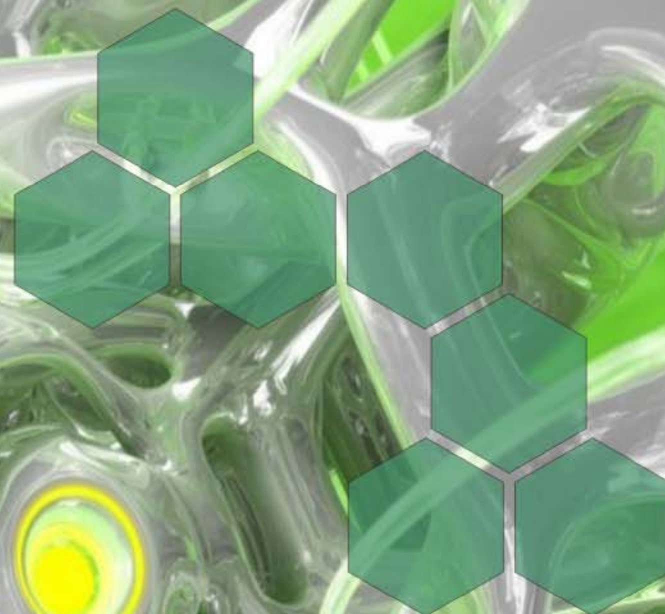
PROCESADORES GRÁFICOS

08 ARQUITECTURA UNIFICADA EN DETALLE: CASO DE ESTUDIO GeForce 8800GTX



Máster Oficial en Informática Gráfica, Juegos y Realidad Virtual

Escuela Técnica Superior de Ingeniería Infomática

CGSociety.org

Índice

- Repaso del modelo de programación
- Siglas
- Arquitectura del G80
- Gestión y ejecución de hebras
- Buffer de instrucciones
- Hardware de la jerarquía de memoria
- Conflictos entre bancos de memoria compartida

"The details are not the details. They make the design."
—Charles Eames

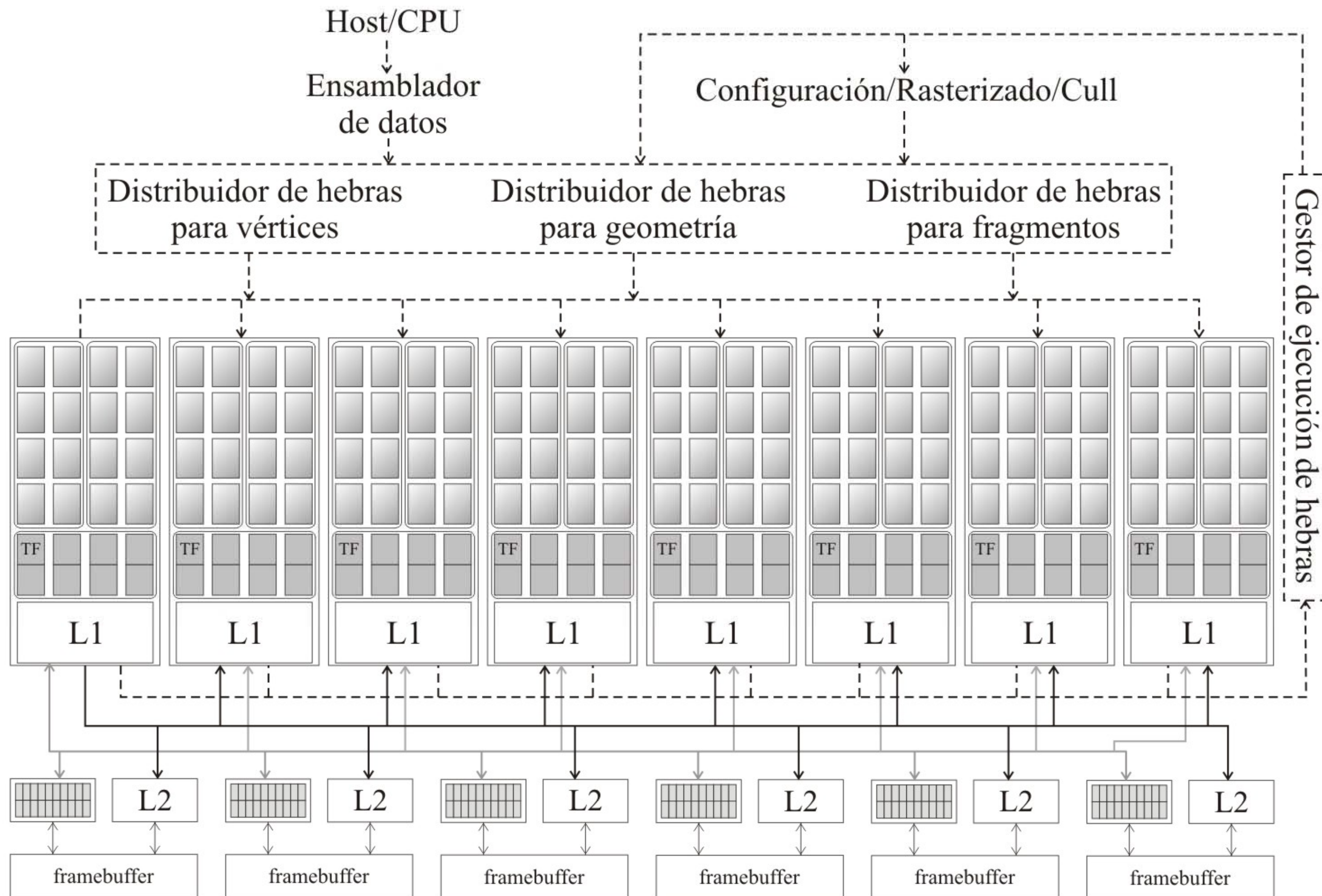
"Reading computer manuals without the hardware is as frustrating as reading sex manuals without the software."
-- Arthur C. Clarke (Clarke's 69th Law, The Odyssey File, 1984)

Aunque en esta clase me voy a apoyar en las transparencias de D. Kirk y WM Hwu la explicación será muy diferente

© Todas las marcas y productos mencionados en estas trasparencias están registradas por sus respectivas compañías, y su uso es de carácter descriptivo con fines docentes.

Parte de las tablas y gráficos están basados en las presentaciones de GPGPU y streaming computing de NVidia y ATI-AMD, en los libros mencionados en la bibliografía, en el curso de David Kirk de la Universidad de Illinois, y el de Simon Green en ARCS08

Qué tenemos dentro del G80/G92?



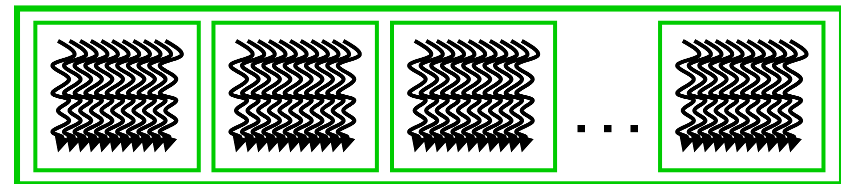
Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  - El código C serie se ejecuta en la CPU
  - El código C paralelizable en forma de kernel que se ejecutan en miles de hebras en organizados en bloques en la GPU

Código serie en CPU

Kernel paralelizado en GPU

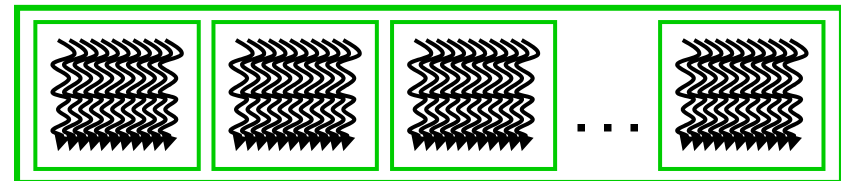
KernelA<<< nBlk, nTid >>>(args);



Código serie en CPU

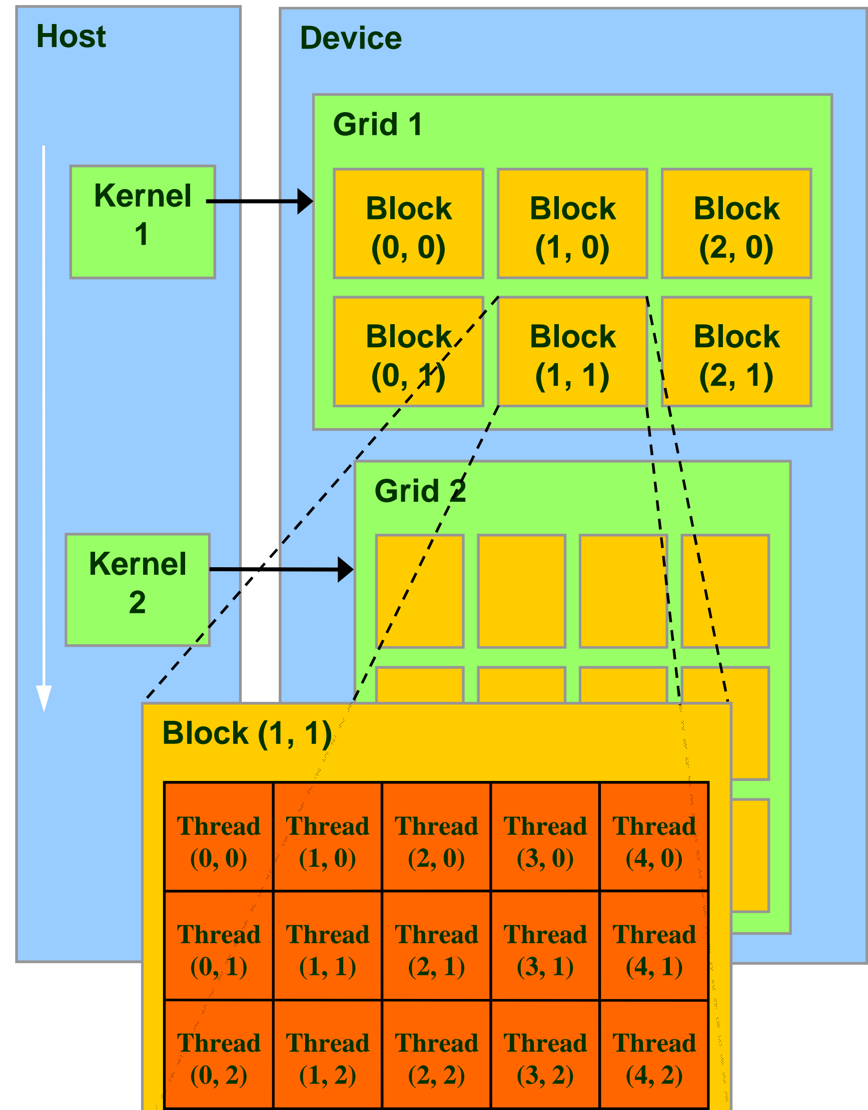
Kernel paralelizado en GPU

KernelB<<< nBlk, nTid >>>(args);



Revisión : Grids y bloques

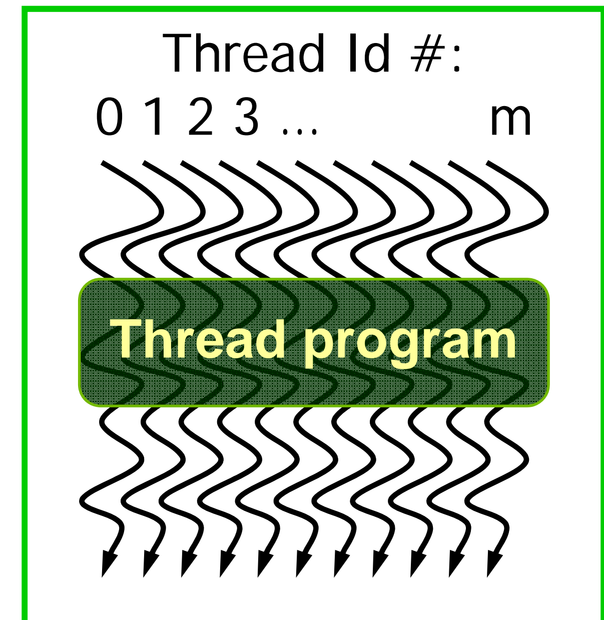
- Un kernes es ejecutado como un grupo (grid) de bloques
  - Las hebras de un mismo bloque pueden compartir datos en memoria compartida
- Un bloque es un cojunto de hebras que pueden cooperar:
  - Sincronizando su ejecución
    - Para evitar riesgos en el acceso a los datos
  - Compartir datos de forma eficiente mediante hardware de baja latencia
  - Hebras de disntos bloques no pueden cooperar (al menos no con estas facilidades)



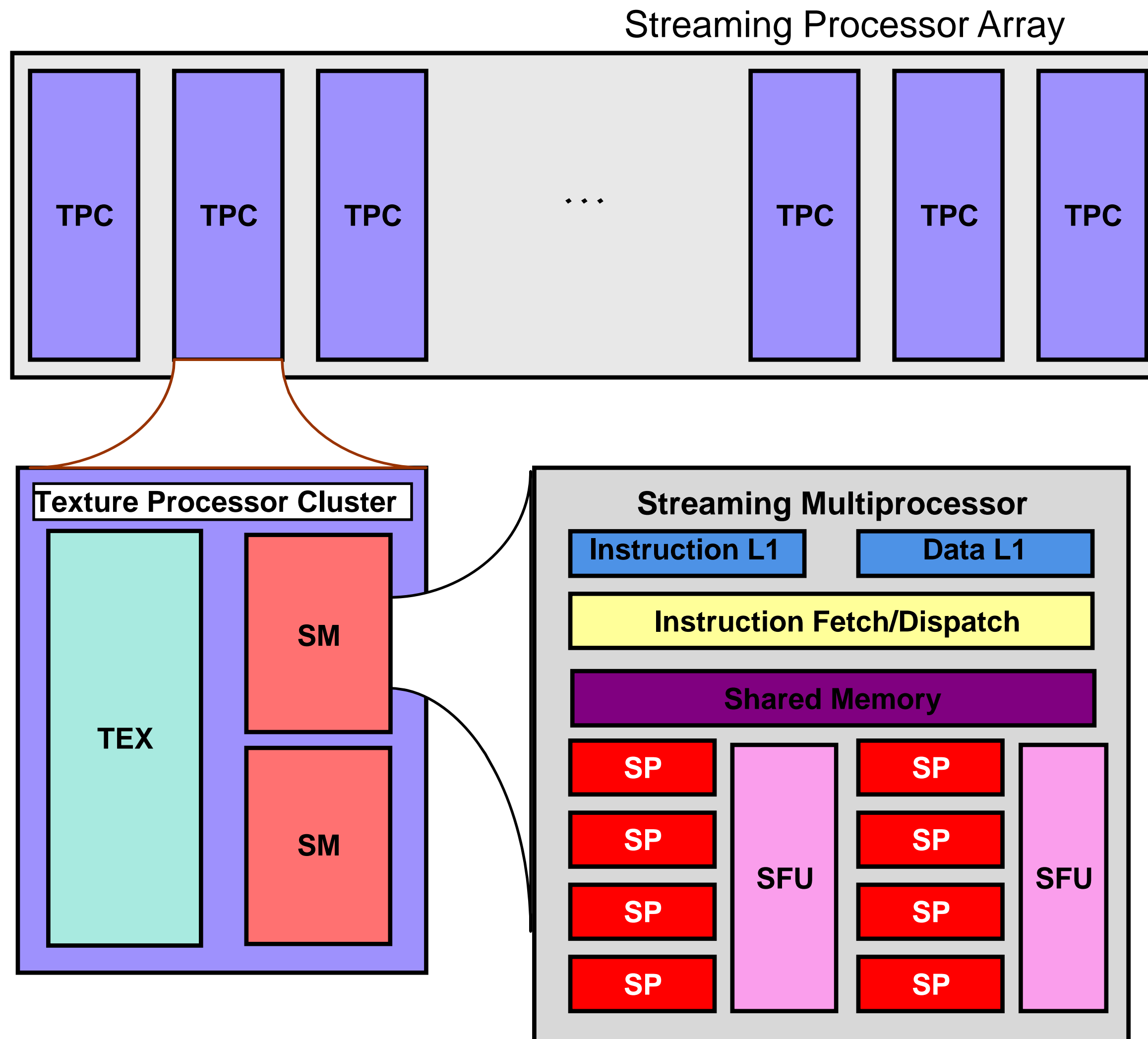
Revisión: Bloque de hebras en CUDA

- El progrador declara el número de bloques atendiendo a un compromiso entre criterios, nos interesa que el número de hebras por bloque:
  - Sea pequeño para que quepan muchos bloques en un mismo multiprocesador y nunca quede bloqueado del todo (// al acceder a memoria principal)
  - Sea grande para que puedan beneficiarse de los mecanismos de comunicación y sincronismo
- Las dimensiones de los bloques pueden
  - Ser 1D, 2D (ó 3D aunque no se usa)
  - Tener de 1 a 512 hebras
  - Consultar los identificadores de los bloques desde cada hebra
- Todas las hebras de un mismo bloque ejecutan el mismo programa
- Las hebras tienen su propio identificador dentro del bloque (thread id)
- Las hebras pueden compartir datos mientras hacen su parte del trabajo
- Los identificadores de bloque y hebra facilitan la asignación de los datos para este trabajo

CUDA Thread Block



Visión panorámica del HW



Terminología de los procesadores de CUDA

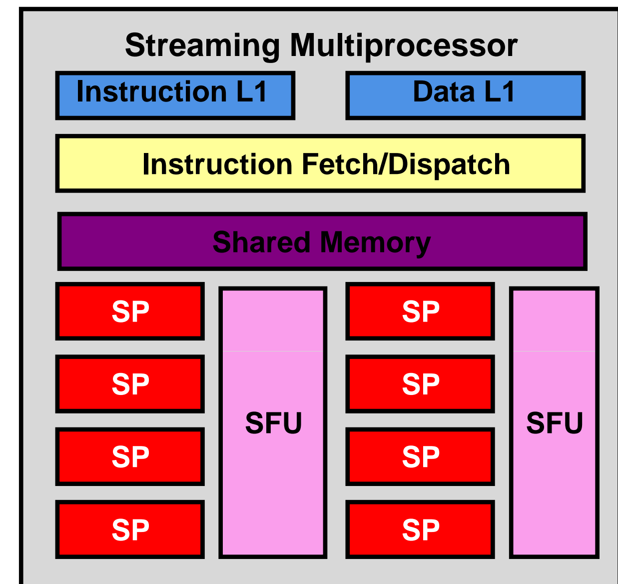
- SPA
  - Streaming Processor Array (en la serie 8 puede cambiar de un modelo a otro, 8 in GeForce8800)
- TPC
  - Texture Processor Cluster (2 SM + TEX)
- SM
  - Streaming Multiprocessor (8 SP)
  - Es un procesador preparado para soporte de multiples hebras
  - Es la unidad de proceso fundamental para los bloques de hebras en CUDA
- SP
  - Streaming Processor
  - Debido a la que la FPU es escalar (de cara a cada hebra)

Utilizaré estas siglas en inglés en las transparencias

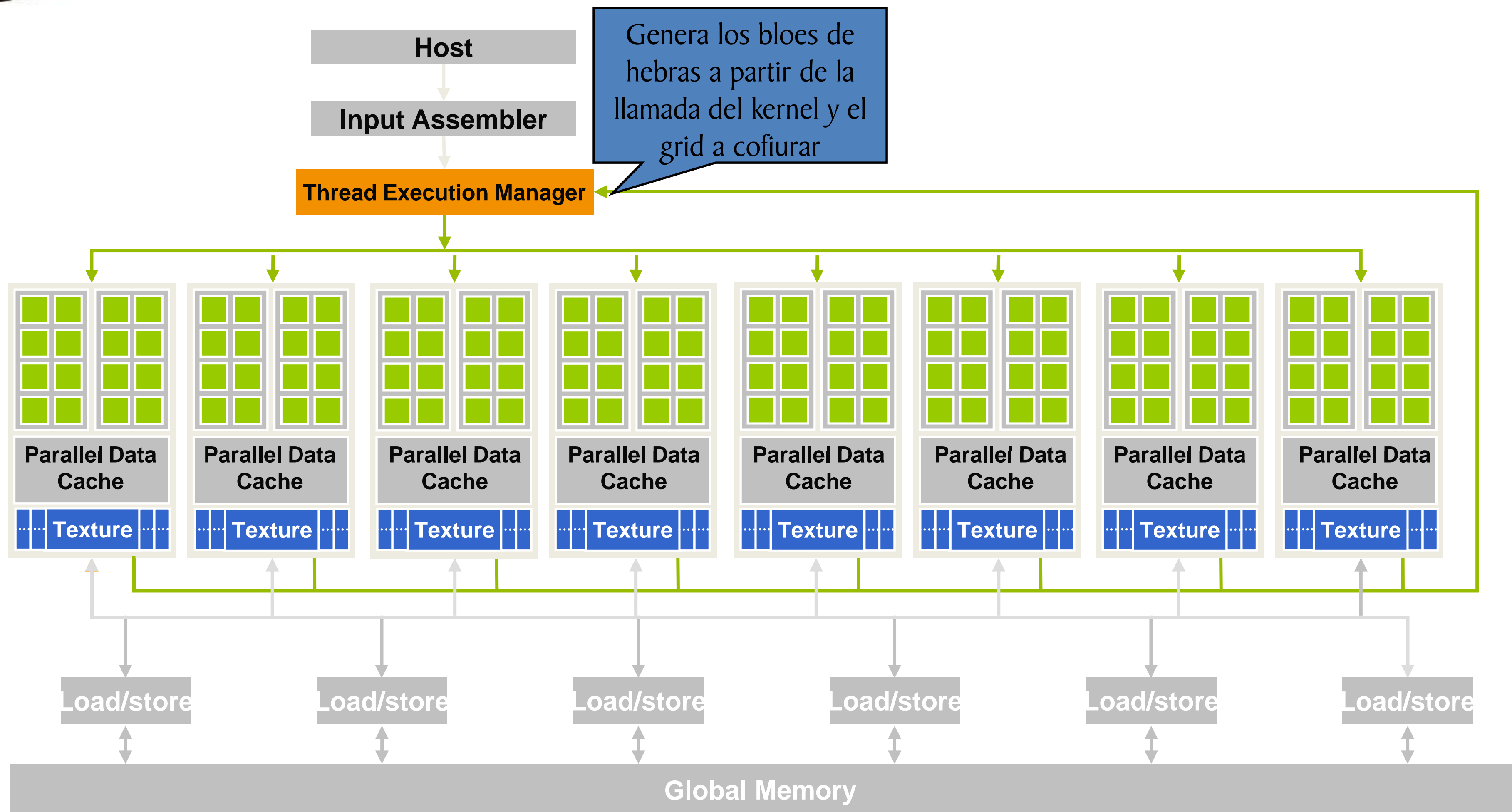
(c) David Miraut Andrés

Streaming Multiprocessor (SM)

- Streaming Multiprocessor (SM)
  - 8 Streaming Processors (SP)
  - 2 Super Function Units (SFU)
- Multi-threaded instruction dispatch
  - De 1 a 768 hebras activas
  - Fetch de instrucciones compartido para subgrupos de 32 hebras
  - Cubre la latencia de las cargas de memoria y texturas
- 20+ GFLOPS
- 16 KB de memoria compartida
- Acceso a memoria de vídeo (y texturas)

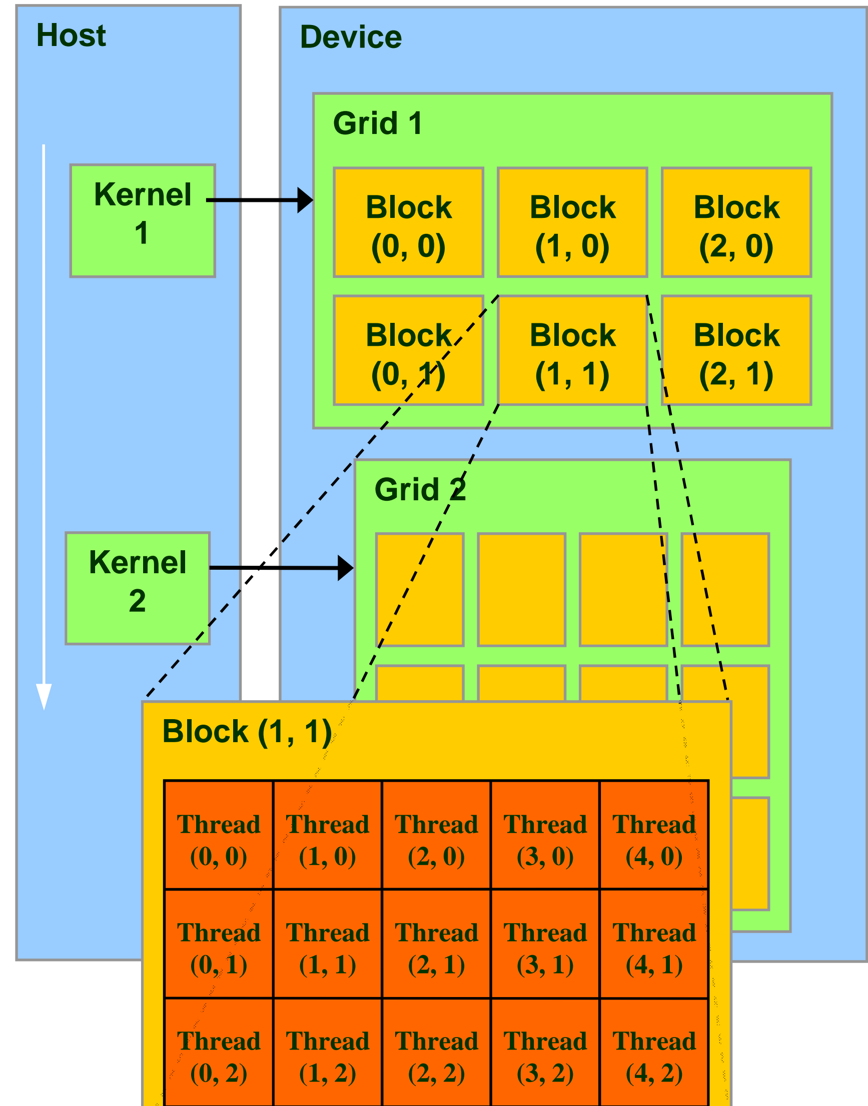


Cauce de computación en las hebras del G80

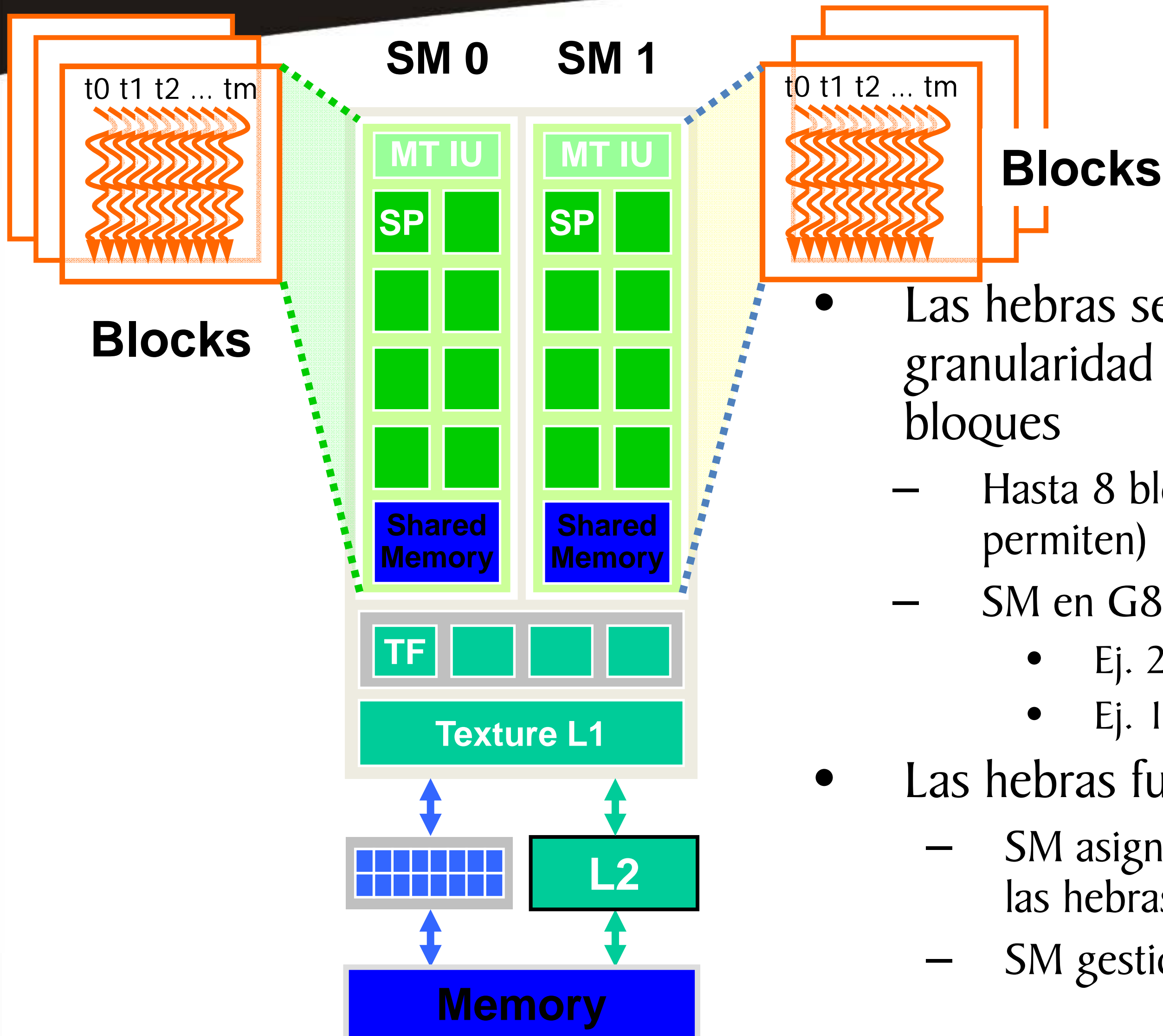


Ciclo de vida de las hebras en HW

- El Grid es lanzado en el SPA
- Los bloques de hebras se distribuyen de forma seralizada en cada SM
  - Potencialmente puede haber más de un bloque de hebras por SM
- Cada SM lanza a su vez Warps de hebras
  - 2 niveles de paralelismo
- El SM gestiona y ejecuta los Warps que están listos para continuar
- A medida que los Warps y los bloques de hebras se completan, los recursos son liberados
  - Y el SPA puede distribuir más bloques de hebras



SM Executes Blocks

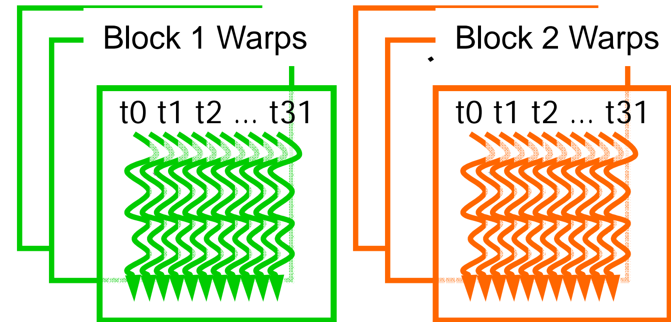


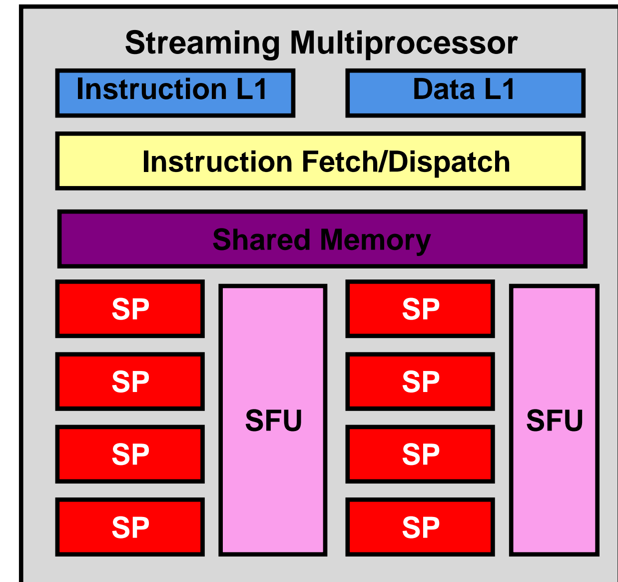
- Las hebras se asignan a los SM en la granularidad elegida por el tamaño de los bloques
  - Hasta 8 bloques por Sm (si los recursos lo permiten)
  - SM en G80 puede tener hasta 768 hebras
    - Ej. 256 (hebras/bloque) * 3 bloques
    - Ej. 128 (hebras/bloque) * 6 bloques, etc.
- Las hebras funcionan concurrentemente
  - SM asigna/mantiene los identificadores de las hebras
  - SM gestiona la ejecución de las hebras

(c) David Miraut Andrés

Gestion y ejecución de hebras

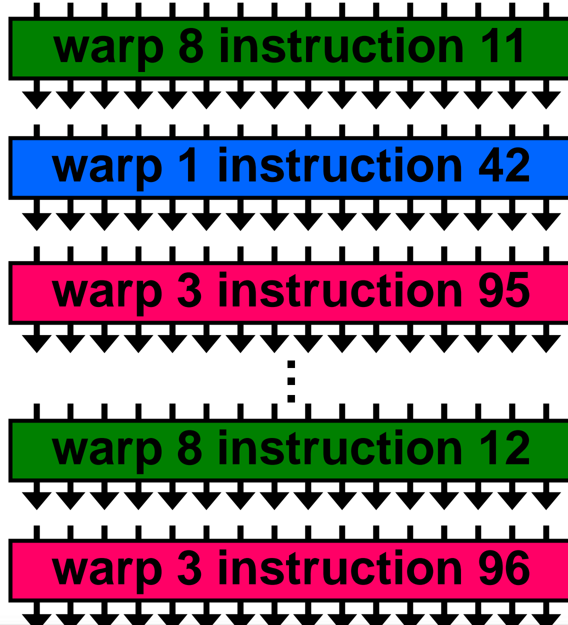
- Cada bloque de hebras es subdividido en Warps de 32 hebras
  - Esta es una decisión de implementación, no parte del modelo de programación de CUDA
  - Hay que tenerlo en cuenta en la actual generación de chips para obtener el máximo rendimiento (si es posible utilizar tamaños multiplos de 32, aunque hay condicionantes más importantes)
- Los Warps son las unidades de organización temporal en el SM





Pregunta

Programación de tiempos de los Warp en los multiprocesadores

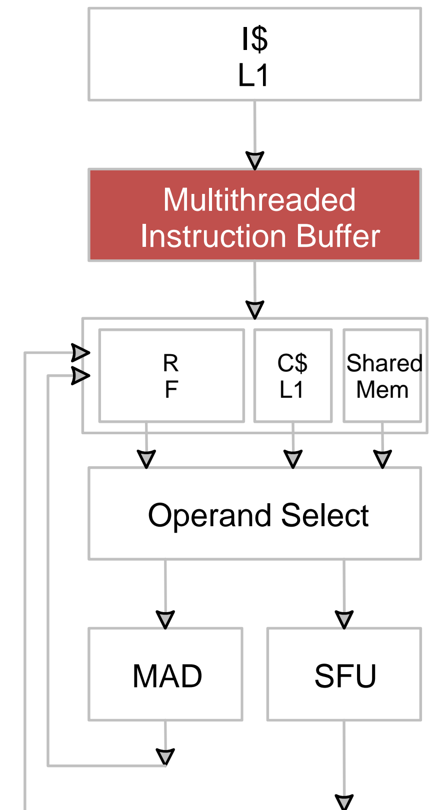


- El hardware de los SM permite realizar el cambio de contexto de los Warps con una sobrecraga nula
  - Los Warps cuya siguiente instrucción tiene operandos listos para ser ejecutados es elegible para su ejecución
  - Esa elección se hace tiendo en cuenta una política de gestion de tiempos de acuerdo a unos criterios de priorización
  - Todas las hebras ejecutan la misma instrucción inicial al ser seleccionadas
- Se requieren 4 ciclos para servir la misma instrucción a todas las hebras en un Warp del G80
  - Si se necesita hacer un acceso a memoria global por cada 4 intrucciones…
  - Necesitamos un mínimo de 13 warps en el mismo bloque para poder esconder ele efecto de la latencia de una memoria que tarde 200 ciclos

(c) David Miraut Andrés

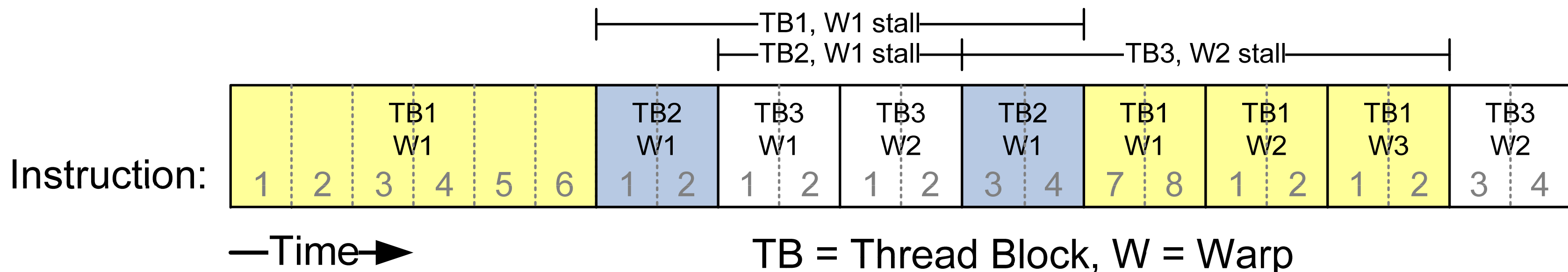
Buffer de instrucciones del SM

- Coge una instrucción por ciclo para el warp
  - De la caché de instrucciones L1
  - En cualquier slot de instrucciones del buffer
- Se da una instruccion lista para ejecutar en el warp por ciclo
  - Desde slot del buffer de instrucciones de un warp cualquiera
  - Se intentan prevenir riesgos mendiante scoreboarding de operandos
- La selección está basada en un criterio de round-robin/age del warp
- El SM envía la misma instruccione a las 32 hebras de un Warp



Scoreboarding

- Todos los registros para operandos de todas las instrucciones en el buffer de instrucciones están scoreboarded
  - Status está listo una vez los valores necesarios han sido introducidos
  - Evitar/prevenir riesgos
  - Las instrucciones limpias son elegibles para ser enviadas (y procesadas)
- Cauces desacoplados de memoria y procesamiento
  - Cualquier hebra puede continuar procesando hasta que el mecanismo de scoreboarding lo evite
  - Esto permite que las operaciones de memoria y procesamiento se vayan ejecutando como operaciones shadow de uno u otro



(c) David Miraut Andrés

Consideraciones de granularidad

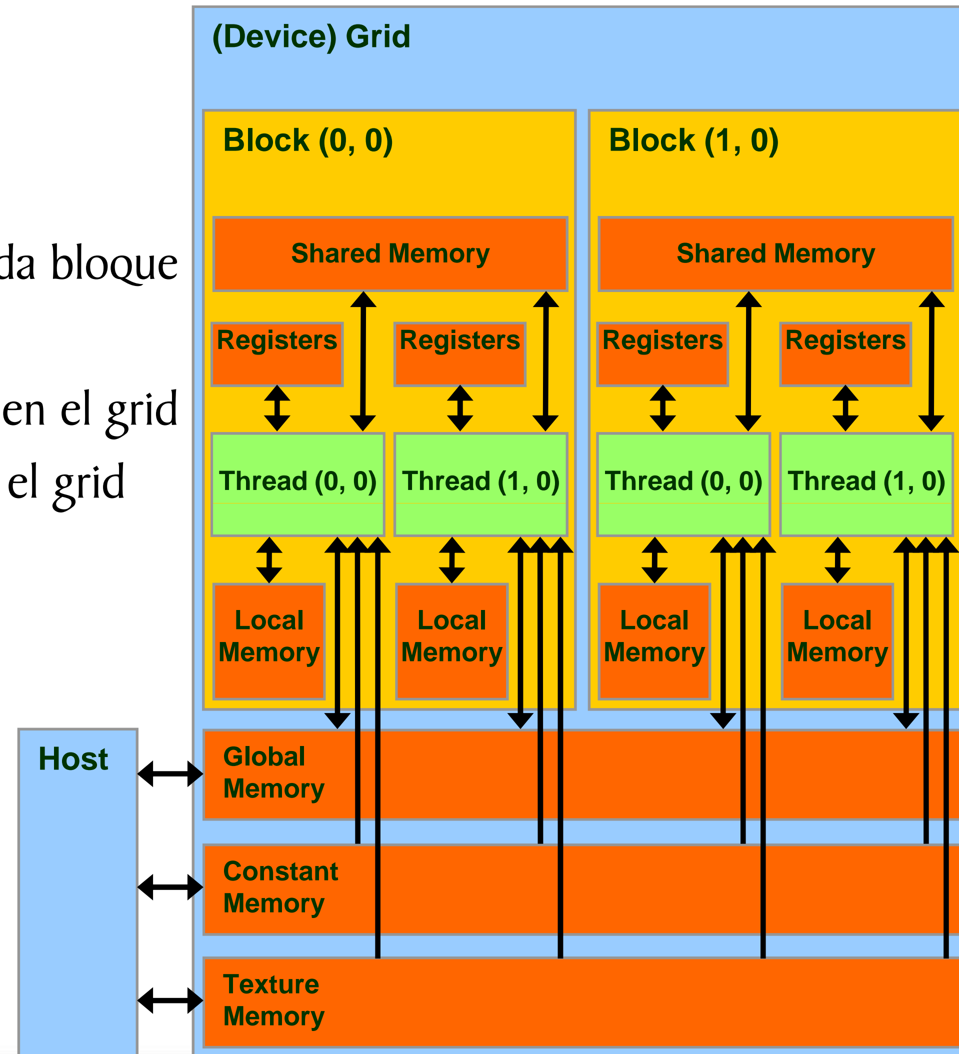
- En el ejemplo de la práctica 8 de multiplicación de matrices, ¿qué tamaño de bloque debería utilizar
  - Para 8X8, tenemos 64 hebras por bloque. Puesto que cada SM puede manejar un máximo de 768 hebras, podremos tener hasta 12 Blocks. Sin embargo, cada SM puede sólo gestionar hasta 8 bloques, y sólo 512 hebras entrarán en cada SM!
  - Para 16X16, tenemos 256 hebras por bloque. Dado que cada puede manejar hasta 768 hebras, podremos tener hasta un máximo de 3 bloques. Esta parece ser la dimensión óptima con la que optimizamos los recursos a menos que otras consideraciones prevalezcan.
  - En el caso de 32X32, tendremos 1024 hebras por bloque. No hay manera de meter esto en un SM!

Hardware de la jerarquía de memoria

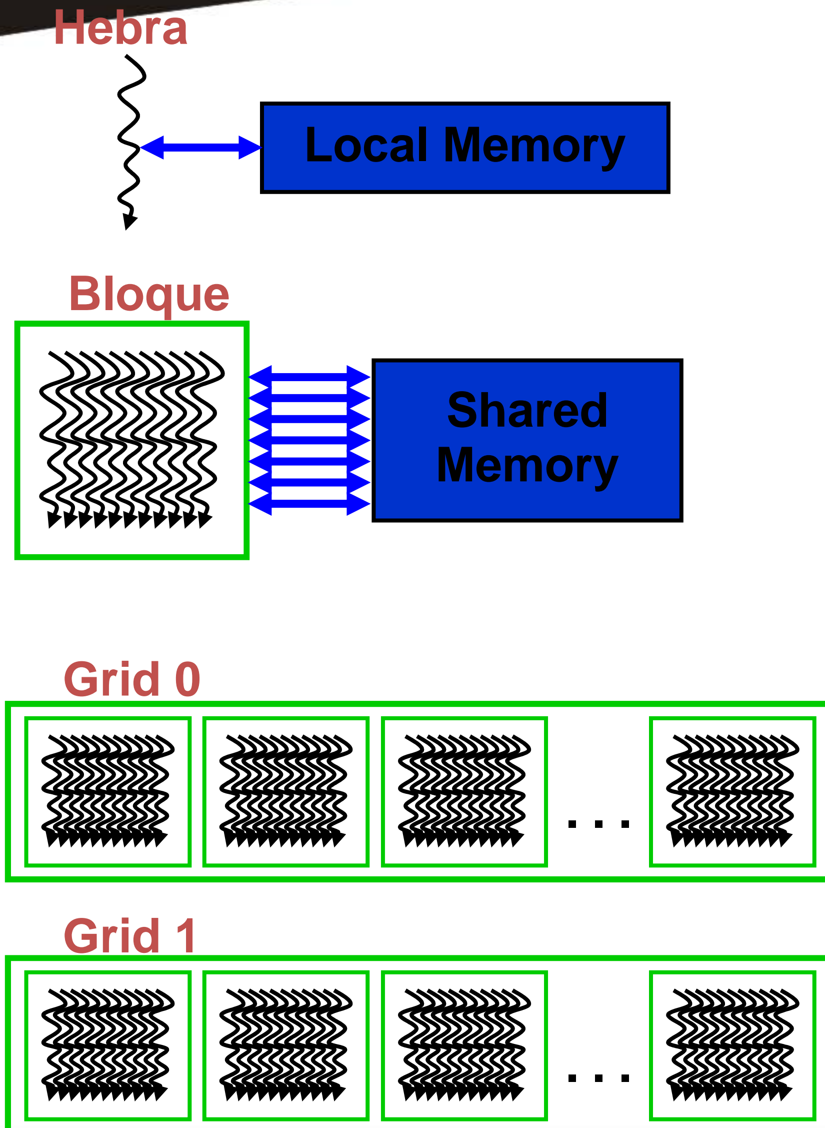
Jerarquía de memoria en CUDA

- Cada hebra puede:
  - R/W registros para cada hebra
  - R/W memoria local a cada hebra
  - R/W memoria compartida para cada bloque
  - R/W memoria global en el grid
  - Consultar memoria de constantes en el grid
  - Consultar memoria de texturas en el grid

- La CPU (host), puede leer y escribir en la memoria global, de constantes, y de textura



Mecanismos de comunicación por memoria compartida



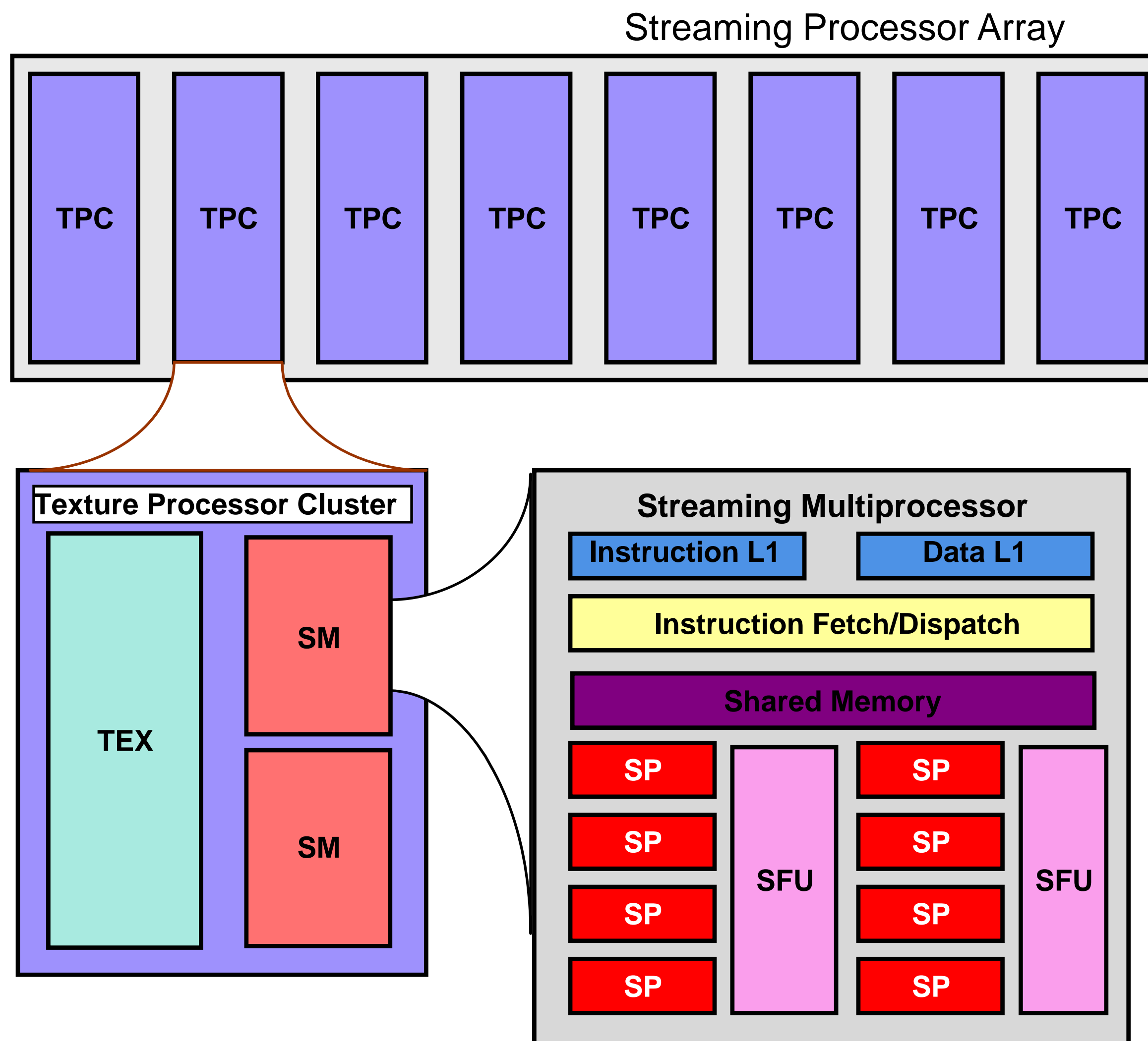
- Memoria local: por-hebra
  - Privada para cada hebra
  - Auto variables, register spill
- Memoria compartida: por-bloque
  - Compartida y accesible por hebras del mismo bloque
  - Nos permite tener un mecanismo de comunicación entre hebras
- Memoria Global: por-aplicación
  - Se comparte entre todas las hebras
  - Nos permite tener un mecanismo de comunicación entre diferentes Grids

Global Memory

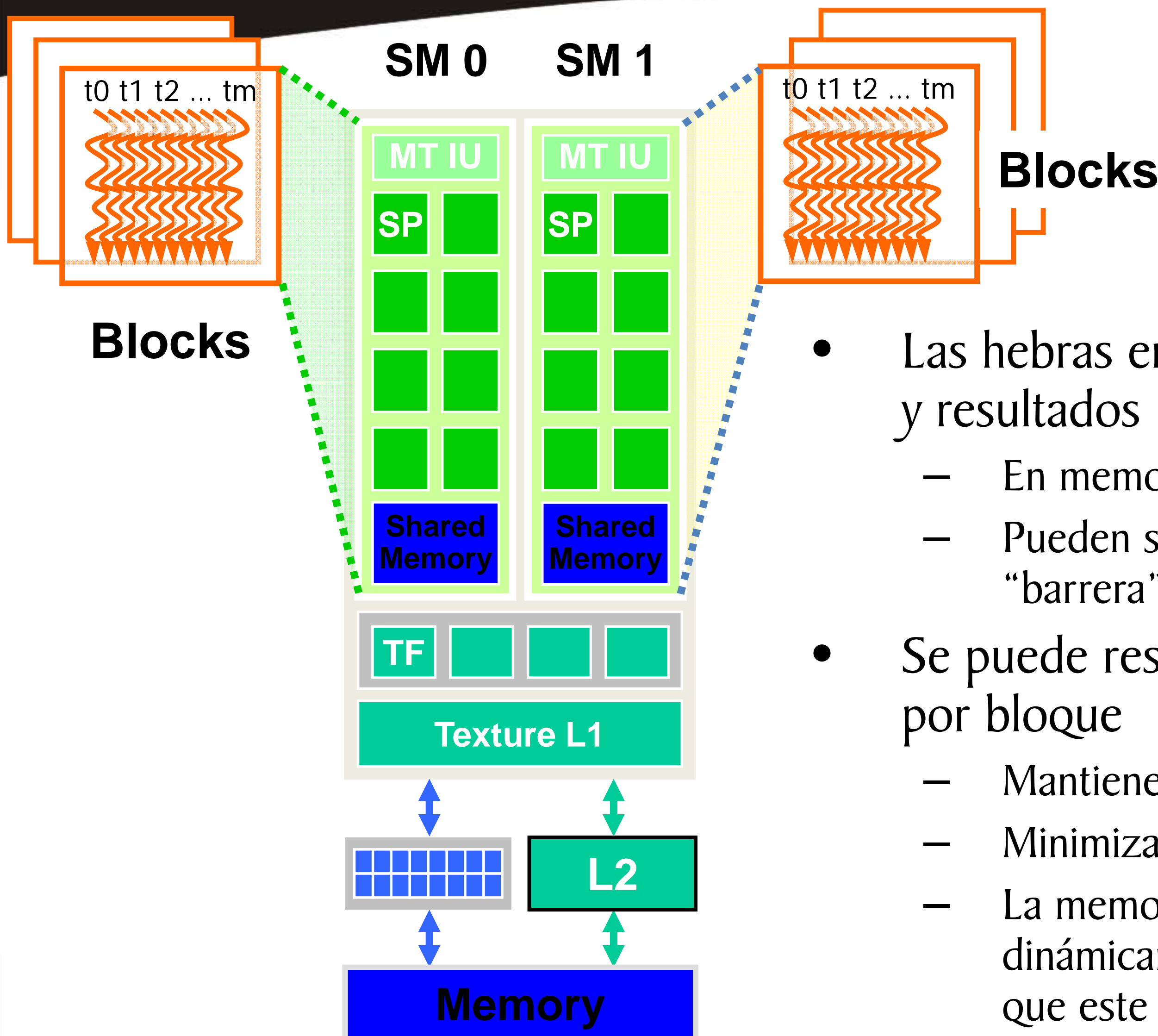
Sequencia de grids en el tiempo

(c) David Miraut Andrés

Resumen del HW



Arquitectura de la memoria del SM

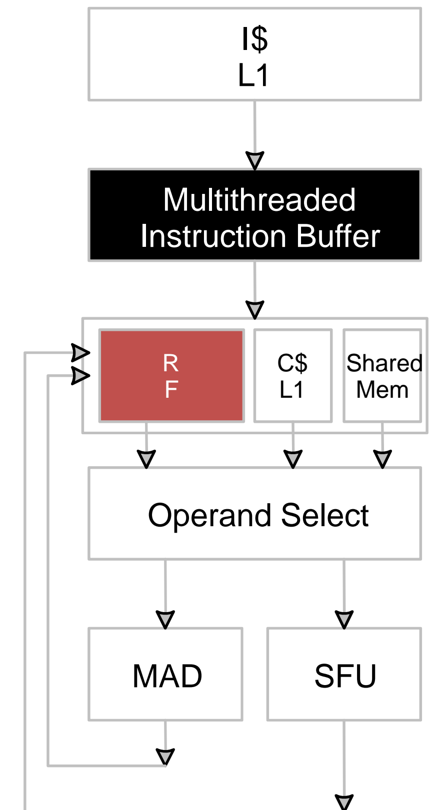


- Las hebras en un bloque comparten datos y resultados
  - En memoria global y compartida
  - Pueden sincronizarse (con instrucciones "barrera")
- Se puede reservar memoria compartida por bloque
  - Mantiene los datos cerca del procesador
  - Minimiza los "viajes" a memoria de vídeo
  - La memoria compartida se reserva dinámicamente para los bloques, por lo que este es uno de los recursos (compartidos) que nos limita

(c) David Miraut Andrés

SM Register File

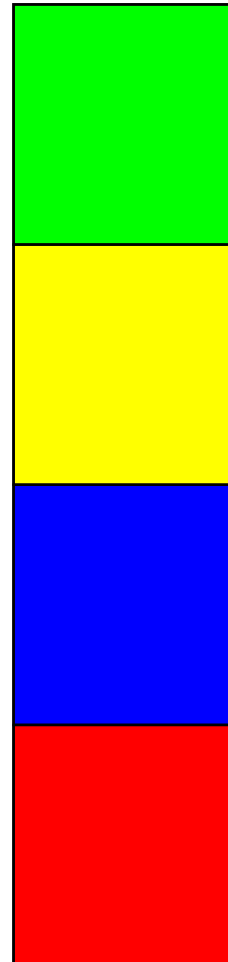
- Register File (RF)
  - 32 KB
  - Puede dar 4 operandos/ciclo de reloj
- El cauce de texturas (TEX pipe) también puede leer y escribir en RF
  - 2 SMs comparten 1 TEX
- El cauce de carga/Almacenamiento también puede leer y escribir en RF



El Register File desde el punto de vista del programador

- Tenemos 8192 registros disponibles en cada SM del G80
  - Es una decisión de implementación, no parte de CUDA
  - Los registros son dinámicamente repartidos entre todos los bloques asignados a un SM
  - Una vez asignados a un bloque, ya no pueden ser accedidos por otros bloques
  - Cada hebra en un bloque sólo puede acceder a los registros que le han sido asignados a ella.

4 bloques



3 bloques

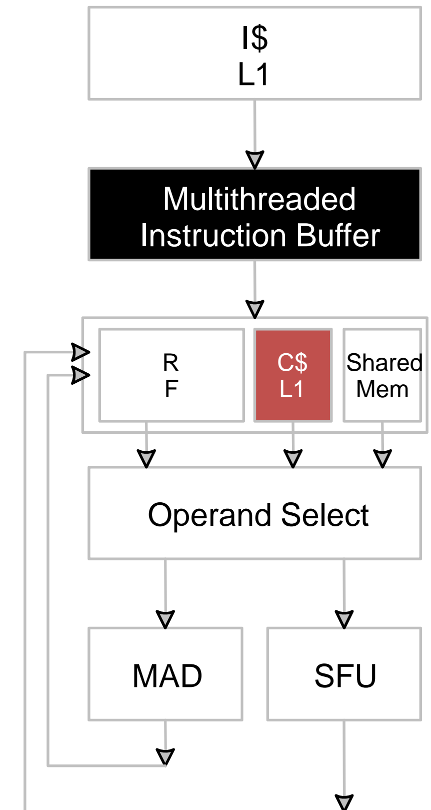
Particionado/Reparto dinámico

- Esta estrategia da más flexibilidad a los compiladores/programadores
  - Se puede ejecutar un número más pequeño de hebras que requieran muchos registros o un gran número de hebras que sólo necesiten un puñado de registros
    - Esto nos permite tener una granularidad más fina en las hebras que los modelos tradicionales de CPU
  - El compilador puede alcanzar un compromiso entre el paralelismo a nivel de instruccion y a nivel de hebras.

(c) David Miraut Andrés

Constantes

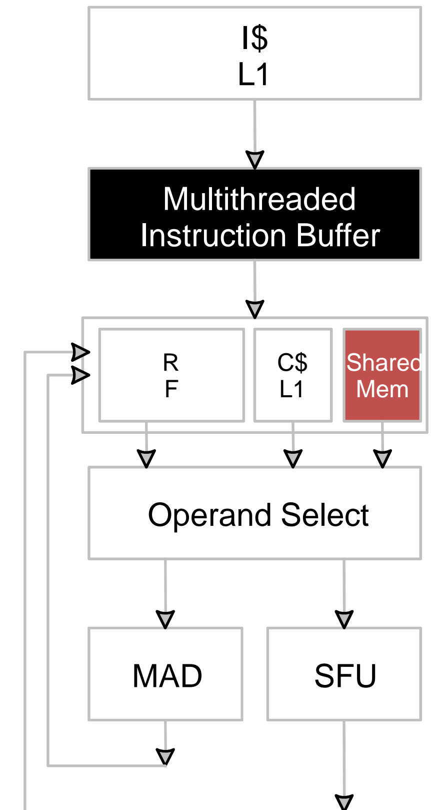
- Contantes de direcciones inmediatas
- Contantes de direcciones indexadas
- Constants son guardadas en memoria de vídeo, pero se acceden a través de una caché em el echip
  - Nivel S1 dentro del SM
- Un valor constante puede ser "emitido" a todas las hebras de un Warp simultáneamente
  - Esta es una forma extremadamente eficiente de acceder a un valor que es común a todas las hebras del bloque



(c) David Miraut Andrés

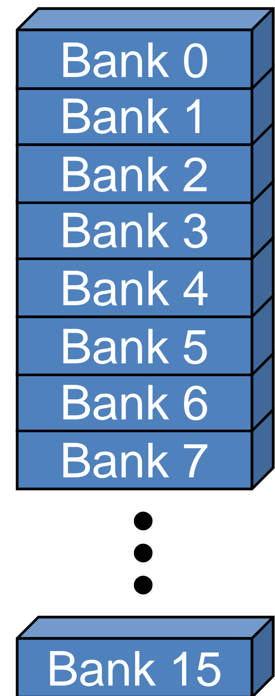
Memoria Compartida

- Cada SM tiene 16 KB de Memoria "Compartida"
  - 16 bancos de palabras de 32bit
- CUDA utiliza esta memoria compartida como almacenamiento visible a todas las hebras del bloque
  - Acceso en lectura y escritura
- No se puede utilizar explicitamente en shaders del cauce gráfico
  - A David Kirk no le mola que los fragmentos hablen unos con otros ;-)
  - Lo más probable es que esta memoria se utilice de alguna forma para tener un buffer entre diferentes etapas de manera que no se queden bloqueadas por tener una carga ligeramente diferente en puntos diferentes de la escena a procesar



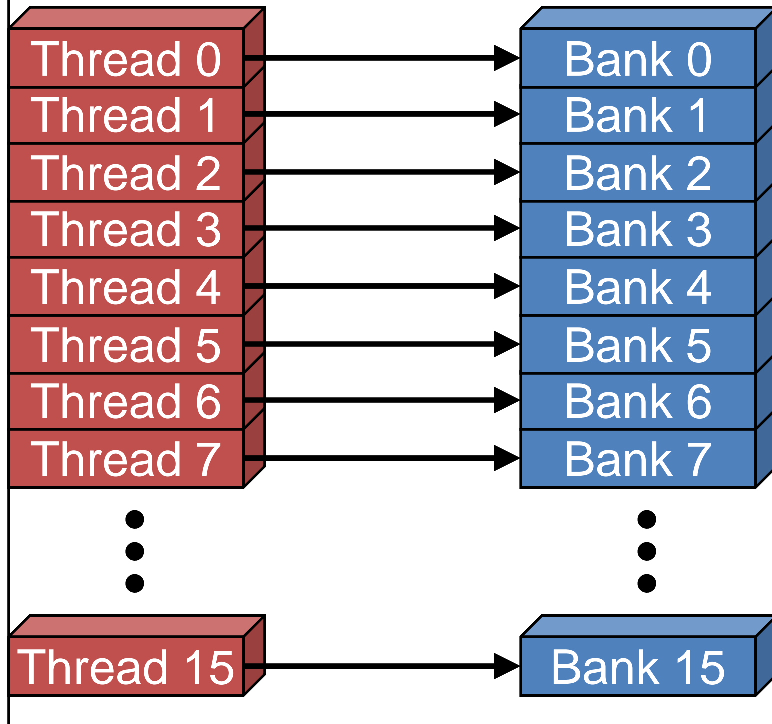
Arquitectura de memoria paralela

- En un sistema paralelo muchos procesadores acceden simultáneamente a memoria
  - Debido a ello, la memoria esta dividida en bancos
  - Es esencial para conseguir un ancho de banda razonable
- Cada banco puede servir una dirección por ciclo
  - Una memoria puede servir tantos accesos simultáneos como bancos tenga
- Accesos multiples a un mismo banco producen un conflicto
  - Los accesos conflictivos son serializados

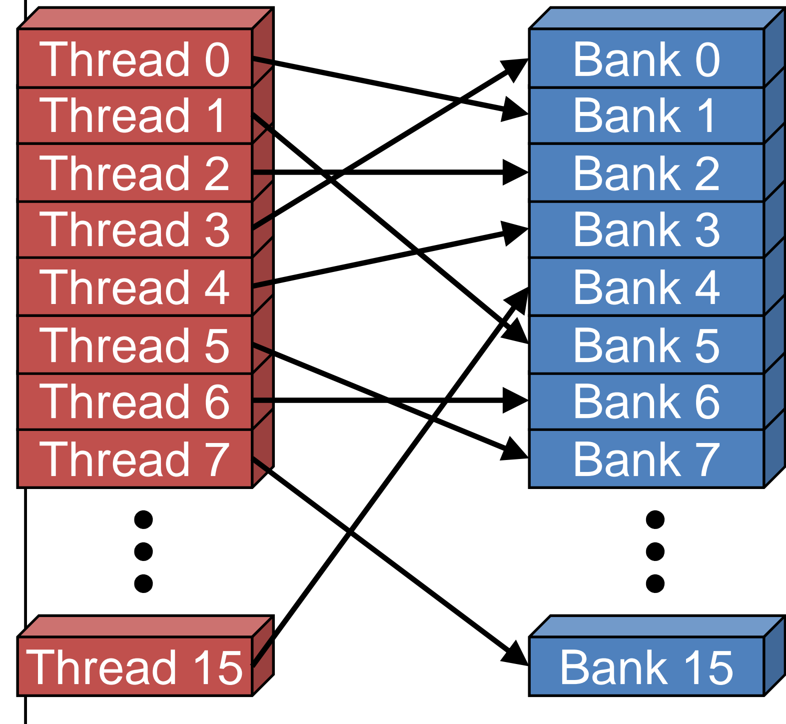


Ejemplos de acceso a los bancos

- Sin conflictos entre bancos
  - Acceso lineal
    paso == 1

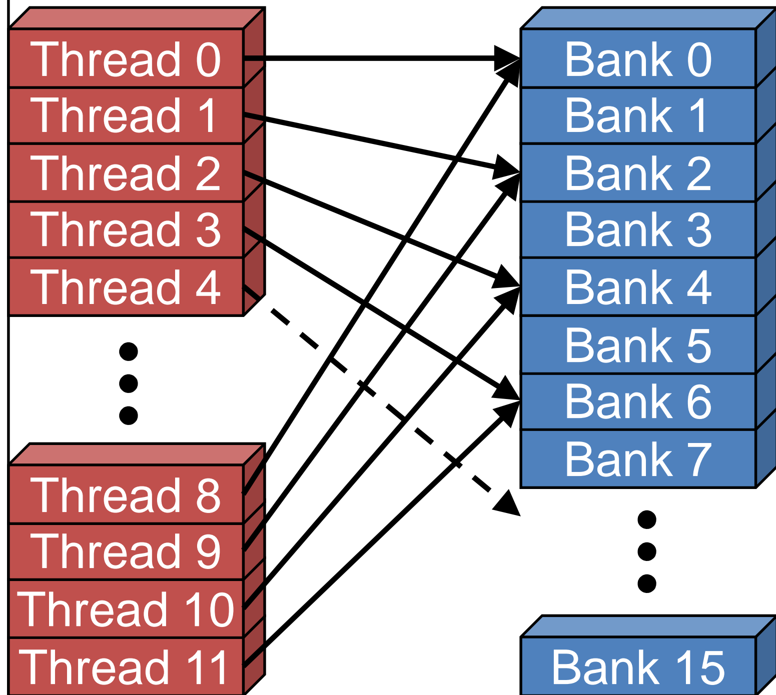


- No Bank Conflicts
  - Aleatorio (permutación 1:1)

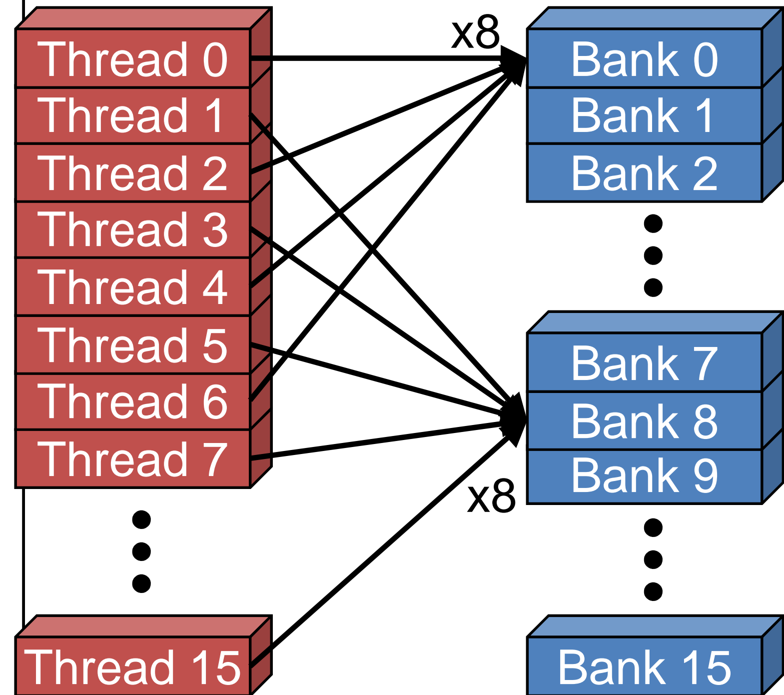


Ejemplos de acceso a los bancos

- Conflicto en los bancos (2 en 2)
  - Acceso lineal paso == 2



- Conflictos en bancos (8 en 8)
  - Acceso lineal paso == 8



Cómo se mapean las direcciones de memoria en el G80

- Cada banco tiene un BW de 32 bits po ciclo de reloj
- Palabras sucesivas de 32 bits son asignadas a bancos sucesivos
- G80 tiene 16 bancos
  - Banco que buscamos = dirección % 16
  - El mismo tamaño que medio warp
    - De modo que no tendremos conflictos entre diferentes mitades de un warp, sólo dentro de una de esas mitades de warp simultáneamente

(c) David Miraut Andrés

Conflictos de bancos de memoria compartida

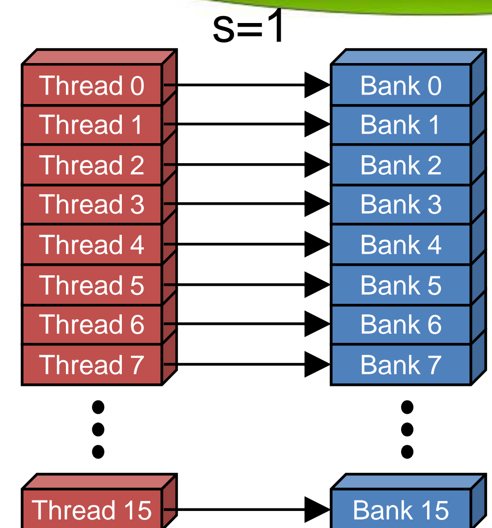
- La memoria compartida es tan rapida como acceder a un registro siempre que no se produzca un conflicto entre bancos

- El caso rápido:
  - Si todas las hebras de una mitad de warp, no hay conflicto de banco
  - Si todas las hebras de una mitad del warp acceden a la misma dirección, tampoco hay conflicto (se "emite" mediante el mecanismo de broadcast)
- El caso lento:
  - Conflicto de banco: varias hebras en la misma mitad de warp acceden al mismo banco
  - Se deben serializar los accesos
  - Coste = max # de los accesos simultáneos a un mismo banco

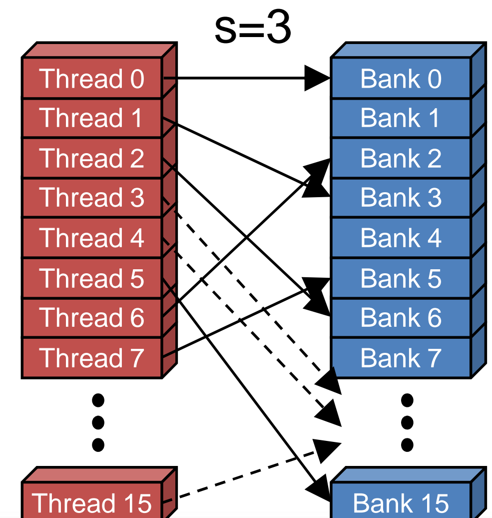
Direccionamiento lineal

- Dado:

```
__shared__ float shared[256];
float foo =
  shared[baseIndex + s *
  threadIdx.x];
```



- Estará libre de conflictos sólo si no hay factor común con el número de bancis
  - 16 en G80, de manea que s debe ser impar

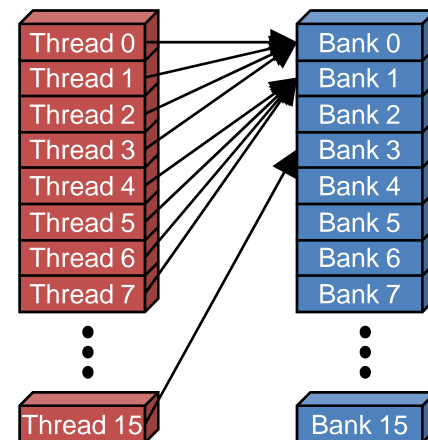


(c) David Miraut Andrés

Tipos de datos y conflictos de bancos

- Aquí no hay conflictos si el tipo de `shared` es de 32 bits:

```
foo = shared[baseIndex + threadIdx.x]
```

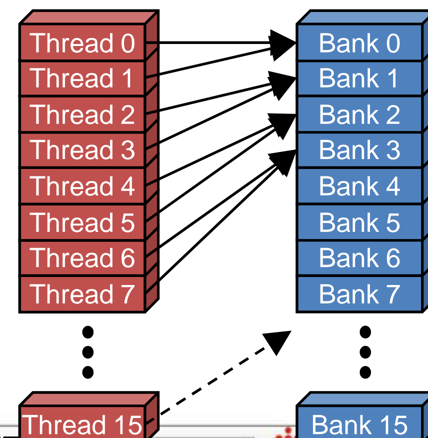


- Pero no funcionará tan bien si el tipo es más pequeño:
  - Conflicto (de 4 en 4):

```
__shared__ char shared[];
foo = shared[baseIndex + threadIdx.x];
```

  - Conflicto (de 2 en 2):

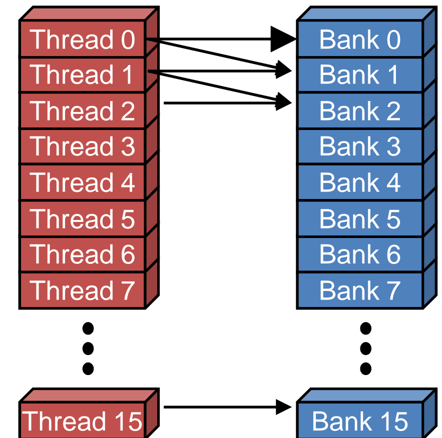
```
__shared__ short shared[];
foo = shared[baseIndex + threadIdx.x];
```



Estructuras y conflictos de bancos

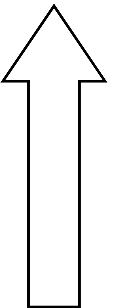
- Las asignaciones en las estructuras de datos se convierten en tantos accesos como miembros haya en la estructura:

```
struct vector { float x, y, z; };
struct myType {
   float f;
   int c;
};
__shared__ struct vector vectors[64];
__shared__ struct myType myTypes[64];
```



- No habría conflictos para un vector en el primer caso si el tamaño de la estructura fuese 3:
  - 3 accesos por hebra, en bancos contiguos (no tenemos un factor común con 16)

```
struct vector v = vectors[baseIndex + threadIdx.x];
```



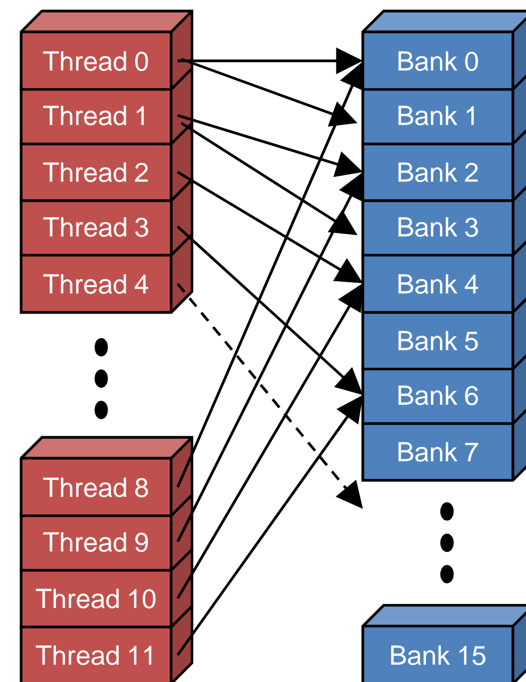
- En el segundo tendriamos conflictos; (2 accesos por hebra)

```
struct myType m = myTypes[baseIndex + threadIdx.x];
```

Conflictos típicos en arrays 1D

- Cada hebra carga dos valores en memoria compartida:
  - Las cargas entrecruzadas dan lugar a conflictos de 2 en 2:

```
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```



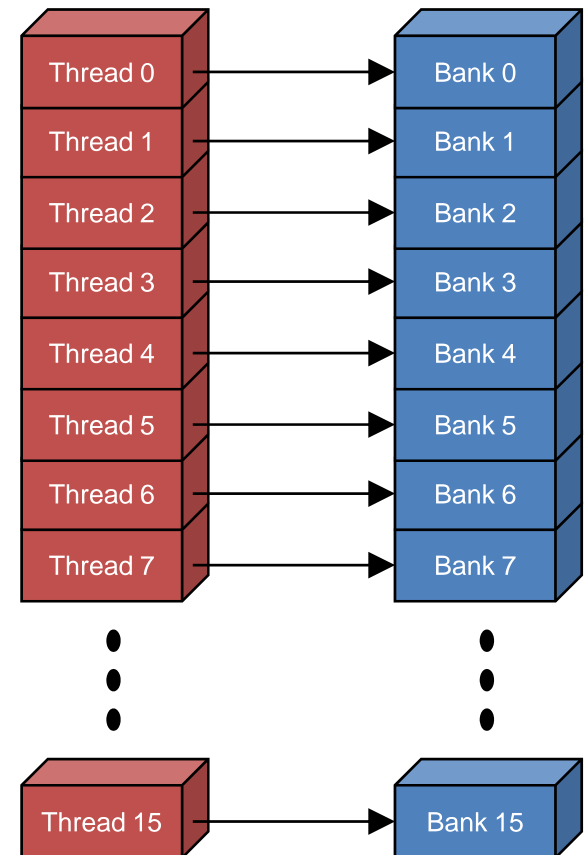
- Este tipo de accesos tiene sentido en el tipo de hebras tradicionales de las CPUs, en las que así se aprovecha la localidad, es uso de líneas de caché y se reduce el tráfico a memoria.
  - Pero no en memoria compartida, donde este patrón de acceso produce efectos adversos

(c) David Miraut Andrés

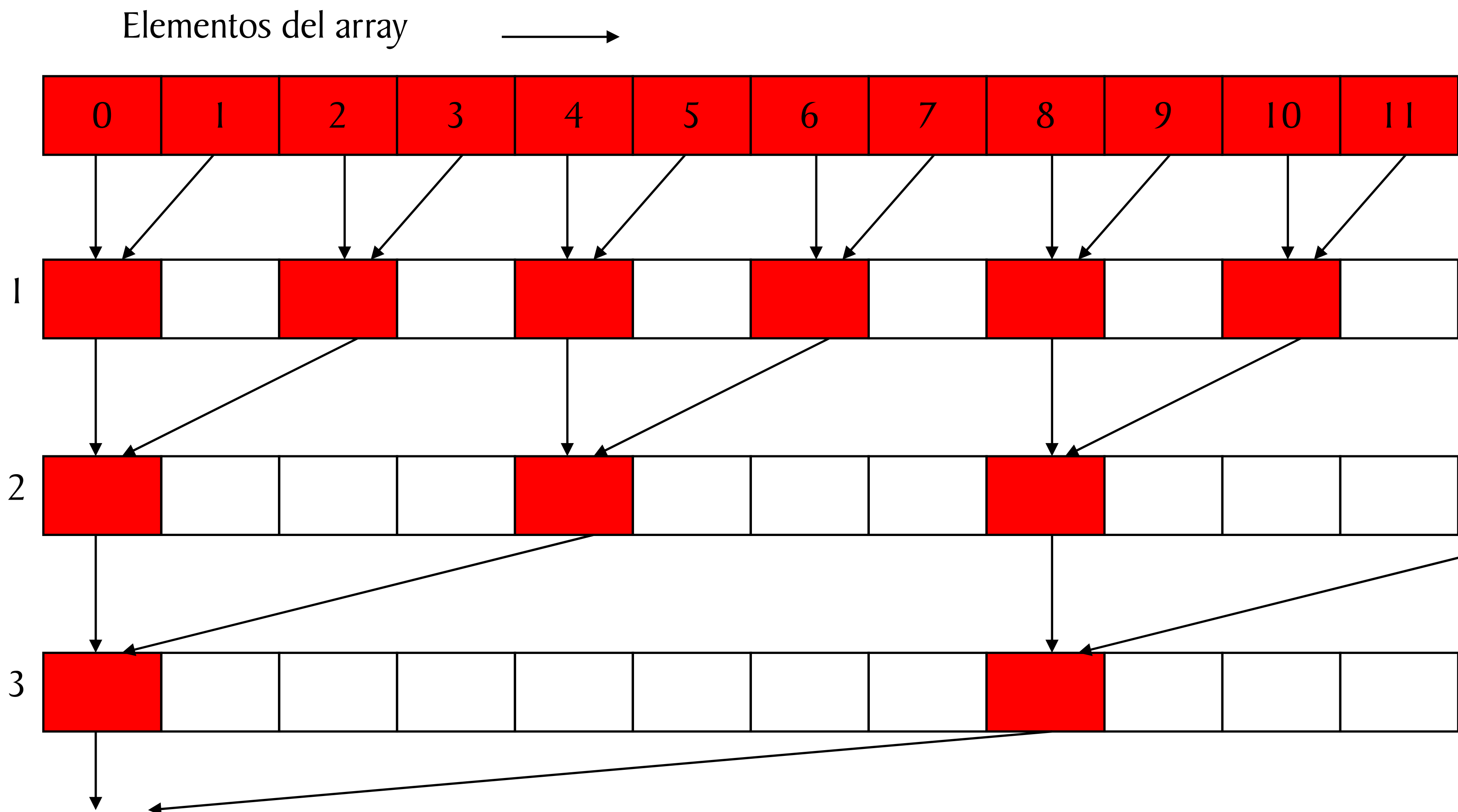
Mejor patrón de acceso a arrays

- Cada hebra carga un elemento en un grupo de elementos blockDim consecutivos

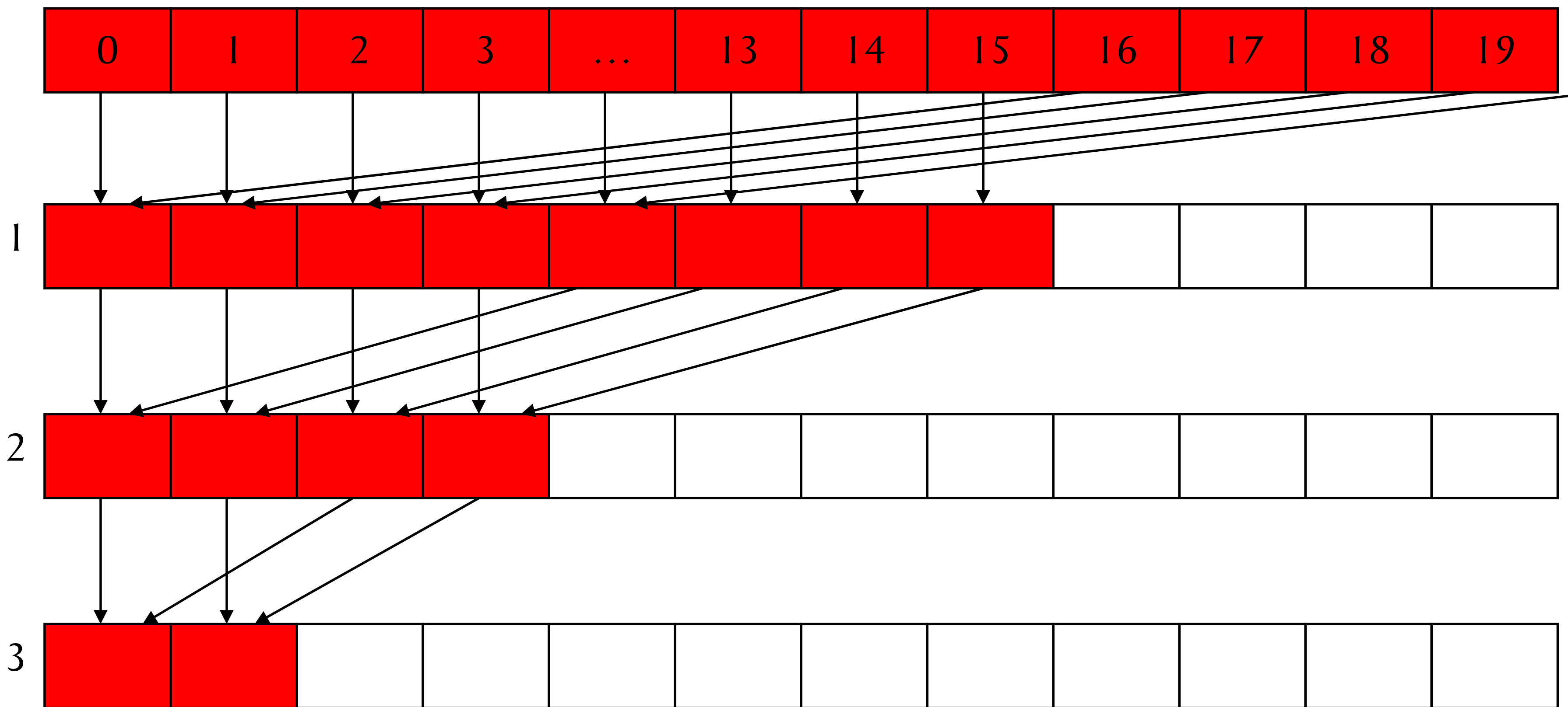
```
shared[tid] =
  global[tid];
shared[tid +
  blockDim.x] =
  global[tid +
  blockDim.x];
```



Reducción de vectores con conflictos entre bancos



Sin conflictos entre bancos



Conflictos típicos en arrays 2D

- Cálculos con arrays 2D de floats en memoria compartida
  - Ej. Procesado de imagen
- Ejemplo: bloque de 16x16
  - Cada hebra procesa una fila
  - De modo que las hebras de un mismo bloque acceden a los elementos de cada columna simultáneamente (Ej. columna 1 en violeta)
  - Tendremos conflictos de 16 en 16, todas las filas comienzan en el banco 0
- Solución 1) rellenar artificialmente las filas
  - Añadir un float al final de cada fila
- Solución 2) transponer la matriz antes de procesar
  - Podemos tener algunos conflictos durante la transposición
  - Pero, posiblemente, los ahorraremos despues

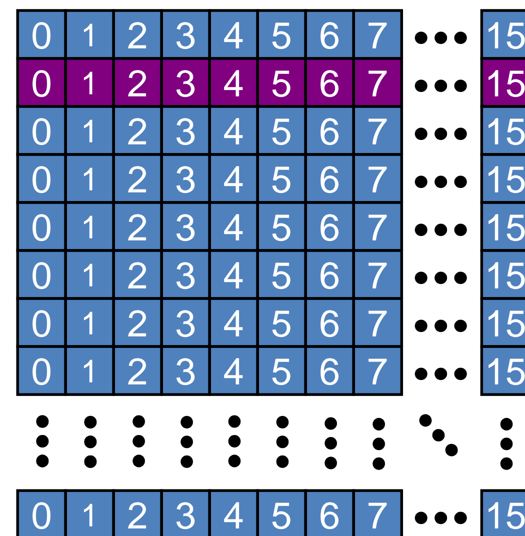
Bank Indices without Padding

0	1	2	3	4	5	6	7	•••	15
0	1	2	3	4	5	6	7	•••	15
0	1	2	3	4	5	6	7	•••	15
0	1	2	3	4	5	6	7	•••	15
0	1	2	3	4	5	6	7	•••	15
0	1	2	3	4	5	6	7	•••	15
0	1	2	3	4	5	6	7	•••	15
0	1	2	3	4	5	6	7	•••	15
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
0	1	2	3	4	5	6	7	•••	15

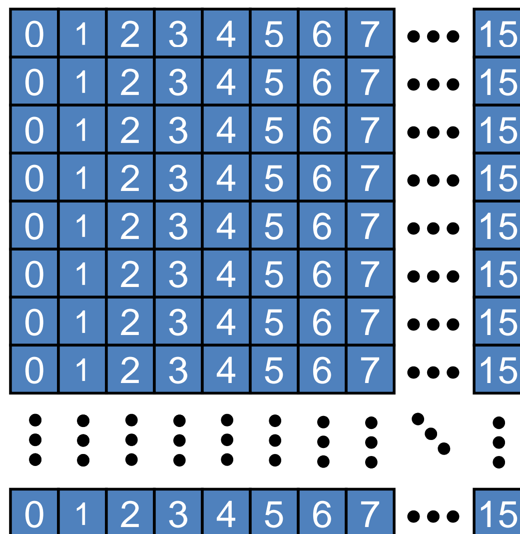
Bank Indices with Padding

0	1	2	3	4	5	6	7	•••	15	0
1	2	3	4	5	6	7	8	•••	0	1
2	3	4	5	6	7	8	9	•••	1	2
3	4	5	6	7	8	9	10	•••	2	3
4	5	6	7	8	9	10	11	•••	3	4
5	6	7	8	9	10	11	12	•••	4	5
6	7	8	9	10	11	12	13	•••	5	6
7	8	9	10	11	12	13	14	•••	7	8
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
15	0	1	2	3	4	5	6	•••	14	15

Tenemos conflictos en la multiplicación de matrices?



Todos los warps en un bloque acceden a la misma fila de M ¡broadcast!



Todos los warps en un bloque acceden a los elementos vecinos en una fila del mismo modo en el que acceden a las columnas vecinas !!

(c) David Miraut Andrés

Agrupamiento de lecturas y escrituras

- Utilizar LD para esconder la latencia (en el caso de que no dependan unos de otros)
  - Utilizar la misma nhebra para esconder los efectos de la latencia
- En lugar de hacer:
  - LD 0 (long latency)
  - Dependent MATH 0
  - LD 1 (long latency)
  - Dependent MATH 1
- Poner smplemente:
  - LD 0 (long latency)
  - LD 1 (long latency - hidden)
  - MATH 0
  - MATH 1
- El compilador se encargará de ordenarlo adecuadamente!
  - Pero para ello necesitamos tener suficientes LDs no dependientes y -sobretodo- operaciones matemáticas con las que rellenar

(c) David Miraut Andrés