

Contenidos

1. Introducción
2. Arquitectura y programación de CUDA
- 3. Optimización y depuración de código**
 - 1. Optimización de código**
 2. Depuración y Medición de uso de recursos
4. Librerías basadas en CUDA
5. Alternativas a NVIDIA/CUDA
6. Conclusiones
7. Bibliografía
8. ANEXO: programación híbrida CPU+GPU

Contenidos

1. Introducción
2. Arquitectura y programación de CUDA
- 3. Optimización y depuración de código**
 - 1. Optimización de código**
 1. Medición de tiempos
 2. Transferencias host ↔ device
 3. Configuración ejecución
 4. Memoria global
 5. Memoria compartida
 2. Depuración y Medición de uso de recursos
4. Librerías basadas en CUDA
5. Alternativas a NVIDIA/CUDA
6. Conclusiones
7. Bibliografía
8. ANEXO: programación híbrida CPU+GPU

3.1. Optimización de código

Medición de tiempos

Uso de temporizadores de GPU y de eventos para medición de tiempo

```
float time;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start, 0);
    kernel<<<grid, threads>>> (parameters);
cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&time, start, stop );

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

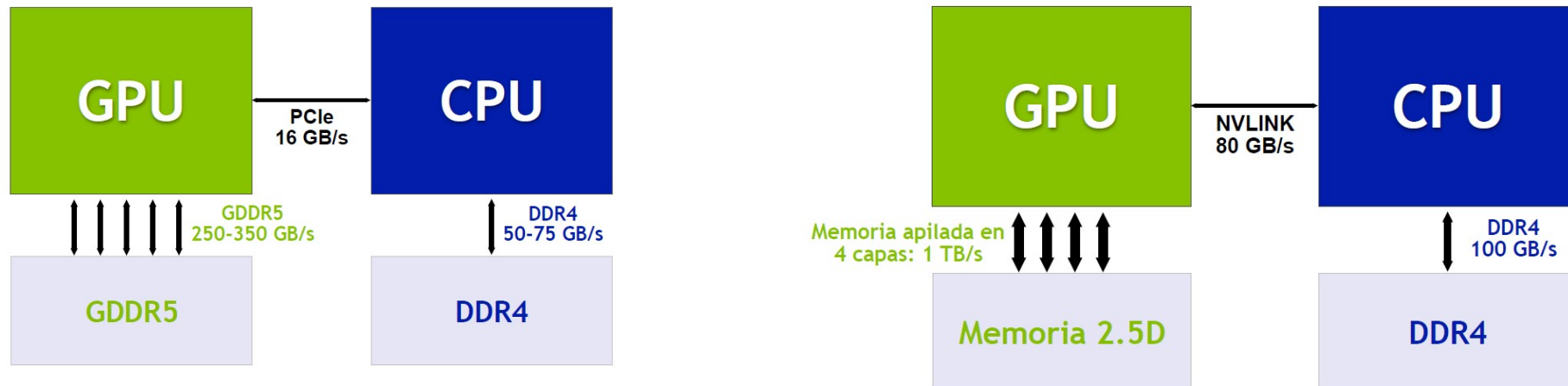
Contenidos

1. Introducción
2. Arquitectura y programación de CUDA
- 3. Optimización y depuración de código**
 - 1. Optimización de código**
 1. Medición de tiempos
 - 2. Transferencias host ↔ device**
 3. Configuración ejecución
 4. Memoria global
 5. Memoria compartida
 2. Depuración y Medición de uso de recursos
4. Librerías basadas en CUDA
5. Alternativas a NVIDIA/CUDA
6. Conclusiones
7. Bibliografía
8. ANEXO: programación híbrida CPU+GPU

3.1. Optimización de código

Transferencias host-device

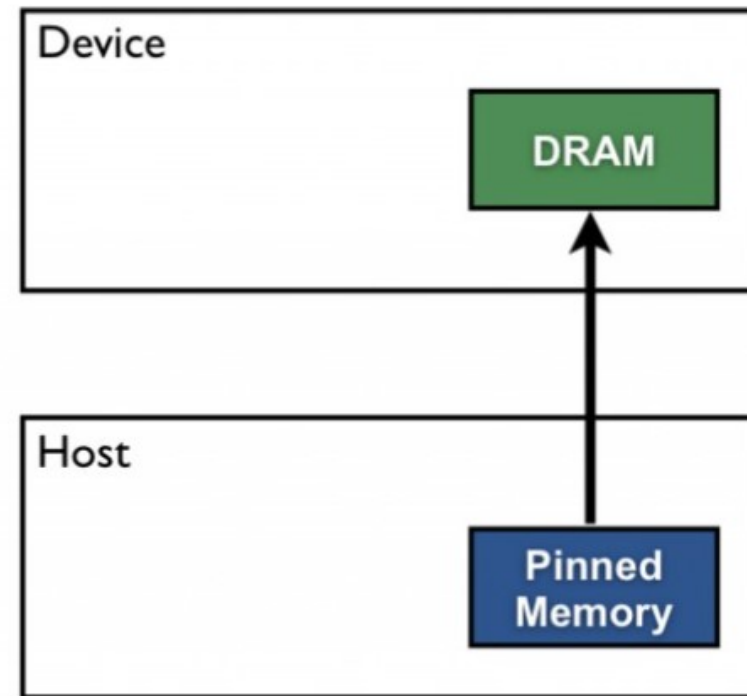
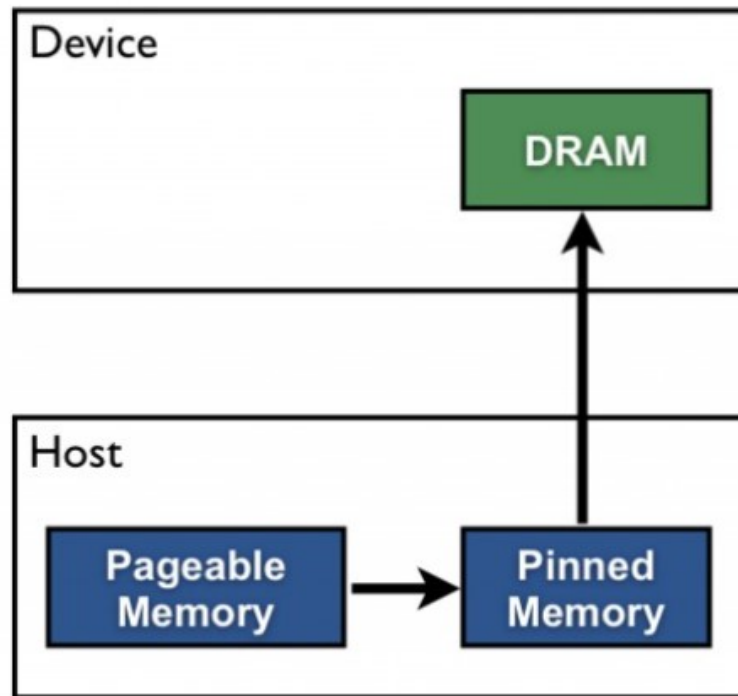
- Mover datos consume tiempo:
host memory \leftrightarrow *device memory*
- El cuello de botella es el ancho de banda del enlace PCIe



3.1. Optimización de código

Transferencias host-device

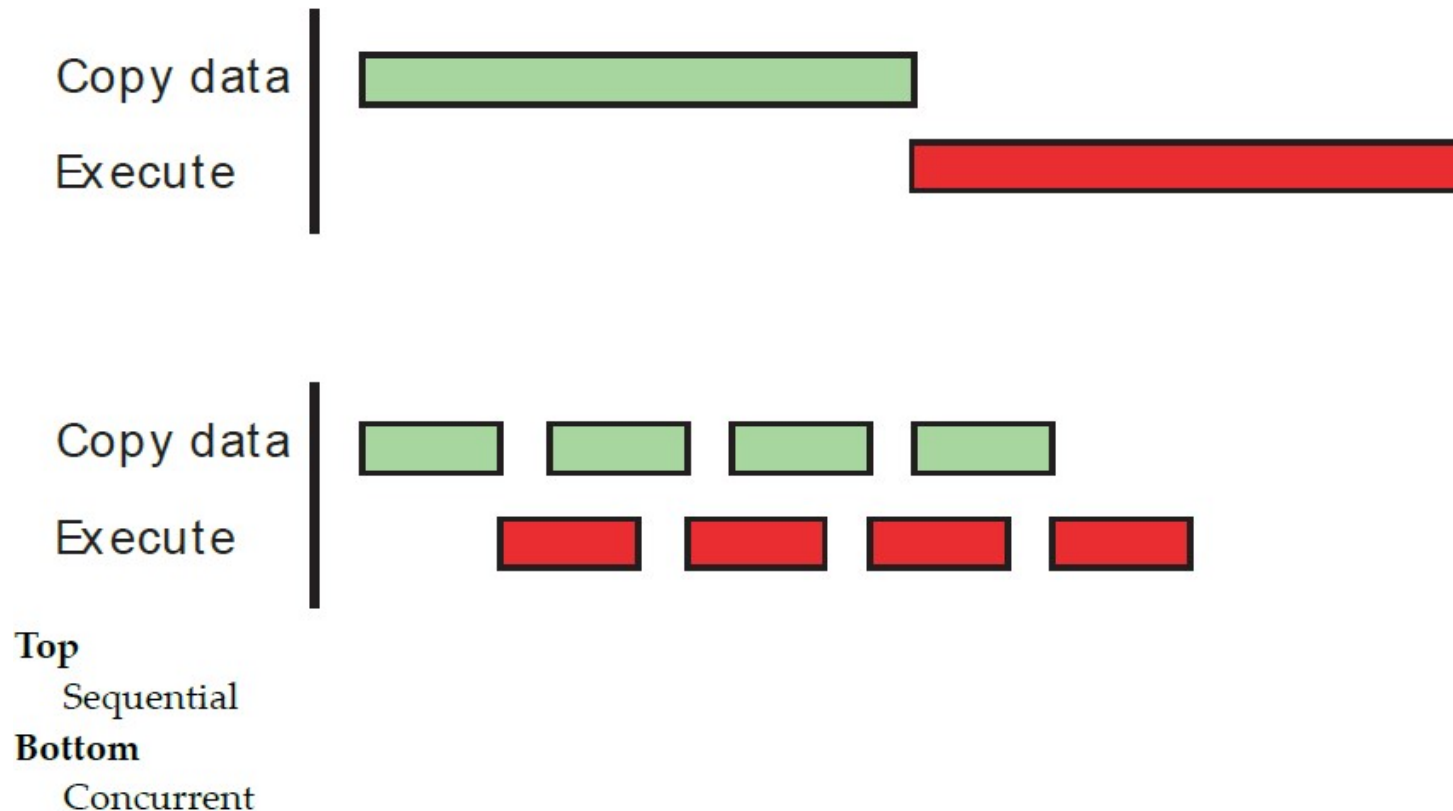
- Mover datos consume tiempo:
host memory \leftrightarrow *device memory*



3.1. Optimización de código

Transferencias host-device

- Se puede reducir el impacto de movimientos de datos...
...solapándolos con otras tareas
(véase Concurrent copy and execution en la salida de `cuda_deviceQuery`)



3.1. Optimización de código

Transferencias host-device

- Se puede reducir el impacto de movimientos de datos...
...solapándolos con otras tareas:
- Opciones:
 1. Programación **explícita** del solapamiento:
 - **SAP:** *streams* + comunicaciones **asíncronas** + memoria **pinned**
 2. Solapamiento programado **semi-implícitamente**:
 - función `cudaHostAlloc()`
 3. Solapamiento **implícito**:
 - Unified Memory Access (**UMA**)

3.1. Optimización de código

Transferencias host-device

Opción 1: streams (+memoria *pinned* + comunicaciones asíncronas)

- *stream*: es una cola de trabajo en la GPU:
 - La CPU pone trabajo en la cola y sigue haciendo otras cosas
 - La GPU va cogiendo trabajo de la cola cuando tiene recursos
- Las operaciones CUDA (Lanzamiento de *kernels*, copias de memoria,...) son colocadas en un mismo *stream*
- Operaciones en el **mismo *stream*** se ejecutan en GPU **en orden (FIFO)** y no se pueden solapar
- Operaciones CUDA en **diferentes *streams*** pueden ejecutarse en GPU **concurrentemente**
- *Stream 0* (por defecto) tiene unas reglas de sincronización especiales:
 - Se ejecuta síncronamente con todos los *streams*
 - Operaciones en *stream 0* no se pueden solapar con otros *streams*

3.1. Optimización de código

Transferencias host-device

Opción 1: streams + comunicaciones asíncronas + memoria *pinned*

- Se puede reducir el impacto de movimientos de datos... solapándolos con otras tareas:
- Usaremos:
 - *Streams* que funcionen simultáneamente
 - `cudaStreamCreate()`
 - Copia de datos asíncrona
 - `cudaMemcpyAsync()`
 - Memoria no paginable (*pinned*):
 - `cudaMallocHost()`

3.1. Optimización de código

Transferencias host-device

Opción 1: streams + comunicaciones asíncronas + memoria *pinned*

GPU TECHNOLOGY CONFERENCE

KERNEL CONCURRENCY

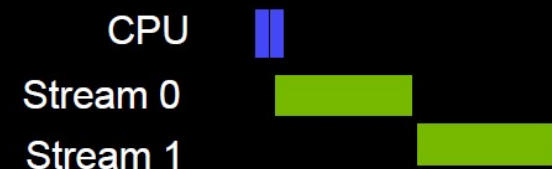
- Assume foo only utilizes 50% of the GPU
- Default stream

```
foo<<<blocks, threads>>>();
foo<<<blocks, threads>>>();
```



- Default & user streams


```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
foo<<<blocks, threads>>>();
foo<<<blocks, threads, 0, stream1>>>();
cudaStreamDestroy(stream1);
```



3.1. Optimización de código

Transferencias host-device

Opción 1: streams + comunicaciones asíncronas + memoria *pinned*



KERNEL CONCURRENCY

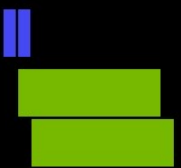
- Assume foo only utilizes 50% of the GPU
User streams

```
cudaStream_t stream1, stream2;  
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
foo<<<blocks, threads, 0, stream1>>>();  
foo<<<blocks, threads, 0, stream2>>>();  
cudaStreamDestroy(stream1);  
cudaStreamDestroy(stream2);
```

CPU

Stream 1

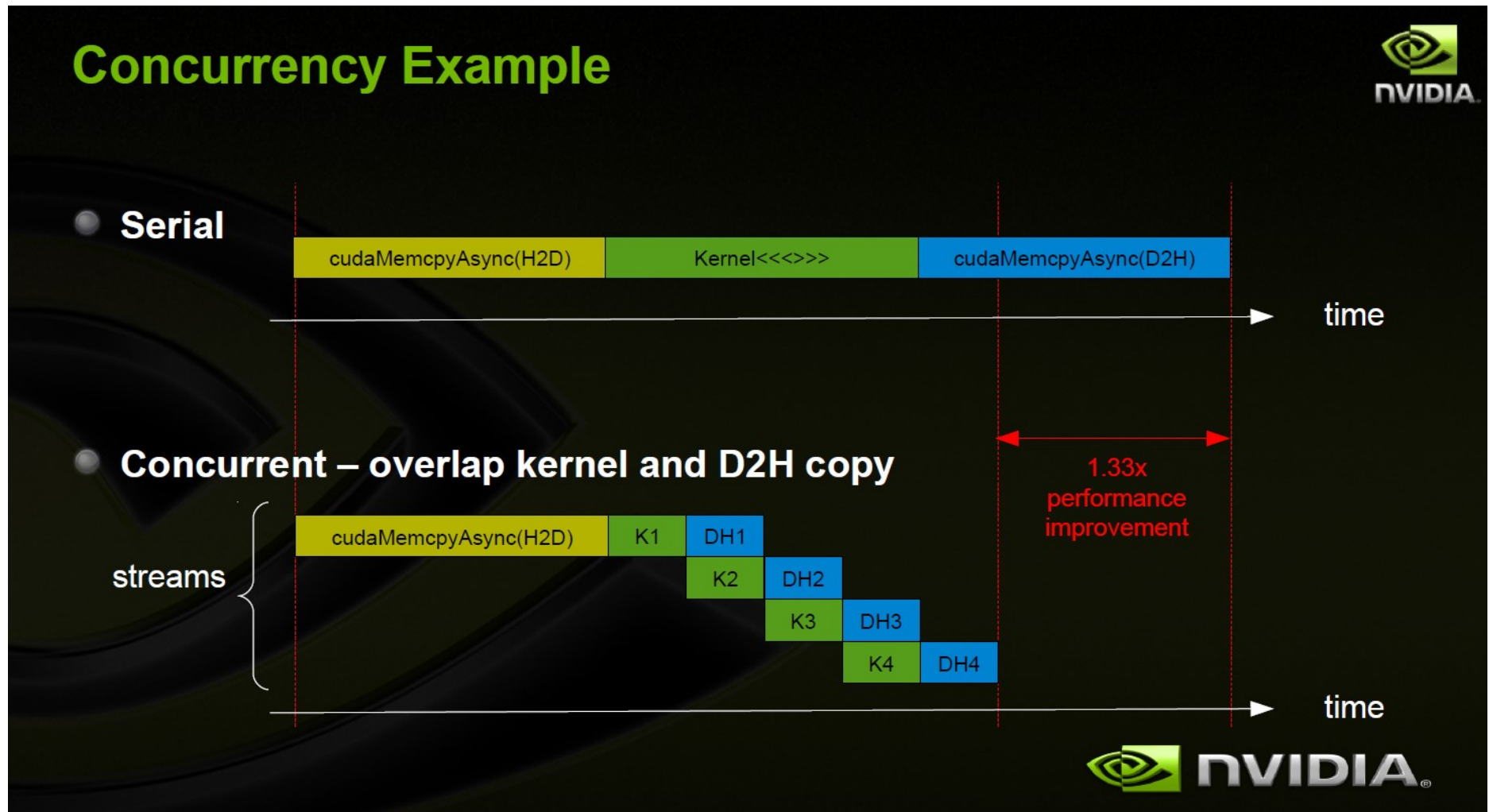
Stream 2



3.1. Optimización de código

Transferencias host-device

Opción 1: streams + comunicaciones asíncronas + memoria *pinned*



3.1. Optimización de código

Transferencias host-device

Opción 1: streams + comunicaciones asíncronas + memoria *pinned*

Amount of Concurrency



Serial (1x)

cudaMemcpyAsync(H2D) Kernel <<<>>> cudaMemcpyAsync(D2H)

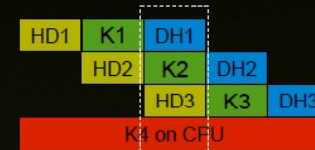
2-way concurrency (up to 2x)



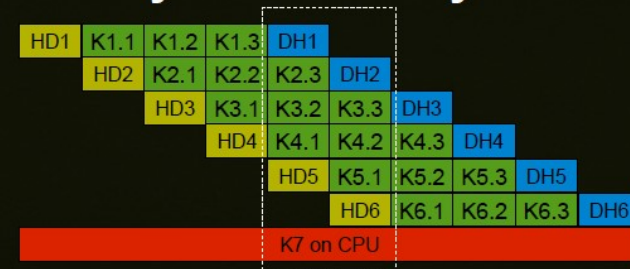
3-way concurrency (up to 3x)



4-way concurrency (3x+)



4+ way concurrency



3.1. Optimización de código

Transferencias host-device

Opción 1: streams + comunicaciones asíncronas + memoria *pinned*

Esquema ejemplo de código (1/3):

```
cudaEvent_t start, stop;
cudaEventCreate(&start); cudaEventCreate(&stop);
cudaStream_t stream[NS];

for (int i = 0; i < NS; ++i)
    cudaStreamCreate(&stream[i]);

float* input_h, *input_d, *output_h, *output_d;

cudaMallocHost((void**)&input_h, NS * size);
cudaMalloc((void**)&input_d, NS * size);

cudaMallocHost((void**)&output_h, NS * size);
cudaMalloc((void**)&output_d, NS * size);

. . .
```

3.1. Optimización de código

Transferencias host-device

Opción 1: streams + comunicaciones asíncronas + memoria *pinned*

Esquema ejemplo de código (2/3):

```
. . .  
  
cudaEventRecord(start, 0);  
for (int i = 0; i < NS; ++i)  
    cudaMemcpyAsync(input_d + i * size, input_h + i * size,  
                    size, cudaMemcpyHostToDevice, stream[i]);  
  
for (int i = 0; i < NS; ++i)  
    myKernel<<<grid, block, 0, stream[i]>>>  
        (input_d + i * size, output_d + i * size, size);  
  
for (int i = 0; i < NS; ++i)  
    cudaMemcpyAsync(output_h + i*size, output_d + i*size,  
                    size, cudaMemcpyDeviceToHost, stream[i]);  
  
cudaEventRecord(stop, 0);  
cudaEventSynchronize(stop);  
cudaEventElapsedTime(&elapsedTime, start, stop);
```


3.1. Optimización de código

Transferencias host-device

Opción 1: streams + comunicaciones asíncronas + memoria *pinned*

Esquema ejemplo de código (3/3):

```
. . .  
cudaFreeHost((void *) input_h);  
cudaFree((void *) input_d);  
cudaFreeHost((void *) output_h);  
cudaFree((void *) output_d);  
  
for (int i = 0; i < NS; ++i)  
    cudaStreamDestroy(stream[i]);  
  
cudaEventDestroy(start);  
cudaEventDestroy(stop);  
. . .
```

3.1. Optimización de código

Transferencias host-device

Opción 2: `cudaHostAlloc (void ** pHost, size_t size, unsigned int flags)`

Opciones indicadas con el campo *flags*:

- *cudaHostAllocDefault*: Se reserva memoria no paginable (*pinned memory*). emulando a `cudaMallocHost()`
- *cudaHostAllocPortable*: Se reserva memoria no paginable para cualquier contexto de CUDA (para cualquier *device*) y no únicamente para el que la invoca
- *cudaHostAllocWriteCombined*: Se reserva memoria como write-combined (**WCM**).
 - **WCM** puede ser transferida más rápida por el PCIe CPU→GPU en algunas configuraciones porque es memoria no cacheable en CPU.
 - Pero, ATENCIÓN: GPU → CPU muy lenta
- *cudaHostAllocMapped*: Se mapea la memoria reservada dentro del espacio de direcciones de CUDA (desde código de *kernel* se accede a memoria host directamente como si fuese memoria del *device*).

→ A continuación más detalles...

NOTA: Definición de flags ortogonal: memoria portable, *mapped* y/o *write-combined*

3.1. Optimización de código

Transferencias host-device

Opción 2: `cudaHostAlloc (void ** pHost, size_t size, unsigned int flags)`

Flag `cudaHostAllocMapped`:

- *Threads* de GPU acceden directamente a memoria del *Host*
- Requiere ***mapped pinned*** (no paginable) memoria
- En GPU integradas siempre ganancia --> evita copias innecesarias
- En GPU independientes:
 - Los datos no son "cacheados" en GPU → ventaja si datos son leídos/escritos solamente una vez
- Automáticamente se solapa transferencia de datos con ejecución de *kernels* (Sin tener que decidir ningún parámetro (como número de *streams* u otro))

3.1. Optimización de código

Transferencias host-device

Opción 2: `cudaHostAlloc (void ** pHost, size_t size, unsigned int flags)`
Flag cudaHostAllocMapped:

Ejemplo código:

```
float *a_h, *a_map;  
...  
  
cudaGetDeviceProperties(&prop, 0);  
if (!prop.canMapHostMemory)  
    exit(0);  
  
cudaSetDeviceFlags(cudaDeviceMapHost);  
  
cudaHostAlloc(&a_h, nBytes, cudaHostAllocMapped);  
  
cudaHostGetDevicePointer(&a_map, a_h, 0);  
  
kernel<<<gridSize, blockSize>>>(a_map);
```

3.1. Optimización de código

Transferencias host-device

Opción 3: Unified Memory Access (UMA)

- Necesario al menos: **GPU cc 3.0** y **SDK-CUDA 6.0**
- Todos los procesadores ven un único espacio de memoria coherente (*managed memory*), compartiendo el espacio de direcciones
- Migración automática de datos a la GPU cuando son accedidos
- GPU cc < 6.0: Cuando se lanza un kernel generalmente se le transfiere a la memoria de la GPU toda la *managed memory* para evitar fallos durante los accesos. No se permiten accesos simultáneos CPU+GPU a un dato.
- GPU cc >= 6.0 : memoria virtual y mecanismo de fallo de página en GPU:
 - Las páginas se migran entre las memorias de las GPUs y CPUs bajo demanda, tras fallo de página.

3.1. Optimización de código

Transferencias host-device

Opción 3: Unified Memory Access (UMA)

- Ejemplo previo, aun **sin usar UMA**:

```
__global__ void AplusB( int *ret, int a, int b) {  
    ret[threadIdx.x] = a + b + threadIdx.x;  
}  
  
int main() {  
    int *ret;  
    cudaMalloc(&ret, 1000 * sizeof(int));  
    AplusB<<< 1, 1000 >>>(ret, 10, 100);  
    int *host_ret = (int *)malloc(1000 * sizeof(int));  
    cudaMemcpy(host_ret, ret, 1000 * sizeof(int), cudaMemcpyDefault);  
    for(int i=0; i<1000; i++)  
        printf("%d: A+B = %d\n", i, host_ret[i]);  
    free(host_ret);  
    cudaFree(ret);  
    return 0;  
}
```

3.1. Optimización de código

Transferencias host-device

Opción 3: Unified Memory Access (UMA)

- Ejemplo **con UMA**. Versión usando `cudaMallocManaged()`:

```
__global__ void AplusB(int *ret, int a, int b) {  
    ret[threadIdx.x] = a + b + threadIdx.x;  
}  
  
int main() {  
    int *ret;  
    cudaMallocManaged(&ret, 1000 * sizeof(int));  
    AplusB<<< 1, 1000 >>>(ret, 10, 100);  
    cudaDeviceSynchronize();  
    for(int i=0; i<1000; i++)  
        printf("%d: A+B = %d\n", i, ret[i]);  
    cudaFree(ret);  
    return 0;  
}
```

3.1. Optimización de código

Transferencias host-device

Opción 3: Unified Memory Access (UMA)

- Ejemplo **con UMA**. Versión usando `__managed__`:

```
__device__ __managed__ int ret[1000];  
__global__ void AplusB(int a, int b) {  
    ret[threadIdx.x] = a + b + threadIdx.x;  
}  
int main() {  
    AplusB<<< 1, 1000 >>>(10, 100);  
    cudaDeviceSynchronize();  
    for(int i=0; i<1000; i++)  
        printf("%d: A+B = %d\n", i, ret[i]);  
    return 0;  
}
```


3.1. Optimización de código

Transferencias host-device

Pruebas comparativas **MARTE**:

```
$ vectorAdd_sobrecarga -n=10000000 -b=200
```

Sin streams

TIEMPO DE EJECUCION: 45.019073 milisegundos

Con streams + comunicaciones asíncronas + pinned memory

Numero streams	TIEMPO
1	22.41 ms
2	22.02 ms
10	21.72 ms
20	21.84 ms

UMA

Memory fault (core dumped)

---→ en marte no funciona UMA porque necesitamos cc>=3.0

3.1. Optimización de código

Transferencias host-device

Pruebas comparativas **VENUS (c.c. 6.1)**:

```
$ vectorAdd_sobrecarga -n=100000000 -b=1000
```

```
-----
Sin streams                                tiempo
                                           210 milisegundos
```

```
-----
Con streams + comunicaciones asíncronas + pinned memory
```

Número de streams	tiempo
2	88 milisegundos
4	80 milisegundos
8	76 milisegundos
16	74 milisegundos
32	75 milisegundos
64	76 milisegundos
128	77 milisegundos

```
-----
Con UMA                                tiempo
                                           1775 milisegundos
```

Contenidos

1. Introducción
2. Arquitectura y programación de CUDA
- 3. Optimización y depuración de código**
 1. **Optimización de código**
 1. Medición de tiempos
 2. Transferencias host ↔ device
 - 3. Configuración ejecución**
 4. Memoria global
 5. Memoria compartida
 2. Depuración y Medición de uso de recursos
4. Librerías basadas en CUDA
5. Alternativas a NVIDIA/CUDA
6. Conclusiones
7. Bibliografía
8. ANEXO: programación híbrida CPU+GPU

3.1. Optimización de código

Configuración de la ejecución

- **Control de flujo (saltos y divergencia)**

- Las instrucciones de control de flujo que implican saltos como `if`, `switch`, `do`, `for` o `while` pueden hacer que los *threads* de un *warp* diverjan
 - Se puede evitar siempre que sea posible expresar la condición a partir del tamaño del *warp* y no sólo del TID
- Todas las funciones `__device__` son ***inline***
 - El programador puede evitarlo con `__noinline__`
- El compilador **desenrolla** los bucles pequeños, por defecto.
 - El programador puede hacerlo con `#pragma unroll N`
- También utiliza **predicción** cuando el número de instrucciones de cada rama es menor que o igual a 7

Contenidos

1. Introducción
2. Arquitectura y programación de CUDA
- 3. Optimización y depuración de código**
 1. **Optimización de código**
 1. Medición de tiempos
 2. Transferencias host ↔ device
 3. Configuración ejecución
 4. **Memoria global**
 5. Memoria compartida
 2. Depuración y Medición de uso de recursos
4. Librerías basadas en CUDA
5. Alternativas a NVIDIA/CUDA
6. Conclusiones
7. Bibliografía
8. ANEXO: programación híbrida CPU+GPU

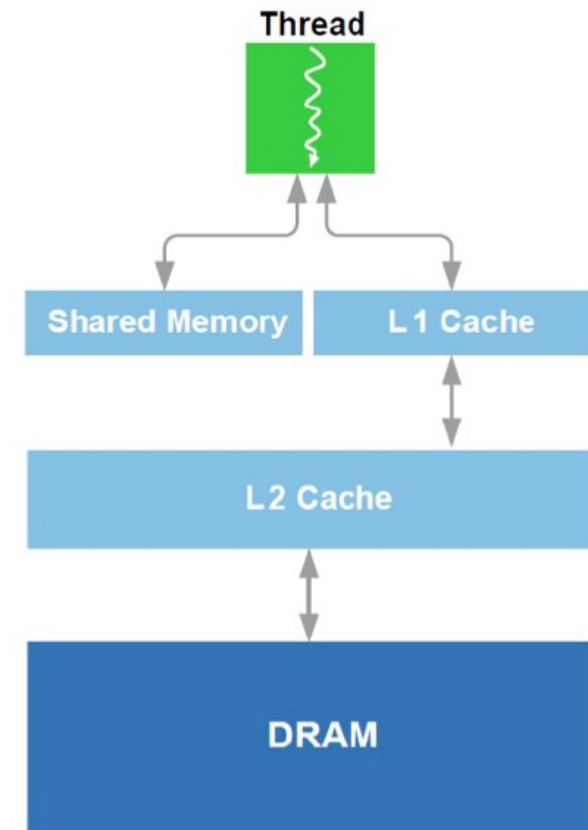
3.1. Optimización de código

Accesos a Memoria Global

Accesos *Coalesced* en GPU

(*Compute Capability 2.x*)

- Los accesos concurrentes de los 32 *threads* de un *warp* se fusionarán en un número de transacciones igual al número de líneas (bloques) de caché necesarias para servir los datos demandados
- Por defecto todos los accesos pasan por caché L1, que tiene líneas de 128 Bytes.
 - Por ejemplo: 128 Bytes --> 32 *threads* accediendo a 32 palabras consecutivas de 4 Bytes cada una
- Se puede tener también en cuenta el paso por L2, que tiene líneas de 32 Bytes.

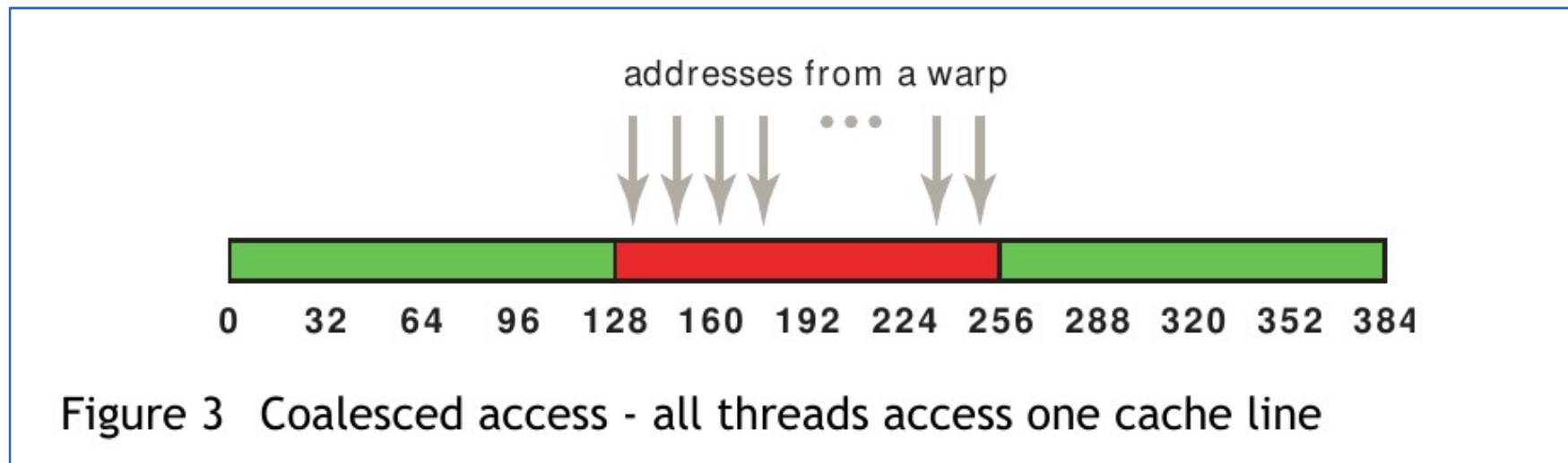


3.1. Optimización de código

Accesos a Memoria Global

Patrón de acceso básico:

- El k -ésimo *thread* de un *warp* accede a la k -ésima palabra de 4 Bytes de una línea de caché (128 Bytes)



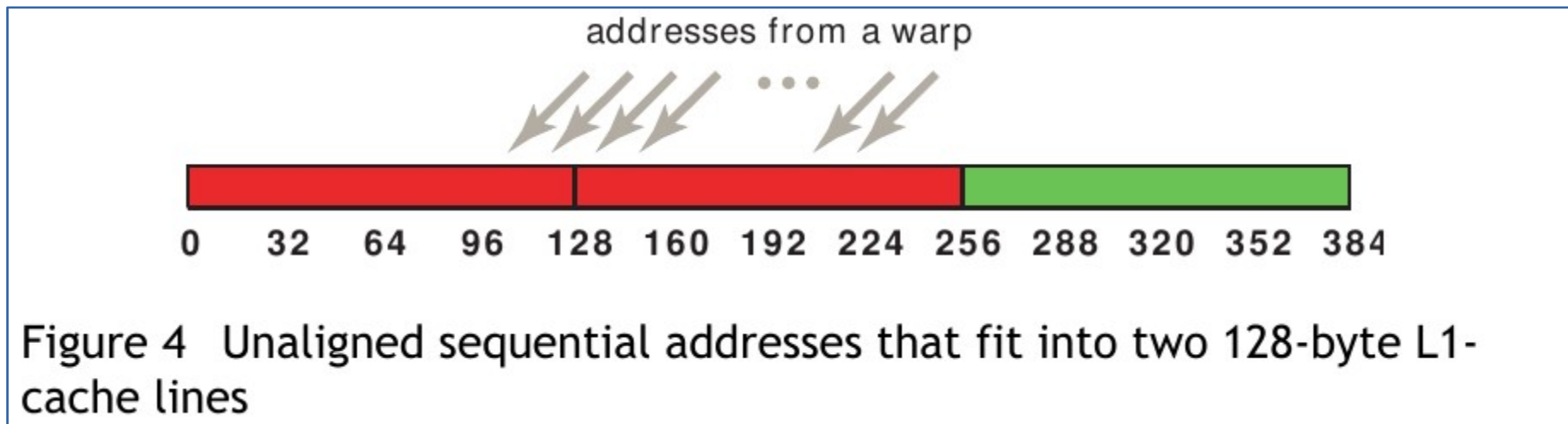
- No es necesario que intervengan todos los *threads* del *warp*
- Varios *threads* pueden acceder a la misma palabra, dejando algunas sin utilizar
- Los accesos a las palabras pueden estar permutados entre los *threads* mientras no se salgan de la línea de caché

3.1. Optimización de código

Accesos a Memoria Global

Patrón de acceso secuencial pero no alineado:

- El k-esimo thread de un *warp* accede a la k-esima palabra de 4 Bytes



- Al no estar alineado a 128 bytes se necesitan leer dos líneas de 128 Bytes de caché L1
- La memoria reservada con CUDA Runtime API (por ejemplo con `cudaMalloc()`) están alineados a 256 Bytes.

3.1. Optimización de código

Accesos a Memoria Global

Patrón de acceso secuencial pero no alineado:

- El k-esimo thread de un *warp* accede a la k-esima palabra de 4 Bytes

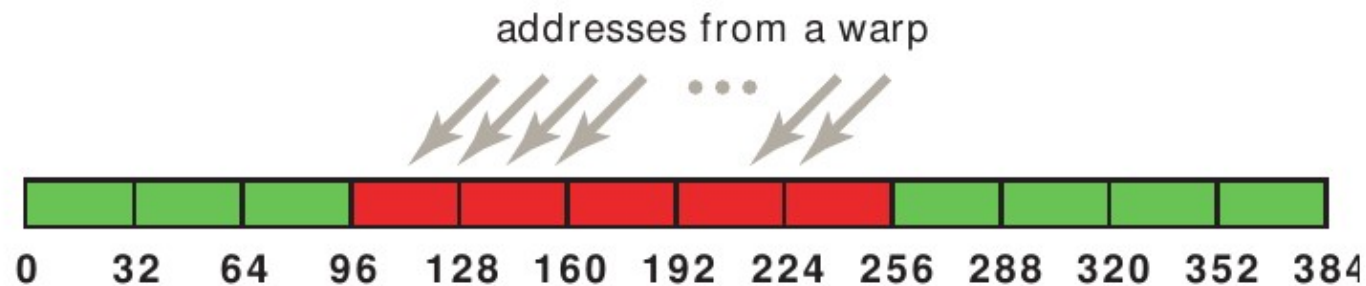


Figure 5 Misaligned sequential addresses that fall within five 32-byte L2-cache segments

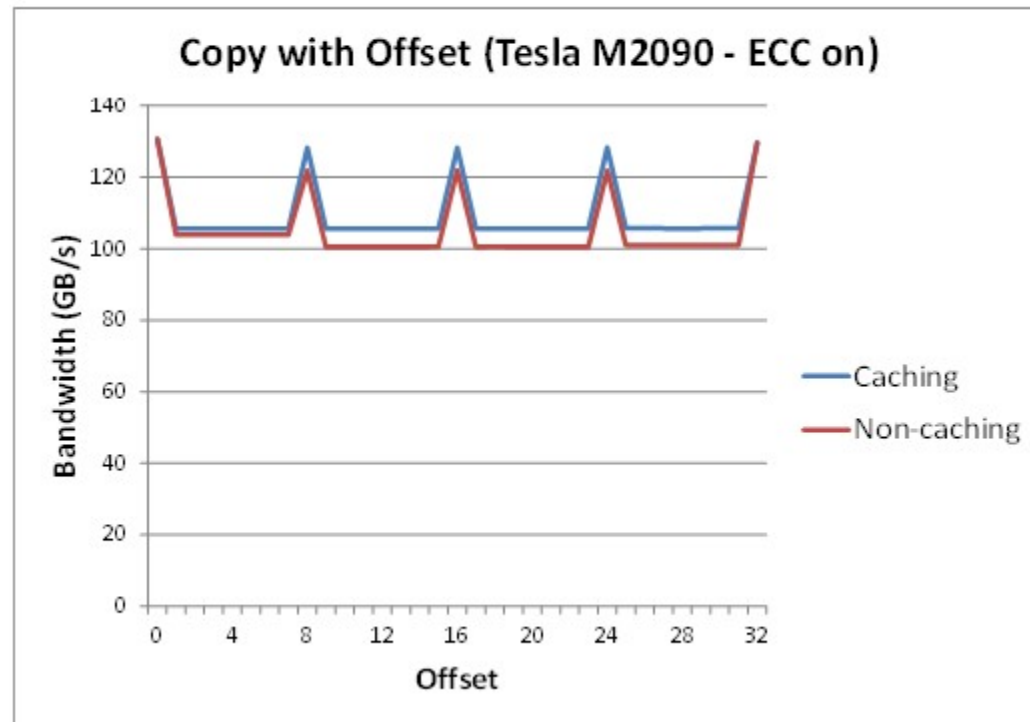
- Si caché L1 deshabilitada → se necesitan 5 líneas de caché L2

3.1. Optimización de código

Accesos a Memoria Global

Patrón de acceso secuencial pero no alineado:

```
__global__ void offsetCopy(float *odata, float* idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```

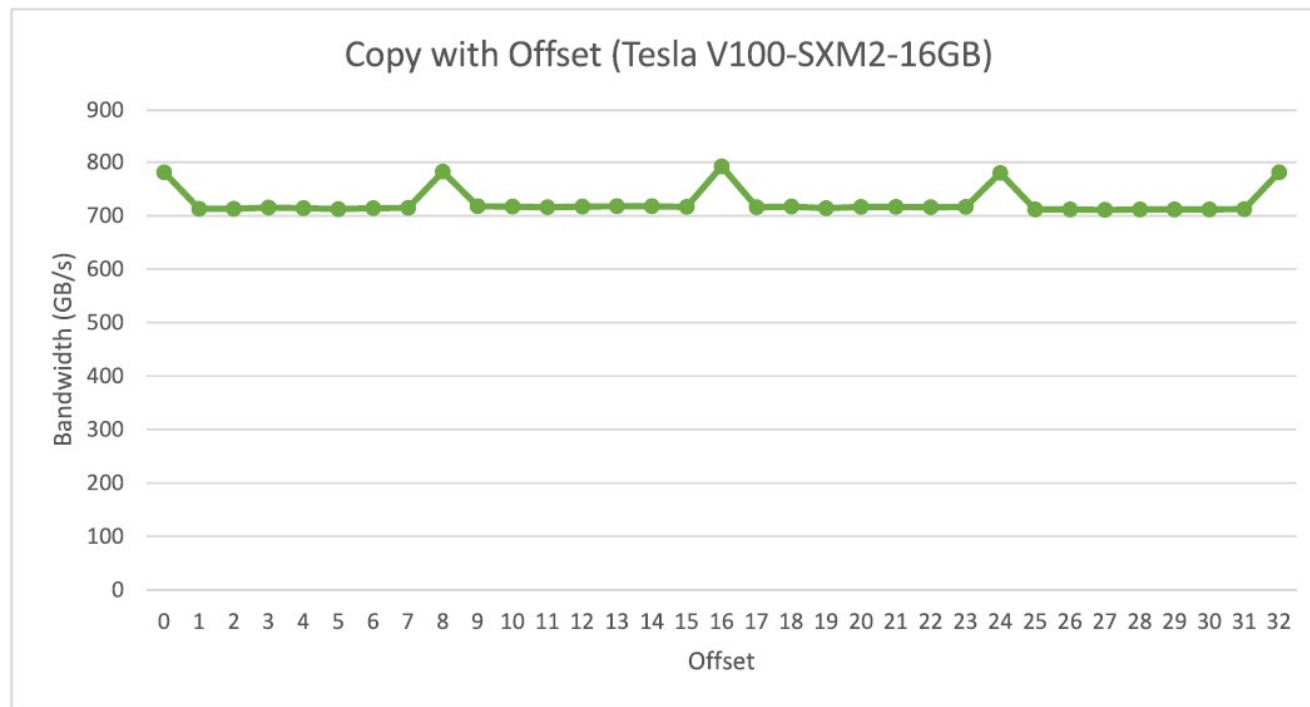


3.1. Optimización de código

Accesos a Memoria Global

Patrón de acceso secuencial pero no alineado:

```
__global__ void offsetCopy(float *odata, float* idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```



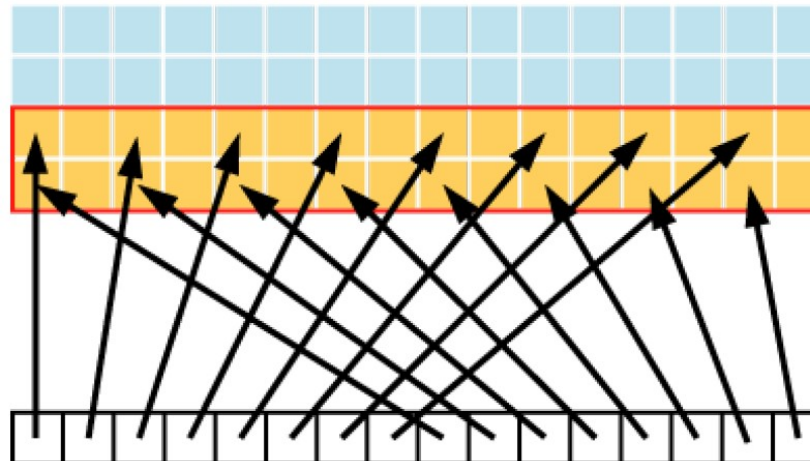
3.1. Optimización de código

Accesos a Memoria Global

Patrón de acceso con intervalo (*stride*) entre elementos

- Por ejemplo, *stride* = 2:

```
__global__ void strideCopy(float *odata, float* idata, int stride)
{
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

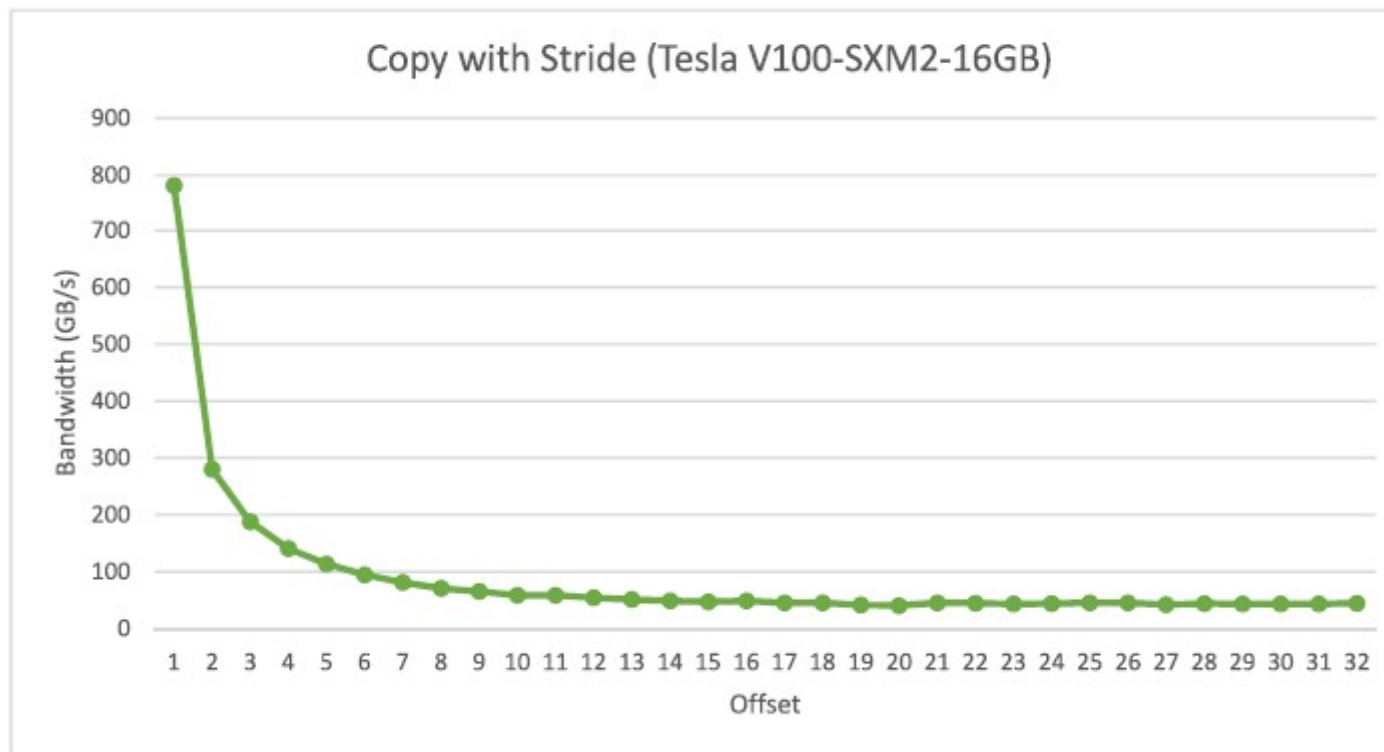


3.1. Optimización de código

Accesos a Memoria Global

Patrón de acceso con intervalo (*stride*) entre elementos

- Eficiencia acceso memoria global disminuye al aumentar el *stride*.



Contenidos

1. Introducción
2. Arquitectura y programación de CUDA
- 3. Optimización y depuración de código**
 1. **Optimización de código**
 1. Medición de tiempos
 2. Transferencias host ↔ device
 3. Configuración ejecución
 4. Memoria global
 5. **Memoria compartida**
 2. Depuración y Medición de uso de recursos
4. Librerías basadas en CUDA
5. Alternativas a NVIDIA/CUDA
6. Conclusiones
7. Bibliografía
8. ANEXO: programación híbrida CPU+GPU

3.1. Optimización de código

Accesos a Memoria Compartida

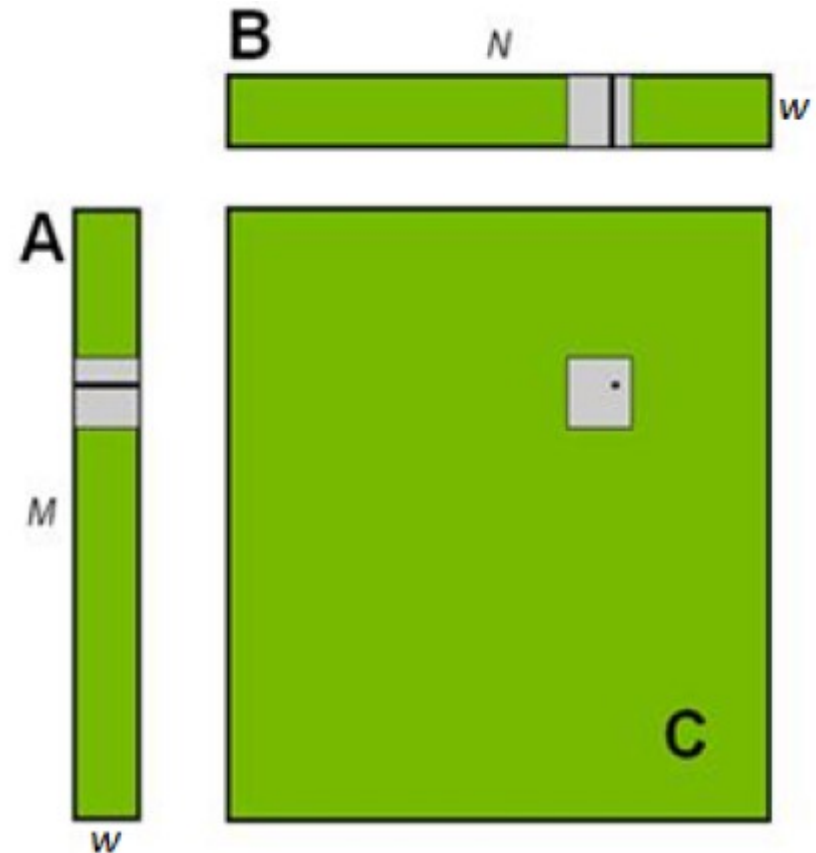
- **La memoria compartida** tiene un tiempo de acceso similar a los registros si no hay conflictos entre los *threads* de un *warp*
- **Accesos sin conflictos a memoria compartida (a partir de *Compute capability* 2.x)**
 - La memoria compartida está compuesta de 32 bancos
 - Elementos de 4 bytes consecutivos se almacenan en los bancos de memoria compartida de manera cíclica
 - Todos los accesos de un *warp* pueden atenderse simultáneamente si todas las direcciones pertenecen a distintos bancos, pudiendo repetirse éstas (*broadcast*)
- Se puede usar la memoria compartida tanto para conseguir accesos *coalesced* a memoria global como para eliminar accesos redundantes a memoria global

3.1. Optimización de código

Accesos a Memoria Compartida

Multiplicación matrices $C = AB$

- Dimensiones:
 - A es $M \times w$
 - B es $w \times N$
 - C es $M \times N$
- Descomposición del problema:
 - Bloque de *threads*: $w \times w$
 - Bloque de datos (*tile*) de datos: $w \times w$
- Por simplificar el *kernel* ejemplo:
 - M y N son múltiplos de w
 - $w = 32$
- *Grid*: $N/w \times M/w$ bloques de *threads*
- Un bloque de *threads* calcula los elementos de un *tile* de C a partir de un *tile* de A y un *tile* de B



3.1. Optimización de código

Accesos a Memoria Compartida

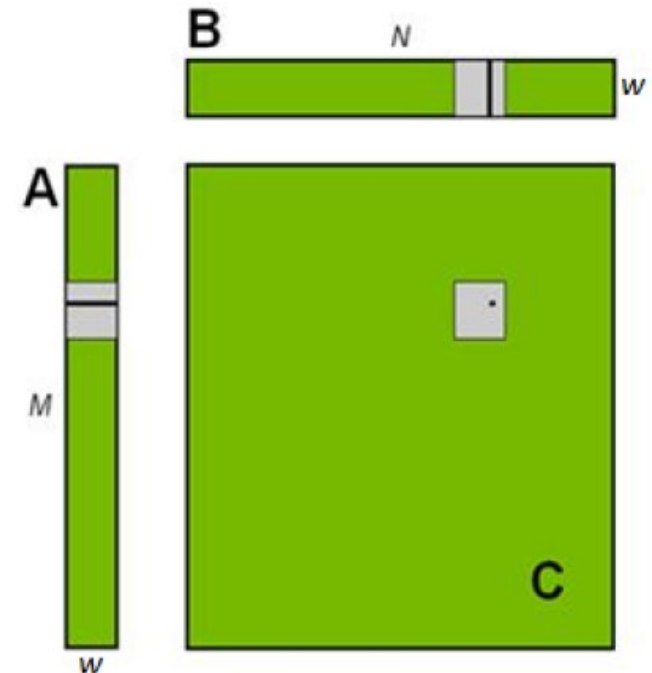
Multiplicación matrices $C = AB$

- **Versión no óptima del kernel: usando memoria global**
 - Cada *thread* calcula el elemento (*row,col*) de C recorriendo la fila *row* de A y la columna *col* de B

```
__global__ void simpleMultiply(float *a, float* b, float *c,int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;

    for (int i = 0; i < TILE_DIM; i++)
    {
        sum += a[row*TILE_DIM+i] * b[i*N+col];
    }

    c[row*N+col] = sum;
}
```



3.1. Optimización de código

Accesos a Memoria Compartida

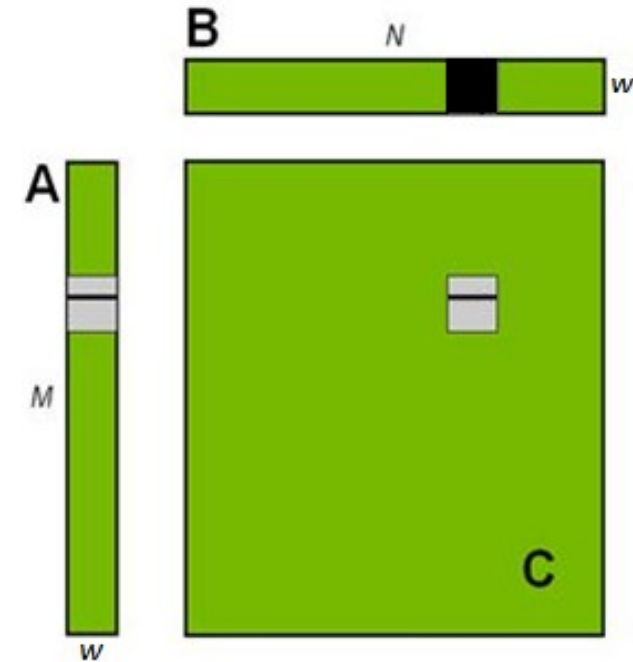
Multiplicación matrices $C = AB$

- **Versión no óptima del kernel: usando memoria global**
 - Cada *warp* calcula una fila de un *tile* de C , a partir de una fila de A y un *tile* de B

```
__global__ void simpleMultiply(float *a, float* b, float *c,int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;

    for (int i = 0; i < TILE_DIM; i++)
    {
        sum += a[row*TILE_DIM+i] * b[i*N+col];
    }

    c[row*N+col] = sum;
}
```

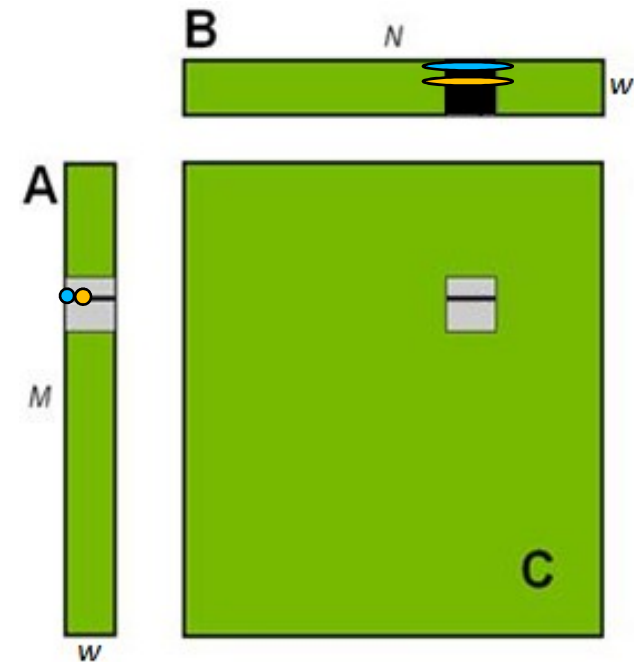


3.1. Optimización de código

Accesos a Memoria Compartida

Multiplicación matrices $C = AB$

- **Versión no óptima del kernel: usando memoria global**
 - Cada *warp* calcula un fila de un *tile* de C , a partir de una fila de A y un *tile* de B
- **Iteración i** del bucle ($i=0 \rightarrow$ azul, $i=1 \rightarrow$ amarillo,...), los *threads* de un *warp*:
 - Leen una fila de $B \rightarrow$ acceso secuencial y coalesced \rightarrow OK
 - Leen un valor concreto de la matriz $A(\text{row}, i)$
 - Una única transacción desde memoria global
 - Supone un desaprovechamiento del ancho de banda (solo se aprovecha 1 palabra de las 32 de la línea de caché)
 - En las siguientes iteraciones se usarían las otras palabras de la línea, pero con tanto *warp* en ejecución simultánea en el mismo SM, puede que la línea sea devuelta a memoria entre las iteraciones i e $i+1$



3.1. Optimización de código

Accesos a Memoria Compartida

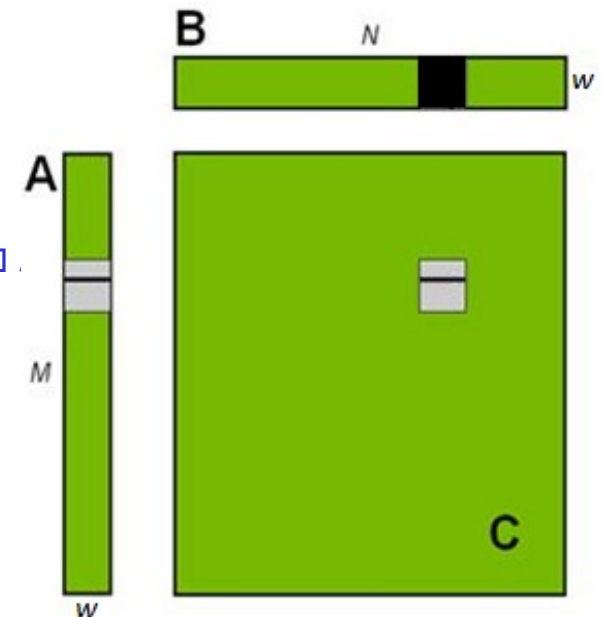
Multiplicación matrices $C = AB$

- **Usando memoria compartida (1/2)**
- Cada elemento del *tile* de A se lee de memoria global a memoria compartida solamente una vez, en forma completamente coalesced, sin desaprovechar ancho de banda
- En cada iteración: un valor de memoria compartida se distribuye a todos los *threads* del *warp*

```
__global__ void coalescedMultiply(float *a, float* b, float *c, int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;

    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];

    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```



3.1. Optimización de código

Accesos a Memoria Compartida

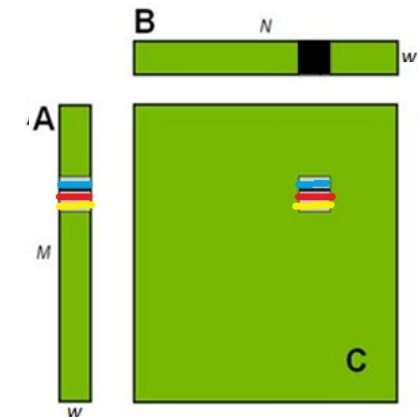
Multiplicación matrices $C = AB$

- **Usando memoria compartida (2/2)**

- Para calcular cada fila del *tile* de C se lee el *tile* entero de B
- Por tanto, para el *tile* entero de C (el trabajo que hace un bloque de *threads*) se lee el *tile* entero de B repetidamente (*w* veces)
- Eliminar esta repetición de la lectura del *tile* de B por un bloque de threads
 - → que ese bloque se encargue de traerlo a memoria compartida

```
__global__ void sharedABMultiply(float *a, float* b, float *c, int N){
    __shared__ float aTile[TILE_DIM][TILE_DIM],
    bTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;

    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
    __syncthreads();          //warp usa datos de B leídos por otro warp del bloque
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i] * bTile[i][threadIdx.x];
    }
    c[row*N+col] = sum;
}
```



3.1. Optimización de código

Accesos a Memoria Compartida

En la máquina **venus** (GPU c.c. 6.1)

$$O(n) = 2 * n * n * w$$

n	Global Memory	Coalesced	sharedAB
512	0.219 ms	0.129 ms	0.078 ms
1024	0.546 ms	0.386 ms	0.251 ms
2048	1.904 ms	1.427 ms	0.815 ms
4096	7.317 ms	5.582 ms	3.097 ms
8192	31.176 ms	23.221 ms	12.451 ms
16384	128.212 ms	94.618 ms	41.993 ms
32768	471.209 ms	285.384 ms	124.677 ms