

Programación Paralela en CUDA

Material traducido desde:

<https://www.youtube.com/playlist?list=PLKK11LigqititwsoZOoGk3SW-TZCar4dK>

CAPITULO 1: Introducción

CAPITULO 2: IDE y Kernels

CAPITULO 3: Threads, Blocks y Grids.

CAPITULO 4: Memory Overview

CAPITULO 5: Algoritmos

CAPITULO 6: Introducción a la Shared Memory

CAPITULO 7: Bank Conflicts

CAPITULO 8: Blocking en la Shared Memory

CAPITULO 9: Librería Thrust

Mauricio Aguilar

Ingeniería En Computación, UNC

rev 1.0

Introducción

CAPITULO 1

Conceptos Básicos (1 / 2)

- **¿Qué es CUDA?**
- GPGPU con tarjetas graficas nVidia. «Compute Unified Device Architecture»
- GPGPU es el proceso de utilizar tarjetas gráficas para procesamiento no gráfico.
- Se recomienda estar familiarizado con los conceptos básicos de C++.
- **Descarga e instalación**
 - **Windows 7 y Visual C++ Express 2010**
 - **Windows 8 y Visual C++ Express 2012**
 - **Linux y Eclipse**
- **Solo disponible en tarjetas graficas nVidia**
- Disponible en tarjetas de la serie 8000 o superior y las de serie 200 o superior tienen CUDA.
- Ver lista completa en: <https://es.wikipedia.org/wiki/CUDA>

Conceptos Básicos (2/2)

- **GPU**
- Las tarjetas graficas están diseñadas para realizar un montón de cálculos simples en paralelo, requeridos por los gráficos 3D. Estos mismos cálculos pueden ser aplicados a múltiples ítems de datos muy rápidamente.
- **CUDA C**
- Hay muchas ramas relacionadas a compilar en CUDA. CUDA C es un lenguaje como C para instrucciones GPUs nVidia. Es un lenguaje de bajo nivel (por debajo corre PTX, que es un pseudo assembler, el cual nVidia no hace público). Otros son PyCuda, C# Cuda.
- **CPU != GPU**
- CPU es un conjunto de pocos procesadores poderosos.
- GPU es un conjunto de muchos procesadores modestos.
- **Descarga e Instalación de CUDA**
- Visite nVidia CUDA Zone y descargue la última versión de CUDA:
- <https://developer.nvidia.com/cuda-downloads>
- Esto instalará el CUDA toolkit, el SDK, y los drivers del desarrollador.

IDE y Kernels

CAPITULO 2

Preparación del IDE (1 / 2)

- **Build Customisation**

- Debes agregar CUDA X.0 a tu proyecto para que el IDE que utilices sepa que los archivos .CU son para el compilador NVCC, el compilador de CUDA.
- Esto debe hacerse antes de agregar los archivos .CU para que los reconozca.

- **Programación y Compilación**

- El código en CUDA C va en un documento .CU en tu proyecto. Estos archivos pueden contener tanto código CUDA C como código C++ normal.
- El compilador de CUDA compilará todo el código de CUDA a PTX y enviará el código C++ restante al compilador C++ para compilar. El código PTX será embebido en un ejecutable y compilado por el driver del dispositivo previamente a ejecutar la aplicación. De esta manera el código PTX puede ser compilado a código de máquina que mejor coincida con tu dispositivo.

Preparación del IDE (2/2)

- **Headers y Libraries**
- Puedes incluir la cabecera “cuda.h” y linkear el archivo “cudart.lib” la cual es la librería principal de tiempo de ejecución de CUDA, necesitas hacer esto para poder llamar a las funciones de la API de CUDA.
- Esto puede requerir agregar los directorios con las librerías CUDA y cabeceras a los directorios de configuración del IDE.
- cudart.lib puede aparecer como sigue en las dependencias del linkeador:
 - `$(CudaToolkitLibDir)\cudart.lib`
- Hay librerías 32 y 64 bit.
- Agregar directorios de librerías y cabeceras a la configuración del IDE.

Conceptos Importantes

- **Host y Device**
- **Host:** Esta es la CPU. La RAM del sistema, discos y otros dispositivos periféricos están todos al control del host.
- **Device:** Esta es la GPU. Tiene su propia RAM, no puede acceder a la RAM del sistema, discos y otros dispositivos periféricos.
- **Kernel**
- Es una función que será ejecutada en la GPU, usualmente por cientos o miles de threads al mismo tiempo. Se escriben en CUDA C. La CPU lanza los kernels con una sintaxis especial para hacer saber a la GPU cuantos threads debe usar.
- CUDA C es parecido a C, solo que no soporta funciones recursivas, además de que los kernels siempre retornan void. Los kernels tampoco pueden tener un número variable de argumentos y no pueden usar la system memory.

Calificadores de Funciones

- **__global__** Llamado por el host, ejecuta sobre el device
 - Los kernels son marcados con este calificador.
- **__device__** Llamado por el device, ejecuta sobre el device
 - Para funciones ayudantes del kernel
- **__host__** (o sin calificador) Función host normal

Copia de Datos

- Debido a que la GPU no puede ver la system RAM, los datos deben ser copiados de ida hacia la GPU y de vuelta hacia la CPU, es decir, entre el host y el device.
- Esto es un masivo cuello de botella y parece ser que en el futuro las memorias serán compartidas.

Funciones de la API de CUDA para la administración de memoria

- **cudaMalloc(void **devPtr, size_t sizeInBytes);**
 - Esta función reserva memoria en el dispositivo sobre la global memory. El primer argumento es el puntero al device y el segundo es la cantidad de bytes a reservar.
- **cudaFree(void **devPtr);**
 - Esta función libera la memoria alojada previamente en el device por cudaMalloc. El device tiene una cantidad limitada de memoria, nunca olvide liberarla una vez que termine de ocuparla.
- **cudaMemcpy(void * dest, void* src, size_t sizeInBytes, enum dirección);**
 - Esta función copia datos hacia y desde las memorias del device y del host. **dest** es el destino, **src** es la fuente de la copia.
 - La dirección usualmente es:
 - **cudaMemcpyDeviceToHost**
 - **cudaMemcpyHostToDevice**

Ejemplo

- Para sumar dos enteros dentro de una GPU (solo sirve a fines conceptuales ya que no explota el paralelismo de la GPU) debemos:
 - **Crear** dos enteros por el host
 - **Alojar** memoria para copiar los valores sobre el device
 - **Copiar** los enteros a la memoria del device
 - **Llamar** al kernel para sumarlos
 - **Copiar** el resultado de vuelta a la memoria del host.
 - **Imprimir** el resultado calculado por la GPU
 - **Liberar** la memoria del device que fue alojada
- Lo último que debemos codificar es la **sintaxis para lanzar el kernel**:
 - **Somekernel**<<<1, 1>>>(parameters);
- Esta es la configuración de lanzamiento / ejecución.
- El primer número especifica **cuantos blocks** lanzar.
- El segundo numero especifica **cuantos threads** por block lanzar.
- En este ejemplo estamos lanzando 1 block con 1 thread.

Código del ejemplo

```
#include <iostream>
#include <cuda.h>

using namespace std;

__global__ void AddIntsCUDA(int* a, int *b)
{
    a[0] += b[0];
}

int main()
{
    int a = 5, b = 9;
    int *d_a, *d_b;
```

```
    cudaMalloc(&d_a, sizeof(int));
    cudaMalloc(&d_b, sizeof(int));

    cudaMemcpy(d_a, &a, sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, sizeof(int), cudaMemcpyHostToDevice);

    AddIntsCUDA<<<1, 1>>>(&d_a, &d_b);

    cudaMemcpy(&a, d_a, sizeof(int), cudaMemcpyDeviceToHost);

    cout<<"The answer is "<<a<<endl;


    cudaFree(d_a);
    cudaFree(d_b);

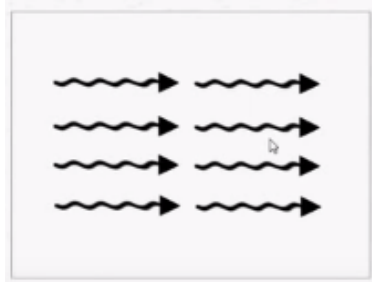
    return 0;
}
```

Threads, Blocks y Grids

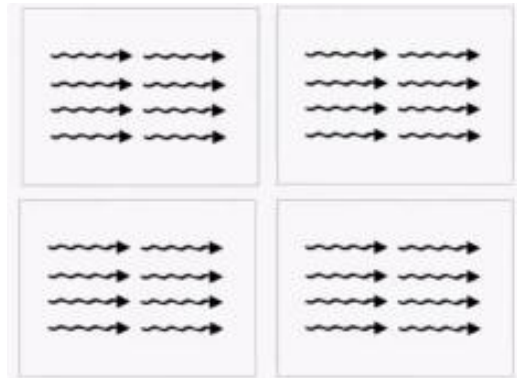
CAPITULO 3

Conceptos Importantes

- **Thread:** Simple unidad de ejecución que ejecuta kernels sobre la GPU. Similar a un thread de CPU, pero hay muchos mas. 
- **Block:** Es un conjunto de threads. Todos los threads dentro de un block pueden comunicarse.



- **Grid:** Es un conjunto de blocks. En CUDA usualmente lanzamos los kernels sobre muchos threads, en grupos de blocks de threads que forman un grid.



Hello Word CUDA! (1 / 2)

- **Arquitectura Básica:** Si queremos sumarle a los valores de un arreglo 1 los valores de un arreglo 2, en C++ haríamos algo como:
 - `for(int a=0; a<6; a++)`
 - `array1[a]+=array2[a];`
- Este programa repetirá la misma operacin 6 veces, una por una.

Array 1	Array 2
35	43
43	55
65	123
12	76
94	33
102	87

Array 1		Array 2
78		43
98		55
188	+	123
12		76
94		33
102		87

Hello Word CUDA! (2/2)

- En CUDA podemos hacer esto con 6 threads. Cada uno realizará una simple suma, y ejecutarán todos a la vez.

Array 1		Array 2	
78	+	43	Thread 0
98	+	55	Thread 1
188	+	123	Thread 2
88	+	76	Thread 3
127	+	33	Thread 4
189	+	87	Thread 5

Tipo de Dato dim3

- dim3 es una tipo de estructura o vector con tres enteros, x, y, y z. Puedes inicializarlos de diferentes formas:
 - `dim3 threads(256);` // Inicializa con x como 256, y y z serán 1
 - `dim3 blocks(100, 100);` // Inicializar x y y, z sera 1.
 - `dim3 coordenadas(10, 54, 32);` // Inicializa los tres valores

Algunos Máximos

- Puedes lanzar hasta **1024 threads** por block
(512 si tu tarjeta tiene capacidad de computo 1.3 o menos)
- Puedes lanzar **$2^{32}-1$ blocks** en una nica ejecución
(o $2^{16}-1$ si tu tarjeta es de capacidad de computo 2.0 o menos)
- Por ejemplo una simple GeForce GT 440 puede lanzar unos 67.108.864 threads sin problemas.
- **Porque Blocks y Threads?**
- Tal vez seria mas fácil lanzar 67 millones de threads en lugar de organizarlos en blocks, pero cada GPU tiene un limite sobre el numero de threads por block, como se vio arriba, pero casi no hay limite sobre el numero de blocks. Cada GPU puede ejecutar el mismo numero de blocks concurrentemente, ejecutando el mismo numero de threads simultáneamente.
- Agregando este nivel extra de abstracción, el rendimiento de las GPUs es mas alto ya que simplemente puede ejecutar mas blocks concurrentemente y realizar los cálculos sin cambiar el código. nVidia hizo esto para permitir la ganancia de

Cada Thread es individual

- Cada uno de los threads ejecutando es individual, ellos saben lo siguiente:
- **threadidx** \leftarrow Índice del thread dentro del block
- **blockidx** \leftarrow Índice del block dentro del grid
- **blockDim** \leftarrow Dimensiones del block en threads
- **gridDim** \leftarrow Dimensiones del grid en blocks
- Cada una de estas son estructuras dim3 y pueden ser leídas en el kernel para asignar trabajos a cualquier thread.
- Dentro del kernel, cada thread puede calcular un único id con:
 - **int id = blockidx.x * blockDim.x + threadidx.x;**

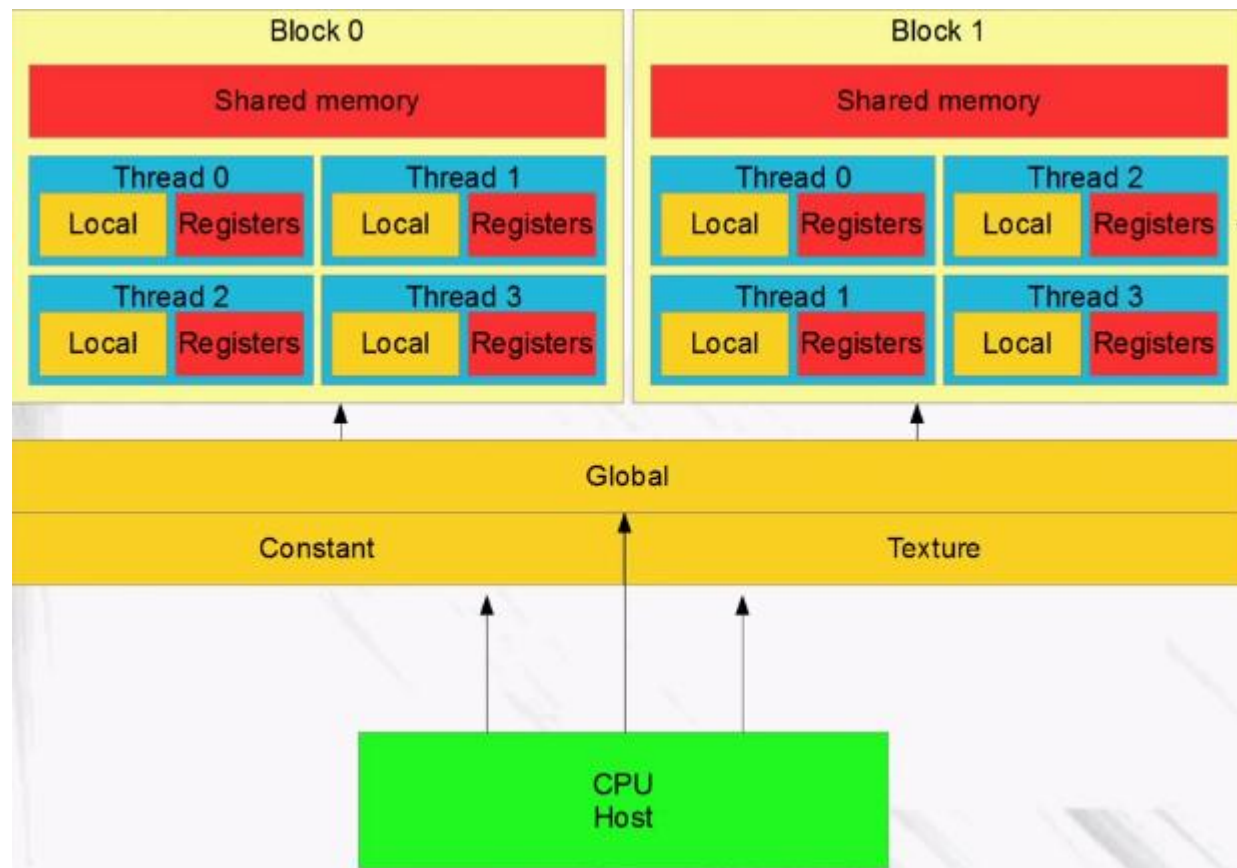
Memory Overview

CAPITULO 4

Memorias de CUDA

- La GPU tiene varios espacios de memoria diferentes. Cada uno tiene características y usos, lectura/escritura, diferentes velocidades y diferentes alcances.
- Programar en CUDA es como un juego de estrategia, el enemigo es la performance pobre.
- Entonces, las memoria de CUDA son:
 - Global
 - Local
 - Caches L1 y L2
 - Constant
 - Texture
 - Shared
 - Registers

Estructura General



Global Memory

- Hay una gran cantidad de global memory. Es mas lenta de acceder que otros tipos de memorias como la shared memory o los registers. Todos los threads ejecutando pueden leer y escribir en la global memory y también puede hacerlo la CPU. Las funciones `cudaMalloc`, `cudaFree`, `cudaMemcpy` y `cudaMemset` todas tratan con la global memory.
- Esta es la memoria principal de almacenamiento de la GPU, cada byte es direccionable. Es persistente entre llamadas del kernel. Para tarjetas con capacidad de computo 2.0 o superior, esta memoria es cacheada.

Local Memory

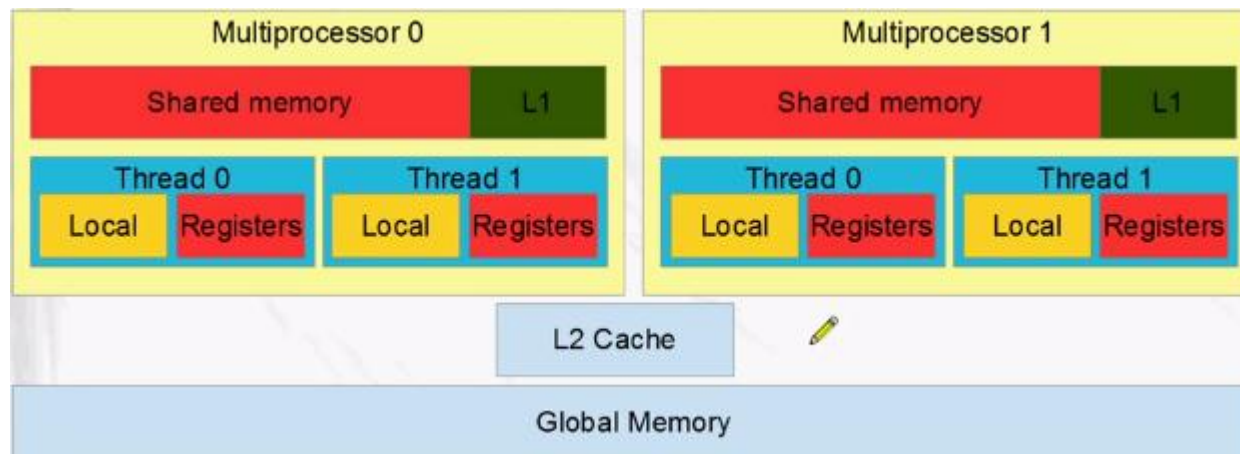
- Es parte de la memoria principal de la GPU (al igual que la global memory), por lo que es generalmente lenta. La local memory es usada automáticamente por el NVCC cuando ejecuta fuera de los registers o cuando los registers no pueden ser usados. Esto se llama **register spilling**. Esto ocurre si hay demasiadas variables por thread que usan registers o si los kernels usan estructuras. También, los arreglos que no son indexados con constantes usan la local memory ya que los registers no tienen direcciones, debe usarse un espacio de memoria direccionable. El alcance para la local memory es por thread.
- La local memory es cacheada en una cache L1 y entonces en una cache L2, entonces el register spilling puede no implicar una disminución dramática en el rendimiento sobre capacidad de computo 2.0 o superior.
- Las estructuras no siempre usan la local memory, algunas pequeñas podrían fácilmente usar registers!.

Caches

- Sobre capacidades de computo 20 y superior hay una cache L1 por multiprocesador. Hay también una cache L2 la cual es compartida entre todos los multiprocesadores. La global memory y la local memory la usan.
- La L1 es muy rápida, con velocidades como la de la shared memory. La L1 y la shared memory son actualmente los mismos bytes, ellas pueden ser configuradas para ser de 48k una y 16k la otra.
- Todos los accesos a la global memory van a través de la cache L2, incluyendo a los accesos de la CPU!
- Usted puede encender y apagar el caching con una opción del compilador.
- Las texture memory y la constant memory tienen sus propias caches separadas.

Estructura de Caches

- Los multiprocessors son el hardware que ejecuta los blocks de threads! Ellos pueden ejecutar mas de un block a la vez siempre que pueda compartir sus recursos.



Constant Memory

- Esta memoria también es parte de la memoria principal de la GPU. Tiene su propia cache. No esta relacionada a la L1 y L2 de la global memory. Todos los threads tienen acceso a la misma constant memory pero ellos solo pueden leerla, no escribirla. La CPU setea los valores en la constant memory antes de lanzar el kernel.
- Es muy rápida (a velocidades de registers y shared memory) si todos los threads ejecutando en un warp leen exactamente la misma dirección.
- Es pequeña, hay solamente 64k de constant memory!
- Todos los threads corriendo comparten la constant memory. En programación grafica, este memoria mantiene las constantes como el modelo, la vista y la proyección de matrices.
- **SM** es Streaming Multiprocessor o solamente Multiprocessor.

Texture Memory

- La texture memory también reside en la memoria del device (al igual que la global, la local y la constant). Tiene su propia cache. La lee solamente la GPU, la CPU la setea.
- La texture memory tiene muchos trucos de direccionamiento extra debido a su diseño de indexado (llamado **texture fetching**) e interpolado de pixeles en una imagen 2D.
- La texture cache tiene un ancho de banda mas lento que la L1 de la global memory, por lo que puede ser mejor apegarse a la L1.
- Todos los threads corriendo comparten la texture memory y las caches son por SM. Debido a que la texture memory es realmente global memory, puedes actualmente cambiar los valores mientras que los threads ejecutan. El asunto es el caching, por lo que si un thread cambia valores en la global memory y algún otro thread lo lee, no se sabe si se leerá la copia cacheada o la copia de la global memory. Nvidia dice que esto es indefinido. Sin embargo, se puede cambiar los valores de la texture memory desde un kernel diferente y entonces asegurarte de que ejecute con seguridad lo que lee.

Shared Memory

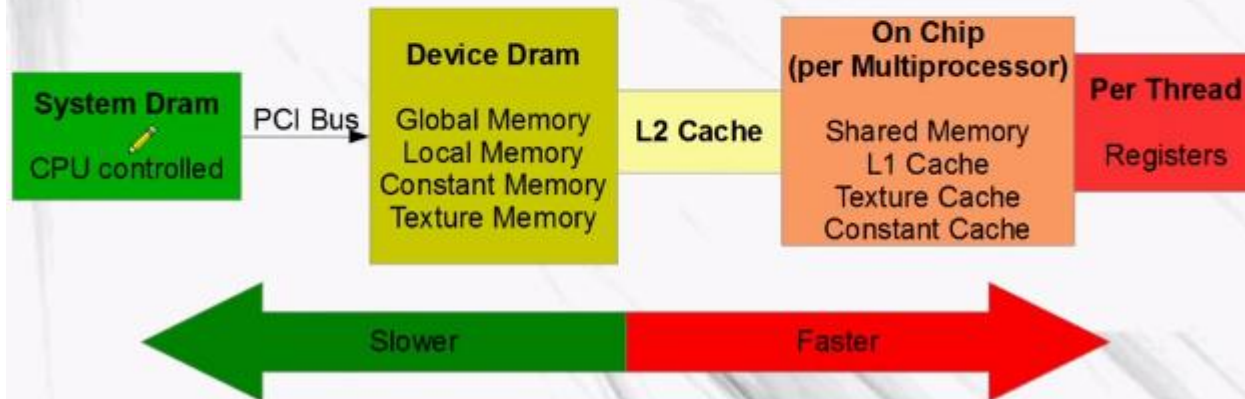
- La shared memory es muy rápida (velocidad de register). Es compartida entre los threads de cada block.
- Los bank conflicts pueden disminuir la velocidad de acceso. **Es mas rápido cuando los threads leen desde diferentes banks o todos los threads de un warp lee exactamente el mismo valor.**
- Words sucesivas (4 bytes) residen en diferentes banks.
- Hay **16 banks** en capacidad de computo 1.0 y **32** en capacidad de computo 2.0.
- La shared memory es usada para permitir una rápida comunicación entre threads en un block. La shared memory y la cache L1 de un SM son los mismos bytes.

Registers

- Los registers son la memoria mas rápida sobre la GPU. Las variables que declaramos en un kernel usará registers a menos que ejecutemos fuera de ellos o no sea almacenado en registers, entonces será usada la local memory.
- El alcance de los registros es por thread. A diferencia de la CPU, hay cientos de registers en un GPU.
- Seleccionando cuidadosamente unos pocos registers en lugar de usar 50 por thread, pueden fácilmente doblar el numero de blocks concurrentes que la GPU puede ejecutar y por lo tanto incrementar el rendimiento substancialmente. Los SM's ejecutan hasta 16 blocks concurrentemente si los dejamos.

Summary

Memory	Access	Scope	Lifetime	Speed	Note
Global	RW	All threads and CPU	All	Slow, cached	Large
Constant	R	All threads and CPU	All	Slow, cached	Read same address
Texture	R	All threads and CPU	All	Slow, cached	Addressing perks
Local	RW	Per thread	Thread	Slow, cached	Register spilling
Shared	RW	Per block	Block	Fast	Fast com. ↔ threads
Registers	RW	Per thread	Thread	Fast	Don't use too many!



Visual Profiler

- Cada una de estas memorias esta situada para diferentes tareas.
- El nVidia Visual Profiler nos dice exactamente como se comporta.

Algoritmos

CAPITULO 5

Embarrassingly Parallel Algorithm

- Los algoritmos paralelos pueden ser definidos en términos de cuanta interacción hay entre threads. Los algoritmos donde no hay interacción entre threads, son llamados **embarrassingly parallel**. Estos algoritmos tienen problemas los cuales son mas fácil de romper en tareas separadas, completamente independientes.
- Usualmente hay algoritmos, aunque no necesariamente mas simples, donde los threads necesitan interactuar.
- Algunos prefieren llamarlos **Pleasingly Parallel**. Son extremadamente útiles y pueden llegar a ser muy complicados.

Introducción a la Shared Memory

CAPITULO 6

Shared Memory

- Área pequeña (unos pocos kylobytes) de memoria muy rápida
- Alojada por block, todos los threads en un mismo block comparten la misma memoria para ese block
- Es 100 o mas veces mas rápida que la global memory.
- Puede servir a 32 threads con memoria en tan solo 2 ciclos de reloj
- Casi cualquier uso de la shared memory sobre la global memory que de ventaja debería realizarse

Alojar shared memory estática en un kernel

- En un Kernel CUDA se puede usar el calificador `__shared__` para indicar que la siguiente variable es declarada en shared memory y la cantidad debe especificarse en tiempo de compilación usando este método

```
__global__ void SomeKernel()  
{  
    __shared__ int i;           // Allocate a single int  
    __shared__ float f_array[100]; // Array of 100 floats  
}
```

- Este método es estático porque la cantidad de shared memory alojada debe ser una constante en tiempo de compilación.

Shared Memory Dinámica: 3er parámetro de configuración de ejecución

- Se logra alojar una cantidad de memoria dinámica, agregando esa cantidad como tercer parámetro cuando se lanza el kernel

```
SomeKernel<<<10, 23, 19>>>();
```

- 10 numero de blocks
 - 23 es el numero de threads por block
 - 19 cantidad de shared memory para alojar por block en bytes
- Cuando se aloja una variable en shared memory, hay que especificar su "extern" en el kernel

```
__global__ void SomeKernel()  
{  
    extern __shared__ char sharedBuffer[];  
    //Será leído como:  
    //extern __shared__ char sharedBuffer[19];  
}
```

La configuración de ejecución solo permite 1 parámetro

- Solo 1 parámetro para especificar la shared memory dinámica.
- Si necesitas alojar dos o mas arreglos de shared memory dinámica, tenés que alojar toda la memoria con el 3er parámetro, y manejarlos con punteros en el kernel.

```
__global__ void Somekernel()
{
    extern __share__ char bothBuffers[];

    char* firstArray = &bothBuffers[0];
    float* secondArray = (float*)&bothBuffers[12];
    // Como se ve, se puede mezclar y compatibilizar los tipos
    // si tus arreglos son de diferentes tipos.
    // La direccion de inicio (12) de los arreglos puede ser pasada
    // como un parametro si tienes varias matrices de tamaño dinamico
}
```


Cache L1 de la global memory y Shared memory

- Están en la misma memoria física.
- Hay 64k de esta memoria y puede dividirse por el programador como lo necesite.
- La L1 es una cache automática regular para la global memory.
- El dato es almacenado en L1 y desalojado según sea necesario por la GPU.
- Por otro lado la shared memory está en control del programador
- En este sentido, **la shared memory es realmente una manera de controlar la cache L1.**
- El programador decide cuando el dato se almacena y cuando debe ser desalojado de la shared memory.

Setear cantidad de Shared Memory y de L1

- Se puede usar 16k de L1 y 48k de shared o al revés.
- Depende un poco de la serie de la tarjeta.
- Llamar a **cudaFuncSetCacheConfig** desde el host para setear la cantidad para un kernel particular previo a su lanzamiento.
- Esta es una configuración "preferida".
- Si solicitas 16k de shared para el kernel pero el kernel actualmente usa mas, entonces esa configuración sera ignorada y obtendrás 48k de shared.

```
cudaFuncSetCacheConfig(kernelName, enum cudaFuncCache);  
cudaFuncCache es uno de los siguiente valores:  
  CudaFuncCachePreferNone    0    Default  
  CudaFuncCachePreferShare   1    48k Shared 16kL1  
  CudaFuncCachePreferL1      2    16k Shared 48k L1  
  CudaFuncCachePreferEqual   3    32k Shared 32k L1*
```

Los riesgos en el paralelismo abundan

- La shared memory es compartida por threads dentro de un block, por lo que es un método útil para la comunicación.
- Esto introduce los riesgos comunes de la programación paralela, mientras que compartir recursos debe considerar las **Condiciones de Carrera**.
- Se pueden usar mutexs y semáforos pero con cuidado porque fuerzan la serialización, la cual anula el propósito general de la GPU

Condiciones de Carrera

- Cuando dos threads cambian la misma variable de shared memory a diferentes valores, no tiene sentido a menos que estén coordinadas
- ejemplo:

```
__global__ void SomeKernel()  
{  
    __shared__ int i;  
    i = threadIdx.x;  
}
```

- Cuando los threads compiten por un recursos se dice que hay una
- Condición de Carrera y el valor final de i esta completamente fuera de nuestras manos. i es **undefined**.

__syncthreads();

- __syncthreads() ofrece una barrera conveniente de ancho de block.
- Ningún thread del block pasara de esta función hasta que todos los threads de el block hayan alcanzado ese punto.

```
__shared__ inti;  
i = 0;  
__syncthreads();  
if(threadidx.x == 0) i++;  
__syncthreads();  
if(threadidx.x == 1) i++;
```

- i es incrementada por dos threads. Esto fuerza a que se haga de manera serial y mas lenta pero segura. La salida es i = 2;

Bank Conflicts

CAPITULO 7

Shared Memory

- La shared memory es mucho mas rápida que ir por la global memory, sin embargo, esta puede hacerse lento, entonces vamos a ver un poco como esta organizada y así la aprovecharemos.
- Lo lógico sería leer la memoria desde la global memory en un block y hacer que todos los threads del block manipulen y computen sobre la shared memory, escribiendo los resultados de vuelta en la global memory si es necesario.
- Esto es muy raro y la única manera real de saber si estas lo usando bien es cronometrar tu código y testear unas pocas diferentes opciones. Puedes obtener sorprendentes speed-ups cambiando algunas pocas cosas en el código pero puedes también obtener una performance pobre.

Warps

- Los threads de un block son ejecutados en grupos llamados **warps**.
- Hay 32 threads en un warp, cada uno tiene un **threadidx** consecutivo.
- Los 32 threads ejecutan código a modo SIMD, todos juntos y muy rápidamente. Ellos solamente ejecutan juntos cuando no hay ramificaciones entre los 32 threads. Si hay ramificaciones dentro de los 32 threads, el warp será dividido y cada rama sería ejecutada una después detrás de otra.

Organización (1 / 3)

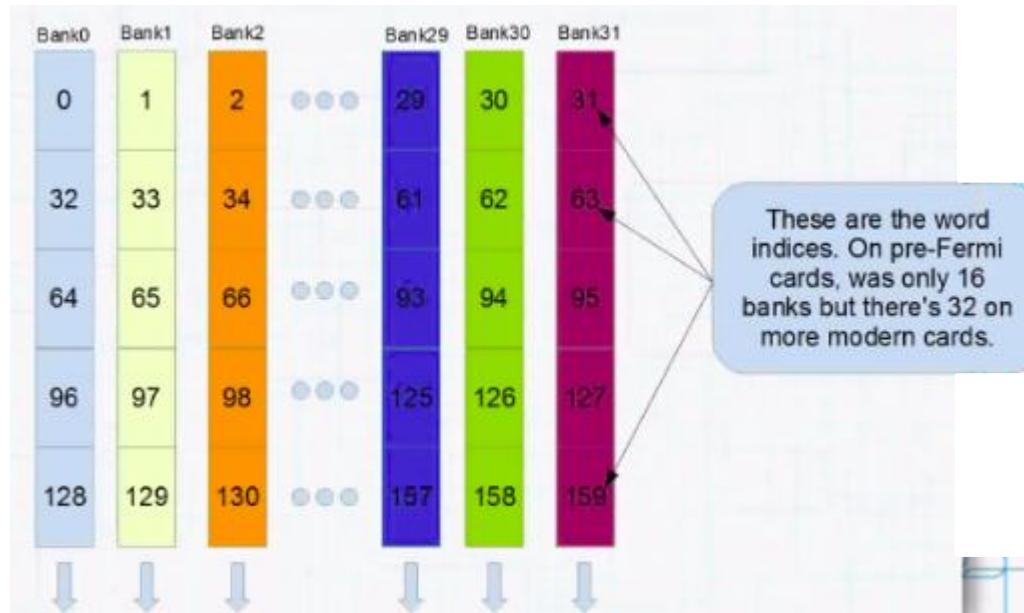
- Los 64k (de shared memory/L1 por SM) es dividida en secciones de 4 bytes llamadas Words. Cualquier word puede ser un int de 32 bit, un float 4 bytes, 2 shorts, la mitad de un double, etc.
- Hay 32 banks. Cada word sucesiva de shared memory pertenece a otro bank (los banks están por color) Así Word[0] pertenece al bank 0, Word[1] al bank 1, etc. el Word[31] pertenece al ultimo bank, el 31. Entonces los banks inician de nuevo y Word[32] es el segundo Word del bank 0, Word[33] pertenece al bank 1, etc.
- La shared memory siempre lee words enteros. Si necesitas leer un único byte, se va a leer los 4 bytes de la word del cual ese byte pertenece. Si necesitas leer una word desalineada (cuya mitad inferior o superior pertenecen a diferentes banks), entonces deben consultarse dos banks. El bank 0 tiene todas los words con índices los cuales son divisibles por 32. El bank 1 tiene los words los cuales dejan resto 1 en la división por 32, y así.



Organización (2/3)

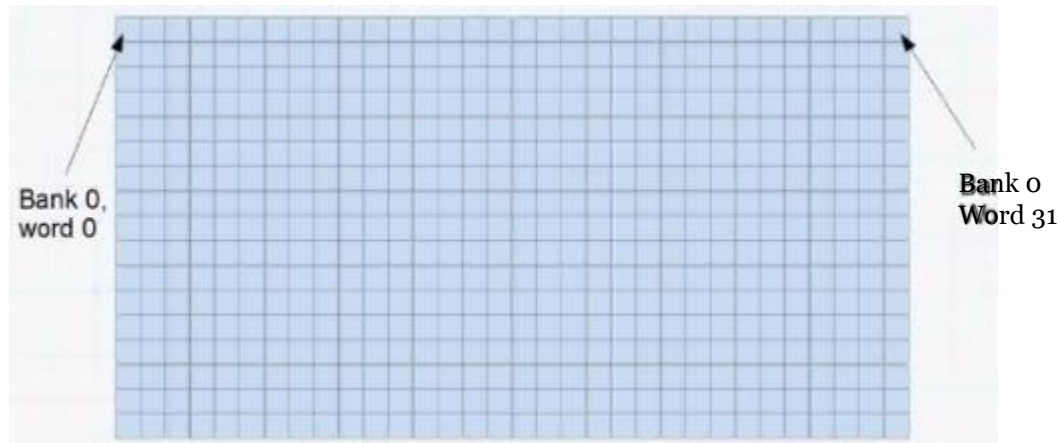
- Las direcciones que un warp consulta pueden referirse a cualquier permutación de los 32 banks como sea.
- La shared memory ejecuta mas rápido cuando hay una consulta desde cada bank por warp.
- Un bank conflict ocurre cuando los threads de un warp requieren diferentes valores de diferentes banks en una única consulta.
- Si todos los threads de un warp consultan exactamente el mismo valor, obtendrás un broadcast. El valor sera leído una vez desde la shared memory y broadcasteado a los threads.
- Si varios threads requieres el mismo valor desde un bank particular, obtendrás un multicast. Esto es una versión mas pequeña del broadcast (solamente para 2.0 o superior). El valor sera leído una vez desde la shared memory.
- Solamente obtendrás broadcast/multicast si una única word desde un bank es referenciado.
- El momento en que diferentes words son consultadas desde cualquier bank único, obtendrás un bank conflict.
- Las consultas son serializadas y el rendimiento baja.

Organización (3/3)



Patrones de Consulta (1 / 7)

- A menudo la forma mas natural de referenciar la shared memory es hacer que cada thread la lea basándose en su threadidx. Supongamos que tenemos un arreglo de words en la shared memory (ellas podrían ser int o float) llamado arr[...] y queremos acceder a las words basándonos sobre varios de los threadidx.



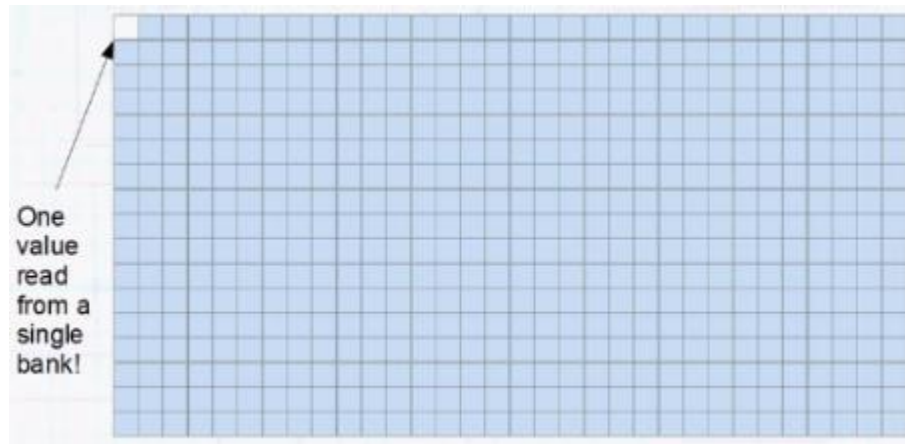
Patrones de Consulta (2/7)

- **Patrón Broadcast.** Obtenemos el valor de una constante para todos los threads, en cada warp hará un broadcast.
- ejemplos:

`arr[threadidx*0]`

`arr[12]`

`arr[blockidx*3]`



Patrones de Consulta (3/7)

- **Patrón por ID.** Usar el thread ID y consultar banks consecutivos, será también realmente rápido igual que Broadcast. Cada thread consulta una word basado en su threadidx:

- ejemplo:

```
arr[threadidx.x] // Accede a una word desde cada bank  
// Todos los threads van a leer desde un  
// diferente bank y será realmente rápido
```

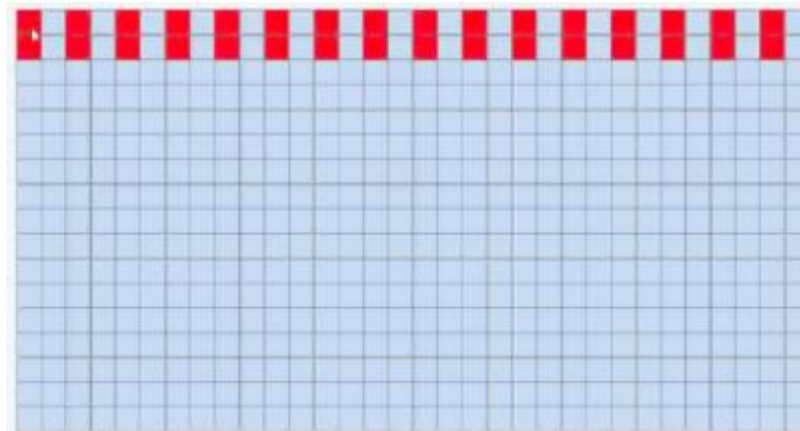


Patrones de Consulta (4/7)

- **Patrón 3:** Los threads de un warp consultan words las cuales son el doble de su threadidx. Esto ocurrirá cuando trabaje con doubles, estructuras de 2 floats, o cualquier otra estructura de datos de 8 bytes.

`arr[threadidx.x*2]`

- Esto será un poco mas lento ya que hay 2 bank conflicts.
- Querrás prevenir patrones como este con banks conflicts.

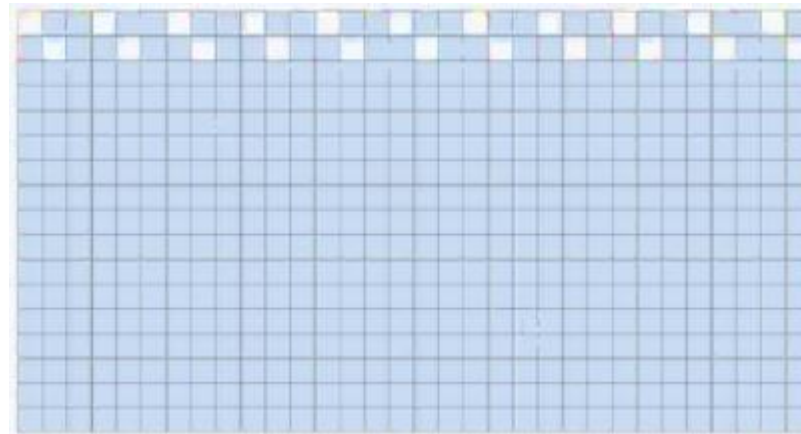


Patrones de Consulta (5/7)

- **Patrón 4:** Estructura de 12 bytes de longitud. Esto puede ocurrir por ejemplo cuando trabajas con puntos 3D X, Y y Z y son floats.

`arr[threadidx.x*3]`

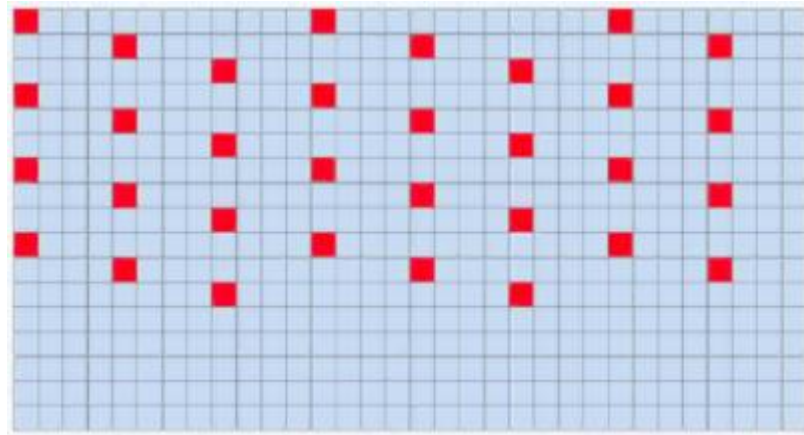
- No hay bank conflicts. Entonces, si estas trabajando con doubles, o alguna otra estructura de 2 words, agregando una word extra de **relleno** puede crear estructuras de 3 words de longitud!



Patrones de Consulta (6/7)

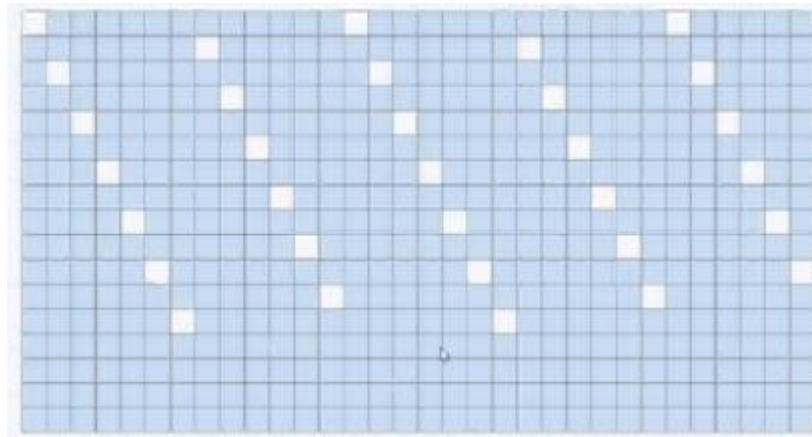
- **Patrón 5:** cada thread accede a las words de 12 partes. Esto puede ocurrir cuando trabaja con estructuras con un tamaño de 48 bytes. Esto producirá un conflicto de bank de 4 vías, cada cuatro bank se pregunta por 4 valores diferentes.

`arr[threadidx.x*12]`



Patrones de Consulta (7/7)

- **Patrón 6:** si agregamos una word extra, el tamaño crece de 48 a 52, el patron de acceso va desde un conflicto de bank de 4 vías a ningun conflicto de banks!
`arr[threadidx.x*13]`
- Cada bank pregunta por un solo valor y obtendrás la misma velocidad que broadcast.



Ocultar la latencia

- El retardo entre threads haciendo consultas desde la shared memory y obteniendo sus datos puede no ser un problema en todos incluso cuando hay banks conflicts.
- Si hay muchos threads corriendo sobre un SM, el scheduler puede simplemente switchear a otro warp mientras los banks descifran el dato consultado.
- La latencia en el peor patrón posible puede estar completamente oculta e irrelevante!
- La única manera real de saber si los banks conflicts están alentando un algoritmo es experimentar con relleno y patrones de acceso.

Conflictos inter-block

- Los threads en diferentes blocks no causan bank conflicts unos con otros. Los bank conflicts son solamente una consideración importante a nivel de warp. Por ejemplo, si tienes un kernel el cual aloja 32 floats de shared memory, entonces la segunda shared memory del block será ubicada directamente después del primero. Pero los dos blocks no tendrán problemas leyendo desde la shared memory al mismo tiempo.
- No importa si un block lee un valor del bank o y otro block lee otro valor del bank o a la vez, no hay conflictos allí.

Usando la función `clock()` para el tiempo

- Una función del device que puede ser usada para grabar el numero de ticks de clock de la GPU. La GPU es mas simple en algunos aspecto a la CPU y los ticks de clock grabados con `clock()` son usualmente mucho mas uniformes que aquellos grabados por lectura del contador time stamp de la CPU.

Conclusión

- Si de los 32 threads de un warp cada uno accede a diferente bank desde la shared memory, la performance es extremadamente rápida!
- Lo mas fácil de hacer es que cada thread en un warp acceda a words consecutivas desde la shared memory.
- La performance alcanzada para unos pocos banks conflicts es mínima, comparada a ir a la cache L2 o a la global memory y el scheduler puede ser capaz de ocultar la latencia si hay suficientes blocks en vuelo a la vez.

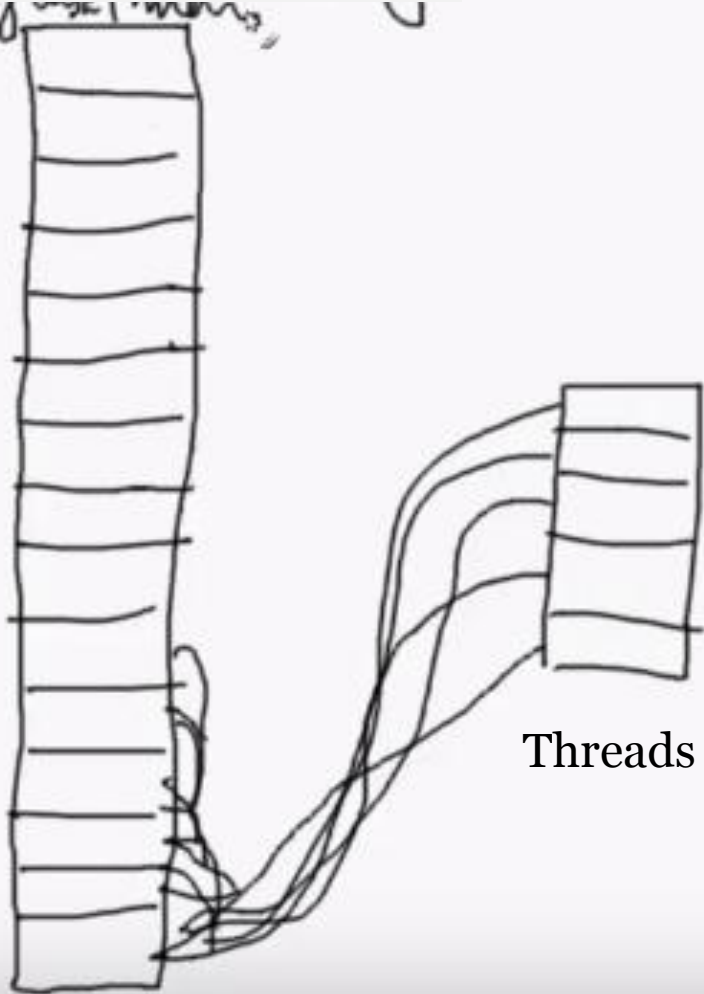
Blocking en la shared memory

CAPITULO 8

Blocking

- El **blocking** es una técnica básica de shared memory realmente útil.
- Supongamos el caso del algoritmo del vecino mas cercano. Si tenemos un manojo de puntos en la global memory copiados desde CPU y un block de threads, cuando queremos bloquear los threads, en el momento que ocurra la copia de los puntos cada thread del block reclamara uno o mas puntos y chequeara la distancia a todos los otros puntos y grabara el mas cercano, todos al mismo tiempo. Esto es un poco lento.
- Si ahora consideramos la shared memory, donde todos los threads comparten una cantidad de memoria, entonces la idea del blocking es leer blocks que son de la global memory desde la shared memory y chequear los puntos desde allí en lugar de volver a la global memory todo el tiempo. De este modo, copiamos un grupo de puntos desde la global memory hacia la shared memory, y aunque los puntos se leen uno a uno, la ventaja es que se hace una sola lectura por punto en la global memory. Una vez que todos los threads chequearon ese grupo de puntos, se copia un nuevo grupo desde la global memory a la shared memory y se repite el proceso.

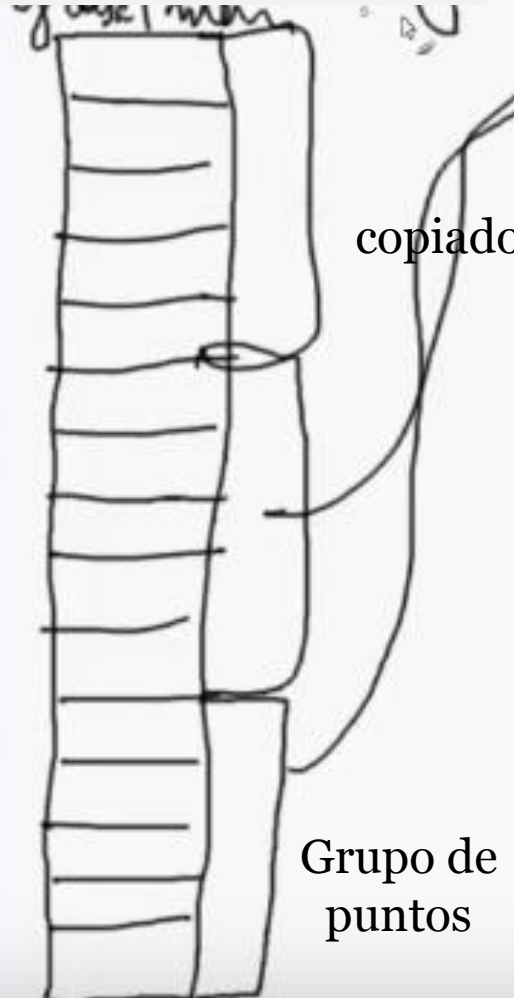
Global Memory



Threads

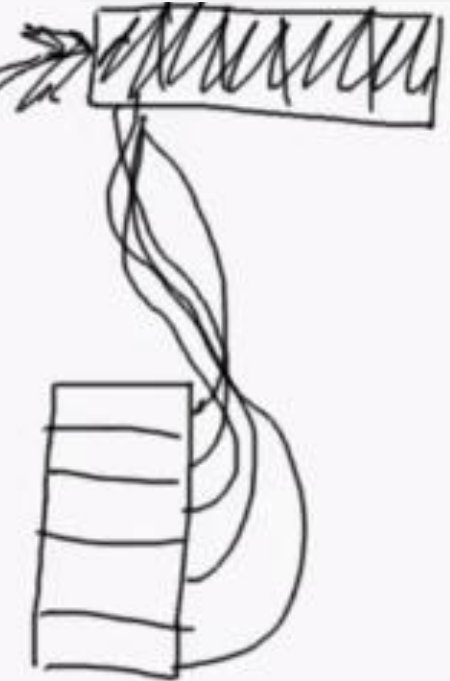
SIN Blocking

Global Memory



Grupo de
puntos

Shared Memory



Threads

CON Blocking

Conclusión

- El Blocking es una técnica donde los blocks de datos son copiados desde la global memory a la shared memory, los threads trabajan sobre los datos desde la shared memory mucho mas rápido. Esto reduce extremadamente la cantidad de trafico sobre el bus de la global memory y permite a los threads usar la shared memory que es mucho mas rápida para la mayoría de los cálculos.
- El Blocking con la shared memory nos da un

Librería Thrust

CAPITULO 9

Librerías CUDA aceleradas por GPU

- CUDA viene con una colección de librerías las cuales nos permiten muy rápidamente implementar algoritmos acelerados por GPU.
- Algunas de ellas son:
 - **Thrust:** Esta es una librería de algoritmos y estructuras de datos para muchas tareas comunes como el ordenamiento y el escaneo. Es una implementación de CUDA de la librería C++ STL.
 - **cuRAND:** Provee pseudo generadores de números aleatorios, para uso por el host y también dentro de los kernels de la GPU.
 - **cuBLAS:** Esta es una librería para subrutinas de algebra lineal básicas, Es una versión de CUDA de la librería BLAS.
 - **cuFFT:** Es una librería de CUDA para calcular la transformada rápida de Fourier usando la GPU.
 - Y hay muchas, muchas mas.

Vectors

- Thrust esta basada en **vectors**, al igual que STL. Hay dos tipos diferentes de vector. El Host vector y el Device vector.
- Los host vectors son almacenados en la system memory, mientras que los device vectors son almacenados sobre la GPU.
- Sus cabeceras son:
 - `<thrust/host_vector.h>`
 - `<thrust/device_vector.h>`

Definiendo los Vectors

- Para definir un nuevo vector use:
 - `// Define un host vector del tipo int y tamaño 100`
 - `thrust::host_vector<int> hv(100);`
 - `//Define un device vector de floats con tamaño 25`
 - `thrust::device_vector<float> dv(25);`
- O puede setear los valores iniciales para su vector suministrando un argumento extra:
 - `thrust::host_vector<int> hv(100,25.Of);`
- La administración de memoria es automática, así que no tienes que alojar ni borrar estos vectors.

Elementos de acceso

- Los elementos de acceso de los vectors pueden hacerse con el operador []:
 - `thrust::device_vector<int> dv(100, 23.4f);`
 - `cout<<"Item 15 es: "<<dv[15]<<endl;`
- Se muy cuidadoso, esto es conveniente para acceder a la device memory, pero hay una copia de memoria desde Device a Host cuando usas un device vector, y eso es muy costoso.
- Igualmente, los vectors pueden ser copiados con el estándar =
 - `thrust::host_vector<int> hv(100, 25.0f);`
 - `thrus::device_vector<int> dv(100);`
 - `dv = hv; // Copia host vector a device`
- O
 - `thrust::copy(hv.begin(), hv.end(), dv.begin());`

Agregando y Removiendo Items

- Es igual que en STL de C++, use `push_back` y `pop_back`:
 - `hv.push_back(1230);` // Sumar 1230 al host vector
 - `dv.push_back(hv.back());` // Sumar 1230 al dev vector
 - `hv.pop_back();` // Elimina 1 elemento desde host
 - `dv.pop_back();` // Elimina 1 elemento desde device

Ordenamiento

- Se puede ordenar un vector con el algoritmo `thrust::sort`.
- Use la cabecera: `<thrust\sort.h>`

```
#include <iostream>
#include <thrust\host_vector.h>
#include <thrust\device_vector.h>
#include <thrust\sort.h>
using namespace std;
int Random() {
    return rand() % 100;
}

int main() {
    thrust::host_vector<int> hv(1000);
    thrust::device_vector<int> dv(1000);
    thrust::generate(hv.begin(), hv.end(), Random);
    dv = hv;    // Copy to device
    thrust::sort(dv.begin(), dv.end()); // Sort
    hv = dv;    // Copy sorted list
    for(int i = 0; i < hv.size(); i++) cout<<"Value: "<<hv[i]<<endl;
    return 0;
}
```

Sumado

- Se puede sumar items en un vector con el algoritmo `thrust::reduce`.
- Use la cabecera `<thrust\reduce.h>`:

```
#include <iostream>
#include <thrust\host_vector.h>
#include <thrust\device_vector.h>
#include <thrust\reduce.h>
using namespace std;
int Sequence() {
    static int i = 0;
    return ++i;
}
int main() {
    thrust::host_vector<int> hv(1000);
    thrust::device_vector<int> dv(1000);
    thrust::generate(hv.begin(), hv.end(), Sequence);
    dv = hv;    // Copy to device
    float sum = thrust::reduce(dv.begin(), dv.end()); // SUM
    cout<<"Sum is: "<<sum<<endl;
    return 0;
}
```