

Sistemas Distribuidos y Paralelos

Ingeniería en Computación



CUDA Streams

Universidad Nacional de La Plata



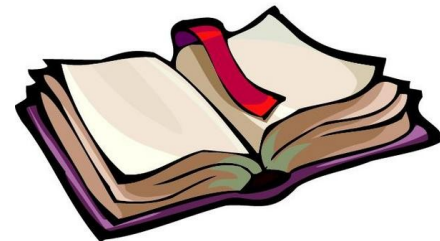
Facultad de Informática



Agenda

2

- I. Flujo de ejecución CUDA y ocultamiento de la latencia
- II. CUDA Streams
 - i. Concurrencia a nivel kernel
 - ii. Concurrencia a nivel de memoria (Pinned Host Memory)
- III. Sincronización y eventos



Agenda

3

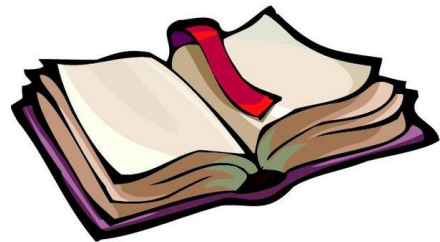
I. Flujo de ejecución CUDA y ocultamiento de la latencia

II. CUDA Streams

i. Concurrencia a nivel kernel

ii. Concurrencia a nivel de memoria (Pinned Host Memory)

III. Sincronización y eventos



Flujo de ejecución CUDA

4

- Flujo de procesamiento típico sobre GPU (Ejecución en serie)
 - 1) Copiar los datos de entrada de CPU a GPU (Transferencia Host to Device - H2D)
 - 2) Ejecutar el Kernel
 - 3) Copiar los resultado de GPU a CPU (Transferencia Device to Host - D2H)



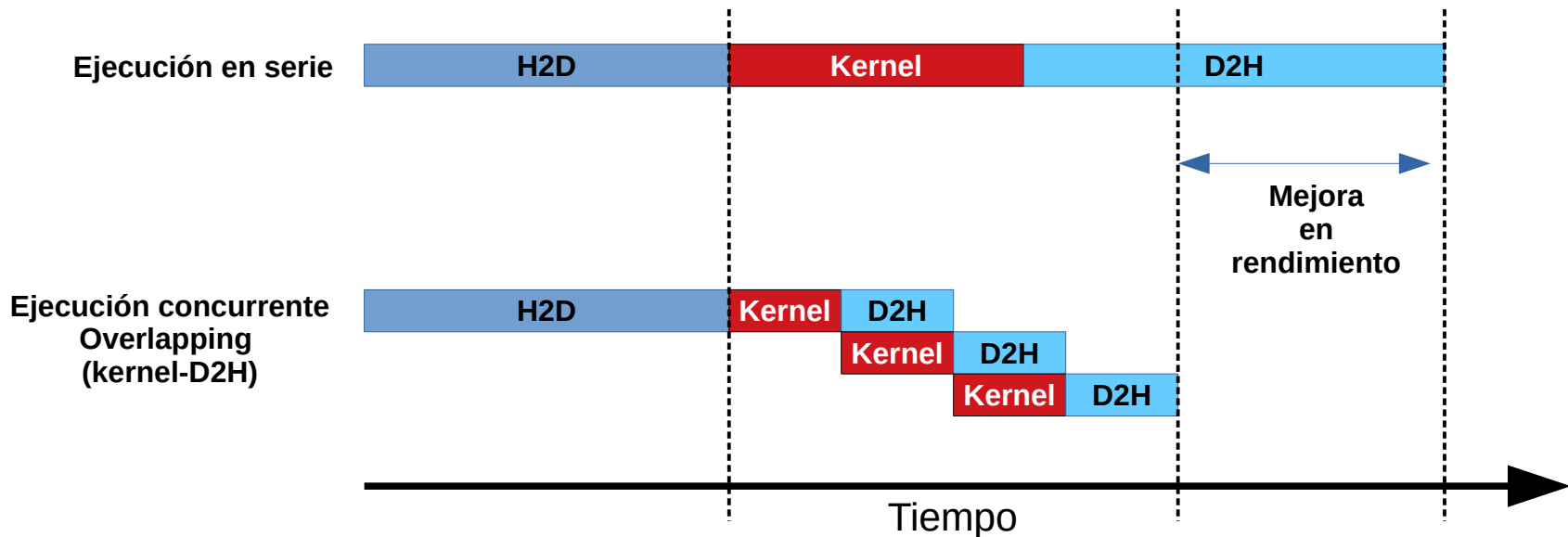
El costo de las transferencias H2D y D2H es alto
¿Es posible ocultar la latencia de estas transferencias?

Flujo de ejecución CUDA

Ocultamiento de la latencia

5

Una alternativa es reemplazar una invocación al kernel por varias invocaciones y superponer la ejecución de cada invocación con las transferencias D2H.

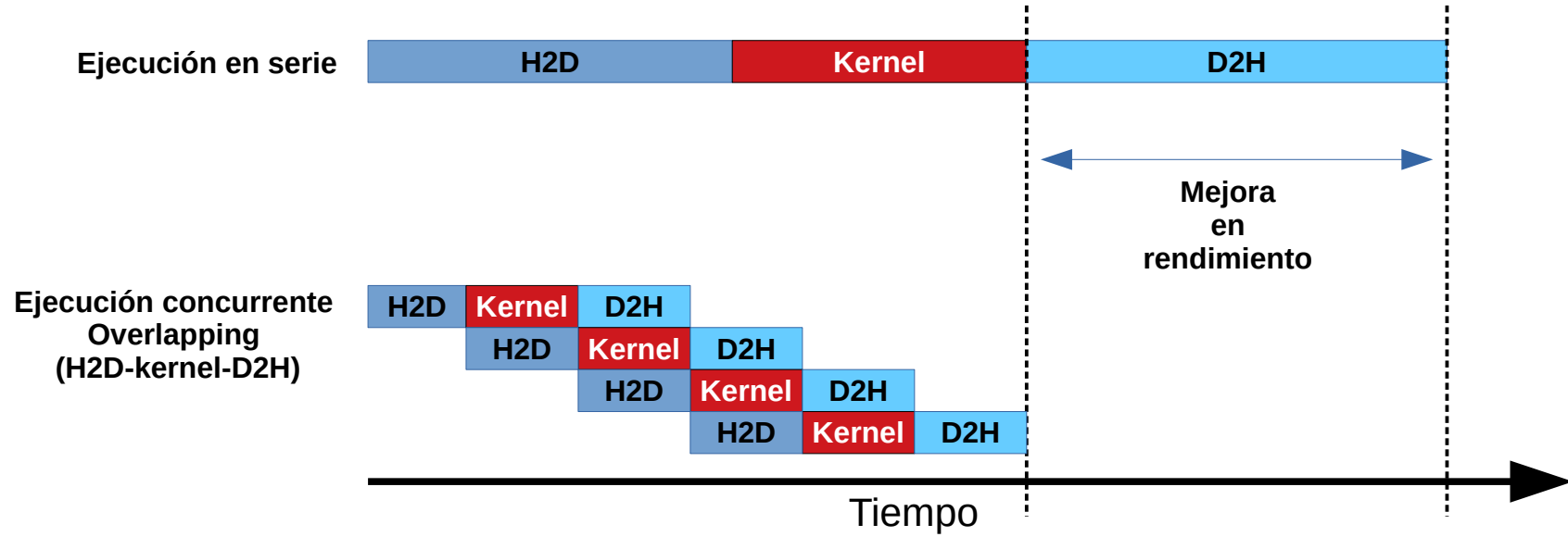


Flujo de ejecución CUDA

Ocultamiento de la latencia

6

Otra alternativa superpone la ejecución de cada invocación al kernel con las transferencias H2D y D2H.

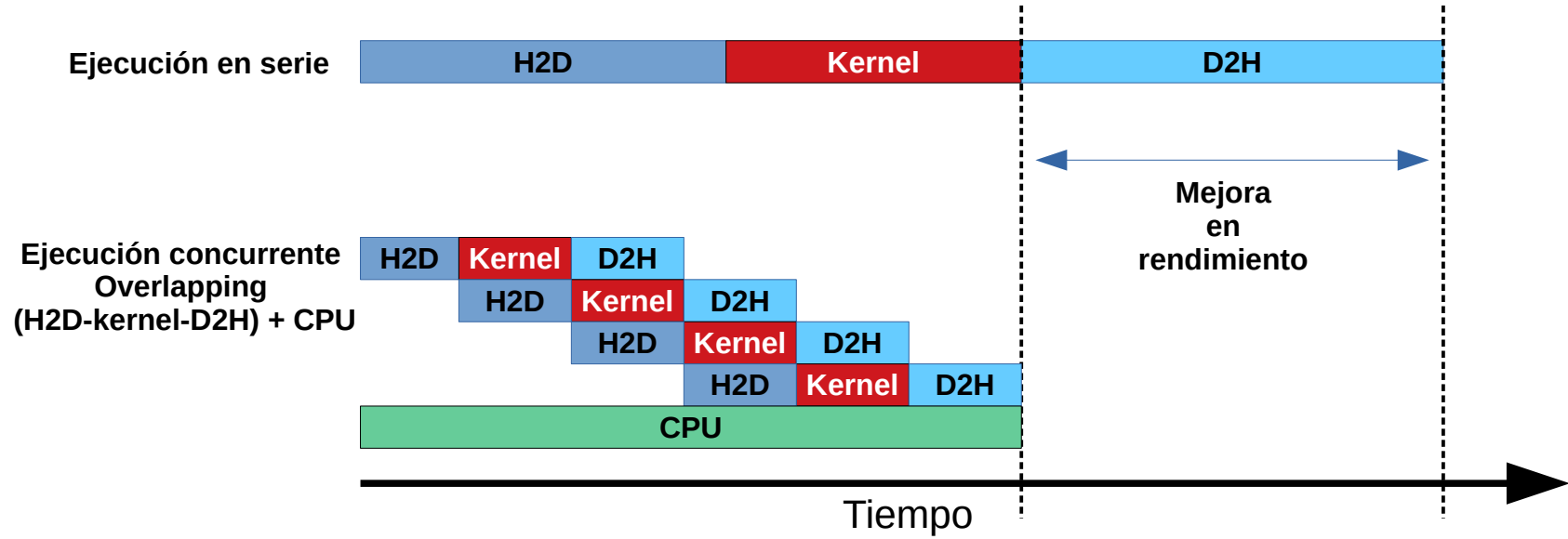


Flujo de ejecución CUDA

Ocultamiento de la latencia

7

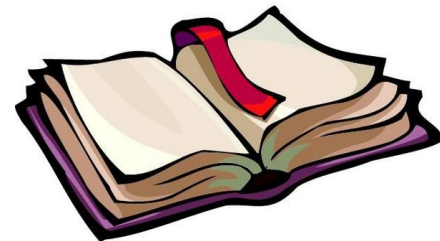
Una alternativa híbrida superpone la ejecución sobre GPU con la ejecución sobre CPU.



Agenda

8

- I. Flujo de ejecución CUDA y ocultamiento de la latencia
- II. CUDA Streams
 - i. Concurrencia a nivel kernel
 - ii. Concurrencia a nivel de memoria (Pinned Host Memory)
- III. Sincronización y eventos



- La forma adecuada de manejar este nivel de concurrencia en CUDA es mediante el uso de **Streams**:
 - Un **stream** es una **cola de trabajo** para el device
 - El host encola el trabajo y continúa inmediatamente
 - Cuando el device está libre se planifica el trabajo de los streams
 - Las operaciones de ejecución del kernel o la copias de memoria se asocian a un stream:
 - Las operaciones del **mismo stream** se ejecutan **secuencialmente** ordenadas (FIFO)
 - Las operaciones de diferentes streams se ejecutan desordenadas y pueden superponerse

CUDA Streams

Stream por defecto

10

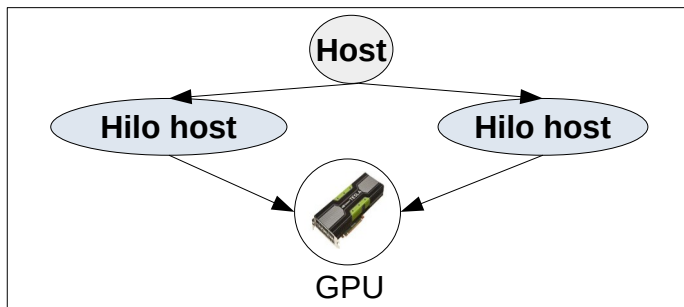
- Cuando no se especifica ningún stream, CUDA ejecuta en el **stream por defecto (null stream)**.
- El **stream por defecto** tiene **características específicas**:
 - Las operaciones de **copia de memoria** son **síncronas** con respecto al host: el host no puede continuar hasta que la copia se complete
 - La **invocación al kernel** es **asíncrona** con respecto al host: el host puede continuar una vez que se invoca al kernel y no espera que termine la ejecución
 - Una invocación por defecto sobre el stream por defecto es síncrona con respecto a la invocación de un kernel sobre otro stream.
- Para lograr concurrencia **NO** se recomienda utilizar el stream por defecto

CUDA Streams

Stream por defecto en hilos de host

11

- Algunas aplicaciones crean varios hilos host y cada hilo utiliza la misma GPU:



- Hasta CUDA versión 7:** los hilos host usarán el mismo stream por defecto.
 - Las operaciones se encolan en la misma cola de trabajo. En operaciones síncronas no puede haber superposición.
- CUDA 7 y posteriores:** se crea un stream para cada hilo host.
 - Las operaciones se encolan en colas de trabajo diferentes y las operaciones pueden superponerse.

CUDA Streams

Gestión de Streams

12

- Para utilizar nuestros propios streams (streams definidos) debemos:

- Declararlos: `cudaStream_t miStream;`

- Crearlos: `cudaStreamCreate(&miStream);`

- Luego de utilizados, destruirlos: `cudaStreamDestroy(miStream);`

- La forma de utilizar un stream es asociando trabajo. Pueden ser:

- Invocando al kernel:

```
miKernel<<<bloques, hilos, bytes de memoria compartida, stream>>>(parámetros);
```

- Realizando una copia de memoria:

```
cudaMemcpyAsync(destino, origen, bytes, tipo de copia, stream);
```

- Por las características específicas del Stream por defecto, el flujo de ejecución requiere sincronizar antes de realizar la copia D2H.
- Cuando se usan Streams definidos por el usuario, todo el flujo de ejecución es asíncrono respecto al host pero las operaciones se ejecutan secuencialmente ordenadas (FIFO) y no se requiere de esa sincronización hasta el final.

Stream por defecto

```
...
int main(int argc, char** argv){
...
    cudaMemcpy(..., cudaMemcpyHostToDevice);
    miKernel<<<...>>>();
    cudaDeviceSynchronize(); //Sincronización
    cudaMemcpy(..., cudaMemcpyDeviceToHost);
...
}
```

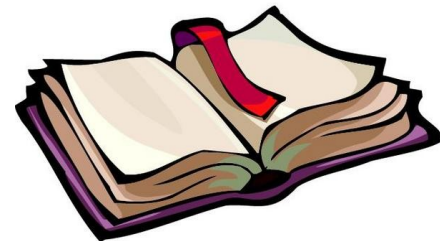
Stream definido por el usuario

```
...
int main(int argc, char** argv){
...
    cudaMemcpyAsync(..., cudaMemcpyHostToDevice, miStream);
    miKernel<<<..., miStream>>>();
    cudaMemcpyAsync(..., cudaMemcpyDeviceToHost, miStream);
    cudaDeviceSynchronize(); //Sincronización
...
}
```

Agenda

14

- I. Flujo de ejecución CUDA y ocultamiento de la latencia
- II. CUDA Streams
 - i. Concurrencia a nivel kernel
 - ii. Concurrencia a nivel de memoria (Pinned Host Memory)
- III. Sincronización y eventos



- Podemos invocar varios kernels y que éstos se ejecuten de forma concurrente en la GPU. Para ello debemos asegurar que cada invocación no utiliza la GPU por completo.
- De todas las posibles invocaciones a kernels se derivan las siguientes combinaciones:
 - 1) Dos o mas invocaciones al kernel sobre el stream por defecto
 - 2) Una invocación al kernel sobre el stream por defecto y otra sobre un stream definido:
 - Sin concurrencia
 - Con concurrencia
 - 3) Invocaciones únicamente con streams definidos

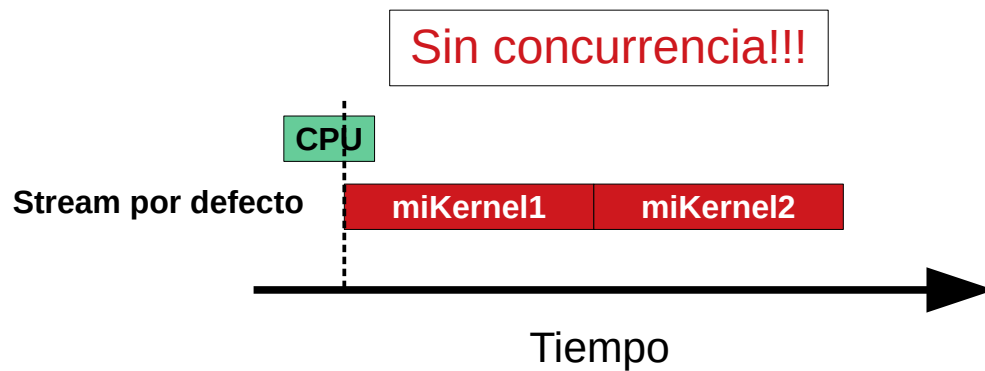
CUDA Streams

Concurrencia a nivel kernel

16

1) Dos o más invocaciones sobre el stream por defecto:

```
...  
int main(int argc, char** argv){  
...  
    //Código CPU  
    miKernel1<<<bloques, hilos>>>();  
    miKernel2<<<bloques, hilos>>>();  
...  
}
```



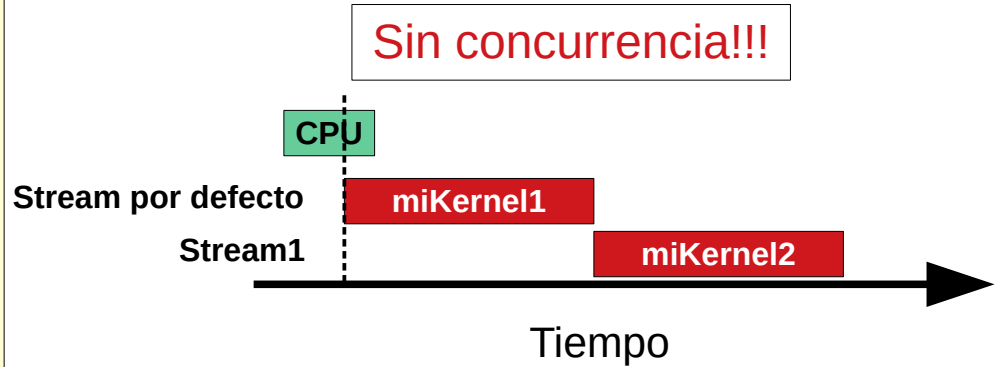
CUDA Streams

Concurrencia a nivel kernel

17

- 2) Una invocación al kernel sobre el stream por defecto y otra sobre un stream definido (sin concurrencia):

```
...  
int main(int argc, char** argv){  
    cudaStream_t stream1;  
    cudaStreamCreate(&stream1);  
    ...  
    //Código CPU  
    miKernel1<<<bloques, hilos>>>();  
    miKernel2<<<bloques, hilos, 0, stream1>>>();  
    ...  
    cudaStreamDestroy(stream1);  
}
```



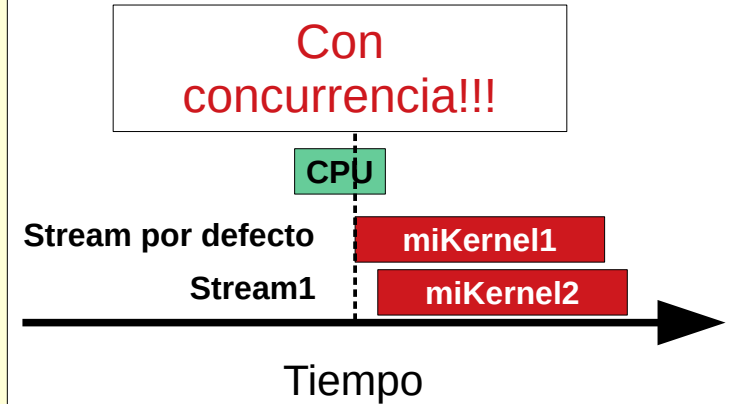
CUDA Streams

Concurrencia a nivel kernel

18

- 2) Una invocación al kernel sobre el stream por defecto y otra sobre un stream definido (con concurrencia):

```
...  
int main(int argc, char** argv){  
    cudaStream_t stream1;  
    cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking);  
    ...  
    //Código CPU  
    miKernel1<<<bloques, hilos>>>();  
    miKernel2<<<bloques, hilos, 0, stream1>>>();  
    ...  
    cudaStreamDestroy(stream1);  
}
```



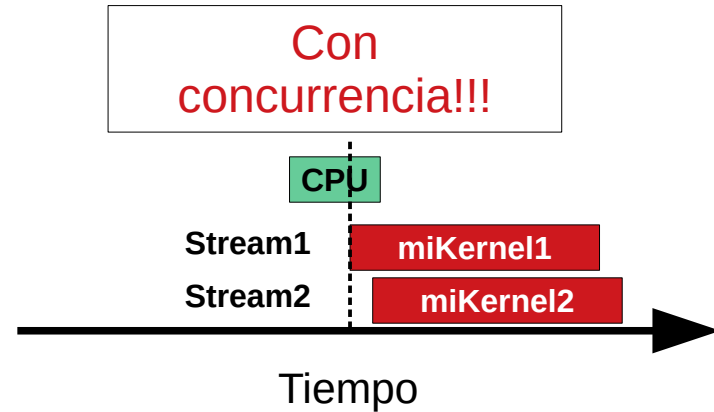
CUDA Streams

Concurrencia a nivel kernel

19

3) Invocaciones únicamente con streams definidos:

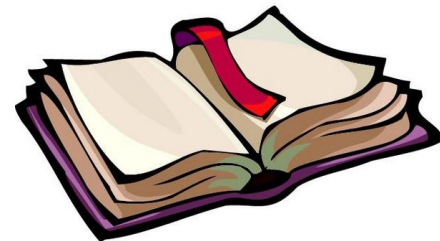
```
...  
int main(int argc, char** argv){  
    cudaStream_t stream1, stream2;  
    cudaStreamCreate(&stream1);  
    cudaStreamCreate(&stream2);  
...  
    //Código CPU  
    miKernel1<<<bloques, hilos,0,stream1>>>();  
    miKernel2<<<bloques, hilos,0,stream2>>>();  
...  
    cudaStreamDestroy(stream1);  
    cudaStreamDestroy(stream2);  
}
```



Agenda

20

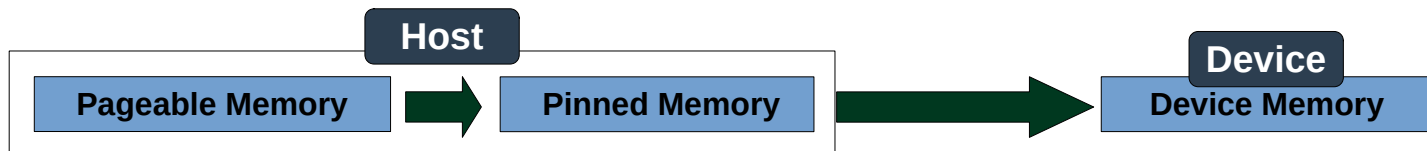
- I. Flujo de ejecución CUDA y ocultamiento de la latencia
- II. CUDA Streams
 - i. Concurrencia a nivel kernel
 - ii. Concurrencia a nivel de memoria (Pinned Host Memory)
- III. Sincronización y eventos



- El runtime system de CUDA considera tres tipos de memoria:
 - **Device Memory** (Memoria del Device):
 - Alocada usando cudaMalloc
 - No está paginada
 - **Pageable Host Memory** (Memoria del Host):
 - Alocación por defecto en el Host (malloc, calloc, etc.)
 - Está paginada por el SO tanto en memoria (In) como en el área de swap (Out)
 - **Pinned -Page locked- Host Memory** (Memoria del Host):
 - Alocación mediante alocadores especiales
 - No puede ser paginada por el SO

- **Pinned Host Memory:**

- Las alocaiones de datos en el host (CPU) son paginables por defecto.
- La GPU NO puede acceder directamente a los datos de la memoria paginable del host.
- En una transferencia H2D, el controlador CUDA debe alocar un host array temporal “page-locked” o “pinned”, copiar los datos del host al pinned-array y luego transferir los datos del pinned array a la memoria del device.



- Podemos evitar el costo de la transferencia entre Pageable Memory Host y Pinned asignando directamente nuestro host array en Pinned Memory



- Podemos alocar en memoria Pinned utilizando alguna de dos funciones:

```
cudaMallocHost(void ** ptr, size_t size)
```

```
cudaHostAlloc(void ** ptr, size_t size, unsigned int flags)
```

- Para liberar espacio en memoria Pinned se utiliza la función:

```
cudaFreeHost(void * ptr)
```

- Ejemplo:

```
...  
int main(int argc, char** argv){  
    int *h_pinned_ptr, *d_ptr;  
    cudaMallocHost(&h_pinned_ptr,bytes) ;  
    cudaMalloc(&d_ptr,bytes) ;  
    ...  
    cudaMemcpy(d_ptr,h_pinned_ptr,bytes,cudaMemcpyHostToDevice) ;  
    ...  
    cudaFreeHost(h_pinned_ptr) ;  
    cudaFree(d_ptr) ;  
    ...  
}
```


- En CUDA, las transferencias entre CPU y GPU pueden hacerse de dos formas:
 - **cudaMemcpy:**
 - Actúa siempre sobre el stream por defecto
 - Síncronas respecto al Host: el host no puede continuar hasta que la copia se complete
 - **cudaMemcpyAsync:**
 - Actúa sobre un stream definido (NO por defecto)
 - Asíncrona respecto al Host: el host puede continuar una vez que se invoca la función y no espera que termine la transferencia
 - La memoria en el host debe estar alocada en **Pinned Host Memory**

- Las invocaciones de copias de memoria asíncronas pueden combinarse con llamados al kernel asíncronos.
- Surgen dos posibilidades:
 - 1) Copias asíncronas y llamado al kernel sobre el mismo stream
 - 2) Copia asíncrona y llamado al kernel sobre diferentes streams

CUDA Streams

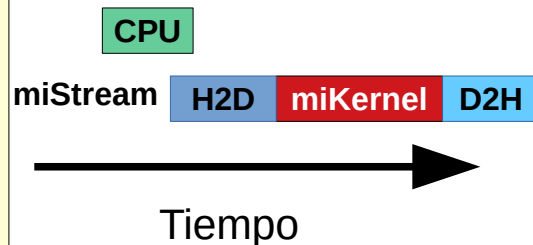
Concurrencia a nivel de memoria

27

1) Copias asíncronas (H2D y D2H) y llamado al kernel sobre el mismo stream:

```
...  
int main(int argc, char** argv){  
    cudaStream_t miStream;  
    int *h_pinned_ptr;  
    cudaStreamCreate(miStream);  
        cudaMallocHost(&h_pinned_ptr, bytes);  
        cudaMemcpyAsync(d_ptr, h_pinned_ptr, bytes, cudaMemcpyHostToDevice, miStream);  
        miKernel<<<bloques, hilos, 0, miStream>>>();  
        cudaMemcpyAsync(h_pinned_ptr, d_ptr, bytes, cudaMemcpyDeviceToHost, miStream);  
        cudaDeviceSynchronize();  
    ...  
        cudaStreamDestroy(miStream);  
        cudaFreeHost(h_pinned_ptr);  
}
```

Sin concurrencia!!!



Recordar: Las operaciones del mismo stream, cuando usamos streams definidos y NO el stream por defecto, son asíncronas respecto al host pero se ejecutan **secuencialmente** ordenadas (FIFO)

2) Copias asíncronas y llamado al kernel sobre diferentes streams:

```
...  
int main(int argc, char** argv){  
    cudaStream_t miStream1, miStream2, miStream3;  
    ...  
    cudaMemcpyAsync(d_ptr, h_pinned_ptr, bytes, cudaMemcpyHostToDevice, miStream1);  
    miKernel<<<bloques, hilos, 0, miStream2>>>();  
    cudaMemcpyAsync(h_pinned_ptr, d_ptr, bytes, cudaMemcpyDeviceToHost, miStream3);  
    ...  
}
```

Con
concurrencia!!!

CPU

miStream1

H2D

miStream2

miKernel

miStream3

D2H

Tiempo



- En CUDA, las copias de memoria se pueden ejecutar concurrentemente siempre y cuando:
 - Se realizan sobre un stream que no es el stream por defecto
 - La copia se hace sobre elementos almacenados en Pinned Host Memory
 - Se utiliza la función de transferencia asíncrona (`cudaMemcpyAsync`)
 - No hay otra copia de memoria ocurriendo en el mismo sentido (H2D o D2H) al mismo tiempo.

CUDA Streams

Concurrencia a nivel de memoria

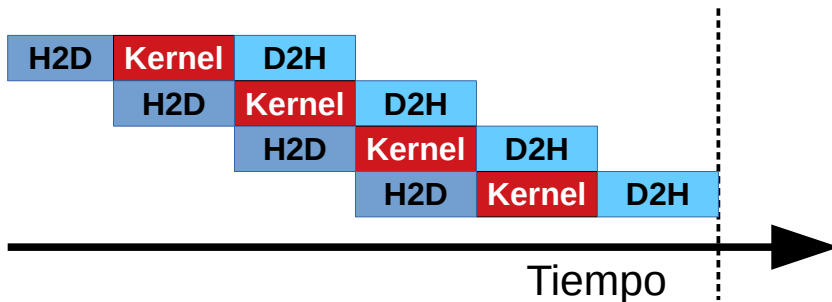
30

- Escenario completo para mejora de rendimiento por ocultamiento de la latencia en las transferencias H2D y D2H.

```
cudaStream_t streams[nStreams];  
  
...  
for(i=0; i<nStreams; i++){  
    offset = i*N/nStreams;  
    cudaMemcpyAsync( d_a + offset, h_a_pinned + offset, NBYTES_SIZE/nStreams, cudaMemcpyHostToDevice, streams[i] );  
    kernel<<<dimGrid, dimBlock, bytesShared, streams[i]>>>(d_a + offset, d_b + offset, N/nStreams);  
    cudaMemcpyAsync( h_b_pinned + offset, d_b + offset, NBYTES_SIZE/nStreams, cudaMemcpyDeviceToHost, streams[i] );  
}  
cudaDeviceSynchronize();
```

Ejecución concurrente
Overlapping
(H2D-kernel-D2H)

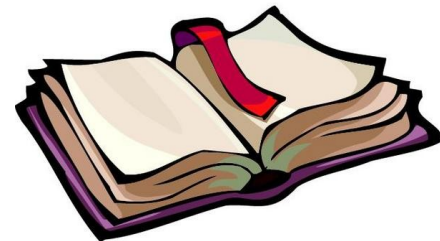
streams[0]
streams[1]
streams[2]
streams[3]



Agenda

31

- I. Flujo de ejecución CUDA y ocultamiento de la latencia
- II. CUDA Streams
 - i. Concurrencia a nivel kernel
 - ii. Concurrencia a nivel de memoria (Pinned Host Memory)
- III. Sincronización y eventos



CUDA Streams y sincronización

32

- Los llamados asíncronos deben converger en algún momento y debe existir alguna función que detenga la ejecución en el Host hasta que estos llamados se completen (**Sincronización**).
- En CUDA, existen dos funciones para realizar este comportamiento:
 - `cudaDeviceSynchronize()`: bloquea la ejecución en el host hasta que todos los llamados CUDA se completen.
 - `cudaStreamSynchronize(stream)`: bloquea la ejecución en el host hasta que todos los llamados CUDA del mismo stream (recibido como parámetro) se completen.

CUDA Streams y eventos

33

- Los **eventos** son mecanismos para señalar si una operación ocurrió en un stream dado.
- Sirven tanto para sincronización como para hacer profiling.
- Los eventos tienen un estado booleano:
 - “Ocurrió”
 - “No ocurrió”
- **Importante:** por defecto el estado es “Ocurrió”
- Para utilizarlos deben:
 - Declararse: `cudaEvent_t miEvento;`
 - Crearse: `cudaEventCreate(&miEvento);`
 - Luego de utilizados destruirse: `cudaEventDestroy(miEvento);`

CUDA Streams y eventos

34

- Una alternativa a crearlos por defecto es la creación con ciertos parámetros de configuración:

```
cudaEventCreateWithFlags(&miEvento, unsigned int flags);
```

- Los valores de los flags pueden ser:
 - **cudaEventDisableTiming**: el evento no necesita registrar el tiempo. **Recomendable: Deshabilitar el tiempo incrementa el rendimiento**
 - **cudaEventBlockingSync**: el evento debe usar sincronización bloqueante
 - **cudaEventInterprocess**: el evento puede usarse como un evento interproceso

CUDA Streams y eventos

35

- Antes de utilizar un evento sobre un stream se debe registrar:

```
cudaEventRecord(&miEvento, stream);
```

- Si no se especifica un stream se pasa como parámetro 0 para indicar que es el stream por defecto
- La función realiza lo siguiente:
 - Inicializa el valor del evento a “No ocurrió”
 - Encola el evento en un stream
- El evento pasa a estado “Ocurrió” cuando se completan las operaciones del stream

CUDA Streams y eventos

36

- Funciones que permiten trabajar con eventos:
 - **cudaEventQuery(evento)**: consulta por el estado del evento. Retorna `CUDA_SUCCESS` si el evento “Ocurrió”
 - **cudaEventSynchronize(evento)**: bloquea al Host hasta que el stream complete todas las llamadas pendientes. (Similar a `cudaDeviceSynchronize`)
 - **cudaStreamWaitEvent(stream, evento)**: bloquea un stream hasta que el evento ocurra. Sólo bloquea los llamados del stream, NO bloquea al host
 - **cudaEventElapsedTime(evento1, evento2)**: obtiene el tiempo transcurrido entre dos eventos.

CUDA Streams y eventos

Ejemplo: Tiempo de ejecución de un kernel

```
int main(int argc, char** argv){
    cudaEvent_t start, stop;
    float milliseconds = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

    cudaEventRecord(start);
    kernel<<<Bloques, Hilos>>>(Parametros);
    cudaEventRecord(stop);

    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&milliseconds, start, stop);

    ...
}
```

CUDA Streams

Sincronización forzada

38

- Si se crea la variable de ambiente `CUDA_LAUNCH_BLOCKING=1` se puede forzar a cuda a que todas las operaciones sean bloqueantes.
- Es útil para detectar condiciones de carrera (Race Conditions):
 - El programa NO tiene una condición de carrera si ejecutó correctamente al tener la variable definida en el ambiente
 - El programa tiene una condición de carrera si NO ejecutó correctamente al tener la variable definida en el ambiente

