TALLER DE PROGRAMACIÓN SOBRE GPUS

Facultad de Informática – Universidad Nacional de La Plata

Dr. Adrián Pousa

Optimizaciones CUDA

Agenda

I. *Introducción a las optimizaciones Nvidia CUDA*

II. *Optimizaciones Nvidia CUDA*

   I. *Relacionadas a la memoria*

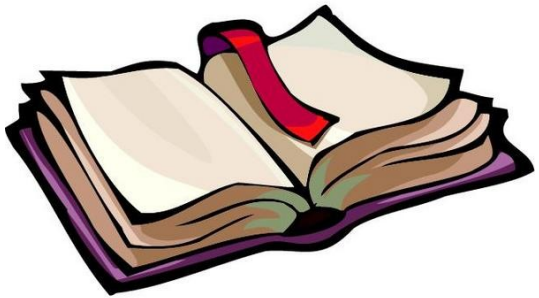
      I. *Acceso coalescente a memoria global y uso de memoria compartida*

      II. *Técnica de carga anticipada de datos (prefetching)*

   II. *Divergencia*

   III. *Mezcla y granularidad*

   IV. *Ocupación*



Agenda

I. ***Introducción a las optimizaciones Nvidia CUDA***

II. *Optimizaciones Nvidia CUDA*

   I. *Relacionadas a la memoria*

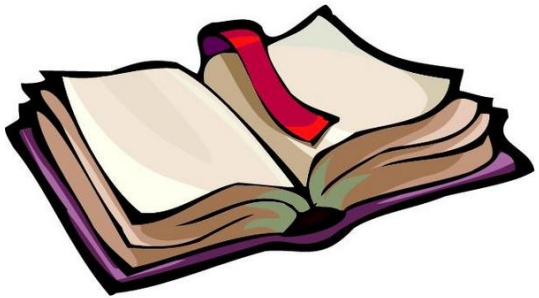
      I. *Acceso coalescente a memoria global y uso de memoria compartida*

      II. *Técnica de carga anticipada de datos (prefetching)*

   II. *Divergencia*

   III. *Mezcla y granularidad*

   IV. *Ocupación*



Introducción a las optimizaciones Nvidia CUDA

- Para alcanzar un buen rendimiento sobre GPUs es necesario aprovechar las ventajas de la arquitectura subyacente.
- Necesitamos conocer las características y factores limitantes en el rendimiento de las GPU.
- Existen distintas técnicas de optimización de una aplicación que abordan distintos aspectos:
  - Relacionadas a la memoria:
    - Organización de los accesos (**Coalescence**)
    - Técnicas de carga anticipada de datos (**Prefetching**).
  - Relacionadas a la ejecución de los threads (**Divergence**).
  - Relacionadas al rendimiento de las instrucciones: **mezcla** y **granularidad**.
  - Relacionadas a la asignación de recursos en un SM (**Occupancy**).

Agenda

I. *Introducción a las optimizaciones Nvidia CUDA*

II. ***Optimizaciones Nvidia CUDA***

   I. ***Relacionadas a la memoria***

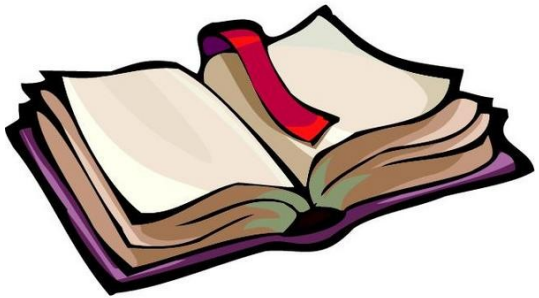
      I. ***Acceso coalescente a memoria global y uso de memoria compartida***

      II. *Técnica de carga anticipada de datos (prefetching)*

   II. *Divergencia*

   III. *Mezcla y granularidad*

   IV. *Ocupación*



Coalescencia

- El acceso a memoria global es uno de los aspectos a considerar cuando se quiere ajustar el rendimiento de una aplicación.
- La memoria global es DRAM: realizan lecturas simultáneas de una posición y sus vecinos más cercanos.
- La misma instrucción se ejecuta por todos los hilos del kernel, esto hace posible la optimización de los accesos a memoria global.
- El patrón de acceso óptimo a memoria se da cuando se realiza a posiciones consecutivas. Si el hardware detecta esto, organiza los accesos combinándolos en un único acceso (**coalescente**).

Coalescencia

- Suponer una matriz A de NxN:

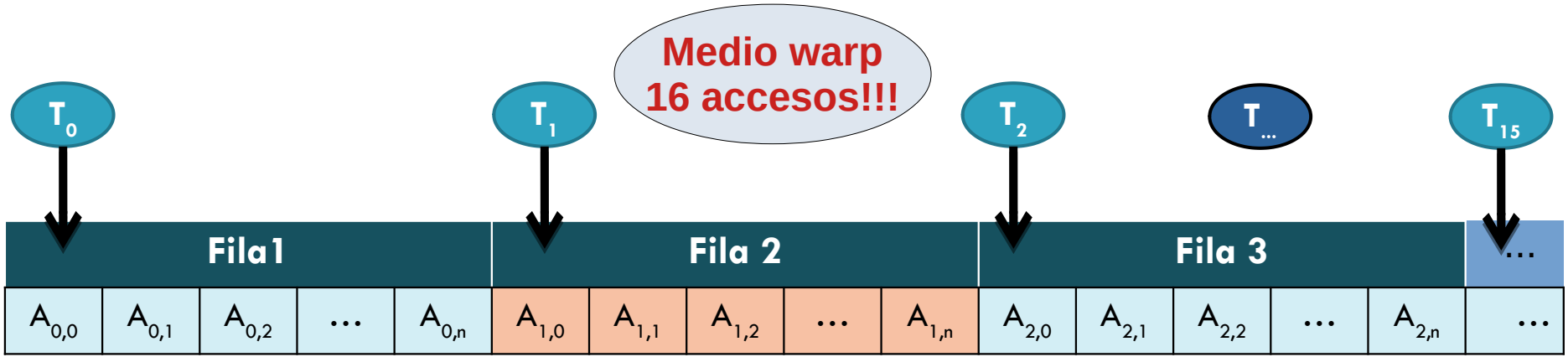
$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	…	$A_{0,N}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	…	$A_{1,N}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	…	$A_{2,N}$
…	…	…	…	…
$A_{N,0}$	$A_{N,1}$	$A_{N,2}$	…	$A_{N,N}$

- Almacenada por filas de la siguiente forma:

Fila1					Fila 2					Fila 3					…
$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	…	$A_{0,n}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	…	$A_{1,n}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	…	$A_{2,n}$	…

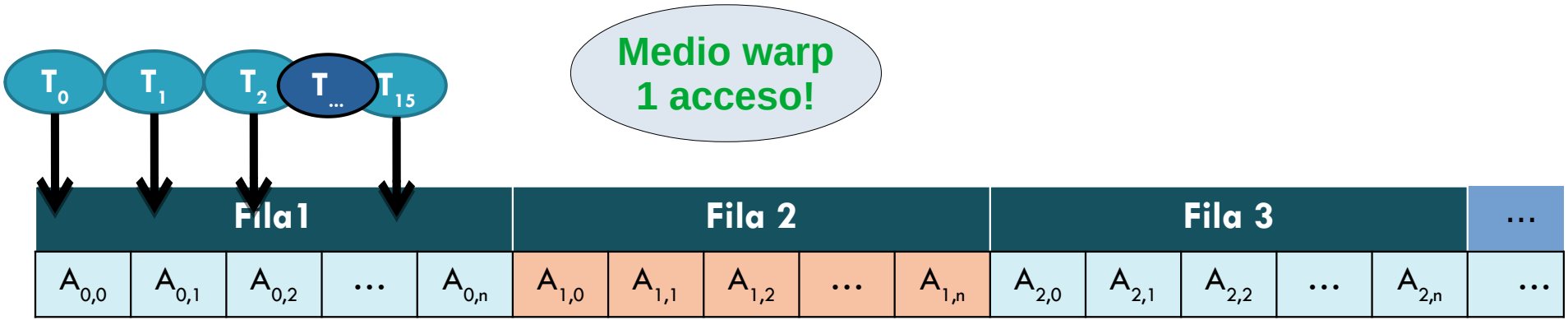
Coalescencia

- Suponer que cada hilo lee el primer elemento de cada fila.
- Por lo tanto, en la primera iteración cada hilo de un **medio warp** accederá a posiciones no consecutivas.
- El acceso es **NO COALESCENTE,** se hacen varios accesos **(16*N).**



Coalescencia

- Suponer que cada hilo lee el primer elemento de cada columna.
- Por lo tanto, en la primera iteración cada hilo de un **medio warp** accederá a posiciones consecutivas.
- El acceso es **COALESCENTE**, se hacen menos accesos (**N/16**).
- Los accesos se fusionan a uno por medio warp trayendo varios datos.



Coalescencia

- En resumen, para medio warp (16 hilos):
  - Si sus hilos acceden a posiciones de memoria no consecutivas (**NO coalescente**) se realizan 16 accesos.
  - Si sus hilos acceden a posiciones de memoria contiguas (**coalescente**), es decir el $Hilo_0$ del medio warp accede a la posición x, el $Hilo_1$ a la posición x+1 y así siguiendo hasta el $Hilo_{15}$ a la posición x+15, todos se unirán en un único acceso a la DRAM. Al mismo tiempo, se accede a varias posiciones de memoria obteniendo una velocidad de acceso cercana al ancho de banda teórico.

Coalescencia y Memoria compartida

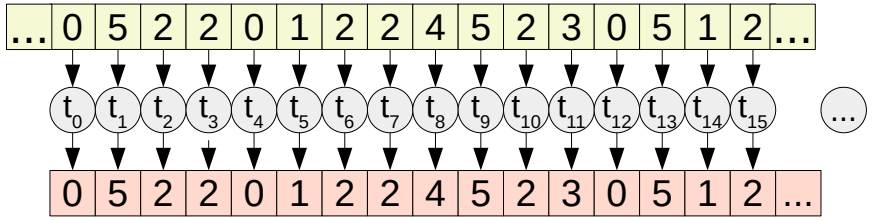
- Una optimización importante combina el acceso coalescente con el uso de la memoria compartida.
- La ejecución del kernel se divide en tres partes:
  1) Todos los hilos leen los datos a procesar coalescentemente desde memoria global y lo almacenan en la memoria compartida.
  2) Se hace el procesamiento sobre memoria compartida aprovechando que es una memoria muy rápida.
  3) Todos los hilos escriben los resultados obtenidos desde la memoria compartida a memoria global coalescentemente.

Coalescencia y Memoria compartida


```
__shared__ datatype datos_shared[];
__global__ void miKernel(datatype *datos_global, datatype *datos_salida_global){
 int idx = blockIdx.x * blockDim.x + threadIdx.x;
     //1) Acceder coalescentemente a la memoria global trayendo datos a memoria compartida
     for(int i=1;i<X;i++)
          datos_shared[indice shared]=datos_global[indice global];
     __syncthreads(); //Sincronización que asegura que los datos a procesar están listos

     "2) Procesamiento sobre memoria compartida"

     __syncthreads(); //Sincronización que asegura que se terminó de procesar
     //3) Acceden coalescentemente a la memoria global para escribir los resultados
     for(int i=0;i<X;i++)
          datos_salida_global[indice global]=datos_shared[indice shared];
}
```

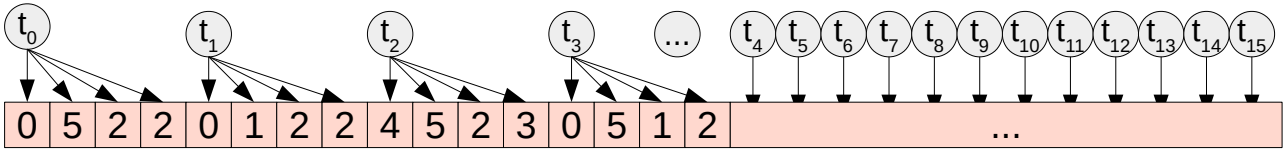
Coalescencia y Memoria compartida

Etapa 1
Los hilos leen los datos coalescentemente de memoria global a memoria compartida



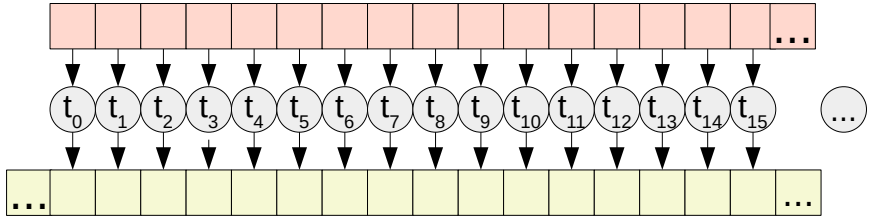
Barrera de sincronización

Etapa 2
Los hilos procesan los datos en memoria compartida.
Cada hilo puede procesar datos que en la etapa 1 trajo otro hilo.



Barrera de sincronización

Etapa 3
Los hilos escriben los resultados coalescentemente de memoria compartida a memoria global.



Coalescencia y Memoria compartida

Por qué sincronizamos varias veces con __syncthreads???

- La memoria compartida se comparte por todos los hilos de un bloque.
- Los hilos de un bloque se dividen en warps cuyo orden de ejecución se desconoce.
- Puede ocurrir que los datos que leen los hilos de un warp no sean los que procese sino que sean procesados por los hilos de otro warp!!!
- Cada warp deberá esperar a que los datos que debe procesar estén disponibles en memoria compartida. Por lo tanto, es necesaria la instrucción __syncthreads.
- __syncthreads es una barrera entre todos los hilos de un bloque.
- Lo mismo ocurre a la hora de transferir los resultados de memoria compartida a memoria global.

Coalescencia y Memoria compartida

¿Cómo se determina la longitud de arreglos en memoria compartida?

- **Estática:** Si se conoce de antemano la dimensión para cada bloque.
- **Dinámica:** Si no se conoce de antemano la dimensión para cada bloque.

Coalescencia y Memoria compartida

- **Estática** con valores constantes:

```
__global__ void Kernel(){
    __shared__ int a[100];
    __shared__ int b[4];
        …
}
```

- **Estática** con variables:

```
__global__ void Kernel(){
        __shared__ int a[DimA];
        __shared__ int b[DimB];
         …
}
```

Tener en cuenta que DimA y DimB deben conocerse en ejecución.

Coalescencia y Memoria compartida

No se permite pasar las dimensiones como parámetros del kernel !!!.

```
__global__ void Kernel(int DimA, int DimB){
        __shared__ int a[DimA];
        __shared__ int b[DimB];
        …
```

Coalescencia y Memoria compartida

- **Dinámica** a través de la invocación al kernel.
- La variable en memoria compartida se define como externa sin dimensión:

```
__global__ void Kernel(){
    extern __shared__ int a[];
     …
}
```

La invocación al kernel requiere un parámetro que indica la cantidad en bytes que utilizará cada bloque de hilo:

```
Kernel<<< gridDim, blockDim, a_sizeInBytesPerBlock >>>();
```

Coalescencia y Memoria compartida

- **Dinámica** con varias variables en memoria compartida.

- Los pasos a seguir son:
  - Calcular el total de bytes que requiere cada variable por cada bloque de hilos.
  - Pasar ese valor en la llamada al kernel.
  - Usar los parámetros para indicar el desplazamiento de cada variable.

Coalescencia y Memoria compartida


```
__global__ void miKernel(int count_a, int count_b){
extern __shared__ int memCompartida[];
 int *a = &memCompartida[0]; //Variable "a" al principio de la memoria compartida
 int *b = &memCompartida[count_a]; //Variable "b" al final de "a"
    …
}
main(){
 ...
 int bytesSharedMemPerBlock = count_a*sizeof(int) + size_b*sizeof(int);
 miKernel <<<numBlocks, threadsPerBlock, bytesSharedMemPerBlock>>> (count_a, count_b);
 ...
}
```

Agenda

I. *Introducción a las optimizaciones Nvidia CUDA*

II. ***Optimizaciones Nvidia CUDA***

   I. ***Relacionadas a la memoria***

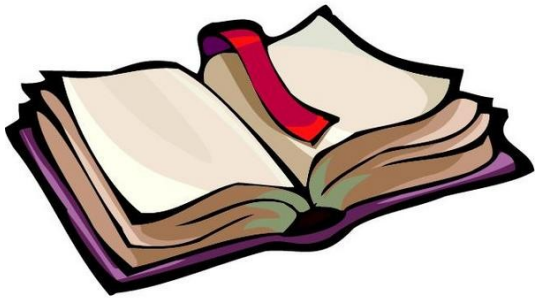
      I. *Acceso coalescente a memoria global y uso de memoria compartida*

      II. ***Técnica de carga anticipada de datos (prefetching)***

   II. *Divergencia*

   III. *Mezcla y granularidad*

   IV. *Ocupación*



Prefetching

- CUDA provee mecanismos para reducir la latencia de la memoria global.
- El modelo de **multithreading por hardware** permite que mientras unos warps esperan por una operación de acceso a memoria otros ejecuten sentencias menos costosas.
- Ejecutar otras sentencias entre accesos a memoria es posible si los hilos se programan de manera que entre las operaciones de accesos a memoria existan **instrucciones independientes,** las cuales se pueden ejecutar mientras se resuelve el acceso.
- Una **instrucción independiente** es aquella que, al momento de ejecutarse, los datos que necesita están disponibles.

Prefetching

- Para mejorar el rendimiento podemos asegurar que todos los datos necesarios para las operaciones estén presentes al momento de ejecutarlas.

- Esta técnica se denomina **prefetching** de datos (carga anticipada), y es otra forma de reducir la latencia de la memoria.

- La idea del **prefetching** es traer con antelación desde la memoria global los datos para trabajar en las operaciones de la siguiente iteración.

Prefetching – Ejemplo 1


```
for (i = 0; i < N; i++) {
    sum += array[i];
}
```

Cada suma espera que sus datos se carguen desde memoria.

Aplicando Prefetching

```
temp = array[0]; //1)
for (i = 0; i < N-1; i++){
 temp2 = array[i+1]; //2) 4)
 sum += temp; //3) 4)
 temp = temp2;
}
sum += temp;
```

1) Carga en **temp** el valor para la primer iteración.
2) Se ejecuta la instrucción que lee de memoria el valor para la iteración siguiente almacenándolo en **temp2**.
3) Mientras se resuelve el acceso (**temp2=array[i+1]**) hace la suma (**sum+=temp**) **en paralelo,** dado que **sum** y **temp** están en registro.
4) La suma y la transferencia se solapan en el tiempo.

Esto implica un uso más intensivo del banco de registros, aspecto a tener en cuenta dado lo limitado de su tamaño.

Prefetching – Ejemplo 2


```
for (int m = 0; m < (N/ Dim_sector); m++) {
  As[threadIdx.y][threadIdx.x] = A[i*N+m*Dim_sector+threadIdx.x];
  Bs[threadIdx.y][threadIdx.x] = B[(m*Dim_sector+threadIdx.y)*N + j];
  __syncthreads();
  for (int k = 0; k <Dim_sector; k++)
     C += As[i][k] * B[k][j];
  __syncthreads();
}
```

Iteración
Carga un bloque de mem global a mem compartida
Sincronización
Calcula el bloque
Sincronización

Prefetching – Ejemplo 2

- Las siguientes dos líneas tienen dos partes dependientes entre si:

```
As[threadIdx.y][threadIdx.x] = A[i*N+m*Dim_sector+threadIdx.x];
Bs[threadIdx.y][threadIdx.x] = B[(m*Dim_sector+threadIdx.y)*N + j];
```

Carga un bloque de mem global a mem compartida

  - Se accede a la memoria global para LEER las posiciones de A y B.
  - Se accede a la memoria compartida para ESCRIBIR los datos leídos.
- Los warps deben esperar a que el acceso se resuelva antes de continuar con otra operación.

Prefetching – Ejemplo 2

Aplicando Prefetching	
`float Ra= A[i*N+threadIdx.x];` `float Rb= B[(threadIdx.y)*N + j];`	Carga un bloque de mem global en registros
`for (int m = 0; m < (N/ Dim_sector); m++) {`	Iteración
`As[threadIdx.y][threadIdx.x] = Ra ;` `Bs[threadIdx.y][threadIdx.x] = Rb;`	Guarda registros en mem compartida
`Ra= A[i*N+(m+1)*Dim_sector+threadIdx.x];` `Rb= B[((m+1)*Dim_sector+threadIdx.y)*N + j];`	Carga un bloque de mem global en registros para la próxima iteración
`__syncthreads();`	Sincronización
`for (int k = 0; k <Dim_sector; k++)` `C += As[i][k] * Bs[k][j];`	Calcula el bloque
`__syncthreads();`	Sincronización
`}`	

Prefetching – Ejemplo 2

- Estas modificaciones permiten reducir la inactividad del sistema, mientras unos hilos estén inactivos esperando que se completen sus accesos a memoria, otros lo están solicitando. Cuando todos completan la lectura, pasan la sincronización, operan los datos presentes en la memoria compartida e inician de nuevo la carga de los datos.

- En la siguiente iteración los últimos elementos cargados se convierten en los actuales y se procede a una nueva carga.

Agenda

I. *Introducción a las optimizaciones Nvidia CUDA*

II. ***Optimizaciones Nvidia CUDA***

   I. *Relacionadas a la memoria*

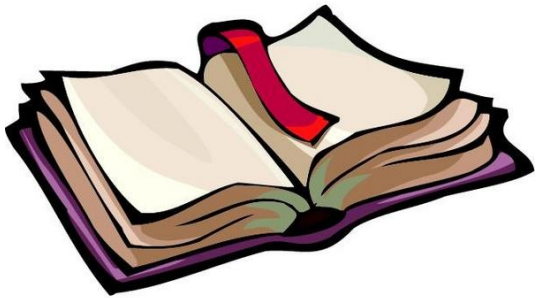
      I. *Acceso coalescente a memoria global y uso de memoria compartida*

      II. *Técnica de carga anticipada de datos (prefetching)*

   II. ***Divergencia***

   III. *Mezcla y granularidad*

   IV. *Ocupación*



Divergencia – Construcción de warps

- Los hilos se asignan a warps según su identificador:
  - Si el bloque es 1D la asignación es directa, según threadIdx.x.
  - Si el bloque es 2D, primero se asignan los hilos con igual identificador threadIdx.y y en orden ascendente por threadIdx.x.
  - Si el bloques es 3D se asignan los hilos con igual identificador threadIdx.z, luego se procede igual que en 2D.
- Si el número de hilos no es múltiplo de 32 el último warp se completa con hilos adicionales.

Divergencia

- Todos los hilos de un warps ejecutan la misma instrucción (SIMT).
- El máximo paralelismo en un warp se alcanza si sus hilos no divergen.
- Una divergencia ocurre cuando los hilos de un warp ejecutan sentencias condicionales y no todos los hilos siguen el mismo camino.

```
if cond
 Sentencias if
else
  Sentencias else
```

- Si existe divergencia la ejecución del *warp se divide en* pasos, uno para cada camino, y cada paso se ejecuta secuencialmente.
- Dependiendo del algoritmo se podría organizar la ejecución de manera que los hilos de un mismo warp sigan todos el mismo camino.

Divergencia

- **Problema de ejemplo:** sumar los elementos de un vector.
- Suponer un escenario hipotético:
  - *Un vector de 8 elementos (N=8).*
  - *Un warp de 4 hilos.*
  - *Un sólo bloque unidimensional de 8 hilos (2 warp).*
  - *El vector está en memoria compartida, por lo tanto lo comparten todos los hilos del bloque.*
  - *El SM ejecuta de a un warp.*

Divergencia

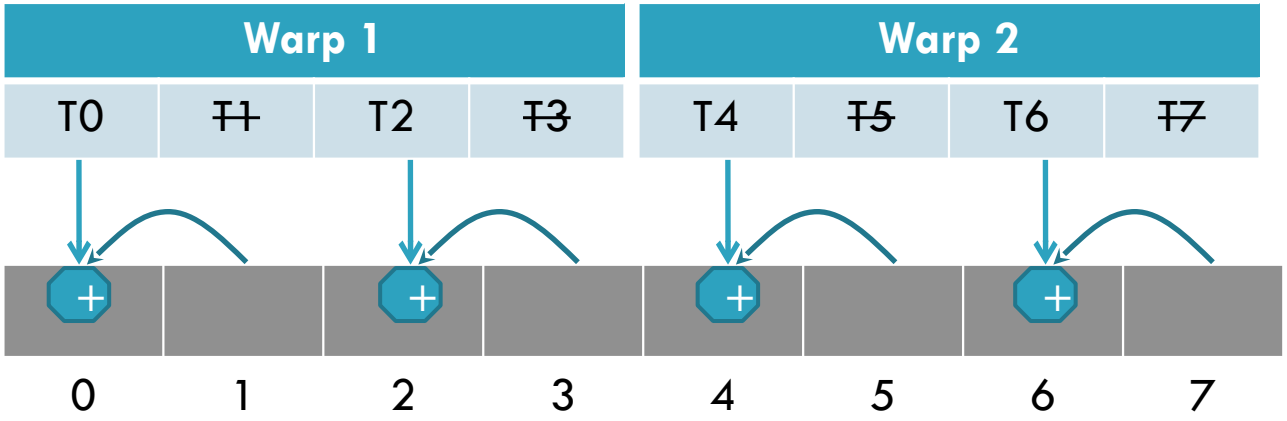
- La solución involucra al menos tres iteraciones.
- El código tiene la siguiente forma:

```
for ( it=1 ;  it <= nrIteraciones ; it++ ){

     if( threadIdx.x % (2^it) == 0 )
          "suma";
     __syncthreads();
}
```

- La divergencia se encuentra en el if.

Divergencia

- En la primera iteración, los hilos que cumplen con la condición del if son los hilos pares y el 0. Cada hilo accede a la posición dada por su threadIdx.x y lo suma a la siguiente (los hilos impares no trabajan). El resultado de cada suma quedará en la posición del primer operando.

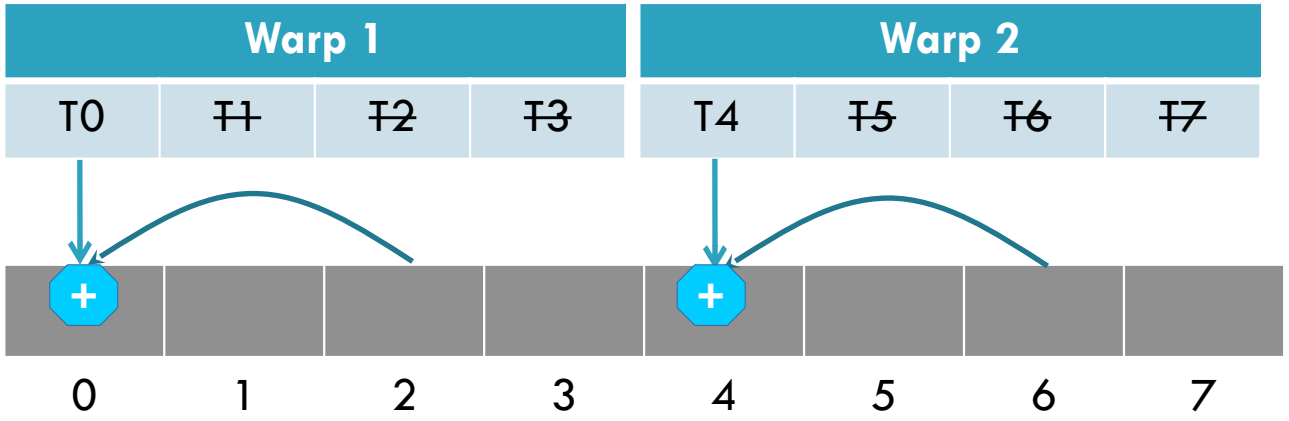


Divergencia

- En la primer iteración los dos warps divergen.
- *En esta iteración, la ejecución del bloque en función del if se da en 4 pasos (2 pasos para el primer warp y 2 para el segundo):*
  - *Paso 1: T0 y T2 (suman datos).*
  - *Paso 2: T1 y T3 (no hacen nada).*
  - *Paso 3: T4 y T6 (suman datos).*
  - *Paso 4: T5 y T7 (no hacen nada).*

Divergencia

- En la segunda iteración, los hilos que cumplen con la condición del if son un hilo de cada warp, el thredIdx.x = 0 y el threadIdx.x = 4. El resultado quedará en la posición del primer operando.

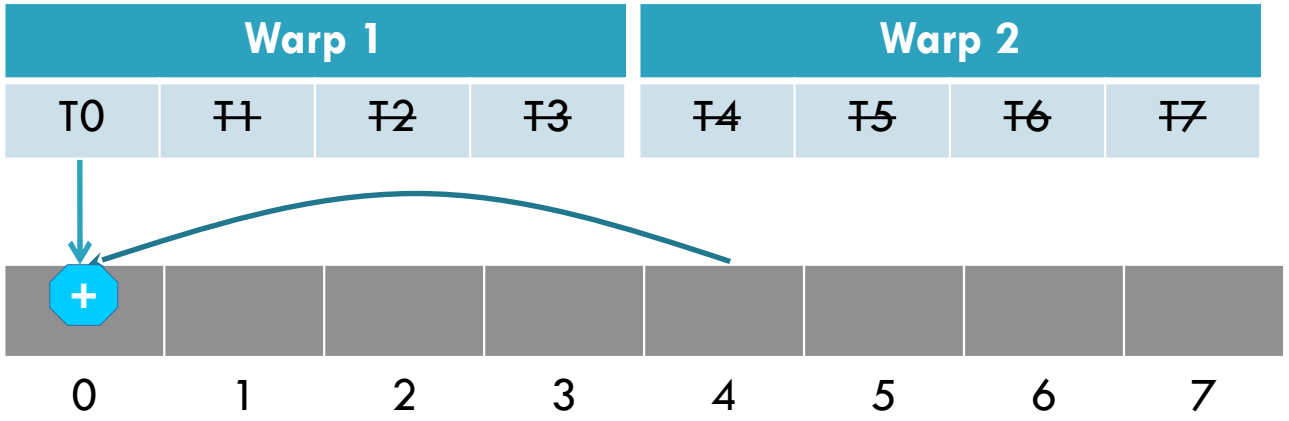


Divergencia

- En la segunda iteración también divergen los dos warps.
- *La ejecución del bloque para esta iteración en función del if también se da en 4 pasos:*
  - *Paso 1: T0 (suma datos).*
  - *Paso 2: T1, T2 y T3 (no hacen nada).*
  - *Paso 3: T4 (suma datos).*
  - *Paso 4: T5, T6 y T7 (no hacen nada).*

Divergencia

- En la tercer iteración, solo el thread con threadIdx.x = 0 sumará y dejará el resultado en la primera posición.



Divergencia

- En la tercera iteración el primer warp diverge pero el segundo warp ya no.
- *En esta iteración la ejecución del bloque en función de los if se da en 3 pasos:*
  - *Paso 1: T0 (suma datos).*
  - *Paso 2: T1, T2 y T3 (no hacen nada).*
  - *Paso 3: T4, T5, T6 y T7 (no hacen nada).*

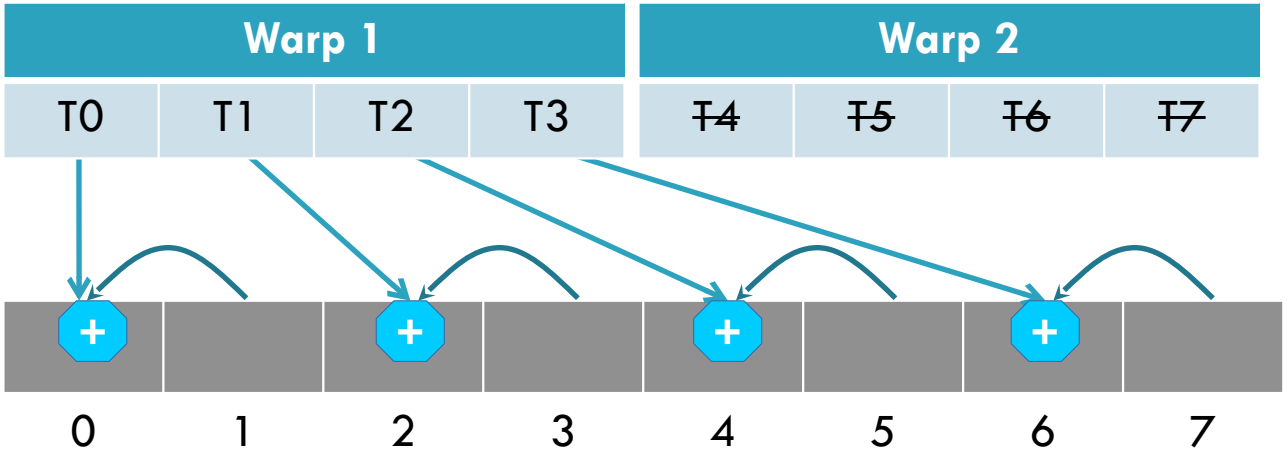
Divergencia

- Para reducir el numero de pasos debido a divergencia es necesario que los hilos que pertenecen a un mismo warp sigan todos el mismo camino.
- Para el problema de ejemplo se puede lograr modificando el código como sigue:

```
for ( it=0 ;  it < nrIteraciones ; it++ ){

     if( threadIdx.x < N/2^(it+1) )
          "suma";
     __syncthreads();
}
```

Divergencia

- En la primera iteración, los hilos que cumplen con la condición del if son los hilos del mismo warp, los hilos del otro warp no trabajarán.



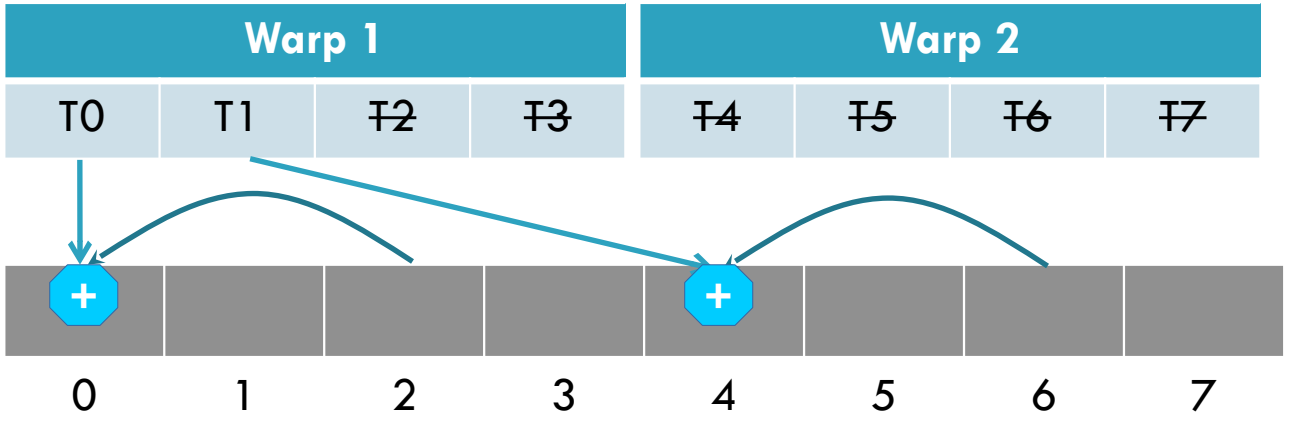
Divergencia

- Ahora, en la primera iteración ningún warp diverge.
- *En esta iteración, la ejecución del bloque en función del if se da en 2 pasos:*
  - *Paso 1: T0, T1, T2 y T3 (suman datos).*
  - *Paso 2: T4, T5, T6 y T7 (no hacen nada).*

- *Esto es una ventaja con respecto a la divergencia de la primera iteración en la primer solución donde se requerían 4 pasos.*

Divergencia

- En la segunda iteración, los hilos que cumplen con la condición del if son los dos primeros del primer warp (thredIdx.x = 0 y threadIdx.x = 1).

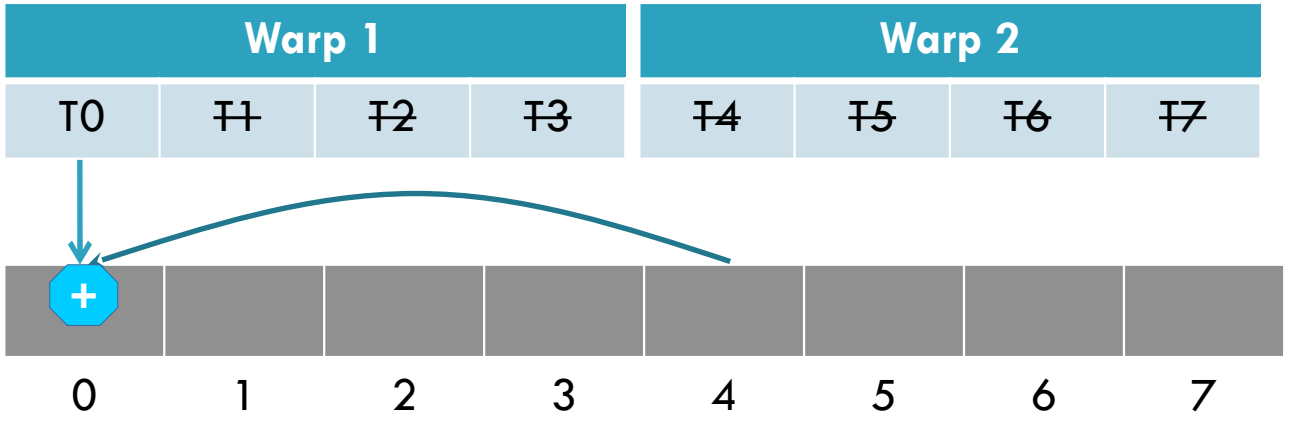


Divergencia

- En la segunda iteración el primer warp diverge y el segundo no.
- *En esta iteración la ejecución del bloque en función del if se da en 3 pasos:*
  - *Paso 1: T0 y T1 (suman datos).*
  - *Paso 2: T2 y T3 (no hacen nada).*
  - *Paso 3: T4, T5, T6 y T7 (no hacen nada).*
- *Nuevamente tenemos una ventaja con respecto a la divergencia de la segunda iteración en la primer solución donde se requerían 4 pasos.*

Divergencia

- En la tercer iteración, sólo el thread con threadIdx.x = 0 sumará y dejará el resultado en la primer posición.
- Esto es igual a la primer solución por lo tanto la cantidad de pasos en este caso se mantiene en 3.



Divergencia

- Conclusión:

  - La primer solución realiza la ejecución del bloque en **11 pasos** (4 de la primera iteración, 4 de la segunda y 3 de la tercera).

  - La segunda solución realiza la ejecución del bloque en **8 pasos** (2 de la primera iteración, 3 de la segunda y 3 de la tercera) aumentando el grado de paralelismo.

Agenda

I. *Introducción a las optimizaciones Nvidia CUDA*

II. ***Optimizaciones Nvidia CUDA***

   I. *Relacionadas a la memoria*

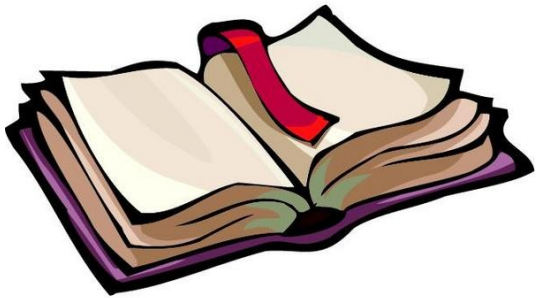
      I. *Acceso coalescente a memoria global y uso de memoria compartida*

      II. *Técnica de carga anticipada de datos (prefetching)*

   II. *Divergencia*

   III. ***Mezcla y granularidad***

   IV. *Ocupación*



Mezcla y granularidad

- No todas las instrucciones implican el mismo tiempo.
- Las operaciones más costosas:
  - Las operaciones de punto flotante.
  - Las instrucciones de acceso a memoria.
  - Las sentencia de branch (iteraciones for).
- Una sentencia de iteración implica varias instrucciones extras: una para la evaluación de la condición (sentencia branch) y otra para la actualización del contador.
- Bajo ciertas condiciones algunas pueden evitarse.

Mezcla y granularidad

- El siguiente código tiene una iteración con varios tipos de instrucciones:

- En la estructura de control (iteración for):
  - Una instrucción que incrementa el contador de la iteración k.
  - Una operación branch (condición de for).

- En la operación (mezcla de instrucciones):
  - Dos instrucciones de punto flotante, una para el producto y otra para la suma.
  - Dos instrucciones enteras al usar k en el índice (desplazamientos).

```
__shared__ float As[N][N];
__shared__ float Bs[N][N];
__global__ miKernel(…){
  float C = 0;
  …
    for (int k = 0; k < N; k++)
      C += As[i][k] * Bs[k][j];
  …
}
```

Mezcla y granularidad

- Cuando existe mezcla de instrucciones unas pocas son de punto flotante útiles para el cálculo. Estas deben competir por el ancho de banda del procesador con instrucciones de control (for) limitando el rendimiento.
- Siempre que la aplicación lo permita, podemos evitar la mezcla de instrucciones tratando de utilizar el máximo ancho de banda para procesamiento de instrucciones útiles.
- Si el programador conoce los índices y límites de la iteración de forma estática, se puede evitar la mezcla de instrucciones eliminando la iteración.

Sin Optimizar	Optimizado
`for (int k = 0; k < N; k++)` `  C += As[i][k]*Bs[k][j];`	`C = As[i][0]*Bs[0][j] + As[i][1]*Bs[1][i]+…+ As[i][N]*Bs[j][N];`

Mezcla y granularidad

- En general, para maximizar el rendimiento de las aplicaciones respecto a la mezcla de instrucciones se debe:
  - Minimizar el uso de instrucciones aritméticas con bajo rendimiento.

    CUDA provee instrucciones propias que pueden utilizarse en lugar de las instrucciones regulares, como también funciones de simple precisión en lugar de doble precisión.
  - Reducir el número de instrucciones evitando la mezcla, en particular cuando existen operaciones muy costosas.
- Idealmente, podemos contar con herramientas de software que realicen estas mejoras, por ejemplo: compiladores que traduzcan una iteración a una instrucción equivalente.

Mezcla y granularidad

- CUDA provee una directiva para el desenrollado de bucles:

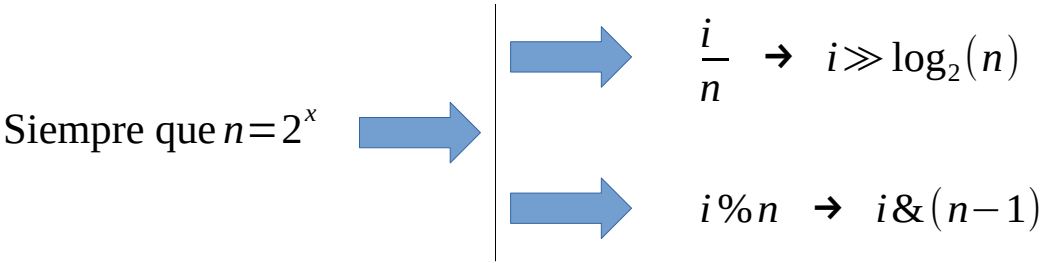
```
__global__ void kernel(float *b, int size){
    int tid = blockDim.x * blockIdx.x + threadIdx.x;

    #pragma unroll
    for(int i = 0; i < size; i++)
        b[i]=i;
}
```

Equivalente

```
b[0]=0;
b[1]=1;
b[2]=2;
 …
b[size] = size;
```

Mezcla y granularidad

- Equivalencia de instrucciones:



Mezcla y granularidad

- Exponenciación/Raíces se resuelven con la función *pow()* *(costosa)*.
- Alternativas a exponenciación pow(x,y):
  - En base 2: *expf2()*
  - En base 10: *expf10()*
- Alternativas a raíces cúbicas pow(x,1/3):
  - Positiva: *cbrtf()*
  - Negativa (-1/3): *rcbrtf()*

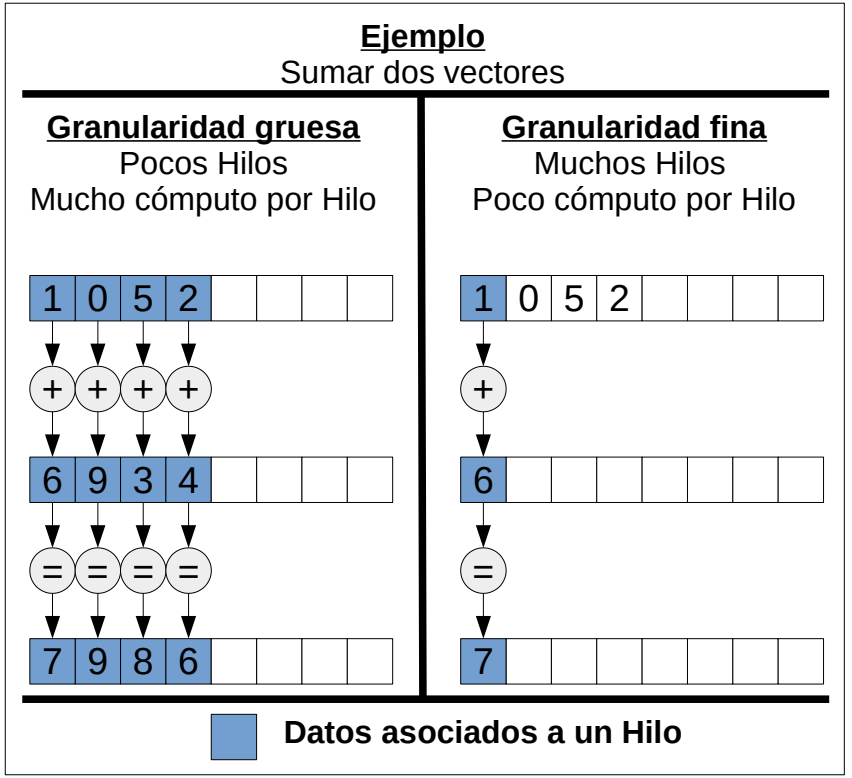
Mezcla y granularidad

- Algunas equivalencias para evitar el uso de pow:

$x^{\frac{1}{9}}$	r = rcbrt(rcbrt(x))
$x^{-\frac{1}{9}}$	r = cbrt(rcbrt(x))
$x^{\frac{2}{3}}$	r = cbrt(x); r = r*r
$x^{\frac{7}{6}}$	r = x*rcbrt(rsqrt(x))
$x^{-\frac{7}{6}}$	r = (1/x) * rcbrt(sqrt(x))

Mezcla y granularidad

- Al desarrollar una aplicación sobre GPU surge la inquietud de cuánto trabajo se asigna a cada hilo (granularidad).
- No existe una metodología para determinar la mejor granularidad de un hilo, se debe evaluar en función de las características de la aplicación y del kernel.
- Sin embargo, las aplicaciones en GPU son, por naturaleza y por las características de la arquitectura, de **grano fino**.



Agenda

I. *Introducción a las optimizaciones Nvidia CUDA*

II. ***Optimizaciones Nvidia CUDA***

   I. *Relacionadas a la memoria*

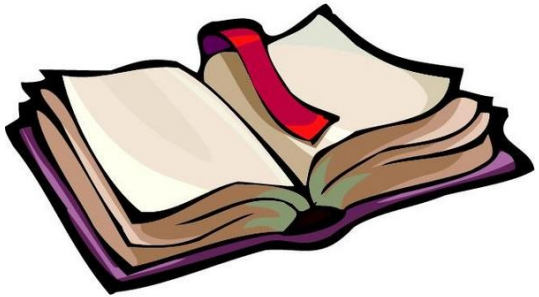
      I. *Acceso coalescente a memoria global y uso de memoria compartida*

      II. *Técnica de carga anticipada de datos (prefetching)*

   II. *Divergencia*

   III. *Mezcla y granularidad*

   IV. ***Ocupación***



Ocupación

- Un SM tiene diferentes recursos que deben distribuirse entre los distintos bloques de hilos:
  - Slots para los bloques (Cantidad de bloques por SM)
  - Slots de hilos (Cantidad de hilos por SM)
  - Registros
  - Memoria compartida
- La cantidad de estos recursos dependen de la capability de la arquitectura que determina los límites de estos y puede influir en el rendimiento de las aplicaciones.

	G80	GT200	GF100
Slots* para *Bloques (Cantidad de bloques por SM)	8		
Slots* para *Threads (Cantidad de hilos por SM)	768	1024	1536
Registros	8KB	16KB	32KB
Memoria Compartida	16KB		48KB

Ocupación – Ejemplo Slots

- En una arquitectura G80 un SM soporta 768 hilos y 8 bloques.

	G80	GT200	GF100
Slots para *Bloques (Cantidad de bloques por SM)*	8		
Slots para *Threads (Cantidad de hilos por SM)*	768	1024	1536
Registros	8KB	16KB	32KB
Memoria Compartida	16KB		48KB

- Si se definen bloques de 256 hilos, sólo 3 bloques podrán residir en el SM al mismo tiempo. (256*3 = 768)
- Si se definen bloques de 128 hilos, 6 bloques podrán residir en el SM al mismo tiempo. (128*6 = 768)

Ocupación – Ejemplo Slots

	G80	GT200	GF100
Slots para *Bloques (Cantidad de bloques por SM)*	8		
Slots para *Threads (Cantidad de hilos por SM)*	768	1024	1536
Registros	8KB	16KB	32KB
Memoria Compartida	16KB		48KB

- Si se definen bloques de 512 hilos, sólo 1 bloque podrá residir en el SM ya que 2 o más bloques superaran los 768 hilos que soporta la arquitectura. En este caso se desperdician 256 hilos que la arquitectura podría soportar.
- Si se definen bloques de 32 hilos, sólo 8 bloques podrán residir en el SM. En este caso se desperdician 512 hilos ya que habrá sólo 8*32 = 256 hilos en ejecución.

Ocupación - Slots

- En teoría, una asignación correcta es la que ocupa la mayor cantidad de bloques por SM y al mismo tiempo la mayor cantidad de hilos por SM.
- En el ejemplo, se deben definir bloques de 96 hilos entonces se podrá cumplir con el límite de 8 bloques residiendo al mismo tiempo en el SM ($96*8 = 768$).
- En la práctica, esta configuración podría no ser la mejor. Dependerá del problema, la relación "cómputo/acceso a memoria" y de la asignación de otros recursos (registros y memoria compartida).
- La mejor asignación de hilos por bloque deberá obtenerse empíricamente.

Ocupación - Registros

- Los registros almacenan todas las variables automáticas escalares de cada hilo.
- Se deben distribuir entre todos los bloques residentes en el SM, por lo tanto es necesario tener en cuenta la cantidad de registros a la hora de definir la cantidad de bloques y de hilos por bloques.

	G80	GT200	GF100
Slots para *Bloques (Cantidad de bloques por SM)*	8		
Slots para *Threads (Cantidad de hilos por SM)*	768	1024	1536
Registros	**8KB**	**16KB**	**32KB**
Memoria Compartida	16KB		48KB

Ocupación – Ejemplo Registros

- La arquitectura GT200 soporta 16KB (16384) registros por SM.

	G80	GT200	GF100
Slots para *Bloques (Cantidad de bloques por SM)*	8		
Slots para *Threads (Cantidad de hilos por SM)*	768	1024	1536
Registros	**8KB**	**16KB**	**32KB**
Memoria Compartida	16KB		48KB

- Si un problema usa 9 registros por hilo y se tienen 256 hilos por bloque, cada bloque necesita $9*256 = 2304$ registros.
- Un SM puede satisfacer las necesidades de registros de 7 bloques ($7*2304 = 16128$) quedando 256 registros libres.

Ocupación – Ejemplo Registros

	G80	GT200	GF100
Slots para *Bloques (Cantidad de bloques por SM)*	8		
Slots para *Threads (Cantidad de hilos por SM)*	768	1024	1536
Registros	**8KB**	**16KB**	**32KB**
Memoria Compartida	16KB		48KB

- Si una nueva implementación requiere 11 registros por hilo y se tienen 256 hilos por bloque, cada bloque necesita $11*256 = 2816$ registros. Ahora, un SM sólo puede satisfacer las necesidades de registros de 5 bloques ($5*2304 = 11520$) restando 2304 registros sin utilizar.
- Esto muestra un débil equilibrio. A pequeñas modificaciones (2 registros más por hilo) se reduce el paralelismo (7 a 5 bloques).

Ocupación – Memoria compartida

- Cada SM tiene una capacidad máxima de memoria compartida.
- El total de memoria compartida se distribuye entre los bloques residentes en el SM.

	G80	GT200	GF100
Slots para *Bloques (Cantidad de bloques por SM)*	8		
Slots para *Threads (Cantidad de hilos por SM)*	768	1024	1536
Registros	8KB	16KB	32KB
Memoria Compartida	**16KB**		**48KB**

Ocupación – Ejemplo Memoria compartida

- Si un problema requiere 2KB de memoria compartida por bloque.
- Las arquitecturas G80 y GT200 soportarán 8 bloques por SM (8*2KB=16KB).
- La arquitectura GF100 también soportará 8 bloques por SM y quedarán 32KB sin utilizar.

	G80	GT200	GF100
Slots para *Bloques (Cantidad de bloques por SM)*	8		
Slots para *Threads (Cantidad de hilos por SM)*	768	1024	1536
Registros	8KB	16KB	32KB
Memoria Compartida	**16KB**		**48KB**

Ocupación – Ejemplo Memoria compartida

- Si un problema requiere 4KB de memoria compartida por bloque.
- Las arquitecturas G80 y GT200 soportarán sólo 4 bloques por SM (4*4KB=16KB) desperdiciando 4 bloques (8 por la capability).
- La arquitectura GF100 soportará 8 bloques por SM (8*4KB=32KB) y quedarán 18KB sin utilizar.

	G80	GT200	GF100
Slots para *Bloques (Cantidad de bloques por SM)*	8		
Slots para *Threads (Cantidad de hilos por SM)*	768	1024	1536
Registros	8KB	16KB	32KB
Memoria Compartida	**16KB**		**48KB**

Ocupación

- Cada uno de los recursos puede determinar una cantidad máxima de bloques por SM, la cual puede no coincidir con la determinada por otro recurso.
- Queda en el programador analizar los recursos necesarios y tomar las decisiones en función de maximizar el rendimiento de la aplicación. (Puede ser una tarea compleja).
- No existe una forma ideal de asignar recursos, existe una propuesta en un paper de ACM: "*Optimization principles and application performance evaluation of a multithreaded GPU using CUDA*".

Trade-off Ocupación-Granularidad

- Usar **granularidad fina** permite utilizar pocos recursos.
- Se debe intentar obtener una organización de hilos que asegure la mayor cantidad de bloques por SM y al mismo tiempo la mayor cantidad de hilos por SM.
- Si el problema es mayor a la cantidad de hilos generados entonces se debería incrementar el grid.
- Aunque el SM no va a poder ejecutar todos los bloques del grid al mismo tiempo, cada vez que ejecute un conjunto de bloques va a ser el conjunto que le brinde mayor concurrencia-paralelismo.