TALLER DE PROGRAMACIÓN SOBRE GPUS

Facultad de Informática – Universidad Nacional de La Plata

Dr. Adrián Pousa

CUDA Streams

Agenda

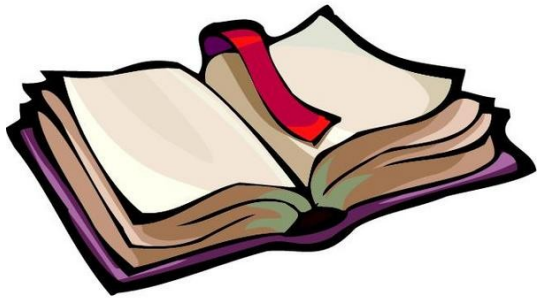
I. *Flujo de ejecución CUDA y ocultamiento de la latencia*

II. *CUDA Streams*

    I. *Concurrencia a nivel kernel*

    II. *Concurrencia a nivel de memoria (Pinned Host Memory)*

III. *Sincronización y eventos*



Agenda

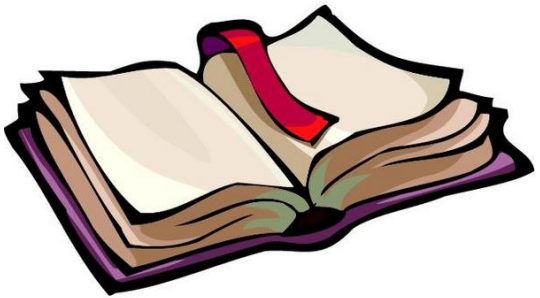
I. ***Flujo de ejecución CUDA y ocultamiento de la latencia***

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

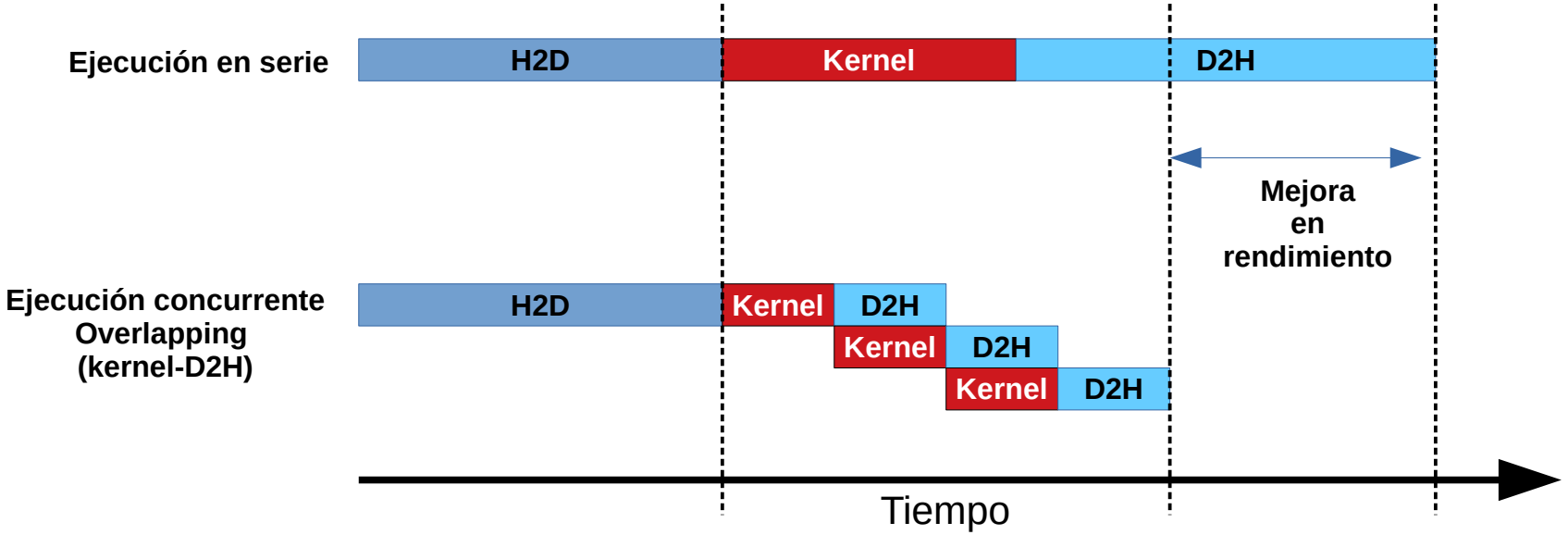
- Flujo de procesamiento típico sobre GPU (**Ejecución en serie**)
  1) Copiar los datos de entrada de CPU a GPU (Transferencia Host to Device - **H2D**)
  2) Ejecutar el Kernel
  3) Copiar los resultado de GPU a CPU (Transferencia Device to Host - **D2H**)

El costo de las transferencias H2D y D2H es alto

¿Es posible ocultar la latencia de estas transferencias?

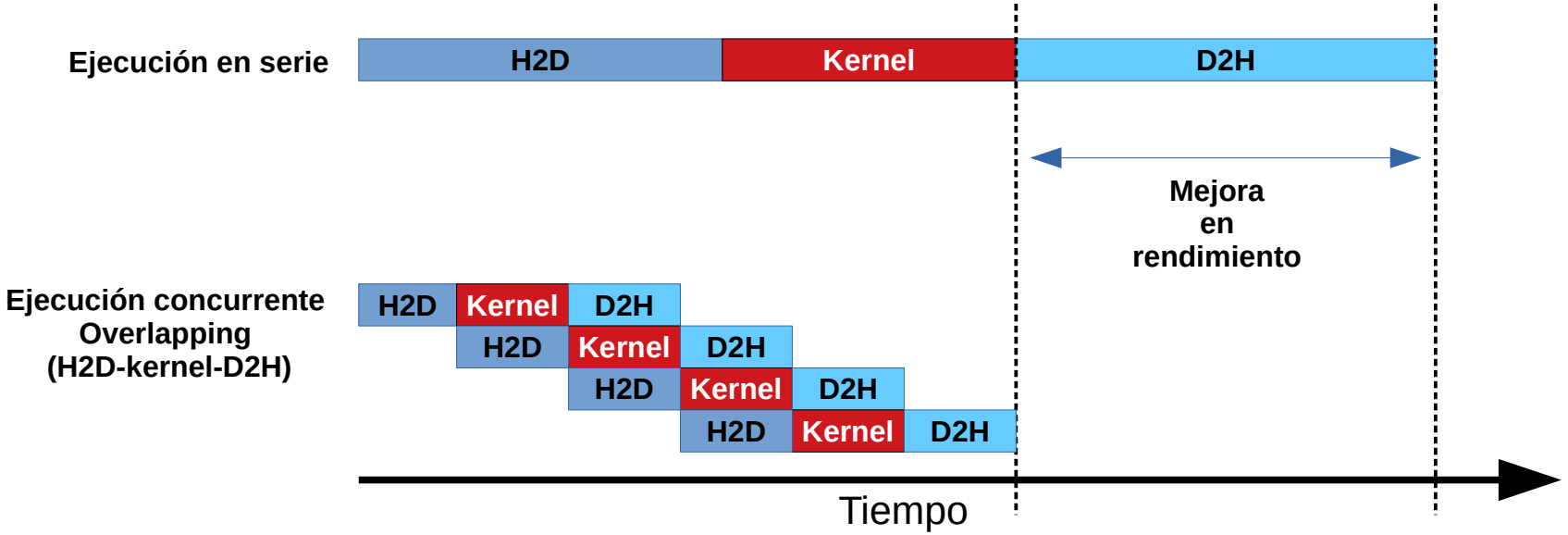
Ocultamiento de la latencia

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



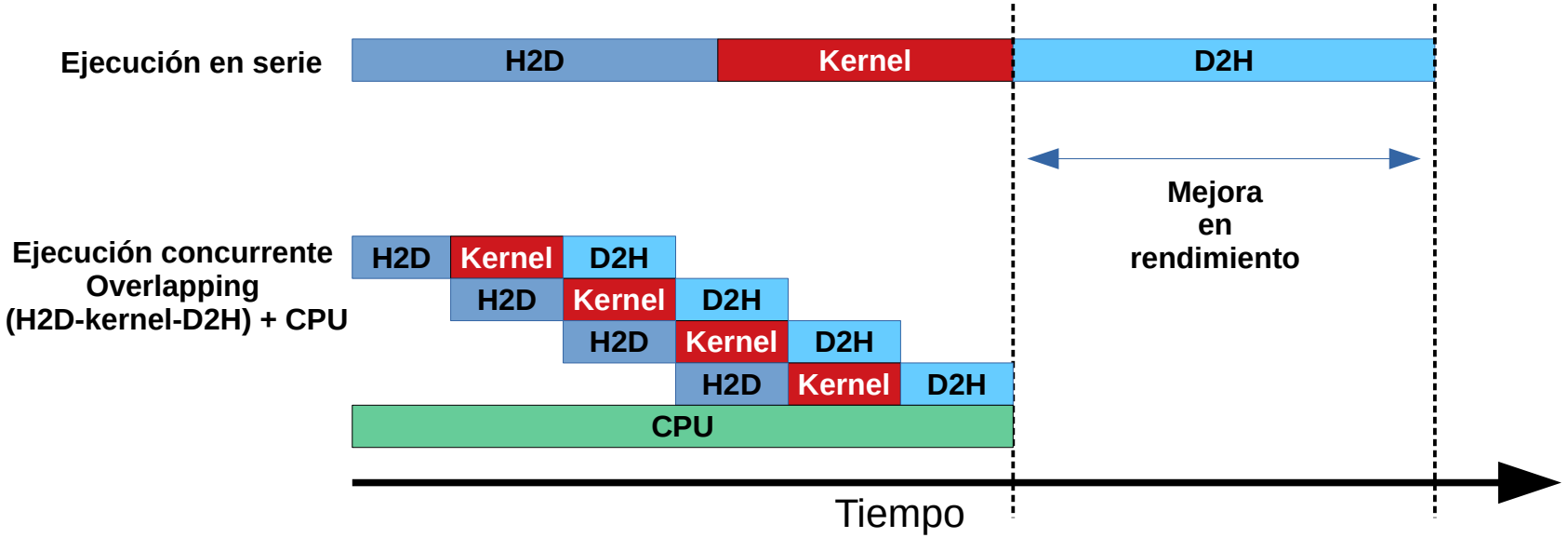
Ocultamiento de la latencia

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



Ocultamiento de la latencia

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



Agenda

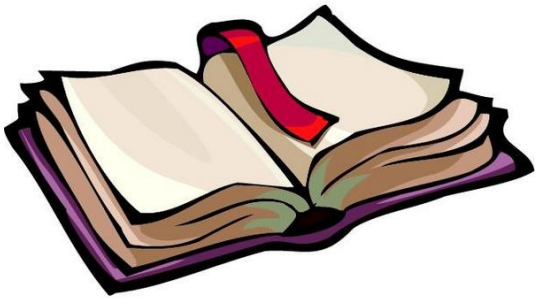
I. *Flujo de ejecución CUDA y ocultamiento de la latencia*

II. ***CUDA Streams***

   I. ***Concurrencia a nivel kernel***

   II. ***Concurrencia a nivel de memoria (Pinned Host Memory)***

III. *Sincronización y eventos*



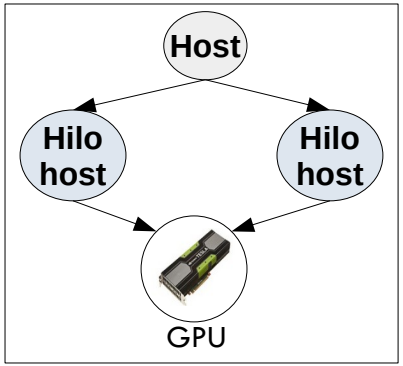
Streams

- 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

Streams por defecto

- Cuando no se especifica ningún stream CUDA ejecuta en el stream por defecto (**null stream**).
- El stream por defecto tiene ciertas características particulares:
  - 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

Streams por defecto e hilos en el host



- Versiones anteriores a CUDA 7: si el host crea dos **hilos host** y cada uno utiliza el device, ambos usarán el mismo stream por defecto.
  - Las operaciones se encolan en la misma cola de trabajo. En operaciones síncronas no puede haber superposición.
- Versiones CUDA 7 y posteriores: se crea un stream para cada **hilo host**.
  - Las operaciones se encolan en colas de trabajo diferentes y pueden superponerse operaciones.

Gestión de Streams

- Si queremos crear nuestros propios streams debemos:
  - Declararlos:

    ```
    cudaStream_t miStream;
    ```

  - Crearlos:

    ```
    cudaStreamCreate(&miStream);
    ```

  - Luego de utilizados, destruirlos:

    ```
    cudaStreamDestroy(miStream);
    ```

Gestión de Streams

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

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

  - Realizando una copia de memoria:

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

Agenda

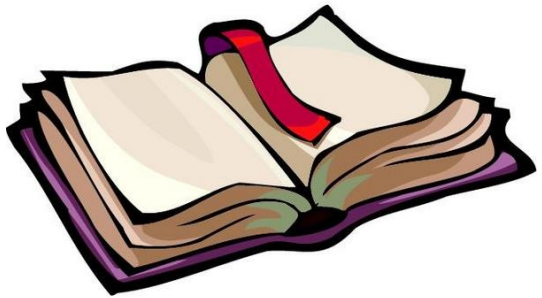
I. *Flujo de ejecución CUDA y ocultamiento de la latencia*

II. ***CUDA Streams***

    I. ***Concurrencia a nivel kernel***

    II. *Concurrencia a nivel de memoria (Pinned Host Memory)*

III. *Sincronización y eventos*



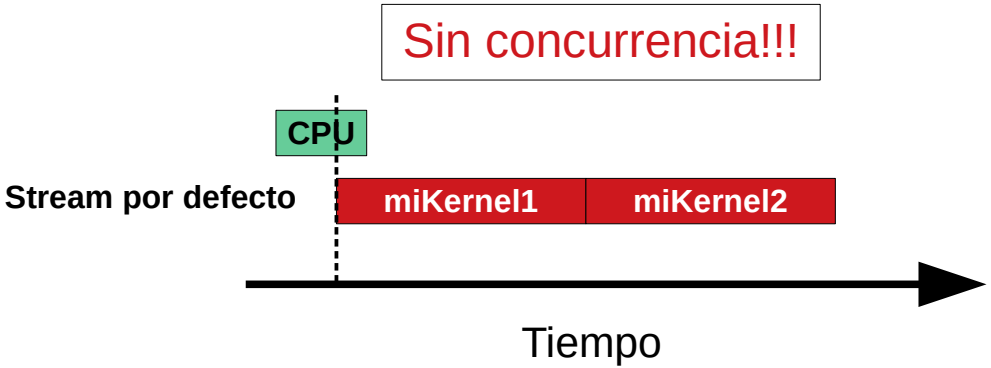
Streams: Concurrencia nivel kernel

- Para lograr concurrencia al invocar varios kernels debemos asegurar que cada invocación no utiliza la GPU por completo.
- De varias invocaciones a kernels se derivan las siguientes combinaciones:
  - Dos o mas invocaciones al kernel sobre el stream por defecto
  - Una invocación al kernel sobre el stream por defecto y otra sobre un stream definido:
    - Sin concurrencia
    - Con concurrencia
  - Invocaciones únicamente con streams definidos

Streams: Concurrencia nivel kernel

- Dos o mas invocaciones al kernel sobre el stream por defecto:

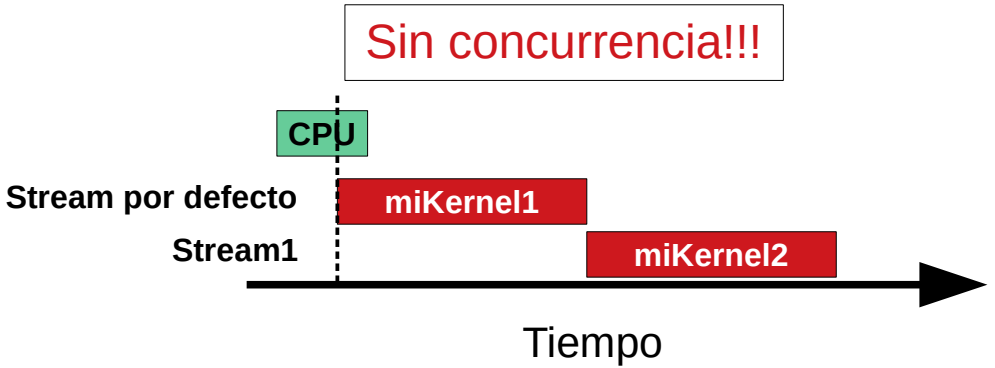
```
...
int main(int argc, char** argv){
 ...
      //Código CPU
      miKernel1<<<bloques, hilos>>>();
      miKernel2<<<bloques, hilos>>>();
 ...
 }
```



Streams: Concurrencia nivel kernel

- Una invocación al kernel sobre el stream por defecto y otra sobre un stream definido:

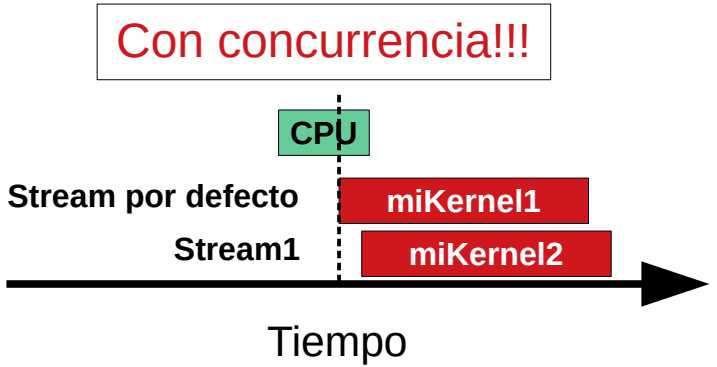
```
...
int main(int argc, char** argv){
 cudaStream_t stream1;
 cudaStreamCreate(&stream1);
...
      //Código CPU
      miKernel1<<<bloques, hilos>>>();
      miKernel2<<<bloques, hilos,0,stream1>>>();
...
      cudaStreamDestroy(stream1);
}
```



Streams: Concurrencia nivel kernel

- Una invocación al kernel sobre el stream por defecto y otra sobre un stream definido

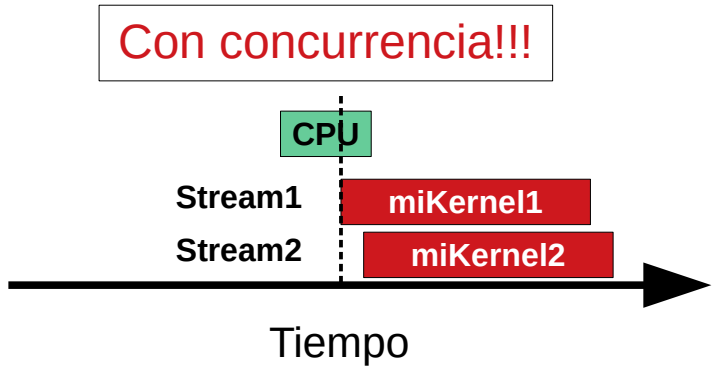
```
...
int main(int argc, char** argv){
 cudaStream_t stream1;
 cudaStreamCreateWithFlags(&stream1,cudaStreamNonBlocking);
...
      //Código CPU
      miKernel1<<<bloques, hilos>>>();
      miKernel2<<<bloques, hilos,0,stream1>>>();
...
      cudaStreamDestroy(stream1);
}
```



Streams: Concurrencia nivel kernel

- 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

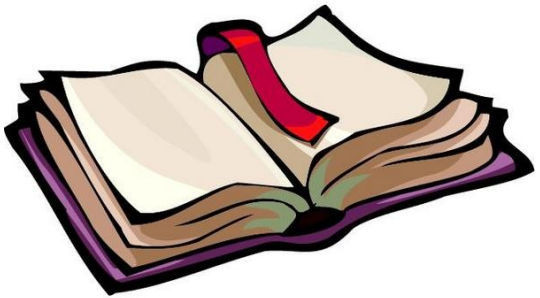
I. *Flujo de ejecución CUDA y ocultamiento de la latencia*

II. ***CUDA Streams***

    I. *Concurrencia a nivel kernel*

    II. ***Concurrencia a nivel de memoria (Pinned Host Memory)***

III. *Sincronización y eventos*

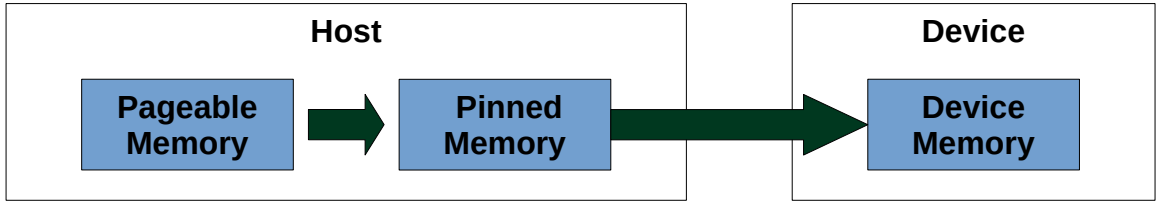


Streams: Concurrencia nivel memoria

- El runtime system de CUDA distingue tres tipos de memoria:
  - Device Memory (Memoria del dispositivo):
    - 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

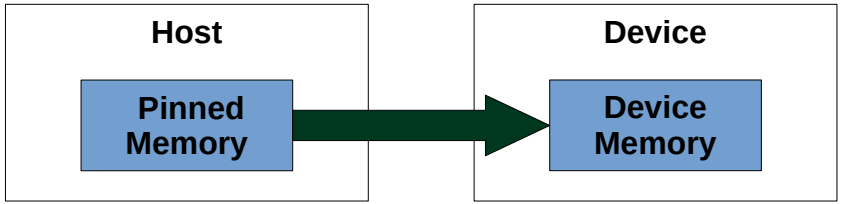
Streams: Concurrencia nivel memoria

- Pinned Host Memory:
  - Las alocaciones 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.
  - Cuando se realiza una transferencia H2D, el controlador CUDA debe alocar primero 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.



Streams: Concurrencia nivel memoria

- Pinned Host Memory se utiliza como un área de preparación para transferencias D2H.
- Podemos evitar el costo de la transferencia entre Pageable Memory Host y Pinned asignando directamente nuestro host array en Pinned Memory



Streams: Concurrencia nivel memoria

- Para alocar memoria en el área pinned se pueden utilizar dos funciones:

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

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

- Se libera utilizando la siguiente función:

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

Streams: Concurrencia nivel memoria

- 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);
}
```

Streams: Concurrencia nivel memoria

- En CUDA, las transferencias entre CPU y GPU se pueden hacer de dos formas:
  - cudaMemcpy:
    - Actúa siempre sobre el stream por defecto
    - Síncronas con respecto al Host: el host no puede continuar hasta que la copia se complete
  - cudaMemcpyAsync:
    - Actúa sobre un stream que no sea el stream por defecto
    - Asíncrona con 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**

Streams: Concurrencia nivel memoria

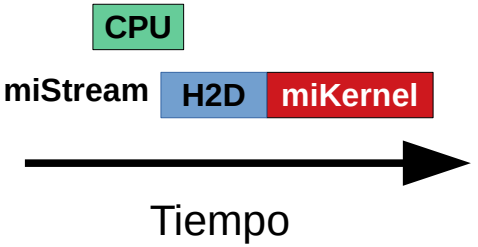
- Podemos derivar las siguientes combinaciones:
  - Copia asíncrona y llamado al kernel sobre el mismo stream
  - Copia asíncrona y llamado al kernel sobre diferentes streams

Streams: Concurrencia nivel memoria

- cudaMemcpyAsync 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>>>();
...
      cudaStreamDestroy(miStream1);
      cudaFreeHost(h_pinned_ptr);
}
```

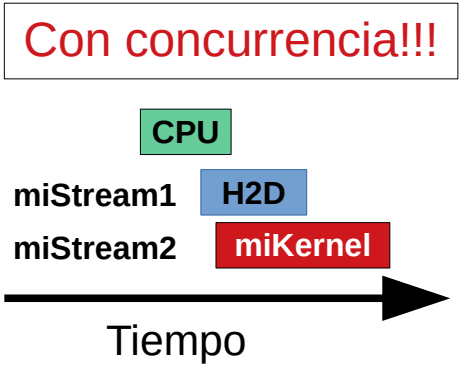
Sin concurrencia!!!



Streams: Concurrencia nivel memoria

- cudaMemcpyAsync sobre diferentes streams:

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

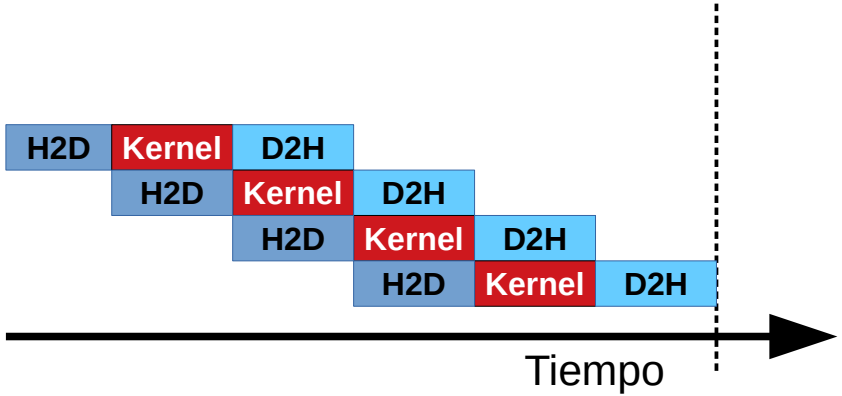


Streams: Concurrencia nivel memoria

- En CUDA, las copias de memoria se pueden ejecutar concurrentemente siempre y cuando:
  - Las copias de memorias 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.

Ocultamiento de la latencia

Ejecución concurrente
Overlapping
(H2D-kernel-D2H)




```
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();
```

Agenda

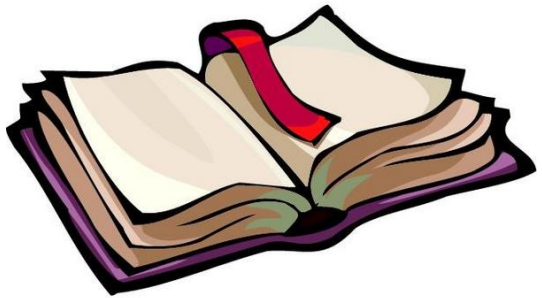
I. *Flujo de ejecución CUDA y ocultamiento de la latencia*

II. *CUDA Streams*

    I. *Concurrencia a nivel kernel*

    II. *Concurrencia a nivel de memoria (Pinned Host Memory)*

III. ***Sincronización y eventos***



Streams: Sincronización

- 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.

Streams: Eventos

- Los eventos son mecanismos para señalizar cuando las operaciones ocurrieron en un stream dado.
- Sirven tanto para sincronización como para hacer profiling.
- Los eventos tienen un estado booleano:
  - **“Ocurrio”**
  - **“No ocurrió”**
- Importante: por defecto el estado es “Ocurrió”

Gestión de eventos

- Antes de utilizarlos los eventos se declaran:

```
cudaEvent_t miEvento;
```

- Y se crean:

```
cudaEventCreate(&miEvento);
```

- Luego de utilizarlos se deben destruir:

```
cudaEventDestroy(miEvento);
```

Gestión de eventos

- 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

Gestión de eventos

- 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

Sincronización de eventos

- 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.

Eventos: Ej. 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);
...
}
```

Sincronización forzada

- 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**