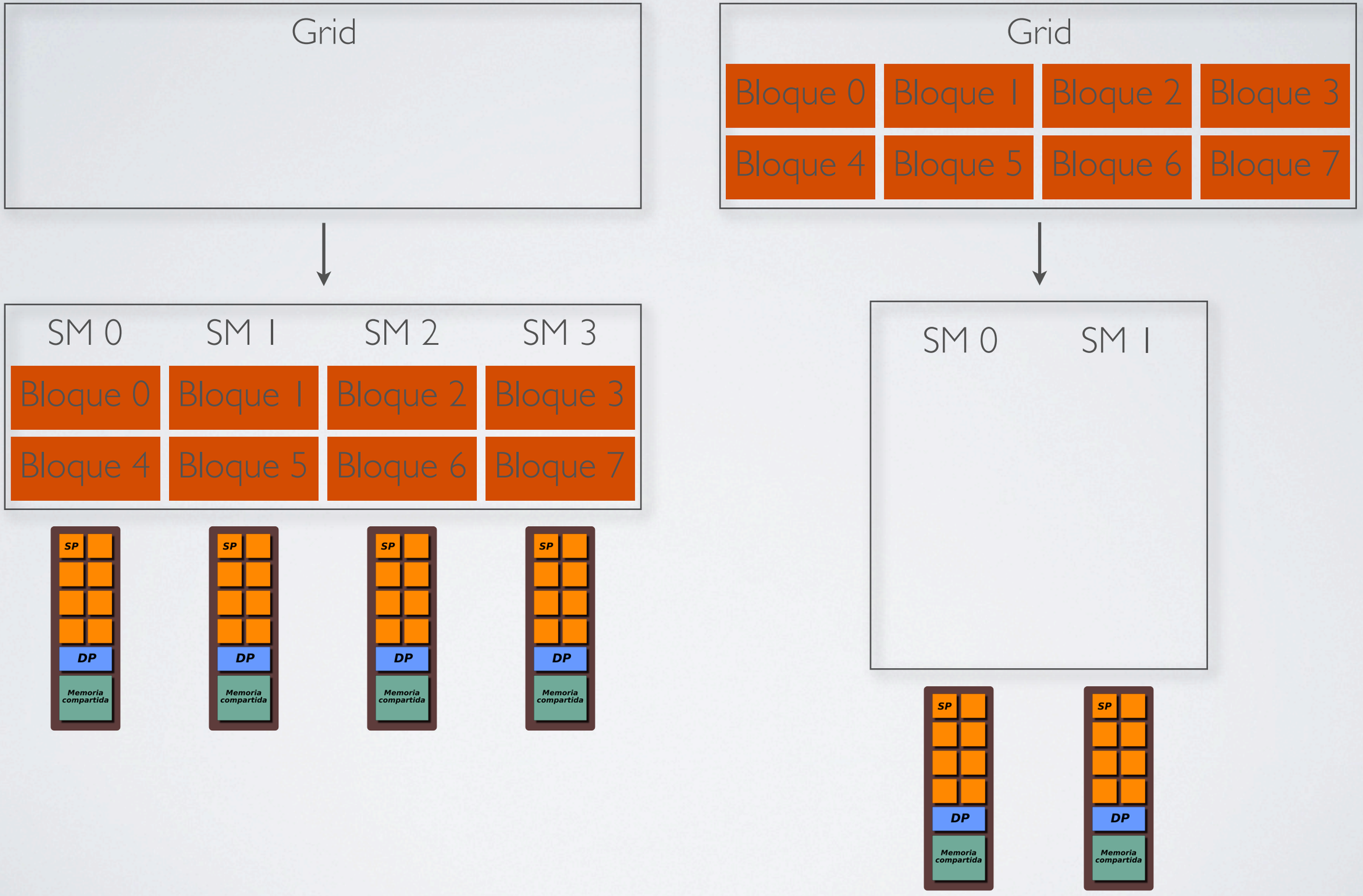


PLANIFICACIÓN

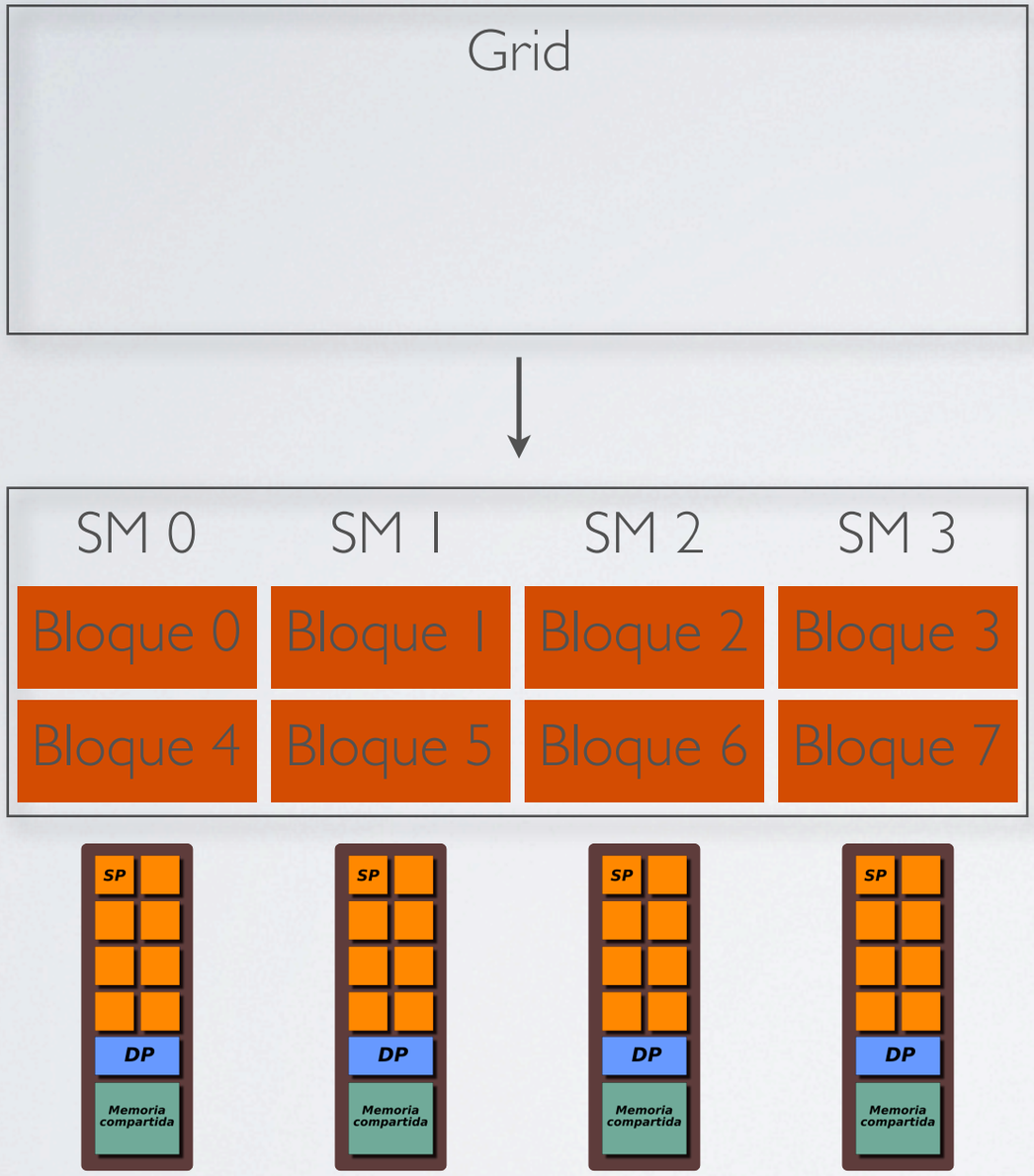


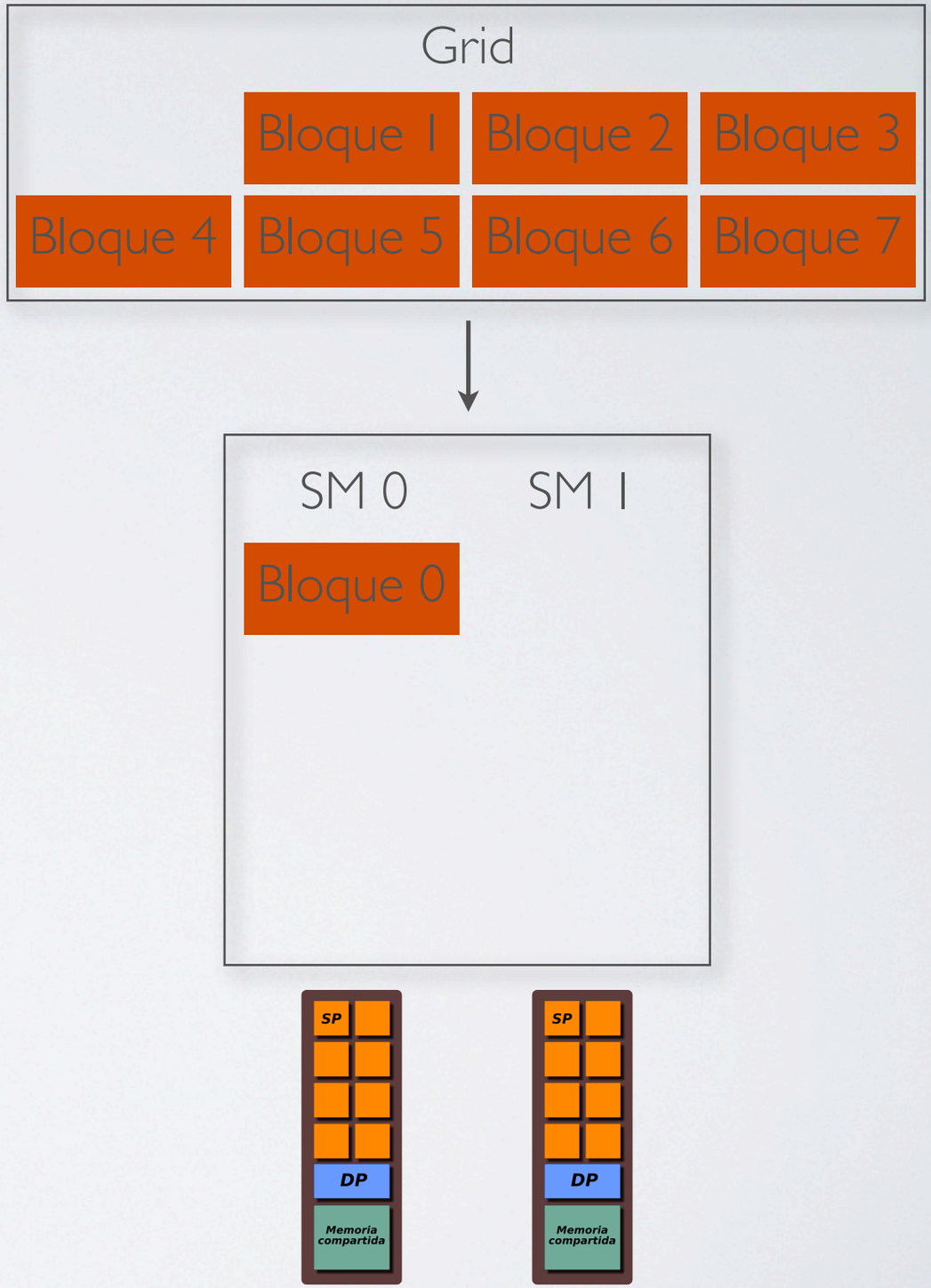


PLANIFICACIÓN

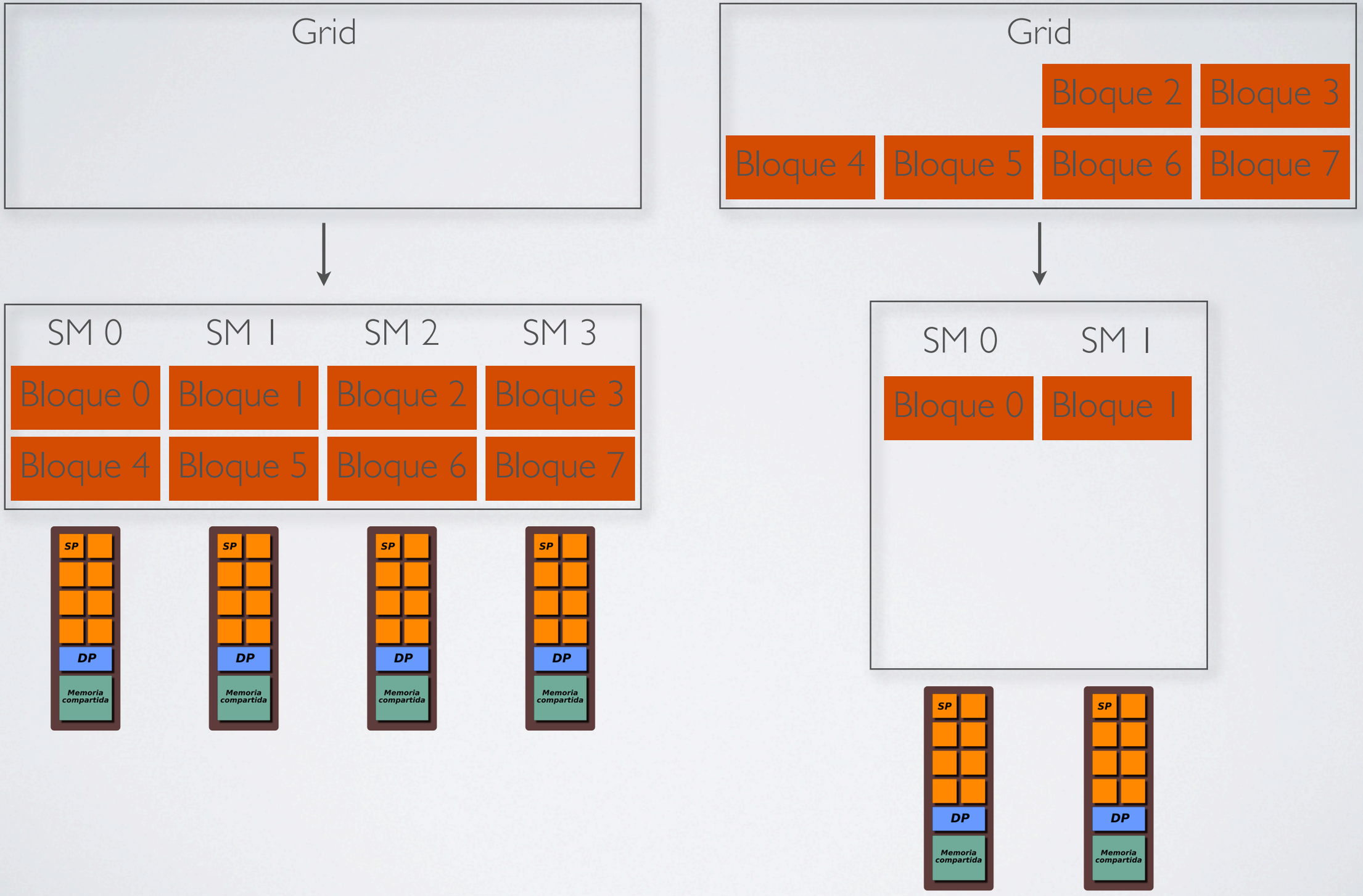


PLANIFICACIÓN

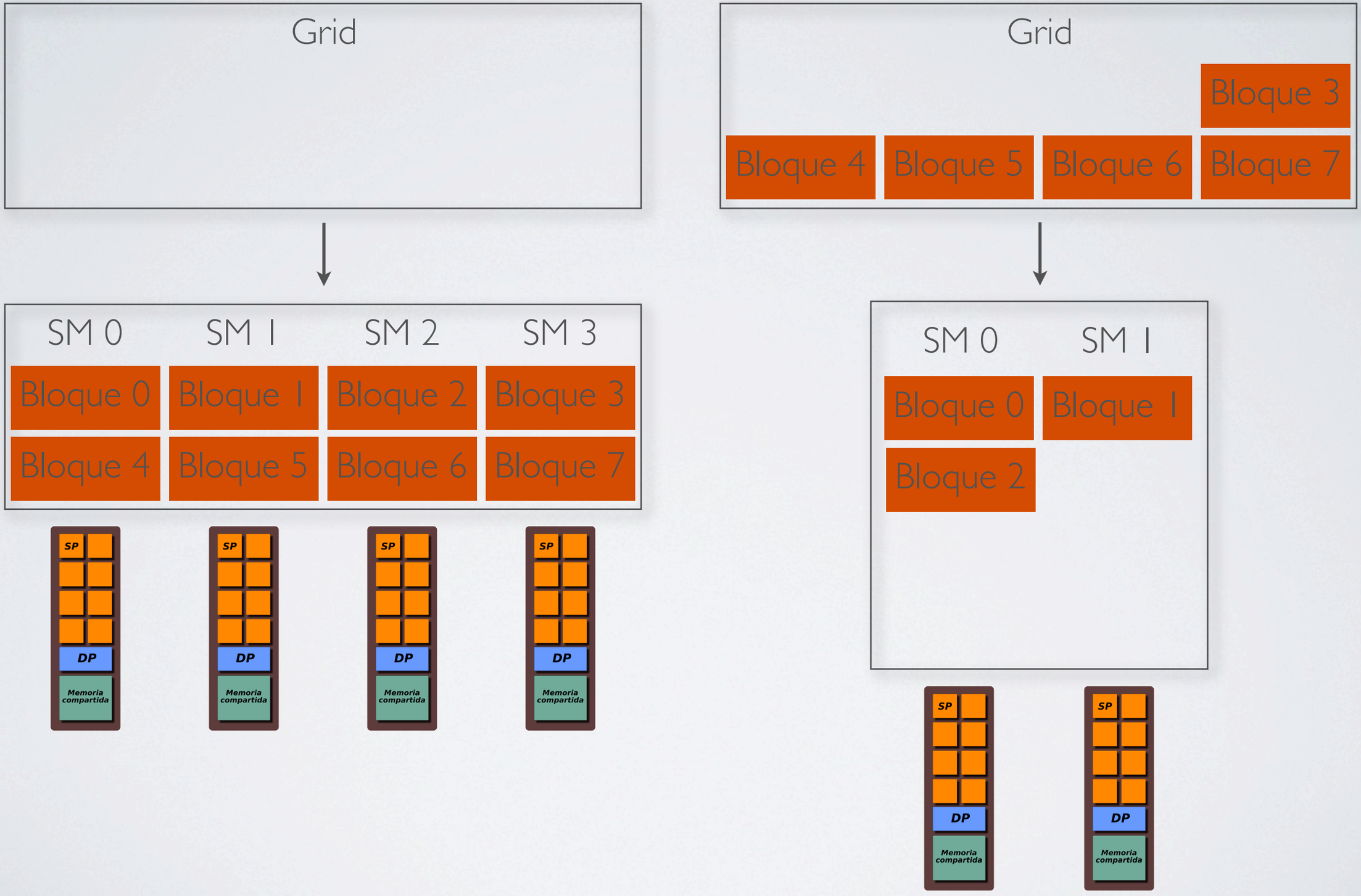




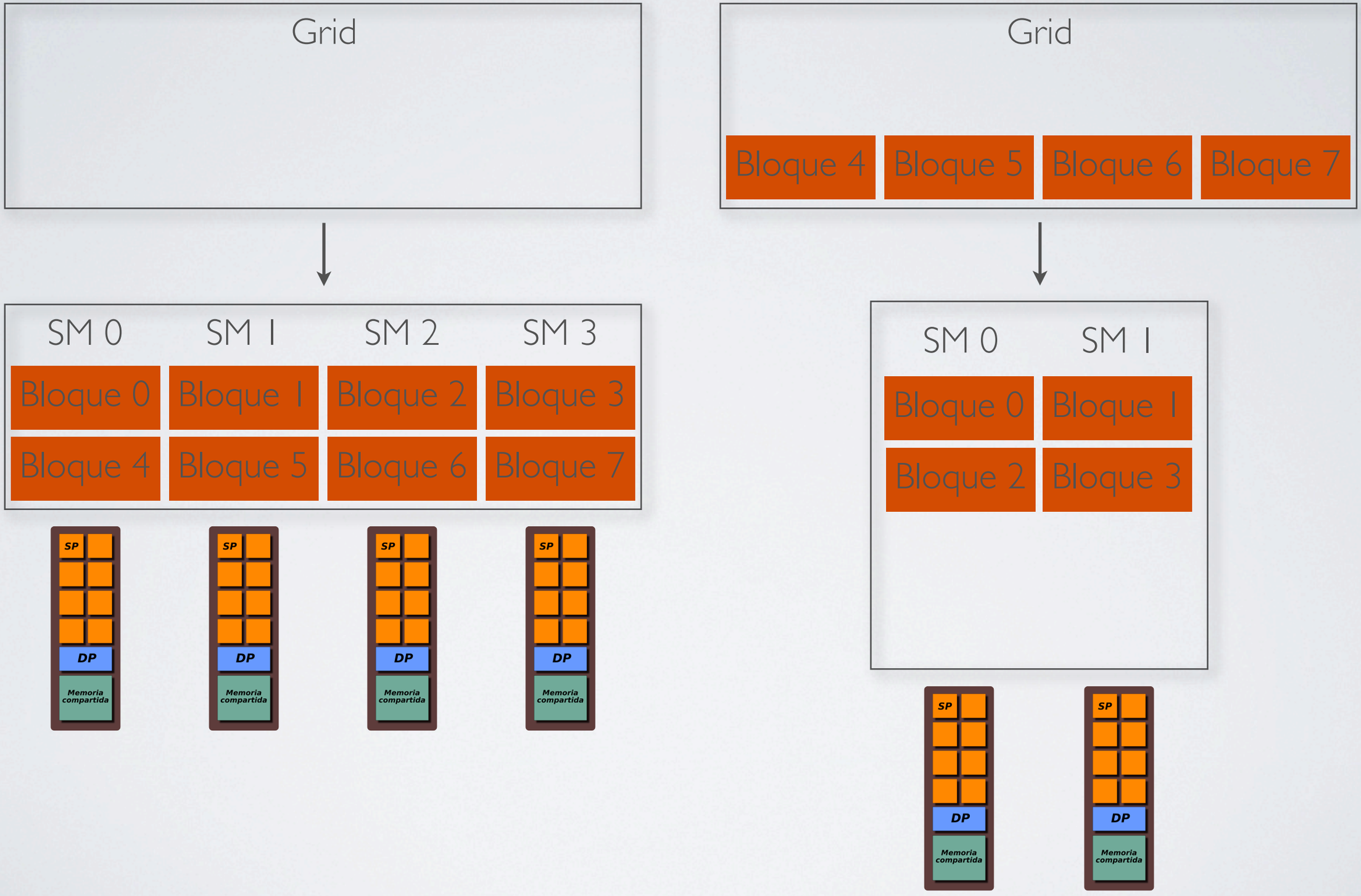
PLANIFICACIÓN



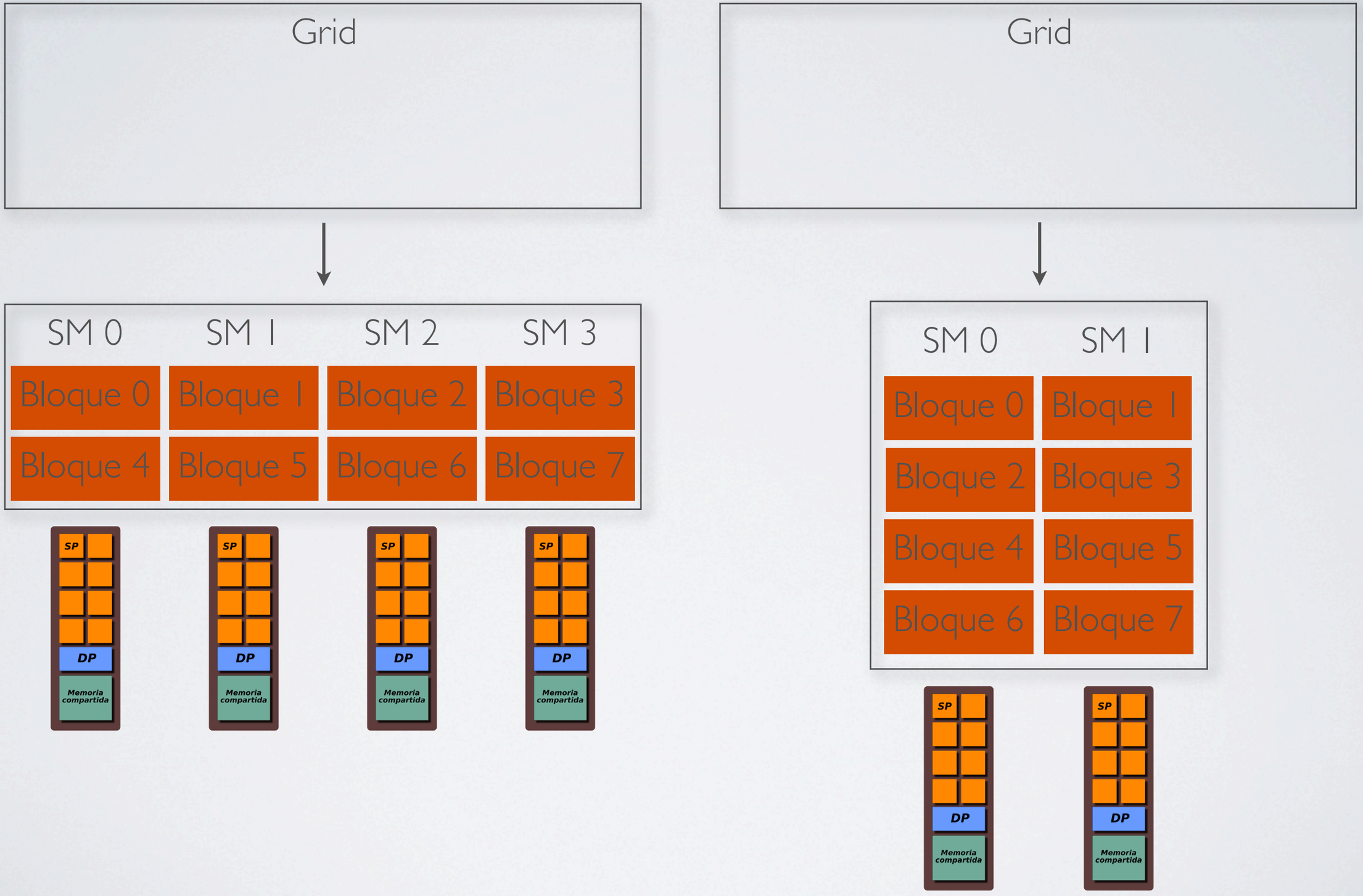
PLANIFICACIÓN



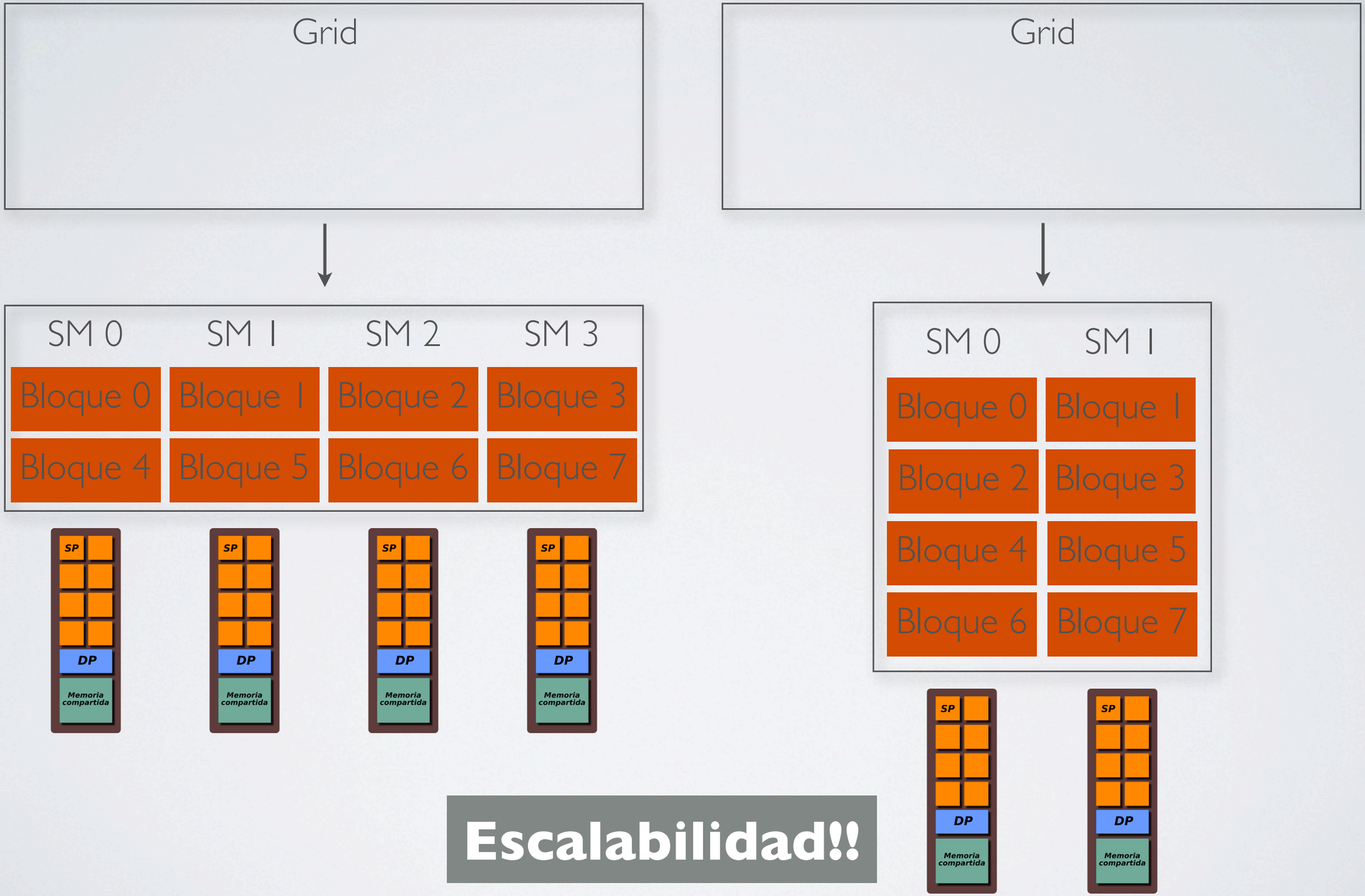
PLANIFICACIÓN



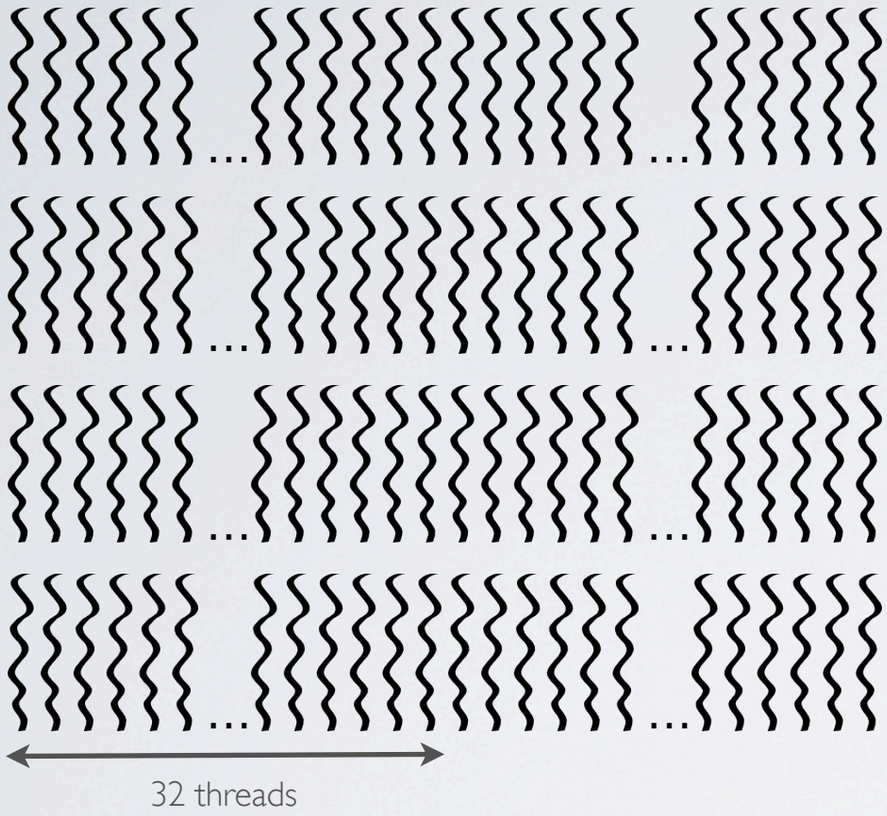
PLANIFICACIÓN

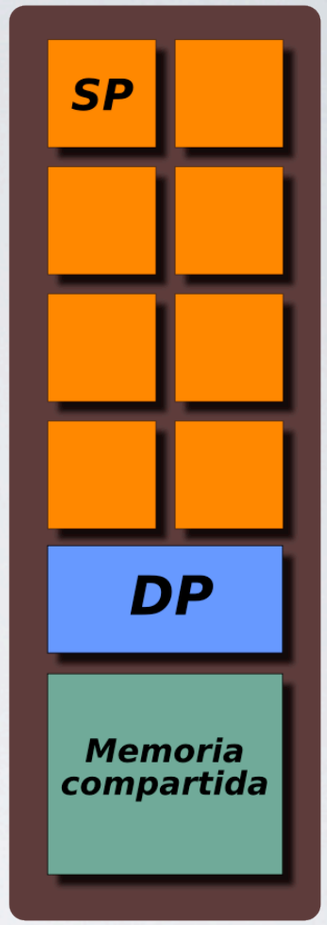


PLANIFICACIÓN



PLANIFICACIÓN

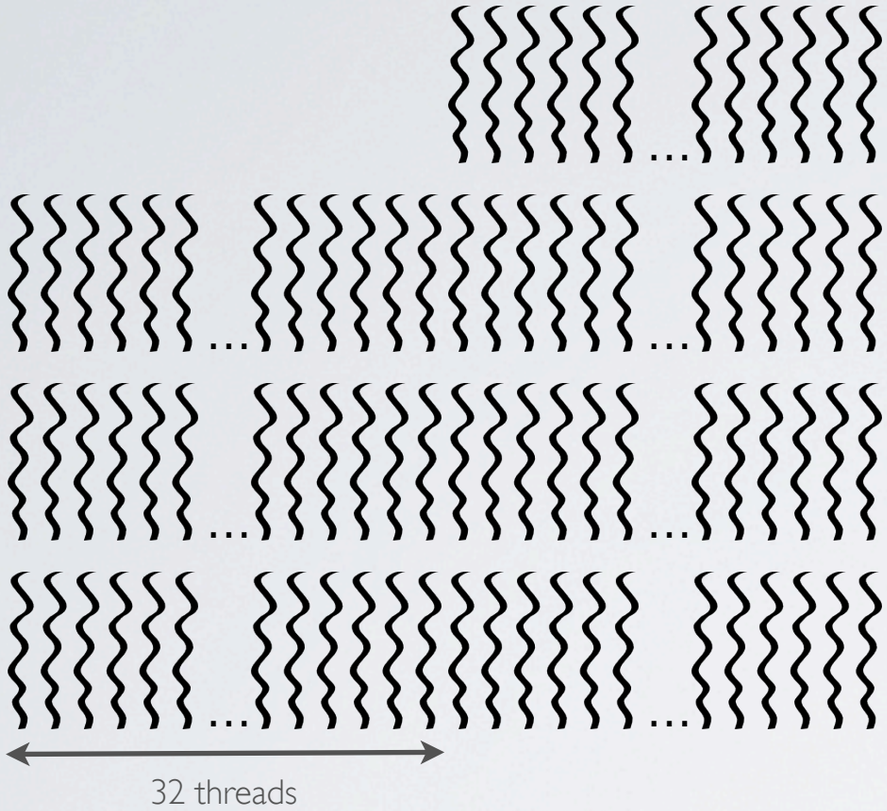




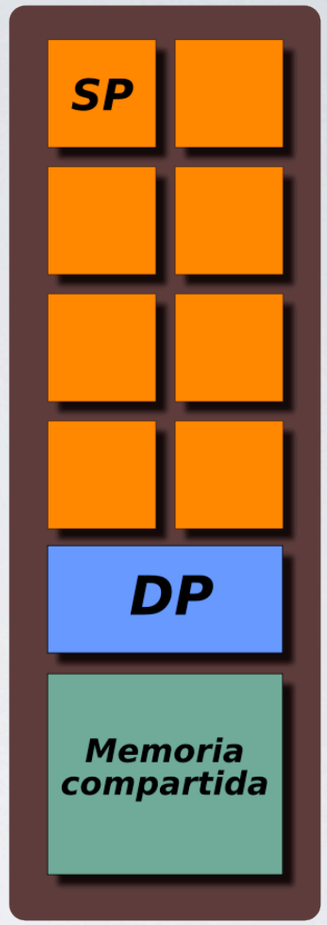
- Unidad de planificación por MP:
  - Crea, planifica y lanza a ejecución threads dentro de un bloque
  - Unidad mínima de planificación: **WARP**

WARP: grupo de 32 threads lanzados a ejecución concurrentemente sobre un MP

PLANIFICACIÓN



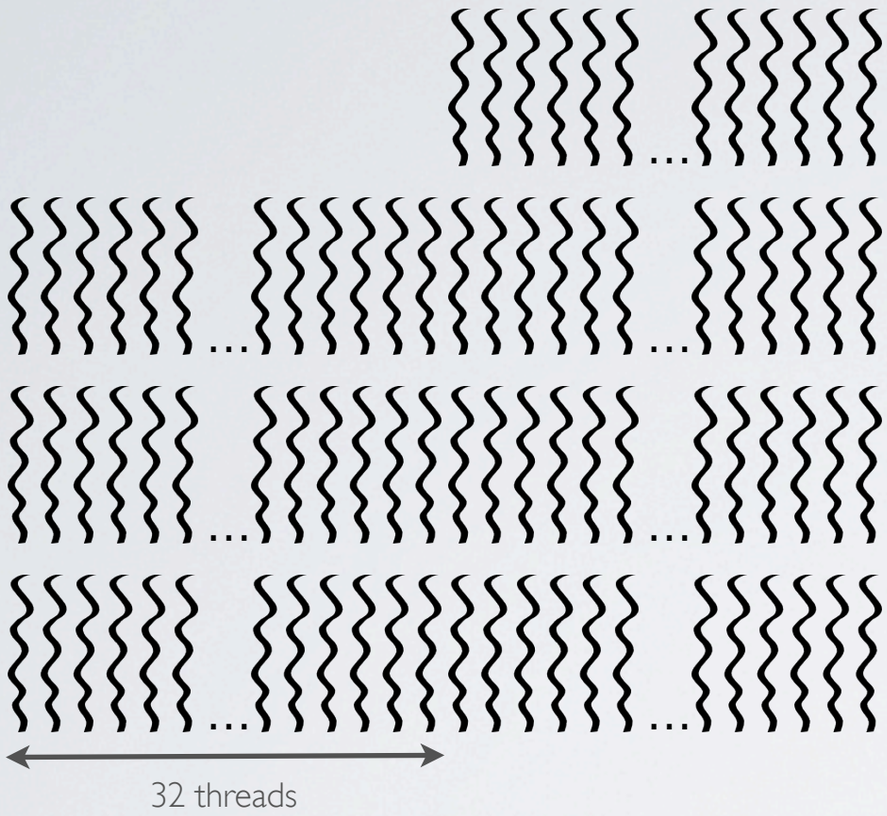


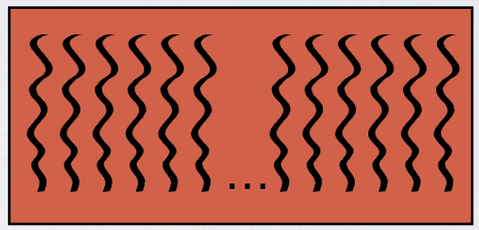


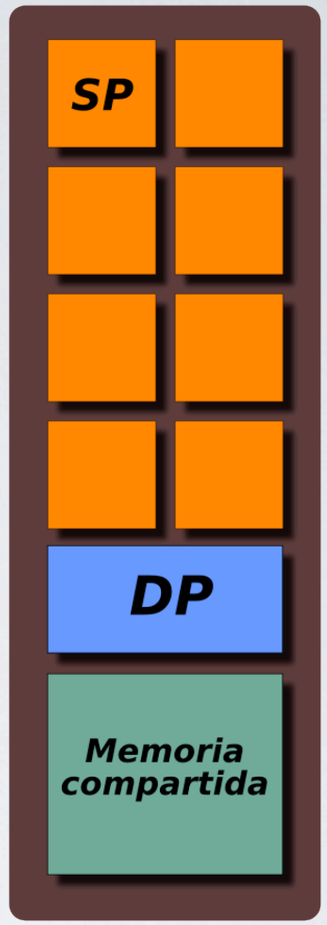
- Unidad de planificación por MP:
  - Crea, planifica y lanza a ejecución threads dentro de un bloque
  - Unidad mínima de planificación: **WARP**

WARP: grupo de 32 threads lanzados a ejecución concurrentemente sobre un MP

PLANIFICACIÓN



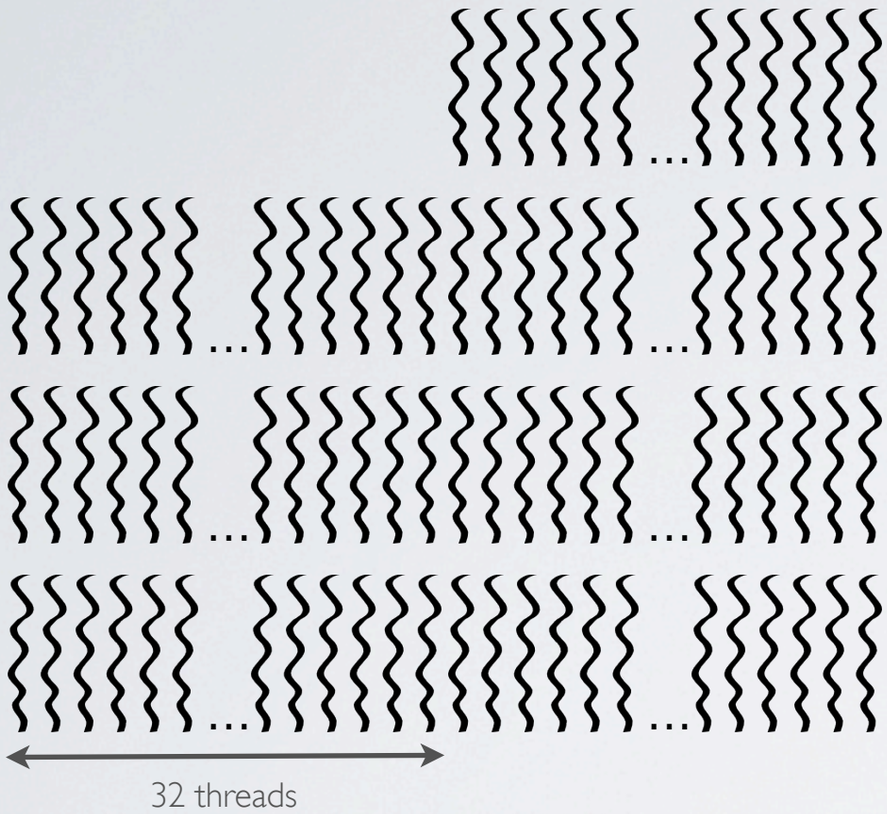


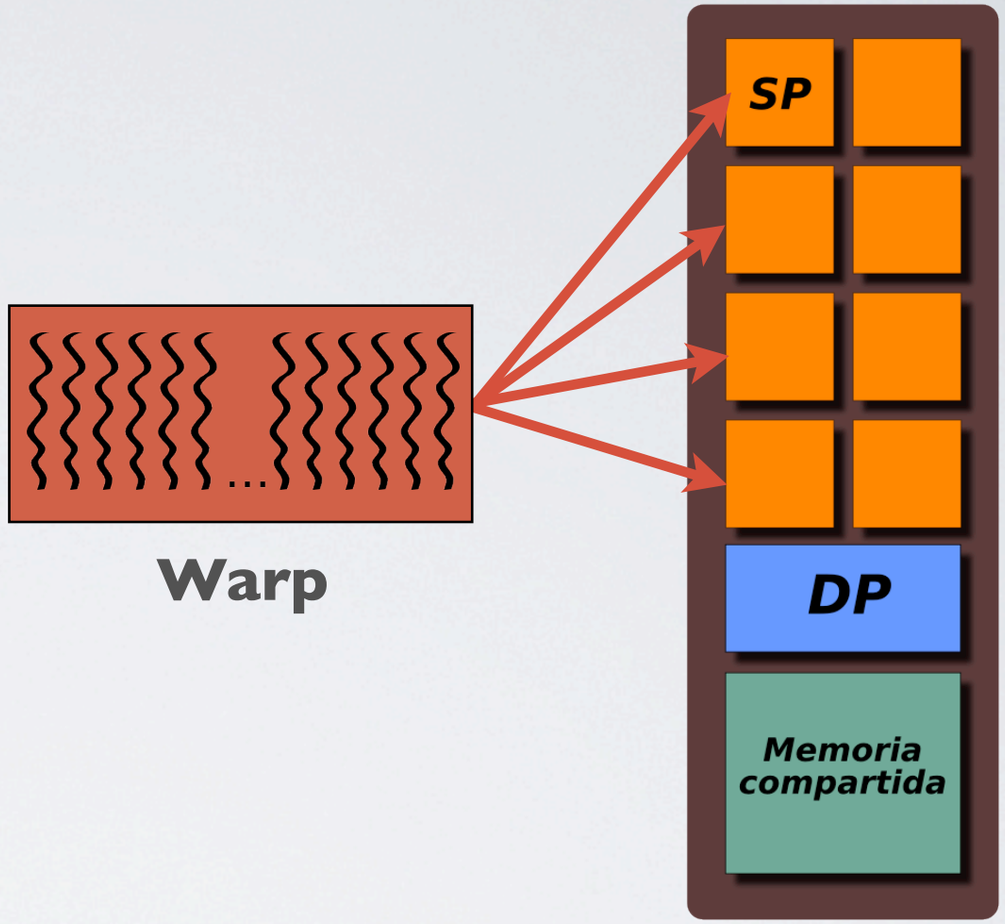


- Unidad de planificación por MP:
  - Crea, planifica y lanza a ejecución threads dentro de un bloque
  - Unidad mínima de planificación: **WARP**

WARP: grupo de 32 threads lanzados a ejecución concurrentemente sobre un MP

PLANIFICACIÓN





- Unidad de planificación por MP:
  - Crea, planifica y lanza a ejecución threads dentro de un bloque
  - Unidad mínima de planificación: **WARP**

WARP: grupo de 32 threads lanzados a ejecución concurrentemente sobre un MP

EJEMPLO BÁSICO

```
1  void suma_cpu( float *A, float b, int N )
2  {
3    for( int idx = 0; idx < N; idx++ )
4      A[ idx ] = A[ idx ] + b
5  }
6
7
8
9  void main( )
10 {
11   ...
12   suma_cpu( A, b, N );
13   ...
14 }
```

Programa en CPU

```
1  __global__ void suma_gpu( float *A, float b, int N )
2  {
3    int idx = blockIdx.x * blockDim.x + threadIdx.x;
4
5    if( idx < N )
6      A[ idx ] = A[ idx ] + b;
7  }
8
9  void main( )
10 {
11   ...
12   dim3 dimBlock( blockSize );
13   dim3 dimGrid( ceil( N / (float)blockSize ) );
14
15   suma_gpu<<<dimGrid, dimBlock>>>( d_A, b, N );
16 }
```

Programa en GPU

EJEMPLO BÁSICO

```
1  void suma_cpu( float *A, float b, int N )
2  {
3    for( int idx = 0; idx < N; idx++ )
4      A[ idx ] = A[ idx ] + b
5  }
6
7
8
9  void main( )
10 {
11   ...
12   suma_cpu( A, b, N );
13   ...
14 }
```

Programa en CPU

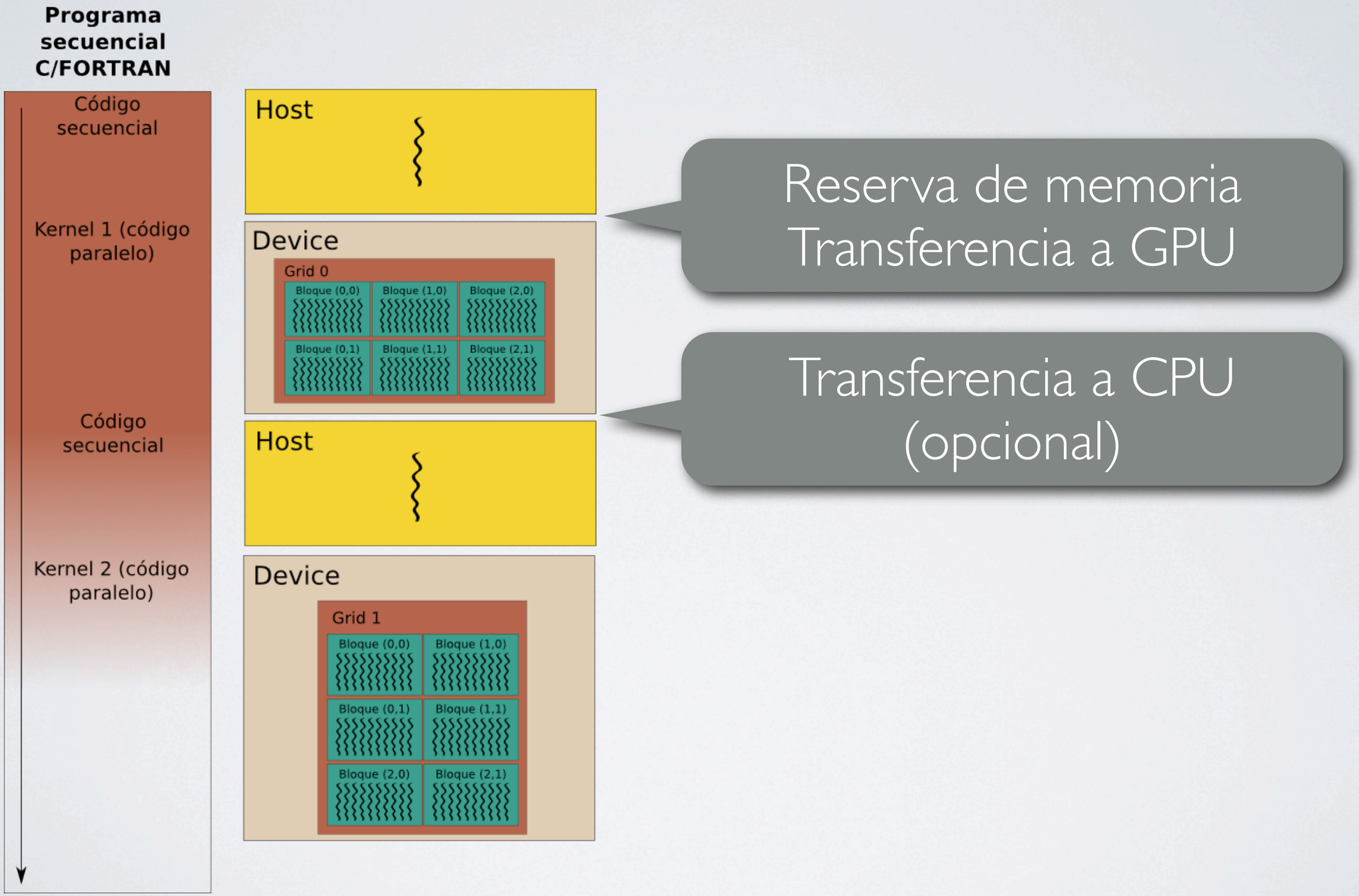
```
1  __global__ void suma_gpu( float *A, float b, int N )
2  {
3    int idx = blockIdx.x * blockDim.x + threadIdx.x;
4
5    if( idx < N )
6      A[ idx ] = A[ idx ] + b;
7  }
8
9  void main( )
10 {
11   ...
12   dim3 dimBlock( blockSize );
13   dim3 dimGrid( ceil( N / (float)blockSize ) );
14
15   suma_gpu<<<dimGrid, dimBlock>>>( d_A, b, N );
16 }
```

Programa en GPU

ÍNDICE DE CONTENIDOS

1. Introducción / Motivación
2. La arquitectura CUDA
3. Espacios de memoria
4. Modelo de ejecución CUDA
5. Programación de kernels
6. Conceptos avanzados

PROGRAMACIÓN HÍBRIDA



PROGRAMACIÓN DE KERNELS

- Código C con algunas restricciones
  - Sólo puede acceder a memoria de la GPU*
  - No está permitido un número variable de argumentos
  - Sin variables estáticas
  - Recursividad no permitida
- Directivas para las funciones de nuestro programa
  - __global__: ejecutado en la GPU, invocado por la CPU
  - __device__: ejecutado en la GPU, invocado por la GPU
  - __host__: ejecutado en la CPU

INVOCACIÓN DE KERNELS

- Configuración de la ejecución
  - **Dimensiones del grid** (2D): 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*
- Ejemplo:

```
...

dim3 gridSize( 16, 16 );

dim3 blockSize( 32, 32 );

kernel<<<gridSize, blockSize, 0, 0>>>( ... );

kernel<<<32, 512>>>( ... );

...
```

EJEMPLO 2

1. A partir de EJEMPLO 1

2. Invocación de kernel para inicializar valores

3. Copia de datos de GPU a CPU

4. Impresión de valores

EJEMPLO 2

```
1  __global__ void kernel( float * A )
2  {
3    a[ blockIdx.x * blockDim.x + threadIdx.x ] = 0.0;
4  }
5
6  int main( void )
7  {
8    int n = 16;
9    int num_bytes = n * sizeof( float );
10
11   float * h_A = NULL;
12   float * d_A = NULL;
13
14   h_A = (float *)malloc( num_bytes );
15   cudaMalloc( (void **)&d_A, num_bytes );
16
17   dim3 blockSize( 4 ); dim3 gridSize( n/block.x );
18   kernel<<<gridSize, blockSize>>>( d_A );
19
20   cudaMemcpy( h_A, d_A, num_bytes, cudaMemcpyDeviceToHost );
21
22   // Mostrar de valores
23   // ...
24
25   free( h_A );
26   cudaFree( d_A );
27 }
```

EJEMPLO 2

```
1  __global__ void kernel( float * A )
2  {
3    a[ blockIdx.x * blockDim.x + threadIdx.x ] = 0.0;
4  }
5
6  int main( void )
7  {
8    int n = 16;
9    int num_bytes = n * sizeof( float );
10
11   float * h_A = NULL;
12   float * d_A = NULL;
13
14   h_A = (float *)malloc( num_bytes );
15   cudaMalloc( (void **)&d_A, num_bytes );
16
17   dim3 blockSize( 4 ); dim3 gridSize( n/block.x );
18   kernel<<<gridSize, blockSize>>>( d_A );
19
20   cudaMemcpy( h_A, d_A, num_bytes, cudaMemcpyDeviceToHost );
21
22   // Mostrar de valores
23   // ...
24
25   free( h_A );
26   cudaFree( d_A );
27 }
```

Configuración de la ejecución:

Número de bloques: 16 / 4 = 4
Threads por bloque: 4

Número total de threads: 16

EJEMPLO 2

```
1  __global__ void kernel( float * A )
2  {
3    a[ blockIdx.x * blockDim.x + threadIdx.x ] = 0.0;
4  }
5
6  int main( void )
7  {
8    int n = 16;
9    int num_bytes = n * sizeof( float );
10
11   float * h_A = NULL;
12   float * d_A = NULL;
13
14   h_A = (float *)malloc( num_bytes );
15   cudaMalloc( (void **)&d_A, num_bytes );
16
17   dim3 blockSize( 4 ); dim3 gridSize( n/block.x );
18   kernel<<<gridSize, blockSize>>>( d_A );
19
20   cudaMemcpy( h_A, d_A, num_bytes, cudaMemcpyDeviceToHost );
21
22   // Mostrar de valores
23   // ...
24
25   free( h_A );
26   cudaFree( d_A );
27 }
```

Configuración de la ejecución:

Número de bloques: 16 / 4 = 4
Threads por bloque: 4

Número total de threads: 16

Invocación del kernel

EJEMPLO 2

```
1  __global__ void kernel( float * A )
2  {
3    a[ blockIdx.x * blockDim.x + threadIdx.x ] = 0.0;
4  }
5
6  int main( void )
7  {
8    int n = 16;
9    int num_bytes = n * sizeof( float );
10
11   float * h_A = NULL;
12   float * d_A = NULL;
13
14   h_A = (float *)malloc( num_bytes );
15   cudaMalloc( (void **)&d_A, num_bytes );
16
17   dim3 blockSize( 4 ); dim3 gridSize( n/block.x );
18   kernel<<<gridSize, blockSize>>>( d_A );
19
20   cudaMemcpy( h_A, d_A, num_bytes, cudaMemcpyDeviceToHost );
21
22   // Mostrar de valores
23   // ...
24
25   free( h_A );
26   cudaFree( d_A );
27 }
```

Configuración de la ejecución:

Número de bloques: 16 / 4 = 4
Threads por bloque: 4

Número total de threads: 16

Invocación del kernel

Transferencia de datos entre GPU y CPU

EJEMPLO

```
__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = 7.0;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = blockIdx.x;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = threadIdx.x;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] =
                    blockIdx.x * blockDim.x + threadIdx.x;
}
```

EJEMPLO

```
__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = 7.0;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = blockIdx.x;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] =
                    blockIdx.x * blockDim.x + threadIdx.x;
}
```

Salida: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

EJEMPLO

```
__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = 7.0;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = blockIdx.x;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] =
                    blockIdx.x * blockDim.x + threadIdx.x;
}
```

Salida: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

Salida: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

EJEMPLO

```
__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = 7.0;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = blockIdx.x;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = threadIdx.x;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] =
                  blockIdx.x * blockDim.x + threadIdx.x;
}
```

Salida: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

Salida: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

Salida: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

EJEMPLO

```
__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = 7.0;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = blockIdx.x;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] = threadIdx.x;
}

__global__ void kernel( float * A ){
  A[ blockIdx.x * blockDim.x + threadIdx.x ] =
                  blockIdx.x * blockDim.x + threadIdx.x;
}
```

Salida: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

Salida: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

Salida: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Salida: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

SINCRONIZACIÓN

- Todas las invocaciones de *kernels* son asíncronas
  - El control vuelve inmediatamente al programa principal (CPU)
  - La ejecución comienza cuando las anteriores llamadas CUDA han finalizado
- cudaThreadSynchronize()
  - Bloqueante, hasta que las anteriores llamadas a CUDA han finalizado
  - Imprescindible en mediciones de tiempo
- Las copias de memoria son síncronas
- Desde CUDA 2.0 existen transferencias asíncronas
  - Posibilidad de solapamiento comunicaciones/ejecución de kernels

MEMORIA COMPARTIDA

- Memoria ***on-chip***
  - Menor latencia - Mayor ancho de banda
  - 16 Kbytes por MP (30 en Nvidia GT200)
- Asociada a un bloque de threads
- Accesible por todos los threads dentro de un mismo bloque
  - NO accesible por otros bloques de threads
- Usos:
  - Comunicación entre *threads*
  - Cache software

MEMORIA COMPARTIDA

Tamaño conocido en tiempo de compilación

```
__global__ void kernel(...)
{
...
__shared__ float sData[256];
...
}

int main( void ){
  ...
  kernel<<<nBlocks,blockSize>>>(...);
  ...
}
```

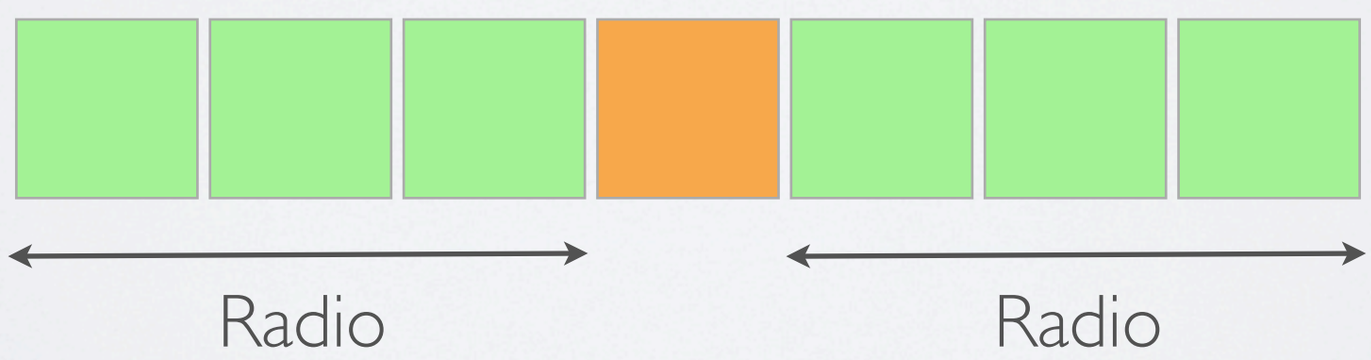
Tamaño conocido en tiempo de ejecución

```
__global__ void kernel(...)
{
...
extern __shared__ float sData[];
...
}

int main( void ){
  ...
  smBytes = blockSize * sizeof(float);
  kernel<<<nBlocks,blockSize, smBytes>>>(...);
  ...
}
```

MEMORIA COMPARTIDA: EJEMPLO

- Stencil 1D
  - Estructura de datos unidimensional
  - Por cada elemento de salida, suma los elementos dentro de un radio
- Ejemplo: Radio = 3
  - Suma de 7 elementos
- Objetivo: reducir accesos a memoria global



MEMORIA COMPARTIDA: EJEMPLO

- Bloques de threads unidimensionaes
  - Particionado de la salida
- Cada bloque de threads computa BLOCK_DIM elementos
  - Lee datos de entrada desde memoria global
    - Necesita BLOCK_DIMX + 2 * RADIO elementos de entrada
  - Transfiere a memoria compartida
  - Computa
  - Escribe a memoria compartida



MEMORIA COMPARTIDA: EJEMPLO

```
1  __global__ void stencil( int *output, int *input,
2                           int dimx, int dimy )
3  {
4    __shared__ int s_a[BLOCK_DIMX+2*RADIUS];
5
6    int global_ix = blockIdx.x*blockDim.x + threadIdx.x;
7    int local_ix  = threadIdx.x + RADIUS;
8
9    s_a[local_ix] = input[global_ix];
10
11   if ( threadIdx.x < RADIUS ) {
12     s_a[local_ix - RADIUS] = input[global_ix - RADIUS];
13     s_a[local_ix + BLOCK_DIMX + RADIUS] =
14           input[global_ix + BLOCK_DIMX + RADIUS];
15   }
16
17   __syncthreads();
18
19   int value = 0;
20   for( offset = -RADIUS; offset<=RADIUS; offset++ )
21     value += s_a[ local_ix + offset ];
22
23   output[global_ix] = value;
24 }
```

MEMORIA COMPARTIDA: EJEMPLO

```
1  __global__ void stencil( int *output, int *input,
2                           int dimx, int dimy )
3  {
4    __shared__ int s_a[BLOCK_DIMX+2*RADIUS];
5
6    int global_ix = blockIdx.x*blockDim.x + threadIdx.x;
7    int local_ix  = threadIdx.x + RADIUS;
8
9    s_a[local_ix] = input[global_ix];
10
11   if ( threadIdx.x < RADIUS ) {
12     s_a[local_ix - RADIUS] = input[global_ix - RADIUS];
13     s_a[local_ix + BLOCK_DIMX + RADIUS] =
14           input[global_ix + BLOCK_DIMX + RADIUS];
15   }
16
17   __syncthreads();
18
19   int value = 0;
20   for( offset = -RADIUS; offset<=RADIUS; offset++ )
21     value += s_a[ local_ix + offset ];
22
23   output[global_ix] = value;
24 }
```



MEMORIA COMPARTIDA: EJEMPLO

```
1  __global__ void stencil( int *output, int *input,
2                           int dimx, int dimy )
3  {
4    __shared__ int s_a[BLOCK_DIMX+2*RADIUS];
5
6    int global_ix = blockIdx.x*blockDim.x + threadIdx.x;
7    int local_ix  = threadIdx.x + RADIUS;
8
9    s_a[local_ix] = input[global_ix];
10
11   if ( threadIdx.x < RADIUS ) {
12     s_a[local_ix - RADIUS] = input[global_ix - RADIUS];
13     s_a[local_ix + BLOCK_DIMX + RADIUS] =
14           input[global_ix + BLOCK_DIMX + RADIUS];
15   }
16
17   __syncthreads();
18
19   int value = 0;
20   for( offset = -RADIUS; offset<=RADIUS; offset++ )
21     value += s_a[ local_ix + offset ];
22
23   output[global_ix] = value;
24 }
```



MEMORIA COMPARTIDA: EJEMPLO

```
1  __global__ void stencil( int *output, int *input,
2                           int dimx, int dimy )
3  {
4    __shared__ int s_a[BLOCK_DIMX+2*RADIUS];
5
6    int global_ix = blockIdx.x*blockDim.x + threadIdx.x;
7    int local_ix  = threadIdx.x + RADIUS;
8
9    s_a[local_ix] = input[global_ix];
10
11   if ( threadIdx.x < RADIUS ) {
12     s_a[local_ix - RADIUS] = input[global_ix - RADIUS];
13     s_a[local_ix + BLOCK_DIMX + RADIUS] =
14           input[global_ix + BLOCK_DIMX + RADIUS];
15   }
16
17   __syncthreads();
18
19   int value = 0;
20   for( offset = -RADIUS; offset<=RADIUS; offset++ )
21     value += s_a[ local_ix + offset ];
22
23   output[global_ix] = value;
24 }
```



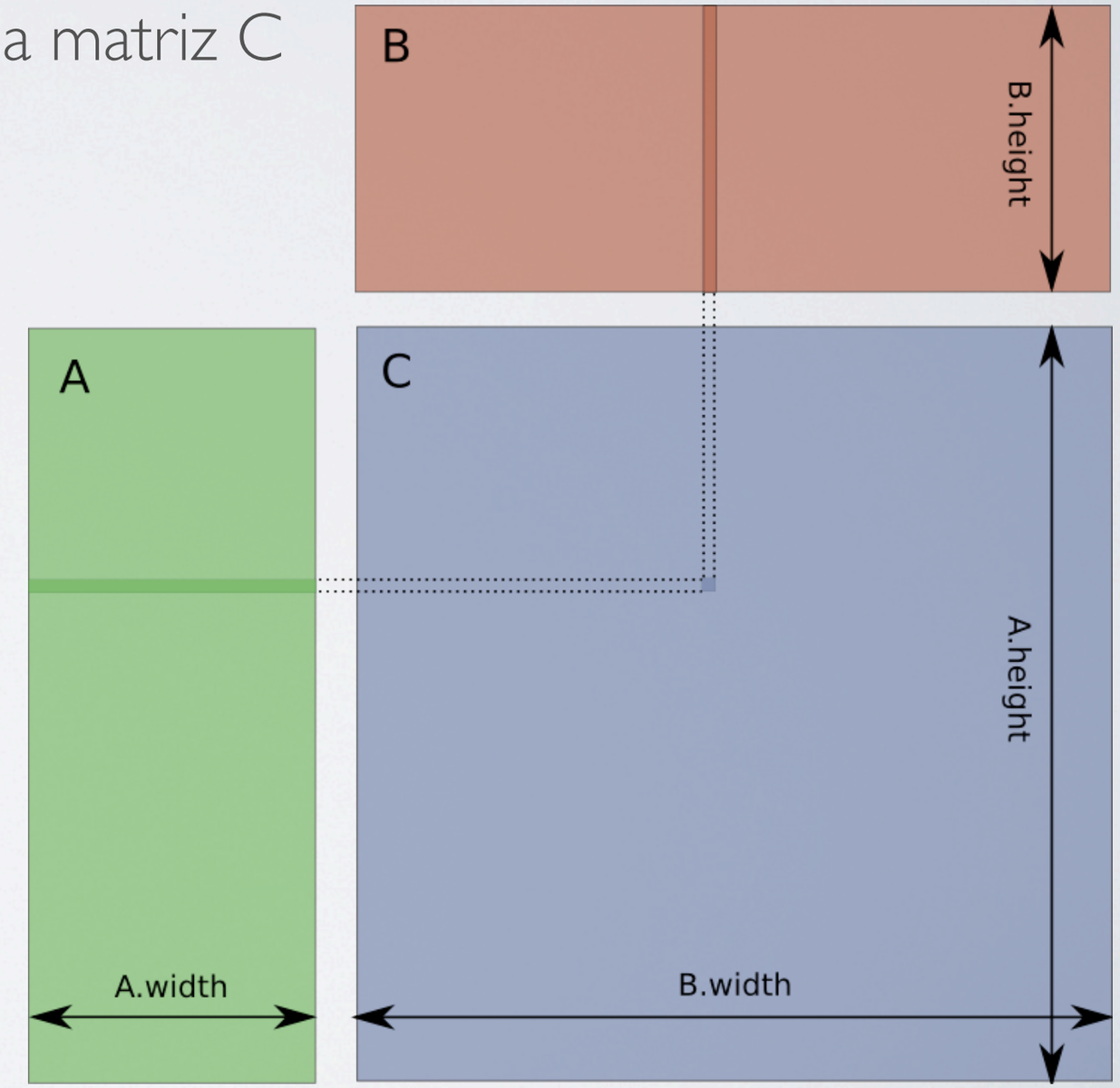
SINCRONIZACIÓN DE THREADS

- void __syncthreads( );
- Sincroniza los threads de un mismo bloque de threads
- Los threads se planifican en tiempo de ejecución
- Evita riesgos RAW/WAR/WAW en acceso a memoria compartida
- Cuando todos los threads llegan a este punto la ejecución continúa
- Precaución al usarlo dentro de condicionales

EJEMPLO: MATMUL

Implementación básica

- Cada thread calcula UN elemento de la matriz C
  - Lee una fila de A
  - Lee una columna de B
- Accesos a memoria global
  - A: B.width lecturas
  - B: A.heigth lecturas



EJEMPLO: MATMUL

```
1  void MatMul(const Matrix A, const Matrix B, Matrix C)
2  {
3    Matrix d_A;
4    d_A.width = A.width; d_A.height = A.height;
5    size_t size = A.width * A.height * sizeof(float);
6
7    // Reserva de memoria para A en GPU
8    cudaMalloc((void**)&d_A.elements, size);
9    cudaMemcpy(d_A.elements, A.elements, size,
10   cudaMemcpyHostToDevice);
11
12   // Reserva de memoria para B y C en GPU
13   // ...
14
15   // Invocacion del kernel
16   dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
17   dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
18   MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
19
20   // Transferencia de C de vuelta a RAM
21   cudaMemcpy(C.elements, Cd.elements, size,
22              cudaMemcpyDeviceToHost);
23
24   // Liberacion de memoria
25   cudaFree(d_A.elements);
26   cudaFree(d_B.elements);
27   cudaFree(d_C.elements);
28 }
```

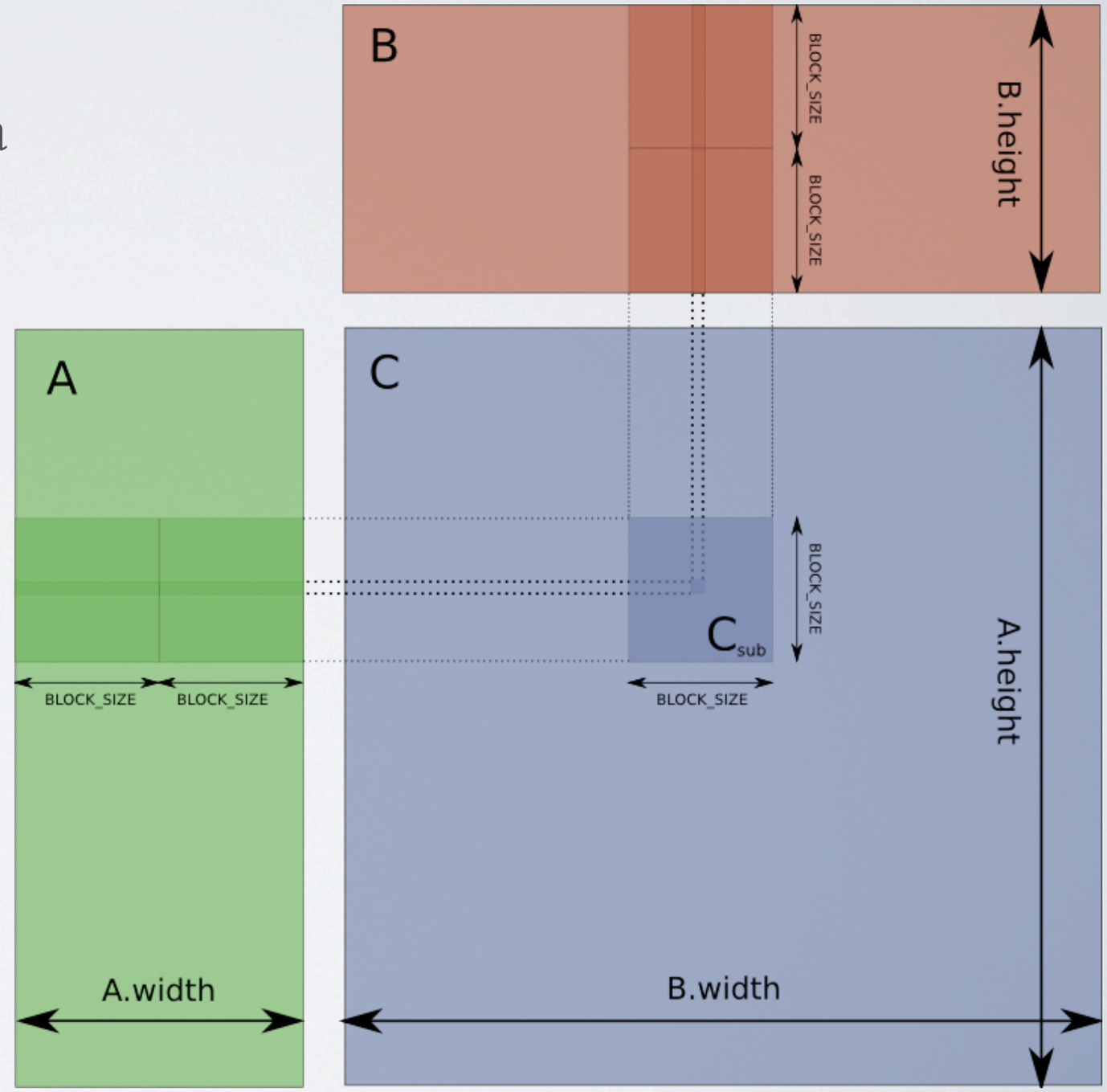
EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
2  {
3    // Cada thread calcula un elemento de C
4    // acumulando el resultado en Cvalue
5    float Cvalue = 0;
6    int row      = blockIdx.y * blockDim.y + threadIdx.y;
7    int col      = blockIdx.x * blockDim.x + threadIdx.x;
8
9    for (int e = 0; e < A.width; ++e)
10     Cvalue += A.elements[row * A.width + e]
11             * B.elements[e * B.width + col];
12
13   C.elements[row * C.width + col] = Cvalue;
14 }
```

EJEMPLO: MATMUL

Implementación optimizada

- Cada bloque de threads calcula una submatriz de C
- Objetivo:
  - Minimizar accesos a mem. global
  - Memoria compartida



EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29 }
```

ID del bloque dentro del grid

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

Submatriz de C a obtener

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

ID del thread dentro del bloque

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

Submatrices de A y B

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

Memoria compartida para Asub y Bsub

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

Carga de A y B en memoria compartida: un elemento por thread

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

Importante:
Sincronización

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

Cálculo de Csub

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

Escritura en memoria global

EJEMPLO: MATMUL

```
1  __global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {
2    int blockRow = blockIdx.y;
3    int blockCol = blockIdx.x;
4
5    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
6
7    int row = threadIdx.y; int col = threadIdx.x;
8
9    // Iteracion sobre submatrices de A y B
10   for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
11     Matrix Asub = GetSubMatrix(A, blockRow, m);
12     Matrix Bsub = GetSubMatrix(B, m, blockCol);
13
14     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
15     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
16
17     As[row][col] = GetElement(Asub, row, col);
18     Bs[row][col] = GetElement(Bsub, row, col);
19
20     __syncthreads();
21
22     // Asub * Bsub
23     for (int e = 0; e < BLOCK_SIZE; ++e)
24       Cvalue += As[row][e] * Bs[e][col];
25       __syncthreads();
26     }
27
28     SetElement(Csub, row, col, Cvalue);
29   }
```

ÍNDICE DE CONTENIDOS

1. Introducción / Motivación

2. La arquitectura CUDA

3. Espacios de memoria

4. Modelo de ejecución CUDA

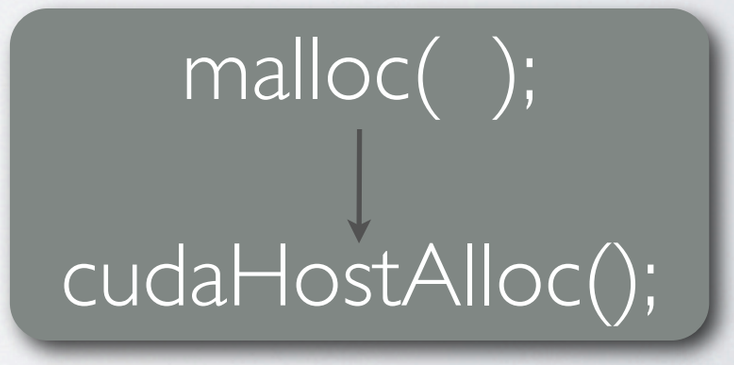
5. Programación de kernels

6. Conceptos avanzados

CONCEPTOS AVANZADOS

Optimización de transferencias

- Agrupar transferencias
- Evitar transferencias pequeñas
- Uso de "*page-locked memory*"
  - Mayor ancho de banda
  - Copias asíncronas (solapamiento cálculo-comunicaciones)
  - Posibilidad de mapear memoria del host en el device
  - Recurso limitado!!



CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones

CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones



CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones



CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones



CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones





CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones

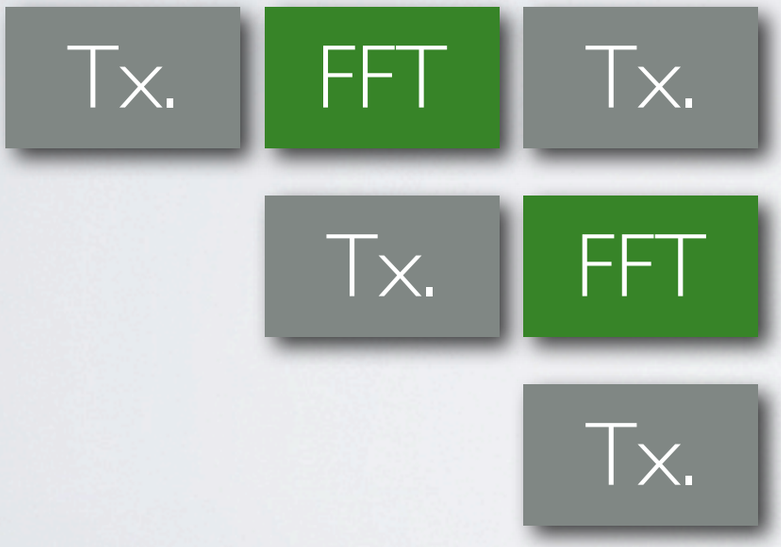




CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones

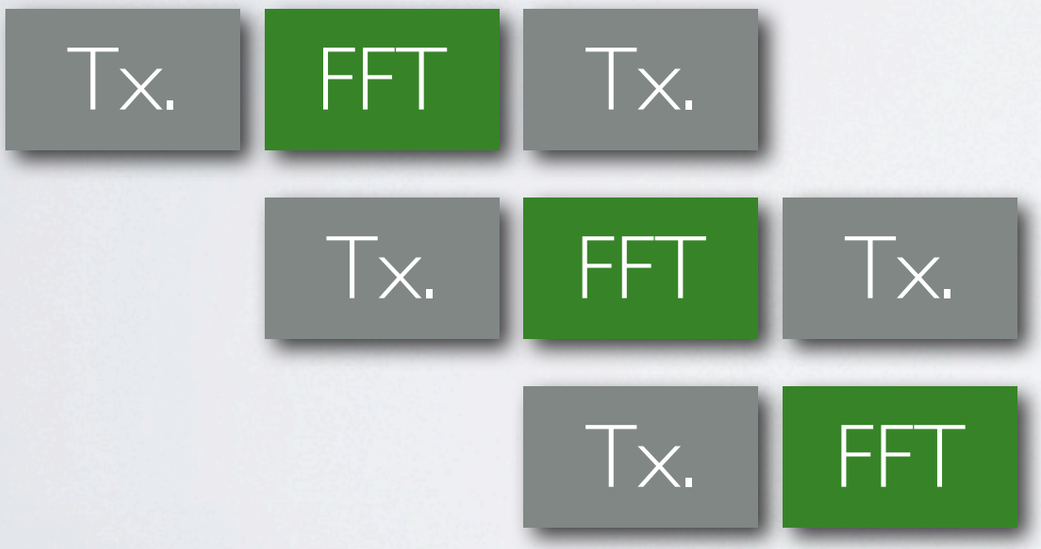




CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones

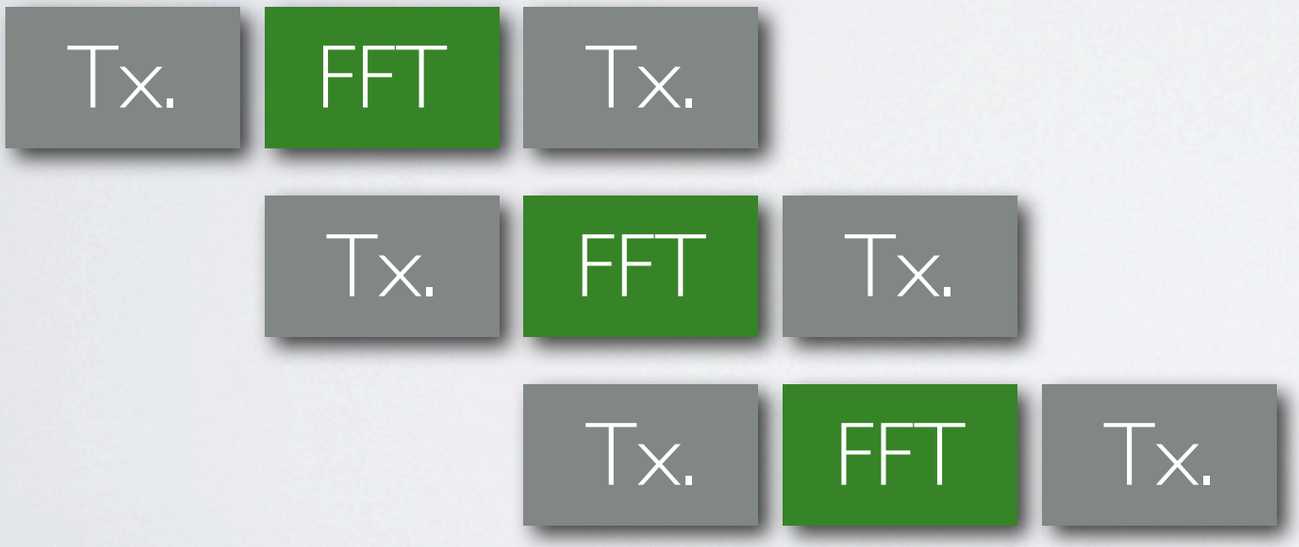




CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones

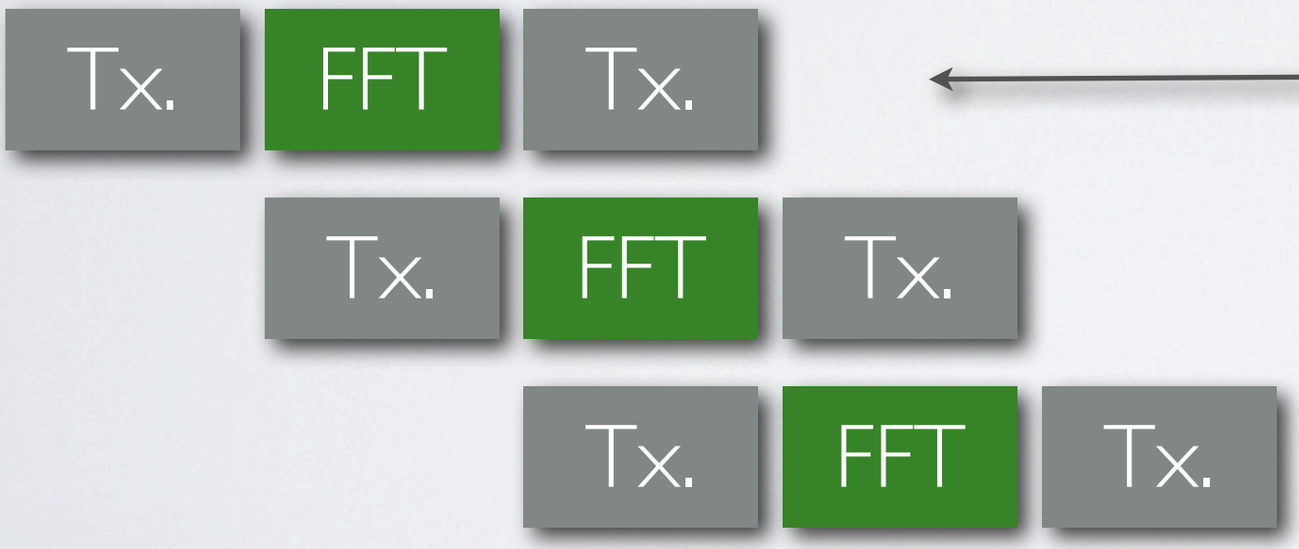




CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones





Stream: secuencia de comandos ejecutados en orden

CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones

```
1  #define NUM_STREAMS 3
2  cudaStream_t stream[ NUM_STREAMS ];
3
4  for( int i=0; i<NUM_STREAMS; i++ )
5    cudaStreamCreate( &stream[ i ] );
6
7  float * hostPtr;
8  cudaMallocHost( (void **)&hostPtr, NUM_STREAMS * size );
9
10 for( int i=0; i<NUM_STREAMS, i++ )
11   cudaMemcpyAsync( inputDevPtr + i * size,
12                    hostPtr + i * size,
13                    size, cudaMemcpyHostToDevice,
14                    stream[ i ] );
15
16 for( int i=0; i<NUM_STREAMS, i++ )
17   fft<<<100, 512, 0, stream[ i ]>>>
18         ( outputDevPtr + i * size,
19           inputDevPtr + i * size,
20           stream[ i ] );
21
22 for( int i=0; i<NUM_STREAMS, i++ )
23   cudaMemcpyAsync( hostPtr + i * size,
24                    outputDevicePtr + i * size,
25                    size, cudadMemcpyDeviceToHost,
26                    stream[ i ] );
27
28 cudaThreadSynchronize();
```

CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones

```
1  #define NUM_STREAMS 3
2  cudaStream_t stream[ NUM_STREAMS ];
3
4  for( int i=0; i<NUM_STREAMS; i++ )
5    cudaStreamCreate( &stream[ i ] );
6
7  float * hostPtr;
8  cudaMallocHost( (void **)&hostPtr, NUM_STREAMS * size );
9
10 for( int i=0; i<NUM_STREAMS, i++ )
11   cudaMemcpyAsync( inputDevPtr + i * size,
12                    hostPtr + i * size,
13                    size, cudaMemcpyHostToDevice,
14                    stream[ i ] );
15
16 for( int i=0; i<NUM_STREAMS, i++ )
17   fft<<<100, 512, 0, stream[ i ]>>>
18         ( outputDevPtr + i * size,
19           inputDevPtr + i * size,
20           stream[ i ] );
21
22 for( int i=0; i<NUM_STREAMS, i++ )
23   cudaMemcpyAsync( hostPtr + i * size,
24                    outputDevicePtr + i * size,
25                    size, cudadMemcpyDeviceToHost,
26                    stream[ i ] );
27
28 cudaThreadSynchronize();
```

Creación de streams

CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones

```
1  #define NUM_STREAMS 3
2  cudaStream_t stream[ NUM_STREAMS ];
3
4  for( int i=0; i<NUM_STREAMS; i++ )
5    cudaStreamCreate( &stream[ i ] );
6
7  float * hostPtr;
8  cudaMallocHost( (void **)&hostPtr, NUM_STREAMS * size );
9
10 for( int i=0; i<NUM_STREAMS, i++ )
11   cudaMemcpyAsync( inputDevPtr + i * size,
12                    hostPtr + i * size,
13                    size, cudaMemcpyHostToDevice,
14                    stream[ i ] );
15
16 for( int i=0; i<NUM_STREAMS, i++ )
17   fft<<<100, 512, 0, stream[ i ]>>>
18         ( outputDevPtr + i * size,
19           inputDevPtr + i * size,
20           stream[ i ] );
21
22 for( int i=0; i<NUM_STREAMS, i++ )
23   cudaMemcpyAsync( hostPtr + i * size,
24                    outputDevicePtr + i * size,
25                    size, cudadMemcpyDeviceToHost,
26                    stream[ i ] );
27
28 cudaThreadSynchronize();
```

Creación de streams

Page-locked memory

CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones

```
1  #define NUM_STREAMS 3
2  cudaStream_t stream[ NUM_STREAMS ];
3
4  for( int i=0; i<NUM_STREAMS; i++ )
5    cudaStreamCreate( &stream[ i ] );
6
7  float * hostPtr;
8  cudaMallocHost( (void **)&hostPtr, NUM_STREAMS * size );
9
10 for( int i=0; i<NUM_STREAMS, i++ )
11   cudaMemcpyAsync( inputDevPtr + i * size,
12                    hostPtr + i * size,
13                    size, cudaMemcpyHostToDevice,
14                    stream[ i ] );
15
16 for( int i=0; i<NUM_STREAMS, i++ )
17   fft<<<100, 512, 0, stream[ i ]>>>
18         ( outputDevPtr + i * size,
19           inputDevPtr + i * size,
20           stream[ i ] );
21
22 for( int i=0; i<NUM_STREAMS, i++ )
23   cudaMemcpyAsync( hostPtr + i * size,
24                    outputDevicePtr + i * size,
25                    size, cudadMemcpyDeviceToHost,
26                    stream[ i ] );
27
28 cudaThreadSynchronize();
```

Creación de streams

Page-locked memory

Cada *stream* lleva asociado:

1. Transferencia asíncrona
2. Ejecución
3. Transferencia asíncrona

CONCEPTOS AVANZADOS

Solapamiento de cálculo y comunicaciones

```
1  #define NUM_STREAMS 3
2  cudaStream_t stream[ NUM_STREAMS ];
3
4  for( int i=0; i<NUM_STREAMS; i++ )
5    cudaStreamCreate( &stream[ i ] );
6
7  float * hostPtr;
8  cudaMallocHost( (void **)&hostPtr, NUM_STREAMS * size );
9
10 for( int i=0; i<NUM_STREAMS, i++ )
11   cudaMemcpyAsync( inputDevPtr + i * size,
12                    hostPtr + i * size,
13                    size, cudaMemcpyHostToDevice,
14                    stream[ i ] );
15
16 for( int i=0; i<NUM_STREAMS, i++ )
17   fft<<<100, 512, 0, stream[ i ]>>>
18         ( outputDevPtr + i * size,
19           inputDevPtr + i * size,
20           stream[ i ] );
21
22 for( int i=0; i<NUM_STREAMS, i++ )
23   cudaMemcpyAsync( hostPtr + i * size,
24                    outputDevicePtr + i * size,
25                    size, cudadMemcpyDeviceToHost,
26                    stream[ i ] );
27
28 cudaThreadSynchronize();
```

Creación de streams

Page-locked memory

Cada *stream* lleva asociado:

1. Transferencia asíncrona
2. Ejecución
3. Transferencia asíncrona

Sincronización

CONCEPTOS AVANZADOS

Compilación. El modo emulación

- **nvcc source.cu**
- nvcc --device-emu source.cu
  - Define la macro __DEVICE_EMULATION__
- Ventajas:
  - Uso de herramientas de depuración (gdb)
  - Invocaciones a printf(...) dentro del kernel
  - Un sólo espacio de direcciones
  - Detección de deadlocks

CONCEPTOS AVANZADOS

Bibliotecas

- CUDA incluye:
  - CUBLAS: implementación BLAS
  - CUFFT: implementación FFT
- CUDPP (Data Parallel Primitives):
  - Funciones de ordenación, reducción, ...

CONCEPTOS AVANZADOS

Bibliotecas. CUFFT

- Características
  - Transformaciones 1D, 2D y 3D sobre datos reales y complejos
  - "*Batched execution*" para múltiples transformaciones en paralelo
  - Hasta 8M elementos para 1D, [2-16384] para 2D y 3D
  - Transformaciones *in-place* y *out-of-place*

CONCEPTOS AVANZADOS

Bibliotecas. CUFFT

```
1  #define NX 256
2  #define NY 128
3
4  cufftHandle plan;
5  cufftComplex *idata, *odata;
6
7  cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
8  cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);
9  ...
10 /* Creacion de plan 2D FFT. */
11 cufftPlan2d(&plan, NX,NY, CUFFT_C2C);
12
13 /* Transformacion OUT-OF-PLACE. */
14 cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);
15
16 /* Transformacion inversa in-place. */
17 cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);
18
19 /* Destruccion del plan. */
20 cufftDestroy(plan);
21
22 cudaFree(idata), cudaFree(odata);
```

Punteros de entrada y salida distintos implican transformación out-of-place

OPTIMIZACIÓN DE RENDIMIENTO

- Ejecución de un *thread* en un MP:

1. Lectura de operandos para cada *thread*
2. Ejecución
3. Escritura de resultados para cada *thread*

- Mecanismos de optimización:

1. Optimización del ancho de banda para cada tipo de memoria
2. Permitir al planificador solapar transacciones de memoria con computación:
   - Gran cantidad de *threads* activos por multiprocesador
   - Programas con elevada intensidad aritmética (número de cálculos por acceso a memoria)

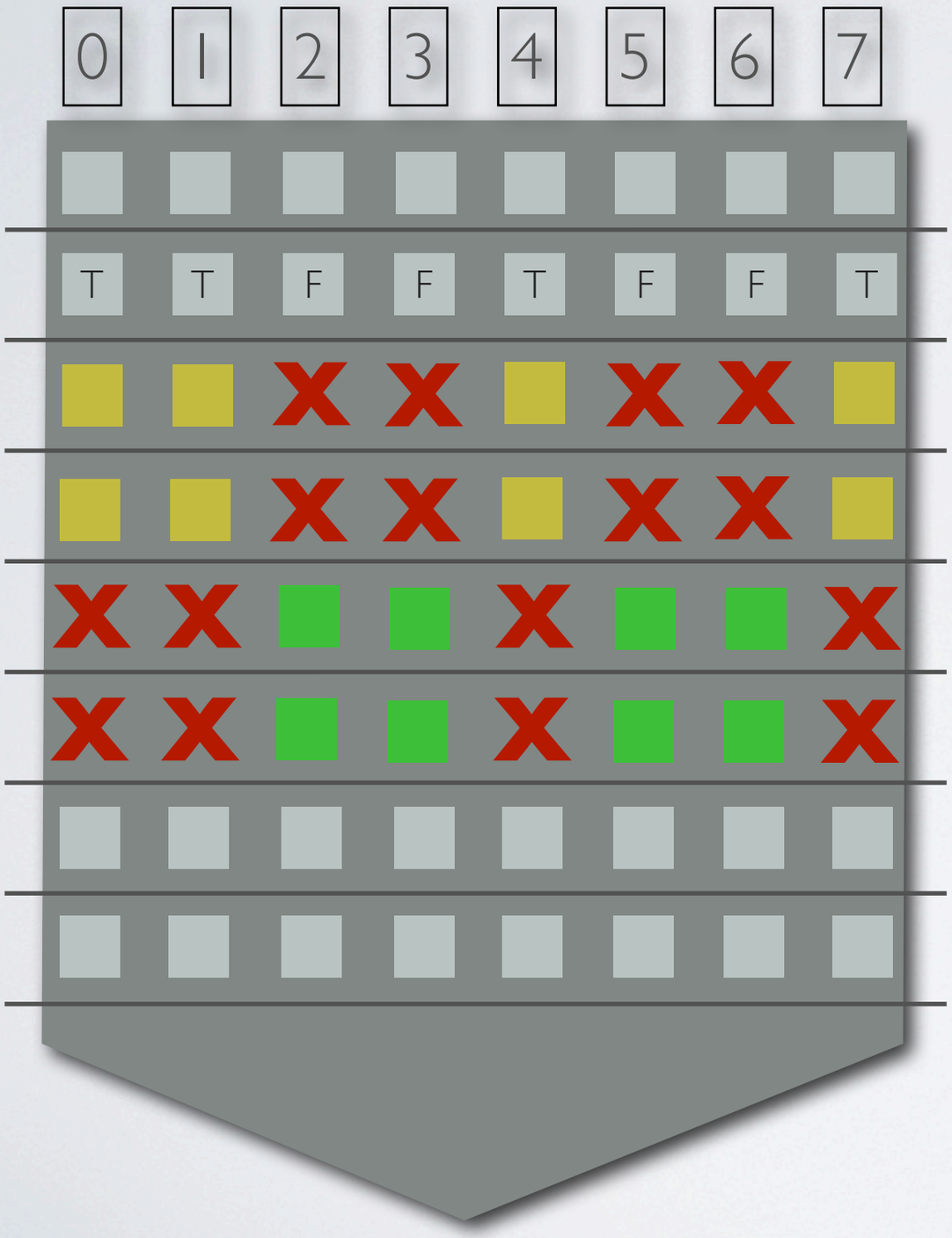
OPTIMIZACIÓN DE RENDIMIENTO

- Optimizar los algoritmos para adaptarlos a la arquitectura
  - Maximizar el paralelismo de datos
  - Maximizar la intensidad aritmética de los kernels (cálculo/accesos a memoria)
  - En ocasiones, mejor recomputar que almacenar
  - En ocasiones, mejor llevar cálculos "anti-GPU" a la GPU antes que aumentar transferencias

OPTIMIZACIÓN DE RENDIMIENTO

- Conocer el modelo de ejecución: usarlo eficientemente
  - Particionado del problema para mantener los MPs ocupados
  - *Kernels* ligeros: los recursos son limitados
  - Evitar saltos y condicionales

SALTOS Y CONDICIONALES




```
if( cond ){
   op1;
   op2;
}
else{
   op3;
   op4;}
```

Evitar condicionales

OPTIMIZACIÓN DE RENDIMIENTO

- Optimizar accesos a memoria
  - Coalesced vs. No-Coalesced = Orden de magnitud
- Optimizar localidad espacial y usar memoria de texturas
- En memoria compartida, evitar conflictos

OPTIMIZACIÓN DE RENDIMIENTO

- USAR MEMORIA COMPARTIDA
  - Cientos de veces más rápida que memoria global
  - Vía de comunicación entre threads
  - Recurso escaso (16 Kbytes por MP)

OPTIMIZACIÓN DE RENDIMIENTO

Memoria global

- Mayor latencia / menor ancho de banda que memoria compartida
- Evitar usarla en la medida de lo posible
- Programa básico por cada thread de un bloque:

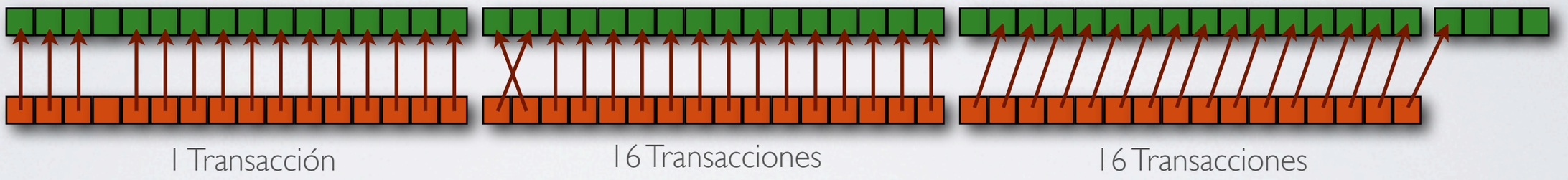
1. Carga de datos desde memoria global a memoria compartida
2. Sincronización de threads
3. Procesamiento usando sólo memoria compartida
4. Sincronización (sin necesario)
5. Escritura en memoria global

OPTIMIZACIÓN DE RENDIMIENTO

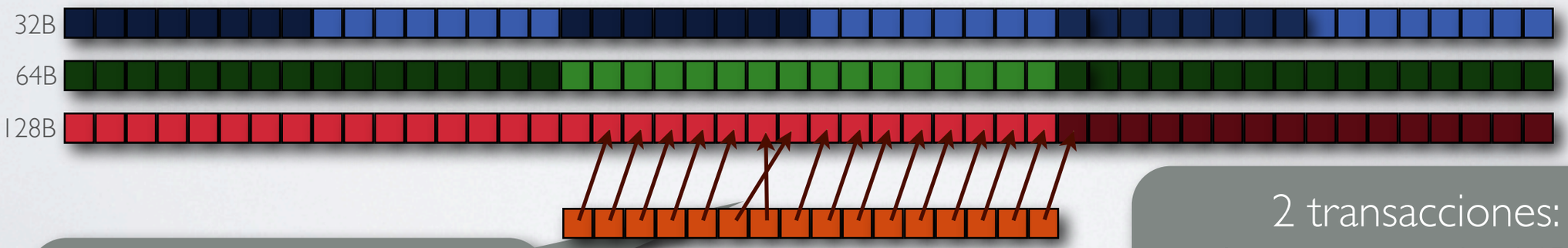
Memoria global

- Accesos a memoria global por *half-warp* (16 *threads*)
- *Coalescing*: grupo de transferencias realizadas en una sola transacción

Coalescing en GPUS 1.1 e inferiores



Coalescing en GPUS 1.2 y superiores



OPTIMIZACIÓN DE RENDIMIENTO

Memoria compartida

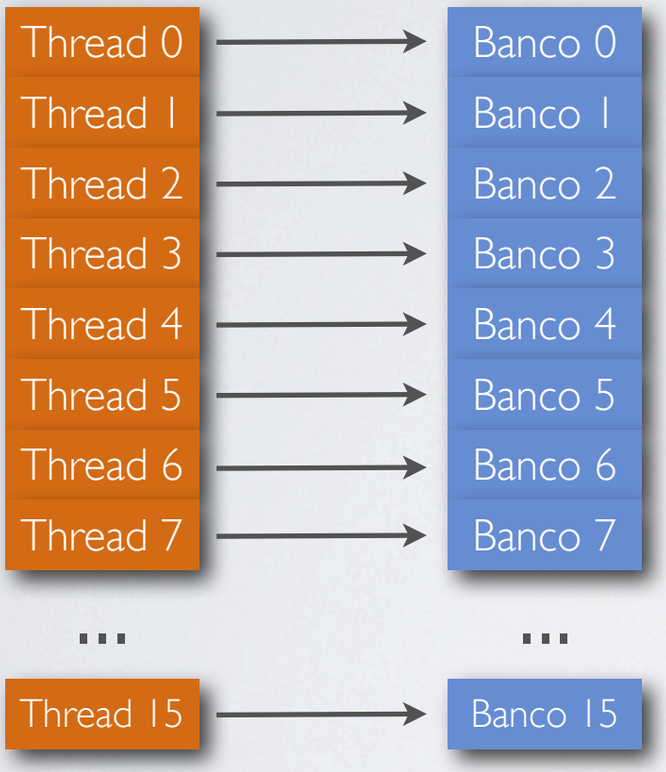
- Tan rápida como acceder a registros (si no hay conflictos)
- Dividida en módulos (bancos). Posibilidad de acceso paralelo a los bancos
- Objetivo: evitar conflictos
  - Acceso simultáneo a un mismo banco por dos threads
- Sucesivas palabras de 32 bits ⟶ sucesivos bancos de memoria
- 16 bancos de memoria en GPUs actuales

```
__shared__ char shared[ 32 ];
char data = shared[ baseIndex + tid ];
```

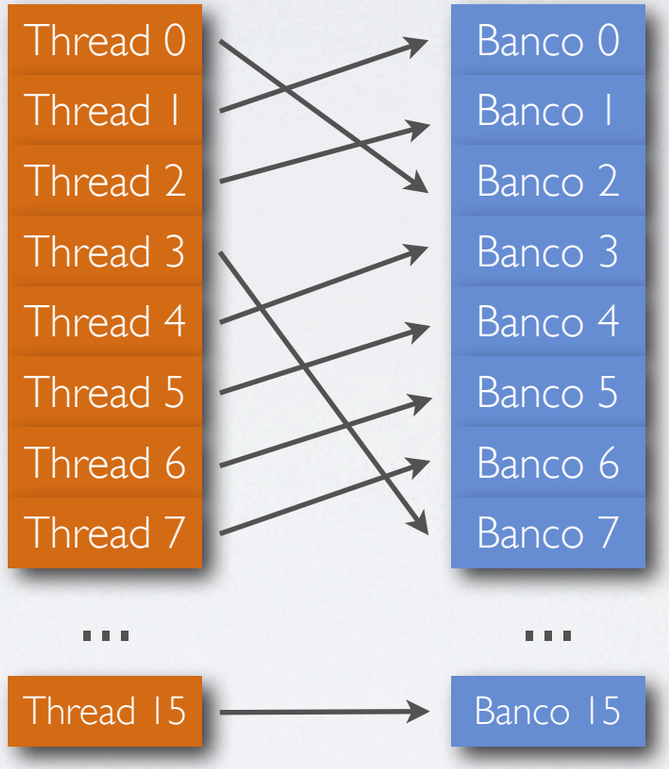
```
__shared__ char shared[ 32 ];
char data = shared[ baseIndex + 4 * tid ];
```

OPTIMIZACIÓN DE RENDIMIENTO

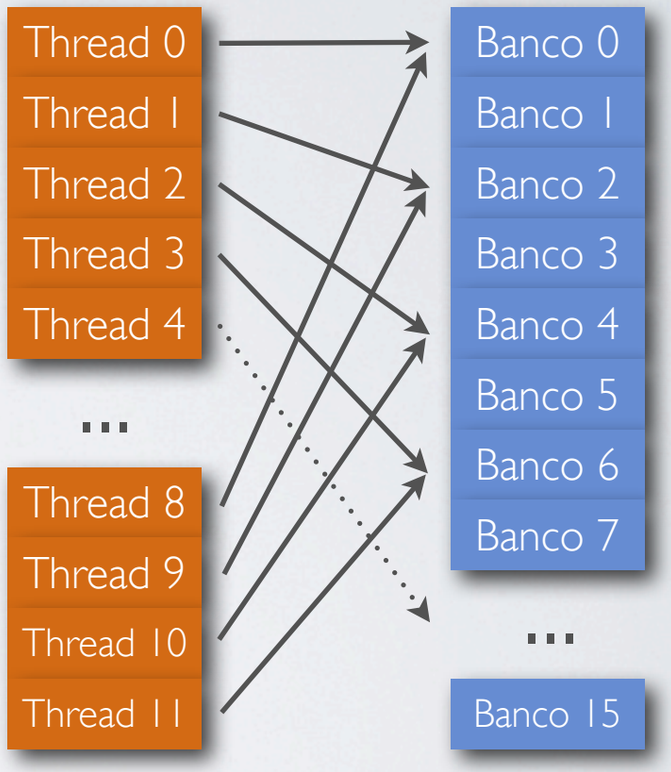
Sin conflictos
Direccionamiento lineal



Sin conflictos
Direccionamiento aleatorio



Con conflictos
Direccionamiento lineal
stride = 2



CONCEPTOS AVANZADOS

Otras memorias

- Memoria de texturas
  - Cache de texturas: optimizada para localidad espacial 2D
  - Ventajas:
    - Si no es posible *coalescing* sobre memoria global
    - Mayor ancho de banda si existe localidad espacial
  - Funciones especiales para acceso a datos (tex2D(...), ...)
- Memoria de constantes
  - Cache de constantes

OPTIMIZACIÓN DE RENDIMIENTO

CONFIGURACIÓN DE LA EJECUCIÓN

Número de threads y bloques

- **¡¡¡Experimentación!!!**
  Pequeñas variaciones pueden ser clave

- La ejecución fallará si:
  - Superamos el máximo número de *threads* por bloque de la arquitectura
  - Nuestro *kernel* requiere más recursos de los disponibles por MP:
    - Recurso: registro y cantidad de memoria compartida

OPTIMIZACIÓN DE RENDIMIENTO

CONFIGURACIÓN DE LA EJECUCIÓN

Número de threads y bloques

- **Número de bloques**
  - Al menos tantos como MPs
  - Si creamos uno por MP:
    - Ciclos perdidos en sincronizaciones y accesos a memoria
  - Dos o más por MP:
    - Solapamiento entre bloques en espera y bloques en ejecución
  - Suficientemente alto para escalar en futuras arquitecturas

OPTIMIZACIÓN DE RENDIMIENTO

CONFIGURACIÓN DE LA EJECUCIÓN

Número de *threads* y bloques

- **Recomendaciones (*CUDA Programming Guide*)**
  - Mínimo número de bloques: 2 x (#MPs)
  - *O(100) bloques*: escalabilidad a futuros dispositivos
  - *O(1000) bloques*: escalabilidad a futuras generaciones
  - Número de threads por bloque:
    - Múltiplo del tamaño de *warp:* evitar *warps* con menos de (32) threads
    - Aconsejable *múltiplos de 64 threads por bloque*
      - Planificador optimizado para evitar conflictos en acceso a registros
    - Tamaños habituales: 192 - 256 threads por bloque

HERRAMIENTAS SOFTWARE

- CUDA Toolkit (versión 2.3, 3.0 Beta ya disponible)
  - Implementaciones para Linux, MacOS y Windows (7)
- CUDA SDK: *ejemplos*, scripts de compilación, utilidades, ...
- CUDA Profiler:
  - Tiempos de ejecución, número de accessos coalesced, conflictos en acceso a memoria compartida, ...
- Nvidia Nexus

DOCUMENTACIÓN

- CUDA Programming Guide
- CUDA Best Practices Guide
- CUDA Zone (http://www.nvidia.com/cuda)
- CUDA SDK (proporcionado con el paquete CUDA)
- GPGPU.org (http://www.gpgpu.org)

GRACIAS POR VUESTRA ATENCIÓN

¿¿...PREGUNTAS...??

figual@icc.uji.es

http://www3.uji.es/~figual
http://www.hpca.uji.es