

Sistemas Distribuidos y Paralelos

Ingeniería en Computación



Optimizaciones CUDA

Universidad Nacional de La Plata



Facultad de Informática



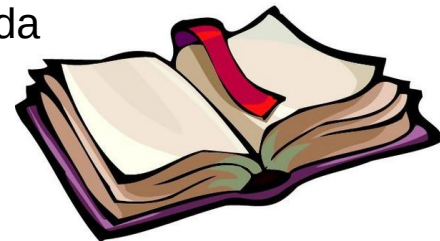
Agenda

2

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



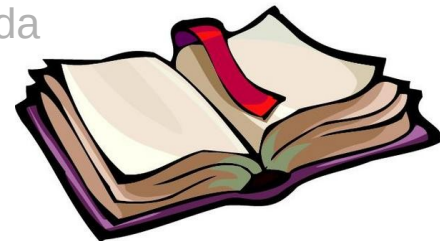
Agenda

3

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

4

- 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**).

I. Introducción a las optimizaciones Nvidia CUDA

II. Optimizaciones Nvidia CUDA

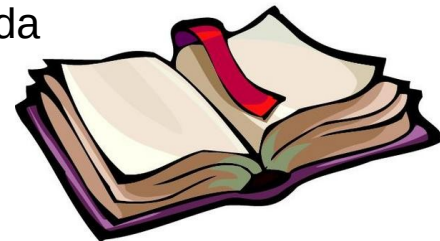
- i. Relacionadas a la memoria
 - i. Acceso coalescente a memoria global y uso de memoria compartida
 - ii. Técnica de carga anticipada de datos (prefetching)
- ii. Divergencia
- iii. Mezcla y granularidad
- iv. Ocupación



I. Introducción a las optimizaciones Nvidia CUDA

II. Optimizaciones Nvidia CUDA

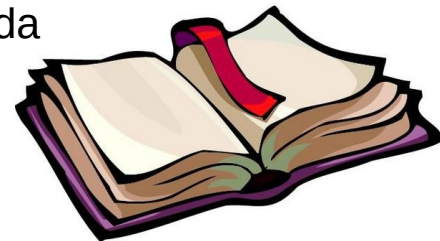
- i. Relacionadas a la memoria
 - i. Acceso coalescente a memoria global y uso de memoria compartida
 - ii. Técnica de carga anticipada de datos (prefetching)
- ii. Divergencia
- iii. Mezcla y granularidad
- iv. Ocupación



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

8

- El acceso a memoria global es uno de los aspectos a considerar cuando se quiere ajustar el rendimiento de una aplicación en GPU.
- 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** se logra cuando se realiza a **posiciones consecutivas**. En este caso, el hardware organiza los accesos combinándolos en un único acceso (**coalescente**).

Coalescencia

9

- Suponer una matriz A de $N \times N$ elementos almacenada en memoria global por filas:

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

Fila 1					Fila 2					Fila 3					...
$A_{0,0}$	$A_{0,1}$	$A_{0,2}$...	$A_{0,n}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$...	$A_{1,n}$	$A_{2,0}$	$A_{2,1}$	$A_{2,2}$...	$A_{2,n}$...

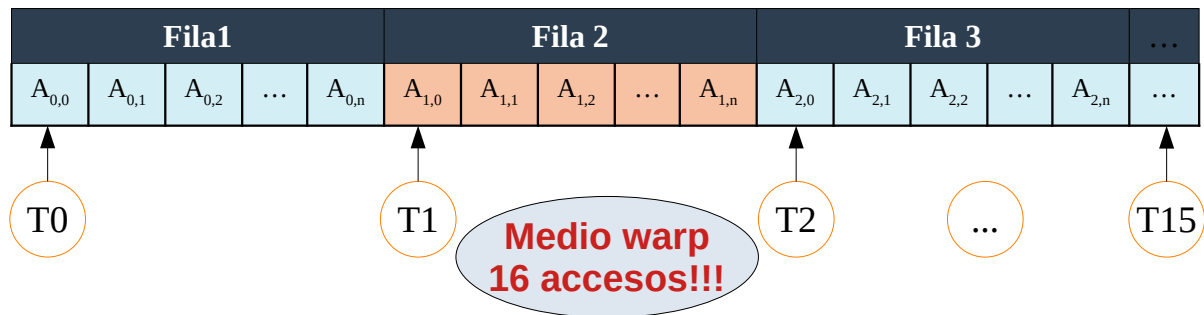
- Suponer además que va a ser accedida por completo por medio warp (16 hilos).
- Recordar que el acceso de medio warp a memoria global es lento pero, dependiendo del patrón de acceso, el hardware puede hacerlo más eficiente.

Coalescencia

10

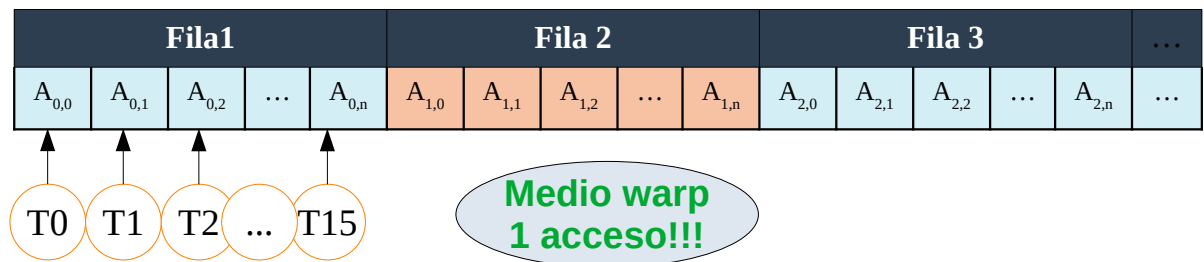
Acceso NO Coalescente

Cada hilo lee los elementos por filas.
Cada hilo de un medio warp accede a **posiciones no contiguas**.
El hardware hace varios accesos (16 en el peor caso)
El total de accesos para toda la matriz:
 $N*N$



Acceso Coalescente

Cada hilo lee los elementos por columnas.
Cada hilo de un medio warp accede a **posiciones contiguas**.
El **hardware unifica** los accesos (1).
El total de accesos para toda la matriz:
 $(N*N)/16$



Coalescencia y memoria compartida

11

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

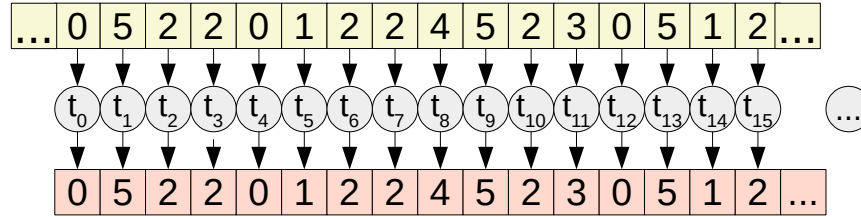
Coalescencia y memoria compartida

12

Etapa 1

Lectura coalescente

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



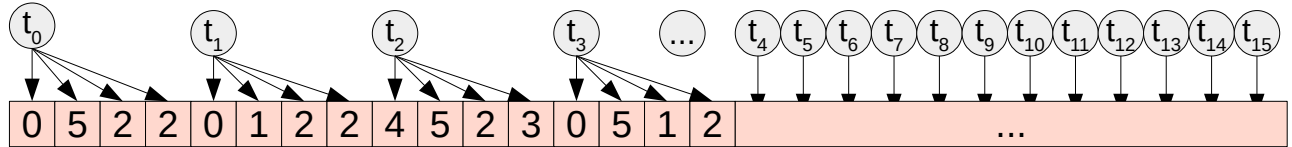
Barrera de sincronización

`_Syncthreads();`

Etapa 2

Procesamiento

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



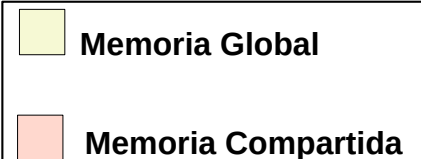
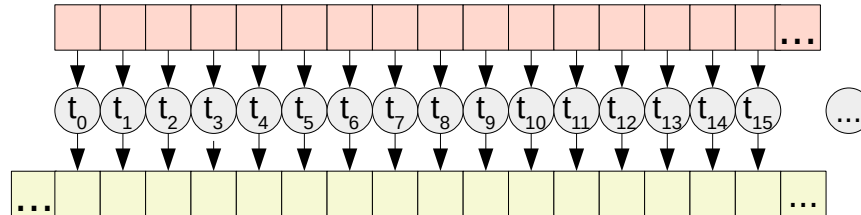
Barrera de sincronización

`_Syncthreads();`

Etapa 3

Escritura coalescente

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



Coalescencia y memoria compartida

Código de ejemplo

13

```
__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

Función `__syncthreads()`

14

Por qué sincronizamos varias veces con `__syncthreads`???

- La **memoria compartida** la comparten 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 traen los hilos de un warp no sean procesados por los hilos de este warp sino 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

Gestión de la memoria compartida

15

- Generalmente las variables en memoria compartida son arreglos
- La longitud del arreglo depende de la cantidad de datos sobre lo que trabajará cada **bloque de hilos**

¿Cómo determinamos la longitud de arreglos en memoria compartida?

- Existen dos formas de hacerlo:
 - **Estática:** Si se conoce a priori la dimensión para cada bloque.
 - **Dinámica:** Si no se conoce a priori la dimensión para cada bloque.

Coalescencia y memoria compartida

Arreglos en memoria compartida – Declaración Estática

16

Declaración estática con valores constantes

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

Declaración estática a partir de variables

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

**DimA y DimB deben conocerse
en tiempo de ejecución.**

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

Arreglos en memoria compartida – Declaración Dinámica

17

Declaración dinámica en la invocación al kernel Una variable

La variable en memoria compartida se define como externa sin dimensión:

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

La dimensión se determina en la invocación al kernel. Se requiere un parámetro adicional en la llamada que indica la cantidad en bytes que utilizará cada bloque de hilos:

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

Coalescencia y memoria compartida

Arreglos en memoria compartida – Declaración Dinámica

18

Declaración dinámica en la invocación al kernel Varias variables

Si se requiere más de una variable en memoria compartida se deben seguir los siguientes pasos:

- 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 de la función para indicar el desplazamiento de cada variable.

```
__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);
    ...
}
```

Coalescencia y memoria compartida

Visibilidad de una variable en memoria compartida

19

- Una variable definida en memoria compartida es la misma para todos los hilos del mismo bloque.
- Por ejemplo:

```
__global__ void Kernel() {  
    __shared__ int a[100];  
  
    ...  
}
```

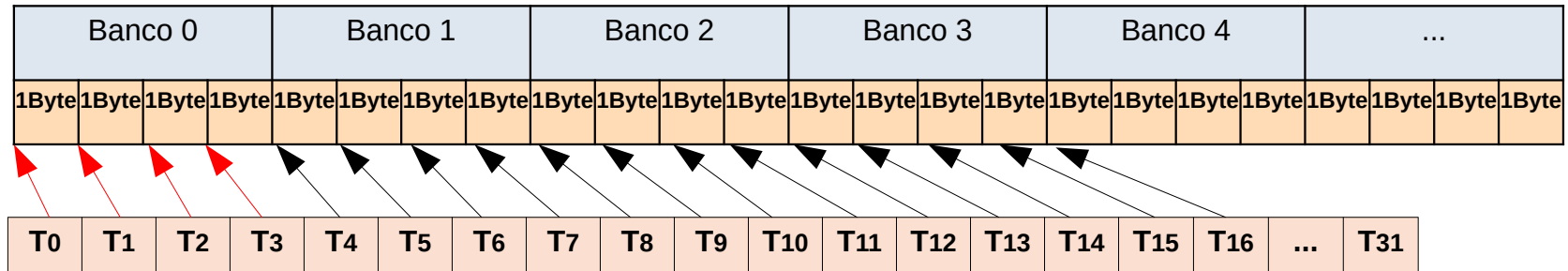
Todos los hilos de un mismo bloque ven el mismo arreglo **a** de 100 posiciones. Si un hilo de ese bloque cambia el valor de una posición de **a**, por ejemplo **a[3]=4**, todos los hilos del mismo bloque verán ese cambio, es decir para todos los hilos de ese bloque **a[3]=4**.

Coalescencia y memoria compartida

Gestión de la memoria compartida - Bancos

20

- La memoria compartida se organiza en bancos de 1KB donde palabras de 32 bits (4 Bytes) consecutivas pertenecen a distintos bancos.
- El uso de memoria compartida requiere considerar los conflictos de bancos.
- Un **conflicto de bancos** implica accesos simultáneos de distintos hilos de un warp a direcciones diferentes del mismo banco de memoria.

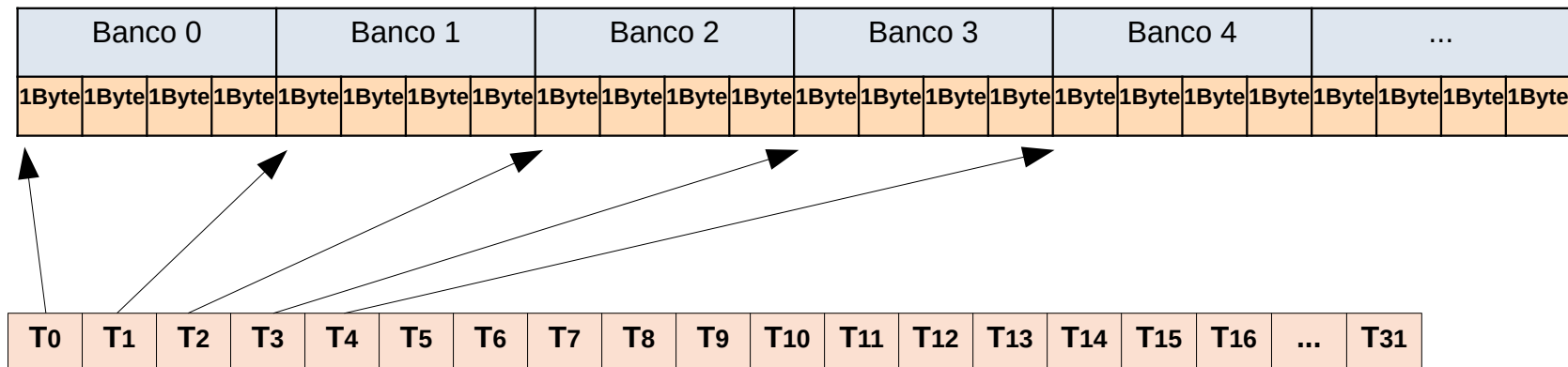


Coalescencia y memoria compartida

Gestión de la memoria compartida - Bancos

21

- Cuando trabajamos con tipos de datos de 4 Bytes (`int` o `float`) no suelen existir inconvenientes si cada hilo del warp accede a un único valor.
- De esta forma cada hilo accede a bancos diferentes.

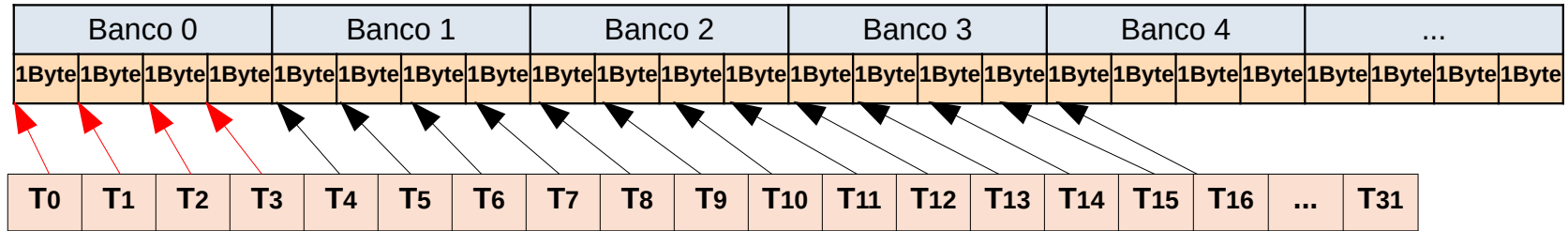


Coalescencia y memoria compartida

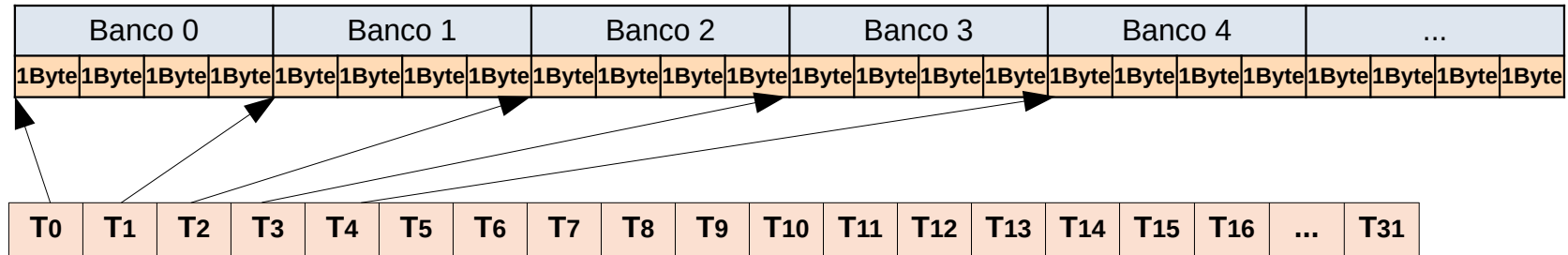
Gestión de la memoria compartida - Bancos

22

- Cuando trabajamos con tipos de datos menores a 4 bytes (`char`, `short`, `int` o `short float`) es muy probable que existan conflictos de bancos.



- En este caso es conveniente que cada hilo acceda y/o procese de a 4 bytes.

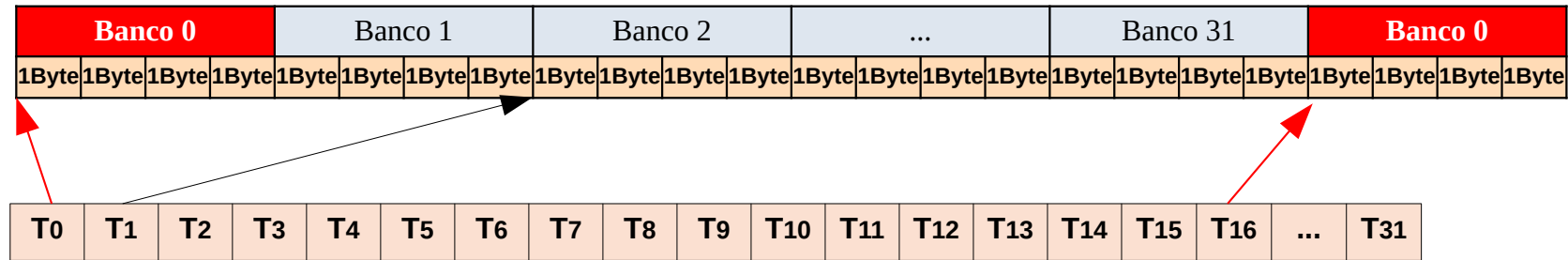


Coalescencia y memoria compartida

Gestión de la memoria compartida - Bancos

23

- Cuando trabajamos con tipos de datos mayores a 4 bytes (`double`) o cada hilo trabaja sobre varios elementos consecutivos es muy probable que existan conflictos de bancos.

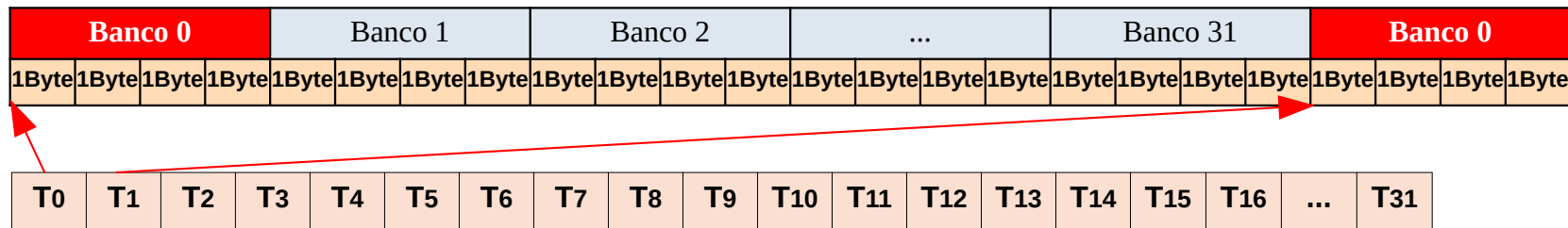


Coalescencia y memoria compartida

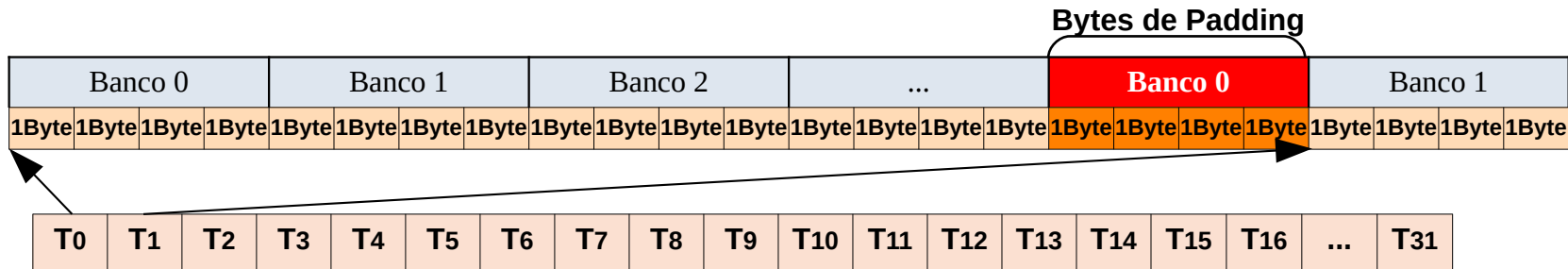
Gestión de la memoria compartida - Bancos

24

- El peor caso: cada hilo lee de a 128 bytes, todos los hilos del warp acceden al mismo banco.



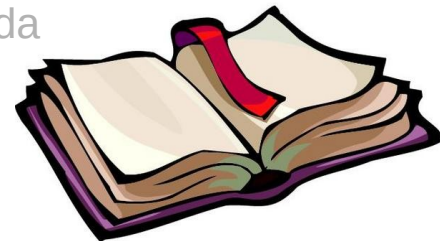
Podemos evitar el conflicto haciendo padding (agregar 4 bytes al final de cada secuencia de 128 bytes).



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

26

- 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 operaciones menos costosas.
- Es posible ejecutar otras operaciones entre accesos a memoria 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

27

- 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

28

Sin prefetching

Cada suma espera que sus datos se carguen desde memoria.

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

Con 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 los operandos están listos (**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

29

Sin prefetching

```
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

Los hilos cargan coalescentemente un bloque de memoria global a memoria compartida

Sincronización

Calcula el bloque

Sincronización

Estas dos líneas tienen dos partes dependientes entre si:

- 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

30

Con prefetching

```
float Ra= A[i*N+threadIdx.x];  
float Rb= B[(threadIdx.y)*N + j];
```

Carga un bloque de memoria 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 memoria 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 memoria 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

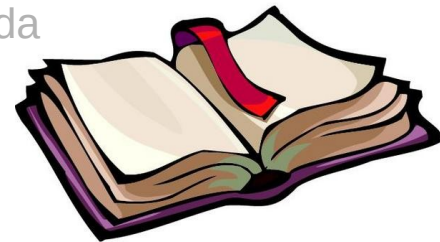
```
}
```

Se reduce la inactividad del sistema, mientras unos hilos estén inactivos esperando se completen sus accesos a memoria, otros lo están solicitando. Cuando todos completan la lectura, pasan la sincronización, computan sobre memoria compartida y vuelven a la carga de los datos. En la siguiente iteración los últimos elementos cargados se convierten en los actuales y se procede a la nueva carga.

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



- 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 bloque 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

33

- Todos los hilos de un warps ejecutan la misma instrucción (**SIMD - 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
```

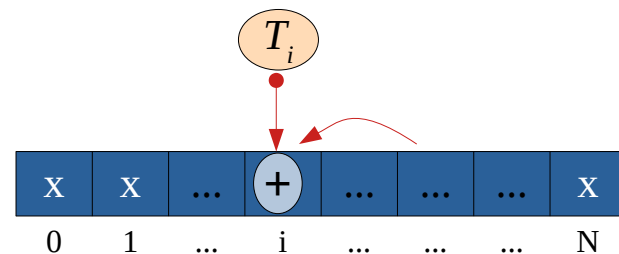
- Si existe divergencia la ejecución del warp se divide en pasos, uno para cada camino, y cada paso se ejecuta secuencialmente.
- Dependiendo del algoritmo se podría organizar la ejecución de manera que los hilos de un mismo warp sigan todos el mismo camino.

Divergencia

Ejemplo

34

- **Problema de ejemplo:** sumar los elementos de un vector.
- Suponer un escenario hipotético:
 - Un sistema donde los **warps** se componen de **4 hilos**
 - El **SM** de este sistema **ejecuta** los hilos de a **un warp**
 - Se creó un sólo **bloque unidimensional de 8 hilos** (2 warp)
 - Inicialmente se utilizaron los 8 hilos para traer desde memoria global un **vector de 8 elementos** ($N=8$) a la **memoria compartida**
 - El vector está en memoria compartida, por lo tanto lo comparten todos los hilos del bloque
 - El bloque de hilos deberá sumar los elementos del vector
 - Las soluciones son **in-place**: cada hilo que suma, requiere dos operandos:
 - El primero en la posición de su threadIdx.x
 - El segundo será determinado por la estrategia de solución.
 - El resultado de la suma queda almacenado en la posición del primer operando.



Divergencia

Ejemplo

35

- Supongamos una primera solución donde el código del kernel que resuelve la suma es el siguiente:

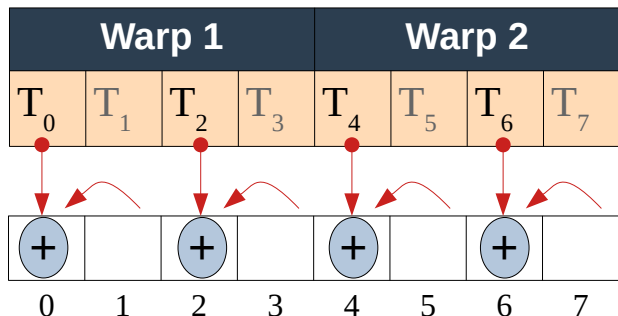
```
for ( it=1 ; it <= nrIteraciones ; it++ ){  
    if( threadIdx.x % (2it) == 0 )  
        "suma";  
    __syncthreads();  
}
```

- Para nuestro ejemplo se requieren 3 iteraciones.

Divergencia

36

Iteración 1



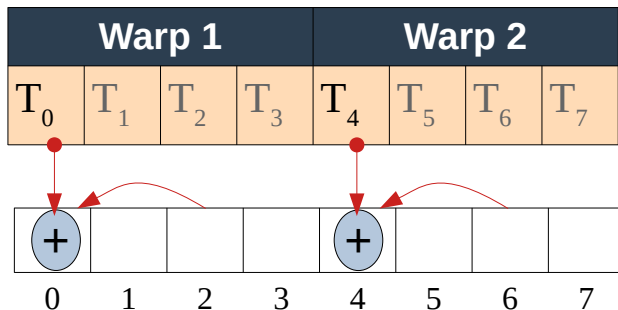
Los hilos 0, 2, 4 y 6 cumplen con la condición del if.

Los dos warps divergen

La ejecución del bloque requiere 4 pasos:

- 1: T_0 y T_2 (suman datos)
- 2: T_1 y T_3 (no computan)
- 3: T_4 y T_6 (suman datos)
- 4: T_5 y T_7 (no computan)

Iteración 2



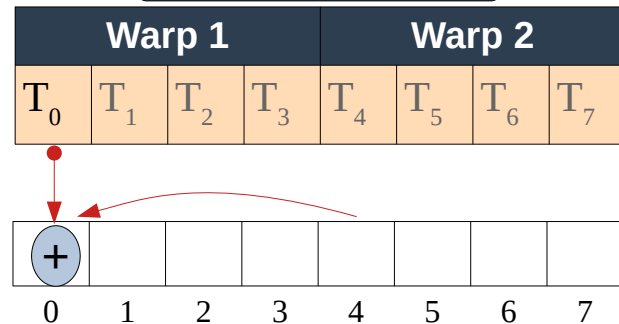
Los hilos 0 y 4 cumplen con la condición del if.

Los dos warps divergen.

La ejecución del bloque requiere 4 pasos:

- 1: T_0 (suma datos)
- 2: T_1 , T_2 y T_3 (no computan)
- 3: T_4 (suma datos)
- 4: T_5 , T_6 y T_7 (no computan)

Iteración 3



El hilo 0 cumple con la condición del if.

El primer warp diverge.

La ejecución del bloque requiere 3 pasos:

- 1: T_0 (suma datos)
- 2: T_1 , T_2 y T_3 (no computan)
- 3: T_4 , T_5 , T_6 y T_7 (no computan)

11 pasos en total

Divergencia

Ejemplo

37

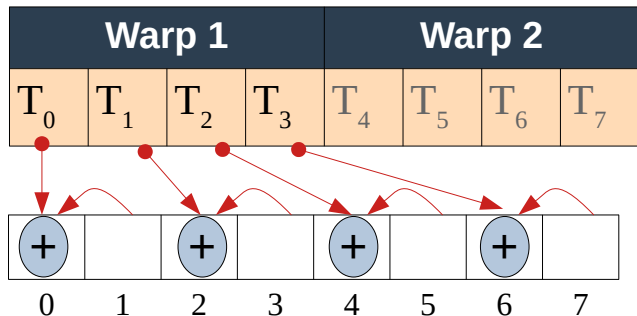
- Para reducir el numero de pasos debido a divergencia es necesario que los hilos que pertenecen a un mismo warp sigan todos el mismo camino.
- Modificamos el código del kernel que resuelve la suma de la siguiente manera:

```
for ( it=0 ; it < nrIteraciones ; it++ ){  
  
    if( threadIdx.x < N/2(it+1) )  
        "suma";  
    __syncthreads();  
}
```

Divergencia

38

Iteración 1



Los hilos del primer warp cumplen con la condición del if.

Ningún warp diverge

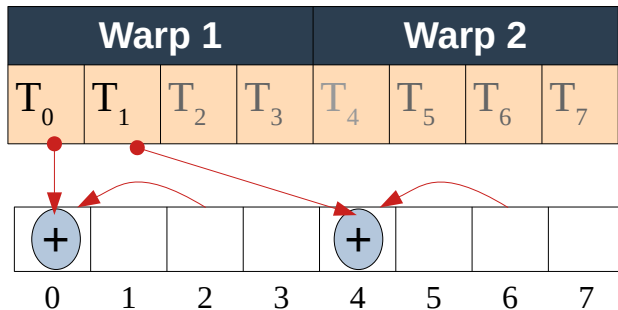
La ejecución del bloque requiere 2 pasos:

1: T_0, T_1, T_2 y T_3 (suman datos)

2: T_4, T_5, T_6 y T_7 (no computan)

Ventaja respecto de la primera iteración de la solución anterior (4 pasos)

Iteración 2



Los hilos 0 y 1 cumplen con la condición del if.

El primer warp diverge.

La ejecución del bloque requiere 3 pasos:

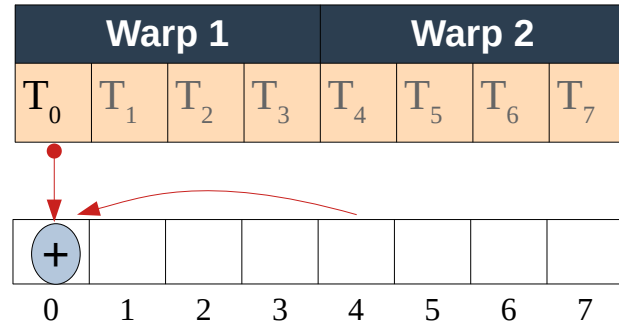
1: T_0 y T_1 (suma datos)

2: T_2 y T_3 (no computan)

3: T_4, T_5, T_6 y T_7 (no computan)

Ventaja respecto de la segunda iteración de la solución anterior (4 pasos)

Iteración 3



El hilo 0 cumple con la condición del if.

El primer warp diverge.

La ejecución del bloque requiere 3 pasos:

1: T_0 (suma datos)

2: T_1, T_2 y T_3 (no computan)

3: T_4, T_5, T_6 y T_7 (no computan)

8 pasos en total

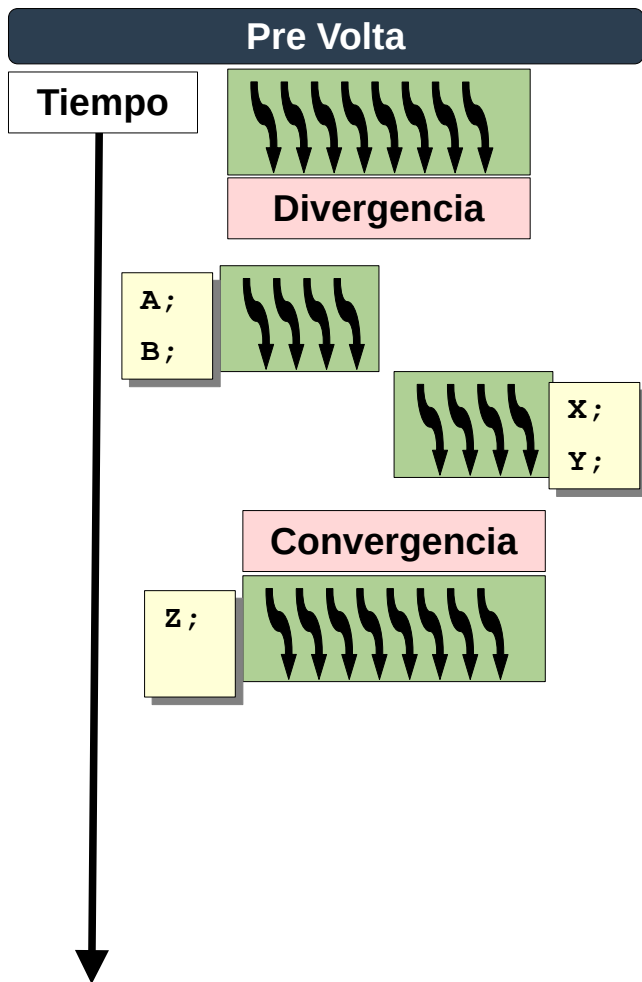
- | | | | | | | | | | | | | | | | | | | |
|------|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| T0 | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 | T15 | T16 | ... | T31 |
| pc | | | | | | | | | | | | | | | | | | |

- [illegible]

Divergencia

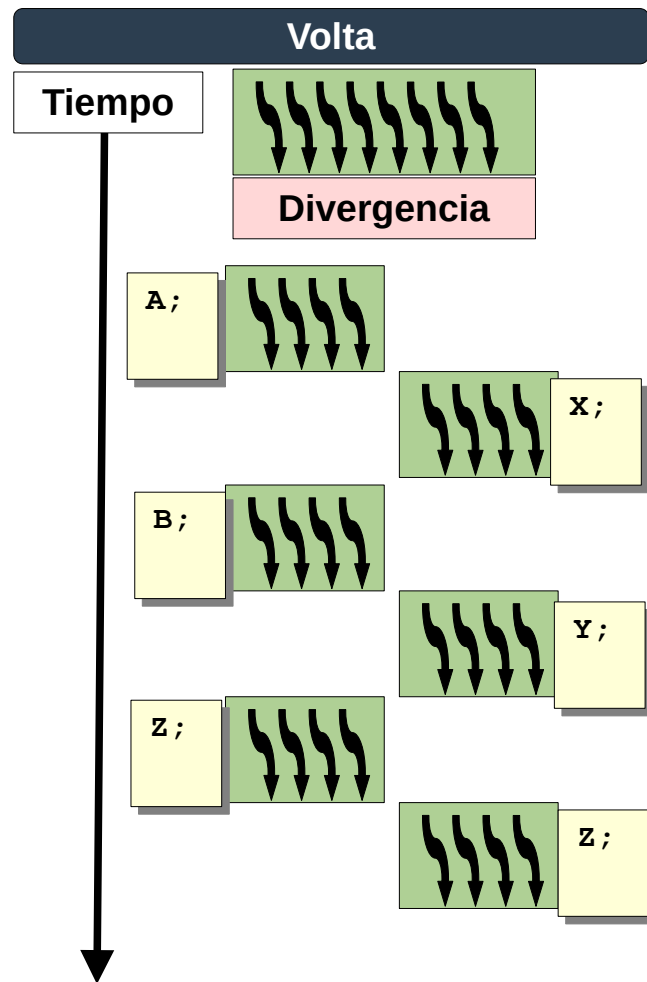
Planificación independiente de hilos en Nvidia Volta

40



```
if (threadIdx.x < 4){  
    A;  
    B;  
}else{  
    X;  
    Y;  
}  
Z;
```

La ejecución sigue siendo SIMT:
en un ciclo de reloj los cores
ejecutan la misma instrucción para
todos los hilos activos en un warp



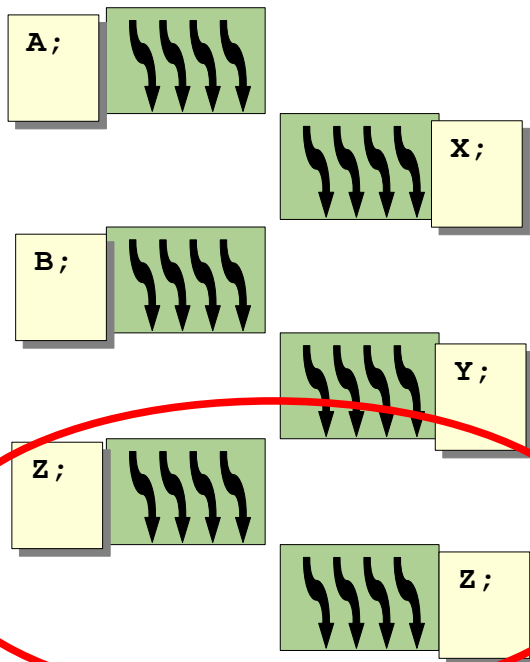
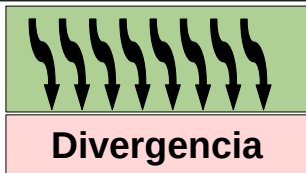
Divergencia

Planificación independiente de hilos en Nvidia Volta

41

Volta

Tiempo



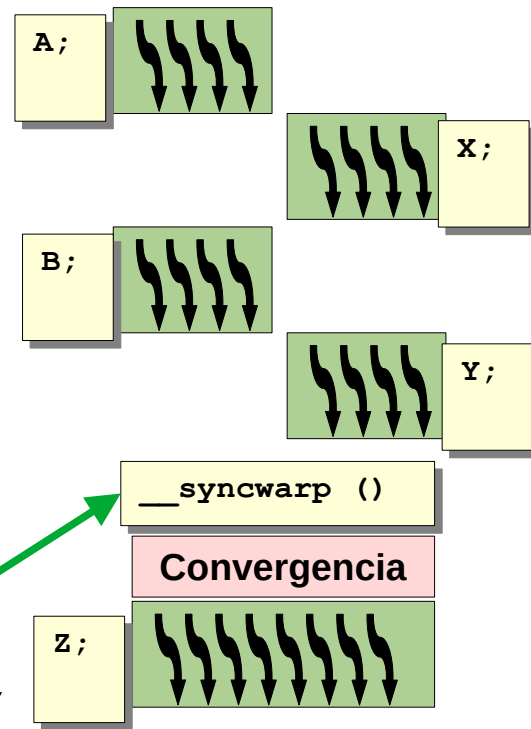
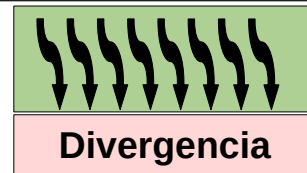
```
if (threadIdx.x < 4){  
    A;  
    B;  
}else{  
    X;  
    Y;  
}  
Z;
```

Pero Z; se encuentra en un área no divergente y no se ejecuta al mismo tiempo por los hilos del mismo warp.

Se puede utilizar la función `__syncwarp()` para forzar la convergencia de los hilos de un warp y permitir una mayor eficiencia del SIMT.

Volta

Tiempo

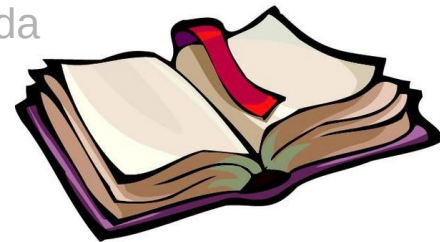


`__syncwarp()`
Convergencia

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

Mezcla de instrucciones

43

- No todas las instrucciones implican el mismo tiempo.
- Las operaciones más costosas:
 - Las operaciones de punto flotante.
 - Las instrucciones de acceso a memoria.
 - Las sentencias 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

Mezcla de instrucciones

44

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

```
__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];
    ...
}
```

Iteración for

- Una operación branch (condición de for): $k < N$
- Una instrucción que incrementa el contador de la iteración: $k++$

Cómputo

Existe mezclas de instrucciones:

Dos instrucciones de punto flotante:

- Otra para la suma

- Una para el producto

Dos instrucciones enteras usan k en el índice (desplazamientos).

- 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
<pre>for (int k = 0; k < N; k++) C += As[i][k]*Bs[k][j];</pre>	<pre>C = As[i][0]*Bs[0][j] + As[i][1]*Bs[1][j]+...+ As[i][N]*Bs[j][N];</pre>

Mezcla y granularidad

Mezcla de instrucciones - Desenrollado de bucles

46

- 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

Mezcla de instrucciones

47

- 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 para utilizar 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 software que realicen estas mejoras, por ejemplo: compiladores que traduzcan una iteración a una instrucción equivalente.

Mezcla y granularidad


Mezcla de instrucciones - Equivalencias


48

- Algunas instrucciones tienen equivalentes menos costosas:

Cocientes y módulos

Siempre que $n=2^x$ 

 $\frac{i}{n} \rightarrow i \gg \log_2(n)$

 $i \% n \rightarrow i \& (n-1)$

Exponenciación

Se suelen resolver con la función **pow(b, e)** pero es **muy costosa**.

Alternativas **pow(b, e)**:

En base 2: **expf2()**

En base 10: **expf10()**

Alternativas a raíces cúbicas **pow(b, 1/3)**:

Positiva: **cbrtf()**

Negativa (-1/3): **rcbrtf()**

- Otras alternativas de exponenciación:

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

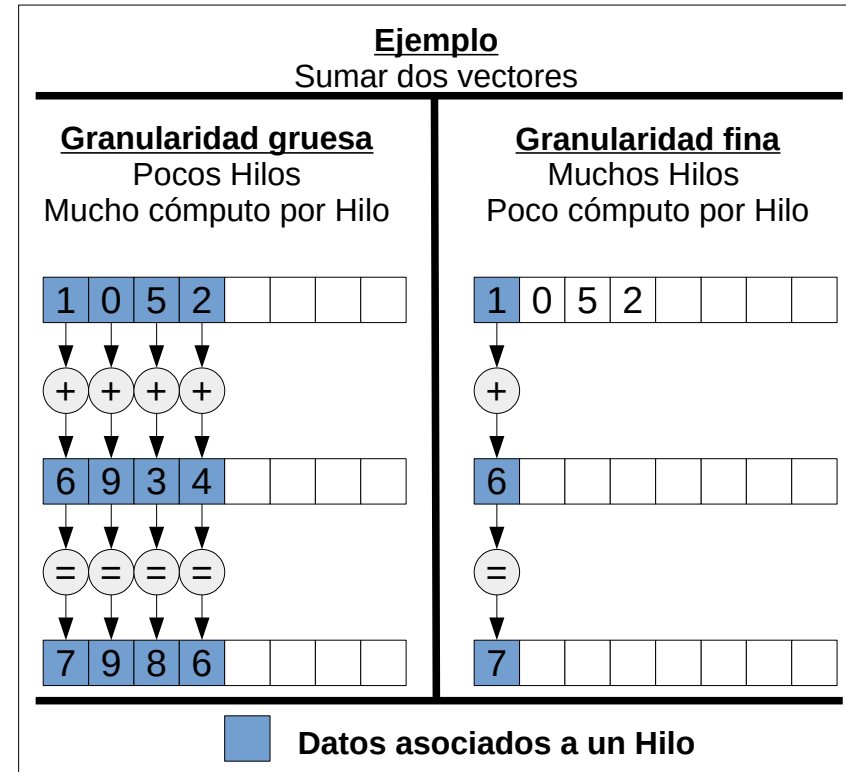
Mezcla y granularidad

Granularidad

50

¿Cuánto trabajo se asigna a cada hilo? (granularidad)

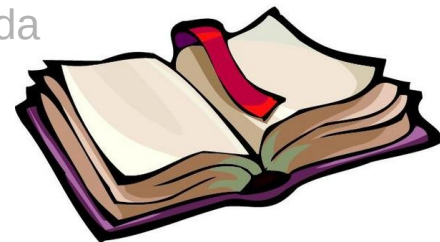
- No existe una metodología para determinar la mejor granularidad de un hilo, se debe evaluar en función de las características de la aplicación y del kernel.
- Sin embargo, las aplicaciones **en GPU** son, por naturaleza y por las características de la arquitectura, de **grano fino**.



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



- 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 y puede influir en el rendimiento de las aplicaciones.

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

Ocupación Slots - Ejemplo

53

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

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

- El mejor aprovechamiento dependerá del número de hilos por bloque:
 - Bloques de 256 hilos:** el SM planificará sólo 3 bloques ($256 \times 3 = 768$)
 - Bloques de 128 hilos:** el SM planificará sólo 6 bloques ($128 \times 6 = 768$)
 - Bloques de 512 hilos:** el SM planificará sólo 1 bloque ya que 2 o más bloques superan los 768 hilos que soporta la arquitectura. Se desperdician 256 hilos que la arquitectura podría soportar.
 - Bloques de 32 hilos:** el SM planificará sólo 8 bloques al mismo tiempo ($8 \times 32 = 256$). En este caso se desperdician 512 hilos que la arquitectura podría soportar.

- **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

55

- 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
<i>Slots para Bloques (Cantidad de bloques por SM)</i>	8		
<i>Slots para Threads (Cantidad de hilos por SM)</i>	768	1024	1536
Registros	8 KB	16 KB	32 KB
<i>Memoria Compartida</i>	16 KB		48 KB

Ocupación Registros - Ejemplo

56

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

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

- Suponer un problema para el cual definimos **256 hilos por bloque**:
 - Se requieren **9 registros por hilo**: cada bloque necesita $9 \times 256 = 2304$ registros. Un SM puede proveer registros para 7 bloques ($7 \times 2304 = 16128$) y quedan 256 registros libres.
 - Se requieren **11 registros por hilo**: cada bloque necesita $11 \times 256 = 2816$ registros. Ahora, un SM sólo puede proveer registros de 5 bloques ($5 \times 2304 = 11520$) y quedan 2304 registros libres.
- Esto muestra un débil equilibrio. A pequeñas modificaciones (2 registros más por hilo) se reduce el paralelismo (7 a 5 bloques).

Ocupación Memoria Compartida

57

- 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
<i>Slots para Bloques (Cantidad de bloques por SM)</i>	8		
<i>Slots para Threads (Cantidad de hilos por SM)</i>	768	1024	1536
<i>Registros</i>	8 KB	16 KB	32 KB
<i>Memoria Compartida</i>	16 KB		48 KB

Ocupación

Memoria Compartida - Ejemplo

58

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

- Si un problema requiere **2KB de memoria compartida por bloque**:
 - **Arquitecturas G80 y GT200**: soportarán 8 bloques por SM ($8 \times 2KB = 16KB$).
 - **Arquitectura GF100**: soportará 8 bloques por SM y quedarán 32KB sin utilizar.
- Si un problema requiere **4KB de memoria compartida por bloque**:
 - **Arquitecturas G80 y GT200**: soportarán sólo 4 bloques por SM ($4 \times 4KB = 16KB$) desperdiciando 4 bloques (8 por la capability).
 - **Arquitectura GF100**: soportará 8 bloques por SM ($8 \times 4KB = 32KB$) y quedarán 18KB sin utilizar.

- 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:
 - Por ejemplo: se puede ajustar un tamaño de hilos por bloque en función del número de registros pero la concurrencia se ve limitada por la cantidad de memoria compartida.
- 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

60

- 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 tamaño del 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-parallelismo.

