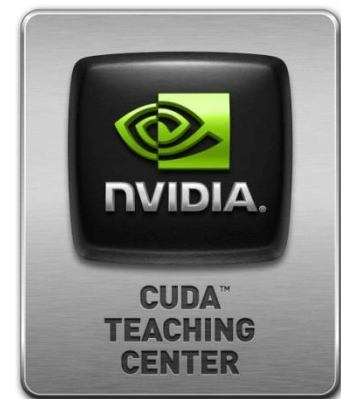


Programación masivamente paralela en procesadores gráficos (GPUs)

E. Dufrechou , P. Ezzatti y M. Pedemonte



Clase 6

Programación CUDA II

Contenido

- **Acceso Coalesced a Memoria Global**
- **Memoria compartida**
 - **Conflicto de bancos**
 - **Tiling**
- **Errores en tiempo de ejecución**
- **Código PTX**
- **Algunas recomendaciones de performance**

Acceso Coalesced a Memoria Global

Acceso Coalesced a Memoria Global

- El acceso a memoria global es por segmentos.
- Incluso cuando solamente se quiere leer una palabra.
- Si no se usan todos los datos de un segmento, se está desperdiciando ancho de banda.
- Los segmentos están alineados a múltiplos de 128 bytes.
- El acceso no alineado es más costoso que el acceso alineado.
- Se desperdicia ancho de banda.

Acceso Coalesced a Memoria Global

- **Coalesced access:**
 - Según Merriam-Webster “to unite into a whole” (unir en un todo).
 - Podríamos traducirlo como acceso unido o fusionado.
- **Cada solicitud de acceso a memoria global de un warp:**
 - se puede partir en varias solicitudes
 - cada solicitud es atendida (issued) independientemente.
- **Los accesos a memoria de hilos de un warp se fusionan en una o más transacciones según características que dependen de las compute capability de la tarjeta.**

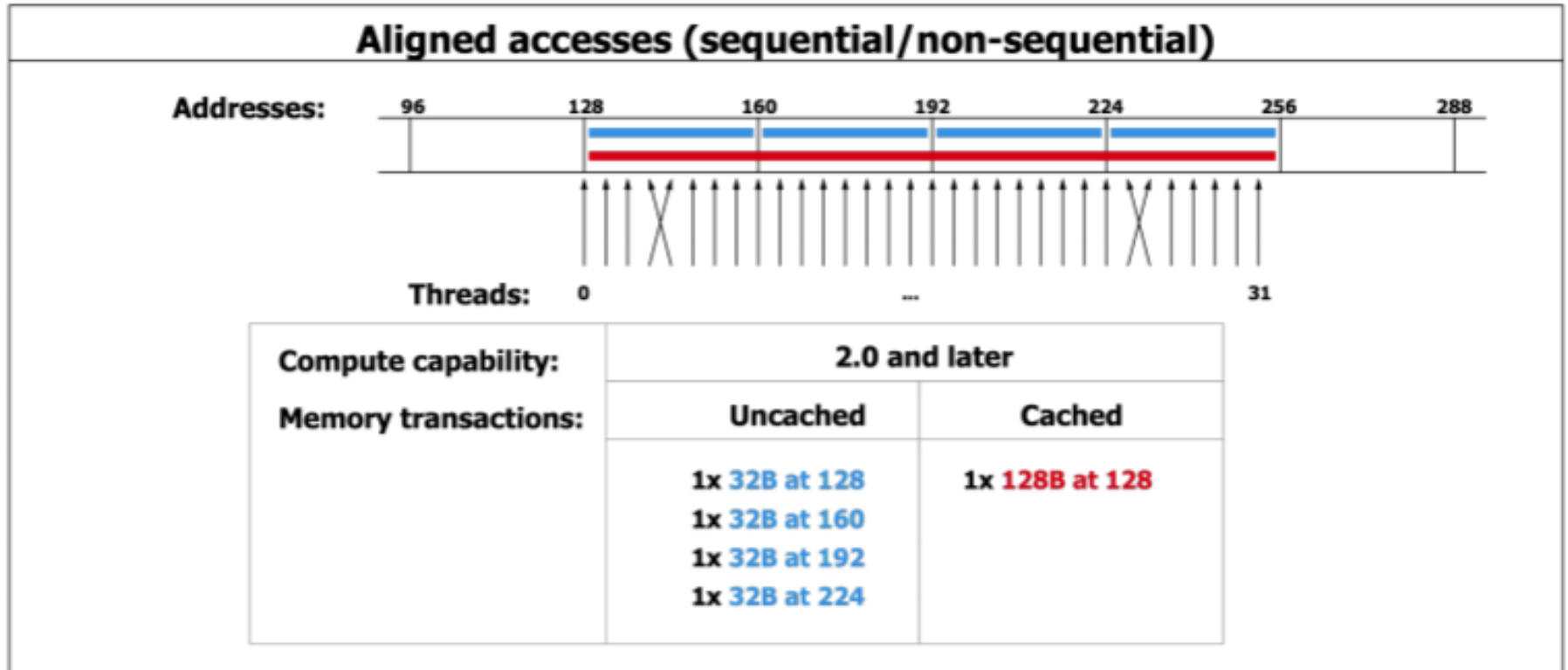
Acceso Coalesced a Memoria Global

- **Desde compute capabilities 3.x:**
 - Las transacciones a memoria global son cacheadas.
 - Hay un caché L1 para cada multiprocesador y un caché L2 compartido por todos los multiprocesadores.
 - Las caché lines son de 128 bytes y se mapean a segmentos alineados de 128 bytes de la memoria global.
 - Los accesos a memoria caché en L1 y L2 usan transacciones de 128 bytes.
 - Los accesos a memoria caché solamente en L2 usan transacciones de 32 bytes. Que los accesos sean solamente a L2 puede configurarse usando modificadores en las instrucciones load y store.

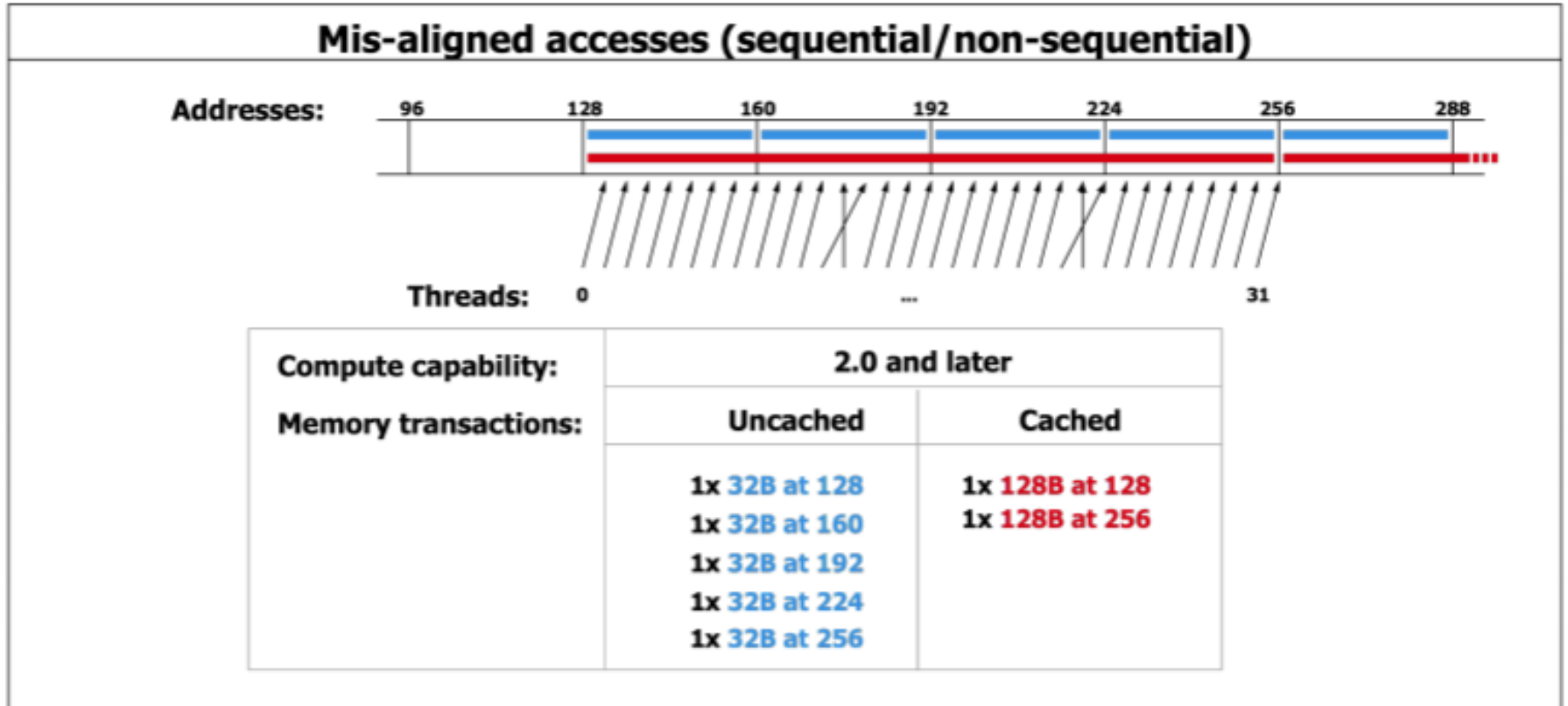
Acceso Coalesced a Memoria Global

- **Desde compute capabilities 3.x:**
 - Si el tamaño de la palabra es de 8 bytes, se realizan dos solicitudes de 128 bytes, una para cada half-warp.
 - Si el tamaño de la palabra es de 16 bytes, se realizan cuatro solicitudes de 128 bytes, una para cada quarter-warp.
 - Cada solicitud se particiona en cache-lines.
 - Si se produce un miss, se accede a la memoria global.
 - Los hilos pueden acceder a las palabras en cualquier orden, incluso a las mismas palabras.

Acceso Coalesced a Memoria Global



Acceso Coalesced a Memoria Global



Acceso Coalesced a Memoria Global

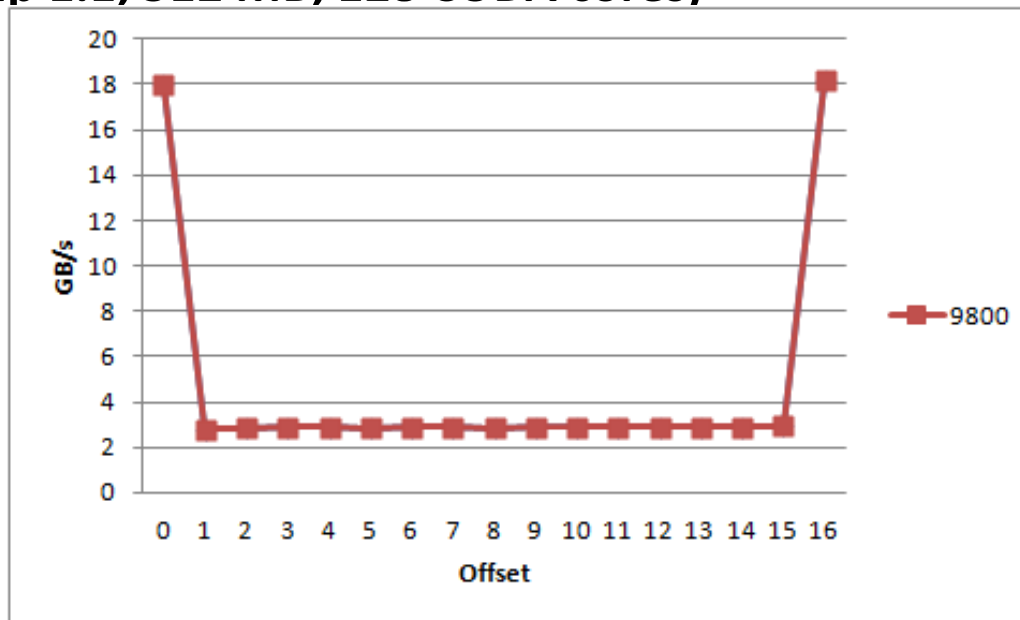
- El espacio de direcciones está particionado en segmentos
- Cuando se solicita una dirección de un segmento, se entregan los datos correspondientes a todas las direcciones del segmento.
- Si todos los hilos del warp acceden al mismo segmento, se hace una sola solicitud y se usan todos los datos.
- Cuando los accesos están distribuidos entre distintos segmentos:
 - Se realizan múltiples solicitudes
 - Hay datos a los que se accede y que se transfieren de la memoria a los multiprocesadores que no son usados por los hilos
- Ejemplo de acceso de un warp serán a direcciones consecutivas:
 - `A[threadIdx.x]`

Acceso Coalesced a Memoria Global

- **Ejemplo 1:**

```
__global__ void CopiaOffset(float *output, float *input, int offset) {  
    int idx = blockIdx.x * blockDim.x + threadIdx.x + offset;  
    output[idx] = input[idx];  
}
```

- **Lanzando 10000 veces 320 bloques de 128 hilos en una 9800 GTX+ (Comp. Cap 1.1, 512 MB, 128 CUDA cores)**



Acceso Coalesced a Memoria Global

- **Ejemplo 1:**

```
__global__ void CopiaOffset(float *output, float *input, int offset) {  
    int idx = blockIdx.x * blockDim.x + threadIdx.x + offset;  
    output[idx] = input[idx];  
}
```

- **Lanzando 10000 veces 320 bloques de 128 hilos en una 480 (Comp. Cap 2.0, 1536 MB, 480 CUDA cores)**

Offset	GB/s
0	49.22
1 a 15	44.88
16	46.24
17 a 31	44.88
32	49.22

El pico teórico es 177 GB/s

Acceso a Memoria Global

- **Attenti al lupo:**
 - Cuando una instrucción no atómica ejecutada por un warp debe escribir en la misma dirección de la memoria global para más de un hilo del warp
 - Además del problema de race condition que se produce, de acuerdo al funcionamiento de CUDA solamente un hilo realiza la escritura
 - Está indefinido cual de los hilos!!!

Acceso Coalesced a Memoria Global

- **Ejemplo 2 (Robert Strzodka):**

Array of Structs (AoS)

```
struct NormalStruct {
    Type1 comp1;
    Type2 comp2;
    Type3 comp3;
};

typedef NormalStruct
    AoSContainer[SIZE];

AoSContainer container;
```

Struct of Arrays (SoA)

```
struct SoAContainer{
    Type1 comp1[SIZE];
    Type2 comp2[SIZE];
    Type3 comp3[SIZE];
};

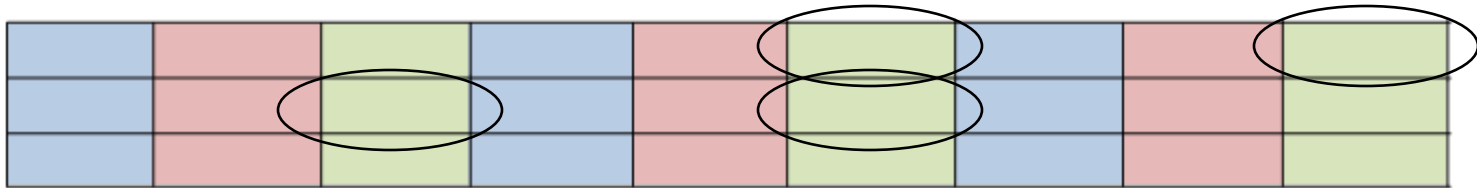
SoAContainer container;
```

Acceso Coalesced a Memoria Global

- Ejemplo 2 (Robert Strzodka):

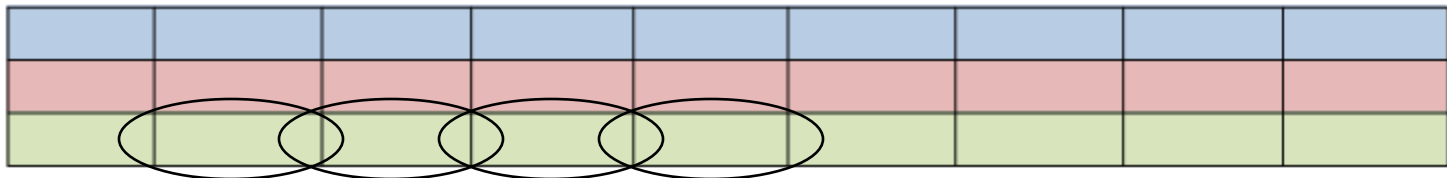
Array of Structs (AoS):

```
container[1].comp3;  
container[2].comp3;  
container[3].comp3;  
container[4].comp3;
```



Struct of Arrays (SoA):

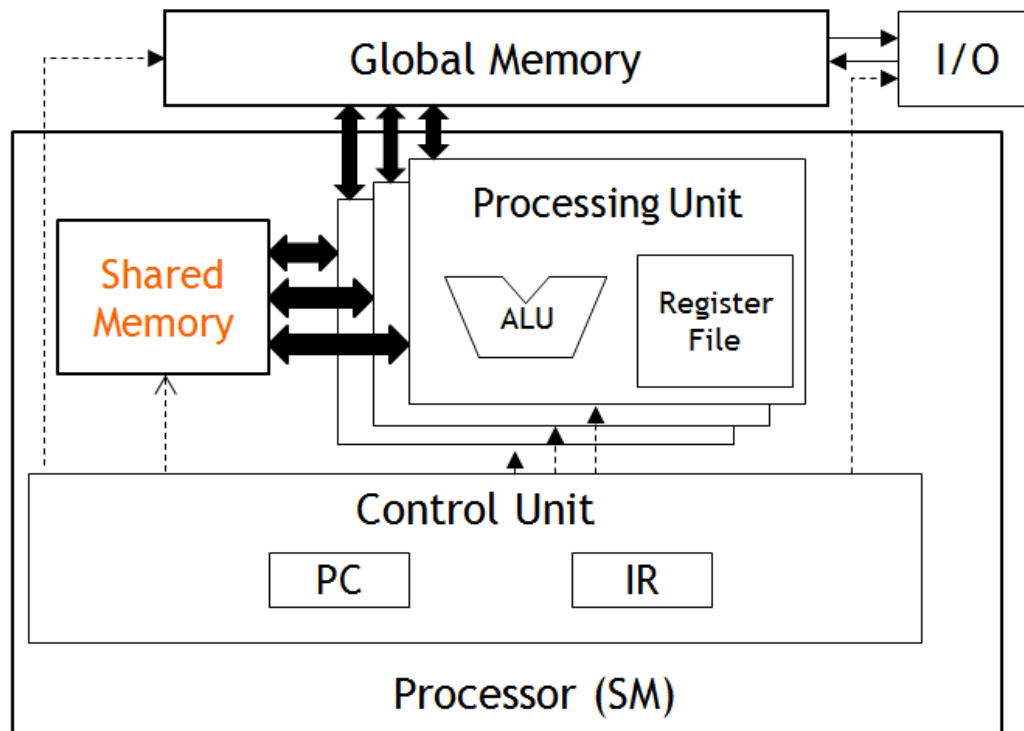
```
container.comp3[1];  
container.comp3[2];  
container.comp3[3];  
container.comp3[4];
```



Memoria Compartida

Memoria Compartida

- Es cientos de veces más rápida (tanto en latencia como en performance) que la memoria global.
- Hay una memoria compartida en cada multiprocesador.



Memoria Compartida

- Los contenidos y el uso de este tipo de memoria deben ser explícitamente definidos por el usuario en el código del kernel.
- El alcance es a nivel del bloque de hilos.
- Por ello permite que los hilos de un mismo bloque puedan cooperar.
- El tiempo de vida corresponde al tiempo de vida del bloque de hilos.
- Es decir que el contenido se pierde cuando los hilos del bloque finalizan su ejecución.

Memoria Compartida

- Para declarar la memoria compartida se puede usar una constante:

```
#define CANT_HILOS 128

__global__ void reduction(float * output, float * input) {
    __shared__ float compartida1[CANT_HILOS];
    __shared__ float compartida2[CANT_HILOS];
    ...
};

reduction <<<N_BLOCK,CANT_HILOS>>> (output,input);
```

Memoria Compartida

- También puede ser una matriz:

```
#define TILE_WIDTH 128

__global__ void reduction(float * output, float * input) {
    __shared__ float compartida[TILE_WIDTH][TILE_WIDTH];
    ...
};

reduction <<<N_BLOCK,CANT_HILOS>>> (output,input);
```

Memoria Compartida

- Para declarar la memoria compartida se puede hacer en forma dinámica con `extern`:

```
__global__ void reduction(float * output, float * input) {  
    extern __shared__ float compartida[];  
    ...  
};  
  
reduction <<<N_BLOCK, CANT_HILOS,  
            CANT_HILOS*sizeof(float)>>> (output, input);
```

Memoria Compartida

- También es posible “partir” el tamaño reservado cuando se usa `extern` en varias estructuras:

```
__global__ void reduction(float * output, float * input) {
    extern __shared__ float auxiliar[];

    float* compartida1 = auxiliar;
    float* compartida2 = (float*)&auxiliar[CANT_HILOS];
    ...
};

reduction <<<N_BLOCK, CANT_HILOS,
            2*CANT_HILOS*sizeof(float)>>> (output, input);
```

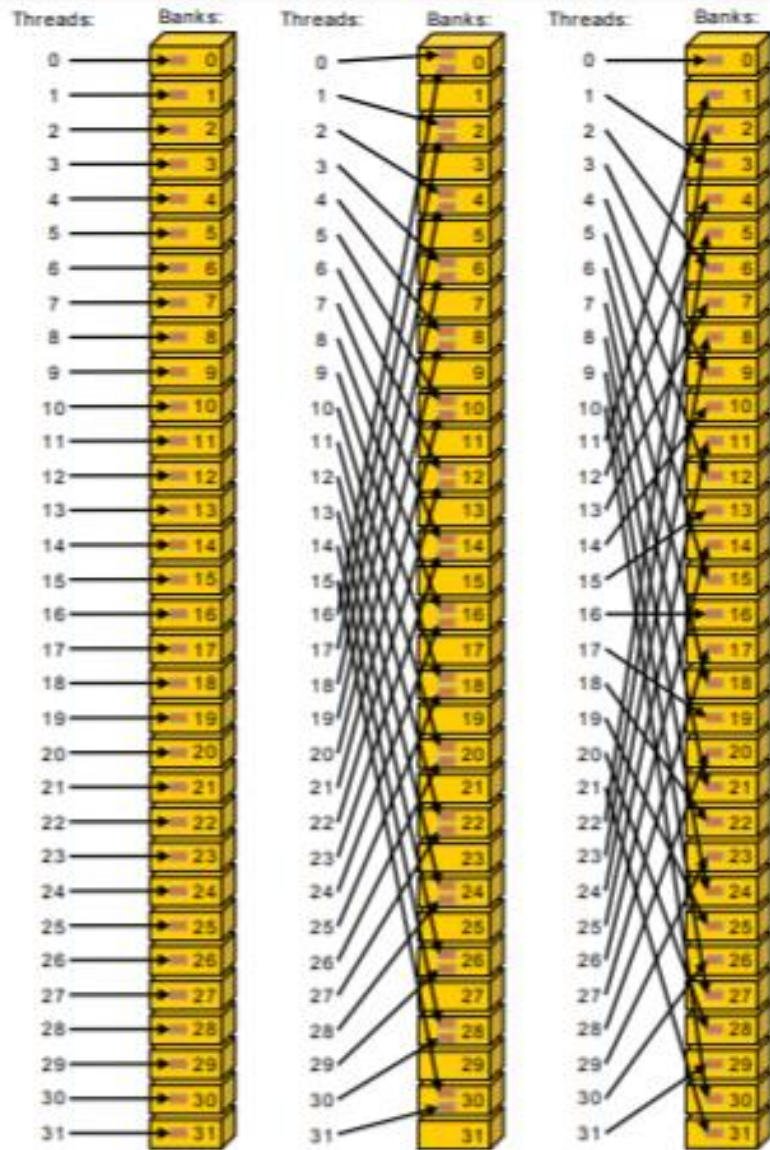
Memoria Compartida

- La memoria compartida se divide en módulos del mismo tamaño llamados bancos.
- En compute capabilities 3.X hay dos modos de direccionamiento: de 32 bits y de 64 bits.
- En compute capabilities posteriores solamente hay direccionamiento de 32 bits, por lo que para la explicación nos centraremos en ese caso.

Memoria Compartida

- **Palabras de 32 bits contiguas están en bancos contiguos.**
- **Los bancos pueden ser accedidos simultáneamente a nivel de warp.**
- **Si las lecturas o escrituras caen todas en bancos distintos, pueden ser atendidas simultáneamente.**
- **Cada banco puede atender una única solicitud por ciclo.**
- **En general, si dos solicitudes caen en el mismo banco de memoria, se produce un conflicto (bank conflict) y el acceso al banco es serializado:**
 - **Lectura de palabras distintas**
 - **Escrituras**

Memoria Compartida



Left

Linear addressing with a stride of one 32-bit word (no bank conflict).

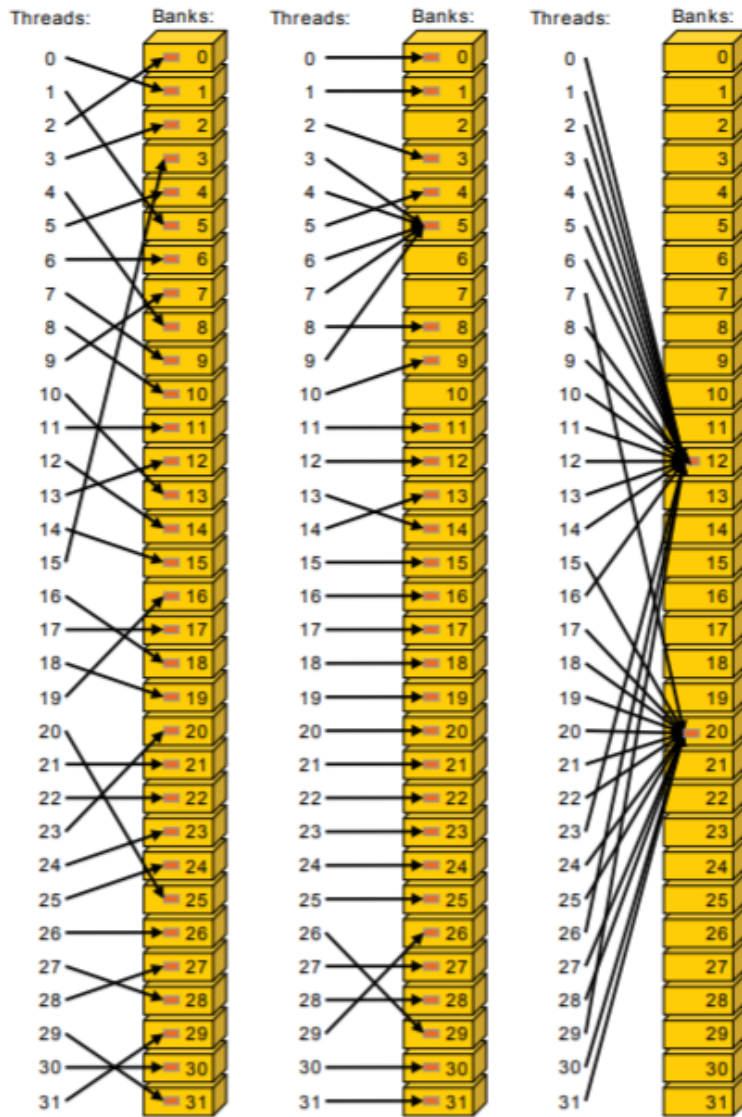
Middle

Linear addressing with a stride of two 32-bit words (two-way bank conflict).

Right

Linear addressing with a stride of three 32-bit words (no bank conflict).

Memoria Compartida



Left

Conflict-free access via random permutation.

Middle

Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.

Right

Conflict-free broadcast access (threads access the same word within a bank).

Memoria Compartida

- **No hay conflicto y el acceso es rápido cuando:**
 - Todos los hilos del warp acceden a diferentes bancos.
 - Hilos del warp leen la misma palabra de un banco (broadcast).

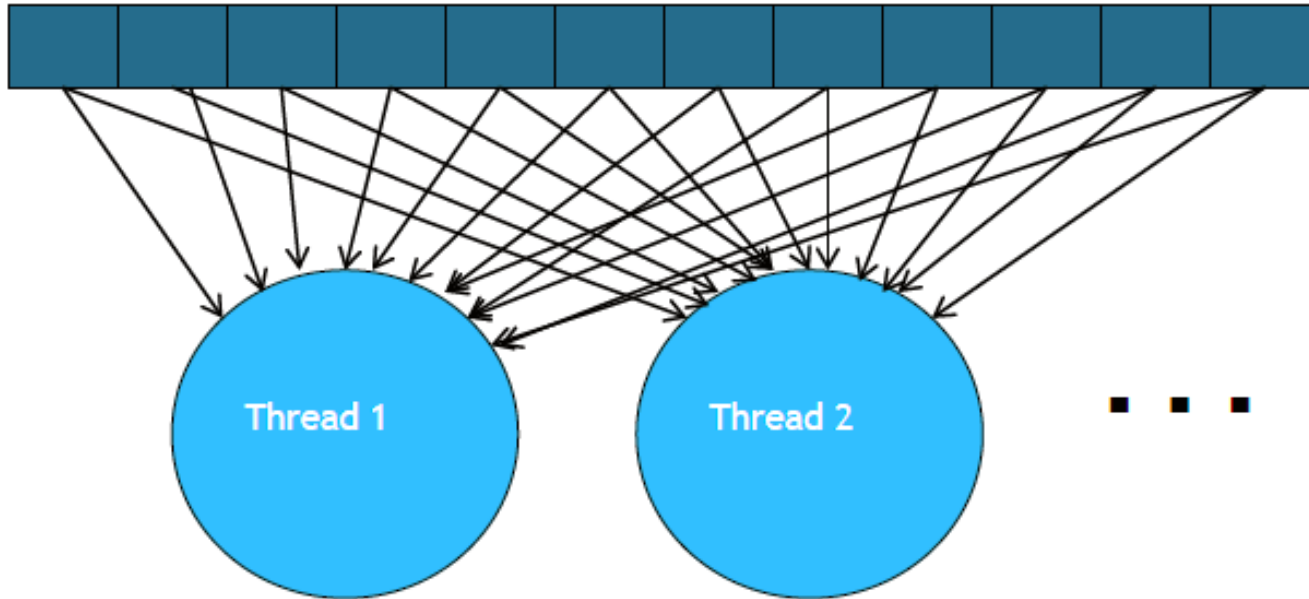
- **El acceso es más lento cuando:**
 - Varios hilos del warp acceden a palabras distintas del mismo banco.
 - Se produce un conflicto y se debe serializar el acceso.
 - Se requieren tantos ciclos como el máximo número de accesos al mismo banco.

Memoria Compartida

- Debido a que es más rápida que la memoria global suele usarse como una especie de caché para reducir los accesos a memoria global.
- También permite evitar accesos no coalesced a la memoria global:
 - Los datos se guardan en forma intermedia en la memoria compartida.
 - Se reordena el acceso a los datos para que cuando se copien de memoria compartida a memoria global el acceso sea coalesced.
- Esto se puede hacer de a pedazos si se debe cargar muchos datos.
- En ese caso, se lo conoce con el nombre de tiling.

Tiling

Global Memory

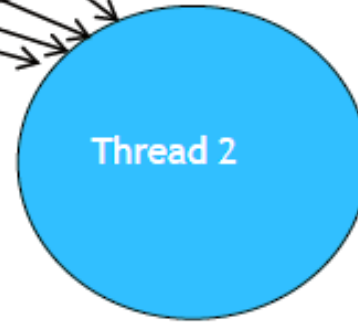
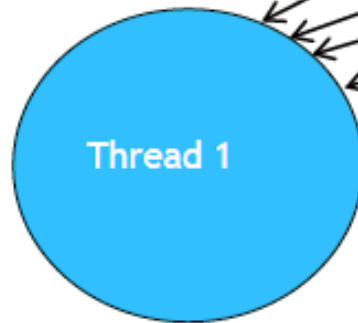


Tiling

Global Memory



On-chip Memory



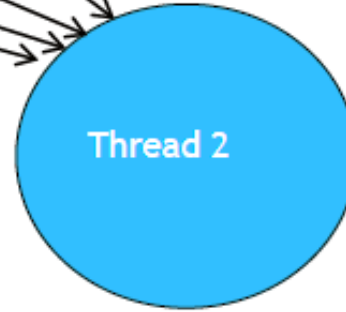
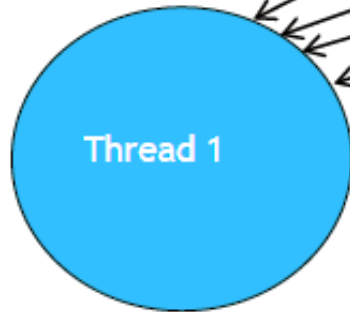
- Se divide el contenido de la memoria global en tiles (bloques?)
- Se concentra el cálculo de los hilos en un tile o un pequeño número de tiles en cada momento.

Tiling

Global Memory



On-chip Memory



- Se divide el contenido de la memoria global en tiles (bloques?)
- Se concentra el cálculo de los hilos en un tile o un pequeño número de tiles en cada momento.

Tiling

- **Esquema general del Tiling:**
 - Identificar un tile de datos de memoria global que deban ser accedidos por múltiples hilos.
 - Cargar el tile desde memoria global a la memoria compartida.
 - Utilizar los mecanismos de sincronización (`__syncthreads()`) para asegurarse que todos los datos necesarios para el procesamiento están cargados.
 - Los hilos hacen los cálculos correspondientes accediendo a los datos en la memoria compartida.
 - Utilizar los mecanismos de sincronización para asegurarse que todos los hilos completaron esta etapa del procesamiento.
 - Se puede pasar a trabajar sobre el siguiente tile.

Errores en tiempo de ejecución

Errores en tiempo de ejecución

- **CUDA no avisa cuando se produce un error.**
- **Veamos un ejemplo:**

```
#include <cuda_runtime.h>

int main(int argc, char *argv[]){

    float * inputGPU = NULL;
    int size = 1000 * sizeof(float);
    float * inputCPU = NULL;
    inputCPU = (float*) malloc (size);

    for (int j=0;j<1000;j++) {
        inputCPU[j]=j+1.1f;
    }

    cudaMemcpy(inputGPU,inputCPU,size,cudaMemcpyHostToDevice);
    cudaFree(inputGPU);
    return 0;
}
```

Errores en tiempo de ejecución

- **CUDA provee cuatro funciones para el manejo de errores:**
 - `cudaError_t cudaGetLastError ()` : **Devuelve el último error de una invocación en tiempo de ejecución. Resetea el estado a `cudaSuccess`.**
 - `cudaError_t cudaPeekAtLastError ()` : **Devuelve el último error de una invocación en tiempo de ejecución. NO resetea el estado a `cudaSuccess`.**
 - `char* cudaGetErrorName(cudaError_t error)` : **Dado un código de error devuelve la string que representa el error.**
 - `char* cudaGetErrorString(cudaError_t error)` : **Dado un código de error devuelve la descripción del error.**

Errores en tiempo de ejecución

- Las invocaciones a operaciones como transferencias o reservas y liberación de memoria ya devuelven un `cudaError_t`.
- Por lo que basta con usar la función `cudaGetErrorString` para desplegar el error.
- Por ejemplo se puede definir:

```
#define CUDA_CHK(ans) { gpuAssert((ans), __FILE__, __LINE__); }  
  
inline void gpuAssert(cudaError_t code, const char *file, int line,  
bool abort=true){  
  
    if (code != cudaSuccess){  
        fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code),  
file, line);  
        if (abort) exit(code);  
    }  
}
```

- Y se usa `CUDA_CHK` como wrapper de la invocación.

Errores en tiempo de ejecución

A ver...

```
#include <cuda_runtime.h>

int main(int argc, char *argv[]){

    float * inputGPU = NULL;
    int size = 1000 * sizeof(float);
    float * inputCPU = NULL;
    inputCPU = (float*) malloc (size);

    for (int j=0;j<1000);j++) {
        inputCPU[j]=j+1.1f;
    }

    cudaMemcpy(inputGPU,inputCPU,size,cudaMemcpyHostToDevice);
    CUDA_CHK( cudaFree(inputGPU) );
    return 0;
}
```

Sigue sin fallar!!!! y entonces????

Errores en tiempo de ejecución

- **Los errores de CUDA son “asíncronos”.**
 - El fallo se produce en `cudaMemcpy`.
 - La llamada a `cudaFree` se realiza pero sin ejecutarse.
- **Sugerencias:**
 - Envolver todas las llamadas a la biblioteca CUDA con `CUDA_CHK()`

Errores en tiempo de ejecución

Ahora si!!!

```
#include <cuda_runtime.h>

int main(int argc, char *argv[]){

    float * inputGPU = NULL;
    int size = 1000 * sizeof(float);
    float * inputCPU = NULL;
    inputCPU = (float*) malloc (size);

    for (int j=0;j<1000);j++) {
        inputCPU[j]=j+1.1f;
    }

    CUDA_CHK(cudaMemcpy(inputGPU,inputCPU,size,cudaMemcpyHostToDevice));
    CUDA_CHK(cudaFree(inputGPU));
    return 0;
}
```

GPUassert: invalid argument prueba2.cu 26 (línea del cudaMemcpy)

Errores en tiempo de ejecución

- **En las invocaciones a kernels:**
 - **No se puede envolver la llamada al kernel con `CUDA_CHK()`**
 - **Como los kernels se ejecutan de forma asíncrona, se debe realizar un `cudaDeviceSynchronize()` luego de la invocación al kernel.**
 - **La invocación a `cudaDeviceSynchronize()` puede ser envuelta `CUDA_CHK()`.**
 - **La sincronización obliga a que el kernel llegue hasta el final de su ejecución, por lo que nos devuelve los errores en la ejecución del kernel.**

Errores en tiempo de ejecución

- **Veamos con un ejemplo, agreguemos kernel1:**

```
__global__ void kernel1(float *v)
{
    int i = threadIdx.x;

    v[i*1000] = v[i]+v[i];
}
```

Errores en tiempo de ejecución

```
#include <cuda_runtime.h>

int main(int argc, char *argv[]){

    float * inputGPU = NULL;
    int size = 1000 * sizeof(float);
    float * inputCPU = NULL;
    inputCPU = (float*) malloc (size);

    for (int j=0;j<1000;j++) {
        inputCPU[j]=j+1.1f;
    }
    kernel1<<<1,10000>>>(inputGPU);
    CUDA_CHK (cudaDeviceSynchronize());
    CUDA_CHK(cudaMemcpy(inputGPU,inputCPU,size,cudaMemcpyHostToDevice));
    CUDA_CHK(cudaFree(inputGPU));
    return 0;
}
```

**GPUassert: invalid argument prueba3.cu 36 (línea del
cudaDeviceSynchronize)**

Errores en tiempo de ejecución

```
#include <cuda_runtime.h>

int main(int argc, char *argv[]){

    float * inputGPU = NULL;
    int size = 1000 * sizeof(float);
    CUDA_CHK (cudaMalloc((void **)&inputGPU, size));
    float * inputCPU = NULL;
    inputCPU = (float*) malloc (size);

    for (int j=0;j<1000;j++) {
        inputCPU[j]=j+1.1f;
    }
    CUDA_CHK(cudaMemcpy(inputGPU,inputCPU,size,cudaMemcpyHostToDevice));
    kernell1<<<1,10000>>>(inputGPU);
    CUDA_CHK (cudaDeviceSynchronize());
    CUDA_CHK(cudaFree(inputGPU));
    return 0;
}
```

NO DA ERROR!!!! Cómo puede ser???

Errores en tiempo de ejecución

- Hay dos errores:
 - Uno en la invocación: no pueden haber 10000 hilos en un bloque.
 - Otra en el propio kernel: que accede a memoria no reservada.
 - El primer error hace que el kernel no se ejecute por lo que no detectamos el error en el `cudaDeviceSynchronize()` pero tampoco es capturado.
 - Para capturar ese error debemos hacer una invocación a `cudaGetLastError()` envuelta en `CUDA_CHK()` entre la llamada al kernel y la sincronización.

Errores en tiempo de ejecución

```
#include <cuda_runtime.h>

int main(int argc, char *argv[]){

    float * inputGPU = NULL;
    int size = 1000 * sizeof(float);
    CUDA_CHK cudaMalloc((void **)&inputGPU, size));
    float * inputCPU = NULL;
    inputCPU = (float*) malloc (size);

    for (int j=0;j<1000;j++) {
        inputCPU[j]=j+1.1f;
    }
    CUDA_CHK(cudaMemcpy(inputGPU,inputCPU,size,cudaMemcpyHostToDevice));
    kernel1<<<1,10000>>>(inputGPU);
    CUDA_CHK (cudaGetLastError() );
    CUDA_CHK (cudaDeviceSynchronize());
    CUDA_CHK(cudaFree(inputGPU));
    return 0;
}
```

**GPUassert: invalid configuration argument prueba3.cu 38 (línea del
cudaGetLastError())**

Errores en tiempo de ejecución

```
#include <cuda_runtime.h>

int main(int argc, char *argv[]){

    float * inputGPU = NULL;
    int size = 1000 * sizeof(float);
    CUDA_CHK cudaMalloc((void **)&inputGPU, size));
    float * inputCPU = NULL;
    inputCPU = (float*) malloc (size);

    for (int j=0;j<1000;j++) {
        inputCPU[j]=j+1.1f;
    }
    CUDA_CHK(cudaMemcpy(inputGPU,inputCPU,size,cudaMemcpyHostToDevice));
    kernel1<<<1,1000>>>(inputGPU);
    CUDA_CHK (cudaGetLastError() );
    CUDA_CHK (cudaDeviceSynchronize());
    CUDA_CHK(cudaFree(inputGPU));
    return 0;
}
```

**GPUassert: an illegal memory access was encountered prueba3.cu 39
(línea del cudaDeviceSynchronize())**

Errores en tiempo de ejecución

- **En resumen:**
 - **Envolver todas las llamadas a la biblioteca CUDA con `CUDA_CHK()`**
 - **Incluir un `CUDA_CHK(cudaGetLastError())` inmediatamente después de la invocación al kernel.**
 - **Incluir un `CUDA_CHK(cudaDeviceSynchronize())` inmediatamente después del paso anterior.**

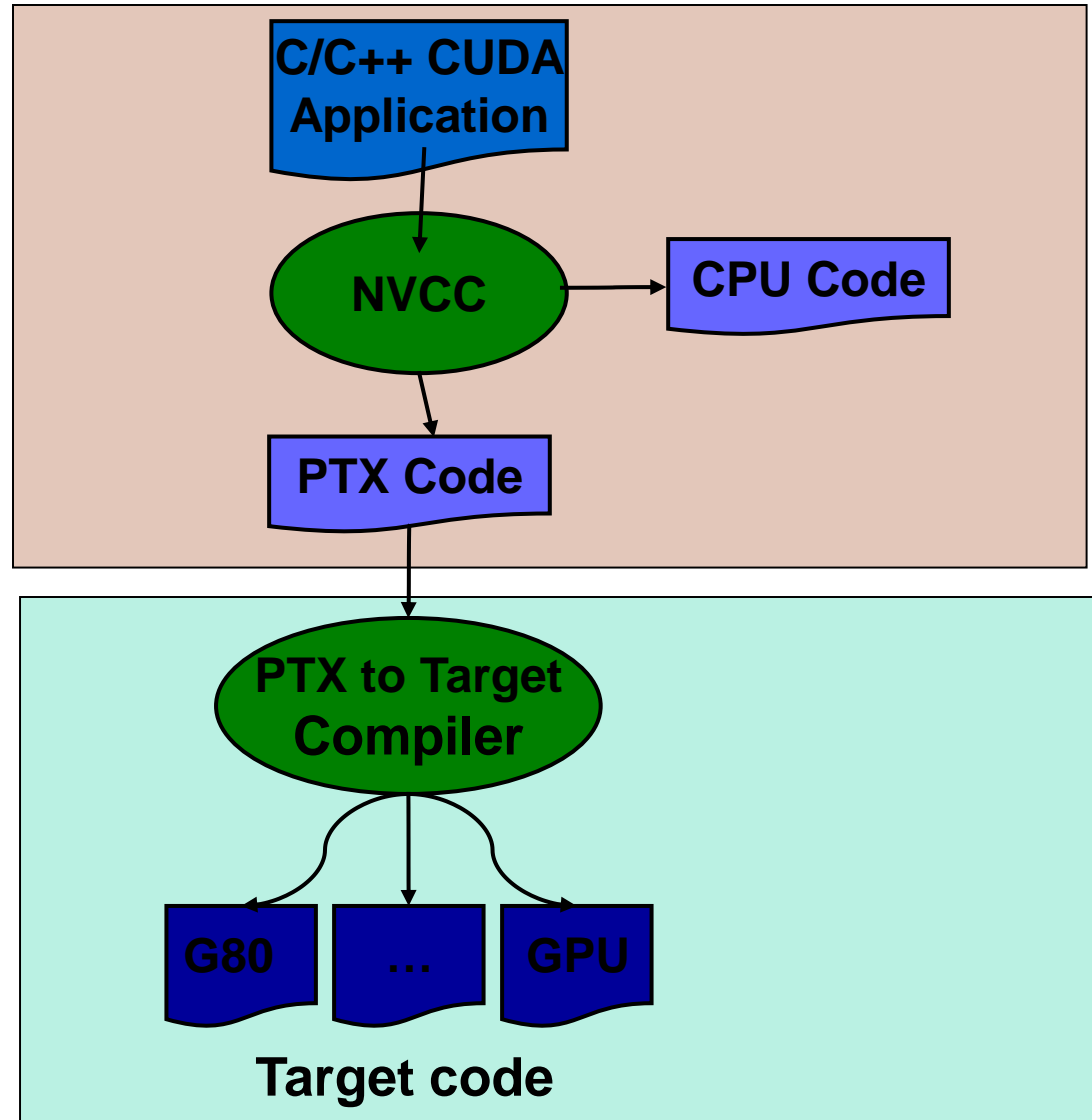
Código PTX

Código PTX

Recordemos la compilación

El código PTX puede obtenerse

Puede dar pistas de posibles optimizaciones del código



Código PTX

- El código PTX se puede obtener compilando con la flag `-ptx`:

```
nvcc -ptx algo.cu
```

- Se genera el archivo `algo.ptx`
- El código PTX permite hilar muy fino en aspectos del código CUDA que impactan en la performance.

Código PTX

```
__global__ void CopiaRara(float * output, float * input){  
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  
    output[idx] = input[idx] * 2.27;  
}
```

- **Inspeccionemos el código resultante (ver archivo1.ptx).**

```
ld.global.f32    %f1, [%rd4+0];           // id:17  
cvt.f64.f32     %fd1, %f1;               //  
mov.f64         %fd2, 0d400228f5c28f5c29; // 2.27  
mul.f64         %fd3, %fd1, %fd2;       //  
cvt.rn.f32.f64 %f2, %fd3;              //
```

Código PTX

```
__global__ void CopiaRara(float * output, float * input){  
    int idx = blockIdx.x * blockDim.x + threadIdx.x;  
    output[idx] = input[idx] * 2.27f;  
}
```

- **Inspeccionemos el código resultante (ver archivo2.ptx).**
- **El mismo fragmento de programa se transformó en:**

```
ld.global.f32    %f1, [%rd4+0];           // id:17  
mov.f32          %f2, 0f401147ae;        // 2.27  
mul.f32          %f3, %f1, %f2;
```

Algunas recomendaciones adicionales sobre performance

Algunas recomendaciones adicionales sobre performance

- Evitar la divergencia de hilos dentro de un warp.
- El número de bloques debe ser mayor al número de multiprocesadores:
 - De forma de mantener a todos los multiprocesadores ocupados.
 - Para permitir ocultar latencias cuando un bloque está trancado con un `__syncthreads()` debe ser mayor al doble del número de multiprocesadores.
- El número de hilos por bloque debe ser un múltiplo de 32 (tamaño de warp).