

INF351 – Computación de Alto Desempeño

Concurrencia y CUDA Streams

PROF. ÁLVARO SALINAS

Concurrencia

La concurrencia es la habilidad de realizar múltiples operaciones de CUDA de forma simultánea (estamos hablando de un paralelismo más allá del esquema multi-thread).

Como hemos visto hasta el momento, las invocaciones de CUDA kernels son un proceso asincrónico para la CPU, mientras las operaciones de memoria (`cudaMalloc`, `cudaMemcpy`, etc.) son sincrónicas.

A continuación veremos una variante asincrónica de `cudaMemcpy`. Esto nos permitirá explotar aun más la potencia de la GPU en algunas aplicaciones.

cudaMemcpyAsync

Al igual que cudaMemcpy, esta función realiza una copia de memoria en una determinada dirección.

La principal diferencia es que esta versión es asincrónica para la CPU.

Su sintáxis es la siguiente:

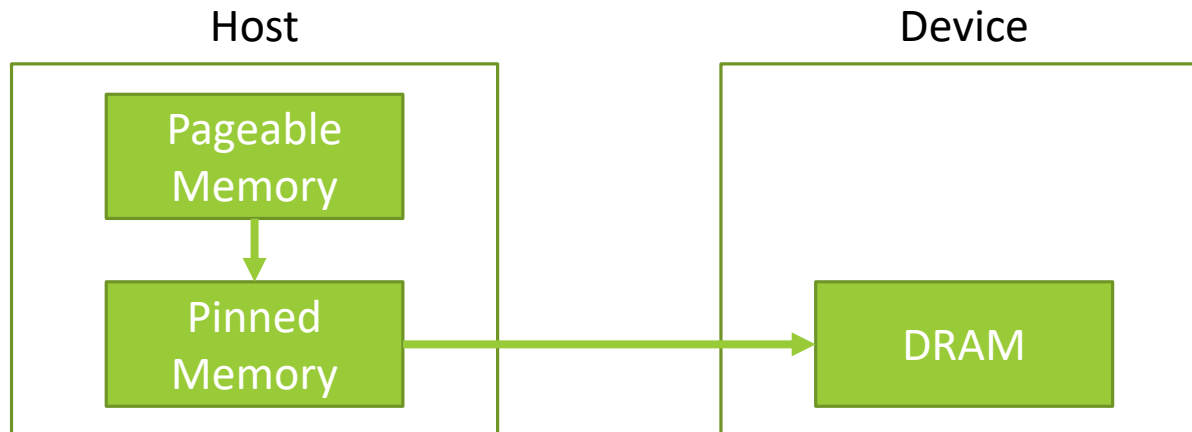
*cudaMemcpyAsync(*dst, *src, size, cudaMemcpyKind, cudaStream)*

donde *dst* y *src* son los punteros de destino y fuente respectivamente, *size* es el tamaño en bytes a ser copiado, *cudaMemcpyKind* corresponde a la dirección de la copia (*cudaMemcpyHostToHost*, *cudaMemcpyHostToDevice*, *cudaMemcpyDeviceToHost* o *cudaMemcpyDeviceToDevice*), y *cudaStream* es algo que veremos en unas diapositivas más.

Pinned Memory

Otra diferencia entre *cudaMemcpy* y su versión asincrónica es el hecho de que esta última solo funciona con page-locked (o “pinned”) memory.

La GPU siempre realiza DMA desde pinned memory, así que cuando la memoria de host es asignada a través de *new* o *malloc*, el driver de CUDA debe realizar en cada *cudaMemcpy* una copia de memoria adicional a un puntero interno a pinned memory.



Pinned Memory

Para asignar page-locked memory, simplemente debemos utilizar:

*cudaMallocHost(**ptr, size)*

debiendo preocuparnos de liberarla con:

*cudaFreeHost(*ptr)*

Es posible utilizar pinned memory con *cudaMemcpy* y su rendimiento será mucho mejor.

Lamentablemente, la asignación de pinned memory (*cudaMallocHost*) es bastante costosa, por lo que solo será útil cuando tengamos varias copias de memoria o cuando necesitemos concurrencia.

Engines

Las GPUs soportan una determinada cantidad de operaciones simultáneas. Por ejemplo, las GPUs Tesla de compute capability 2.0 (microarquitectura Fermi) soportan:

- Hasta 16 CUDA kernels
- 2 *cudaMemcpyAsync* (en diferentes direcciones).
- Cálculos en CPU

Esto se debe a que poseen 2 copy engines y 1 compute engine. Estos engines son colas en donde las operaciones son enlistadas para su posterior ejecución.

Esta configuración depende de la tarjeta gráfica utilizada.

CUDA Stream

Un stream se define como una serie de operaciones que se ejecutan de forma secuencial en la GPU.

Hasta ahora, todas las operaciones que hemos realizado en la GPU han sido secuenciales. Siempre hemos estado usando un CUDA stream por defecto (llamado stream 0).

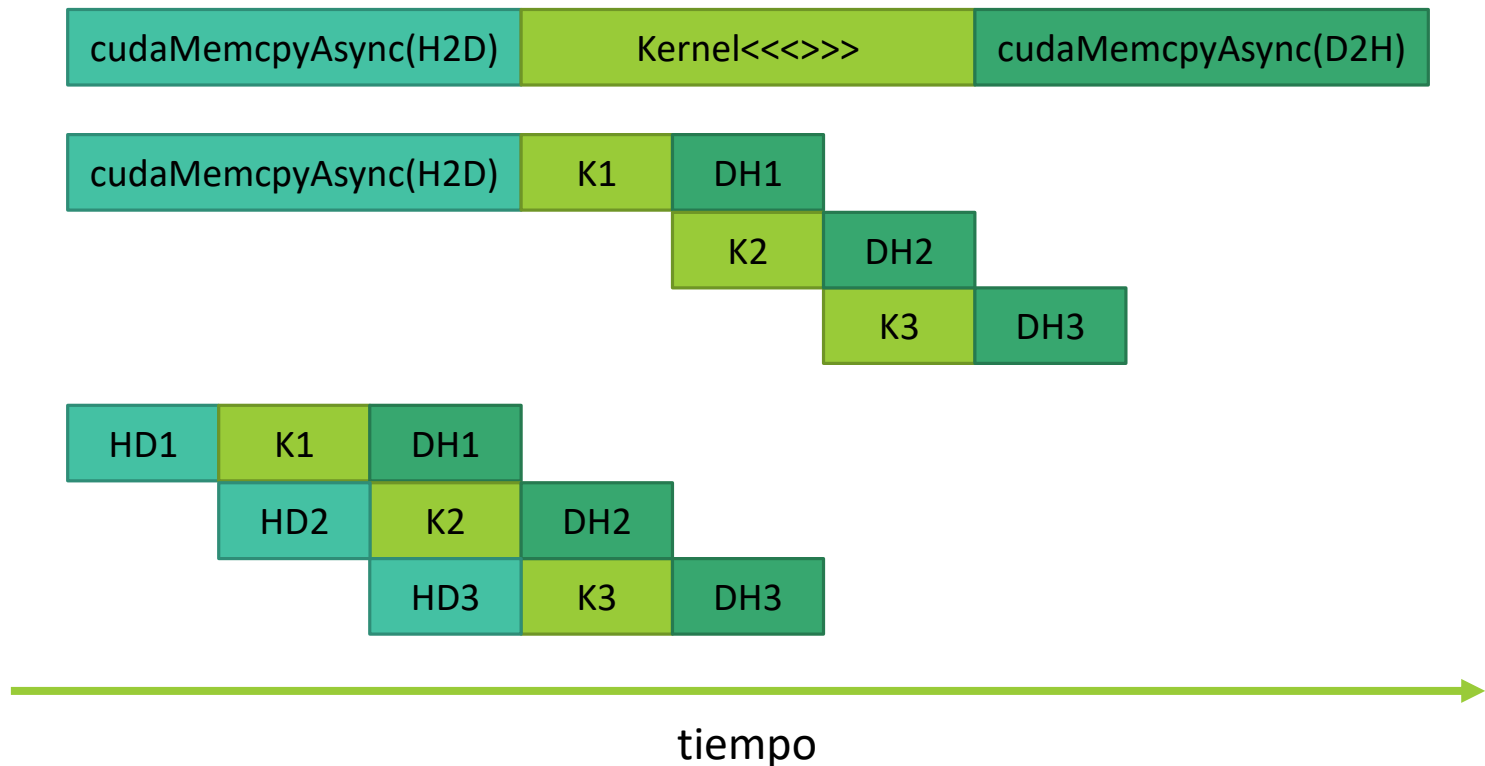
¿Qué pasaría si usásemos más streams?

¡Operaciones en distintos CUDA streams pueden ser ejecutadas de forma simultánea!



CUDA Stream

Es posible alcanzar distintos niveles de concurrencia:



CUDA Stream

Distintos CUDA streams son creados con:

```
cudaStream_t stream;  
cudaStreamCreate(&stream);
```

De esta forma, *stream* es el quinto parámetro que podríamos usar en *cudaMemcpyAsync*.

Para definir en qué stream se ejecuta un kernel, debemos darle un cuarto parámetro en su configuración:

```
kernel<<<grid_size, block_size, sharedmem_size, stream>>>(...)
```

Requerimientos concurrencia

Para que las operaciones de CUDA se ejecuten de forma simultánea se deben cumplir los siguientes requerimientos:

- Deben estar en streams diferentes.
- Copias de memoria deben ser asincrónicas desde pinned memory.
- Los datos utilizados deben ser independientes.
- Deben haber suficientes recursos:
 - Engines para copias de memoria (diferentes direcciones).
 - Recursos de la GPU (Shared Memory, registros, etc.).

Ejemplos

Ejecución sincrónica:

```
int *dev1;  
cudaMalloc(&dev1, size);  
int *host1 = (int*)malloc(size);  
...  
cudaMemcpy(dev1, host1, size, cudaMemcpyHostToDevice);  
kernel2<<<grid_size, block_size>>>(..., dev2, ...);  
kernel3<<<grid_size, block_size>>>(..., dev3, ...);  
cudaMemcpy(host4, dev4, size, cudaMemcpyDeviceToHost);  
...
```

Ejemplos

Ejecución asincrónica sin streams:

```
int *dev1;
cudaMalloc(&dev1, size);
int *host1 = (int*)malloc(size);
...
cudaMemcpy(dev1, host1, size, cudaMemcpyHostToDevice);
kernel2<<<grid_size, block_size>>>(..., dev2, ...);
funcion_CPU();
kernel3<<<grid_size, block_size>>>(..., dev3, ...);
cudaMemcpy(host4, dev4, size, cudaMemcpyDeviceToHost);
...
```

Ejemplos

Ejecución asincrónica con streams:

```
cudaStream_t stream1;  
cudaStreamCreate(&stream1);  
...  
int *dev1, *host1;  
cudaMalloc(&dev1, size);  
cudaMallocHost(&host1, size);  
...  
cudaMemcpyAsync(dev1, host1, size, cudaMemcpyHostToDevice, stream1);  
kernel2<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...);  
kernel3<<<grid_size, block_size, 0, stream3>>>(..., dev3, ...);  
cudaMemcpyAsync(host4, dev4, size, cudaMemcpyDeviceToHost, stream4);  
funcion_CPU();  
...
```

Sincronización

A continuación, veremos los distintos tipos de sincronización que podemos utilizar al trabajar con streams:

- Para sincronizar la totalidad de la aplicación (bloquear la CPU hasta que todas las operaciones de CUDA hayan sido resueltas) utilizaremos *cudaDeviceSynchronize()* tal y como hemos hecho hasta ahora.
- Para sincronizar el host con un determinado stream utilizaremos *cudaStreamSynchronize(stream)*.
- Finalmente, podemos también sincronizar mediante *cudaEvents*. Esto permite imponer una sincronización entre streams.

Sincronización con events

Para lograr esta sincronización, primero debemos agregar un stream como parámetro en la llamada de `cudaEventRecord`:

`cudaEventRecord(event, stream1)`

lo que vincula el evento *event* al stream *stream1*. Luego, podemos utilizar

`cudaStreamWaitEvent(stream2, event)`

para lograr que otro stream *stream2* espere a la llamada más reciente de `cudaEventRecord` sobre el evento *event* para continuar ejecutando sus operaciones.

Ejemplo

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
...
cudaEvent_t event;
cudaEventCreate(&event);
...
cudaMemcpyAsync(dev1, host1, size, cudaMemcpyHostToDevice, stream1);
cudaEventRecord(event, stream1);
cudaMemcpyAsync(host2, dev2, size, cudaMemcpyDeviceToHost, stream2);
cudaStreamWaitEvent(stream2, event);
kernel<<<grid_size, block_size, 0, stream2>>>(..., dev1, dev2, ...);
...
```

Implementación y orden

Dado que se cuenta con recursos limitados, es importante entender el funcionamiento de las colas de operaciones:

- Las operaciones son asignadas a las colas en el orden en que fueron llamadas en CPU.
- Una vez en cola, las operaciones son ejecutadas solo si todos los trabajos anteriores en la cola fueron resueltos, todos los trabajos anteriores del mismo stream fueron resueltos y hay recursos disponibles.
- Aunque haya un solo compute engine, kernels de distintos streams pueden ejecutarse simultáneamente.
- En GPUs que soportan ejecución concurrente de kernels, kernels llamados secuencialmente (aunque sean de distintos streams) bloquean las demás colas hasta la resolución del último kernel.

Ejemplos

Bloqueo por stream:

```
kernel<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...);  
cudaMemcpyAsync(dev2, host2, size, cudaMemcpyHostToDevice, stream1);  
kernel<<<grid_size, block_size, 0, stream1>>>(..., dev2, ...);  
cudaMemcpyAsync(host2, dev2, size, cudaMemcpyDeviceToHost, stream1);
```

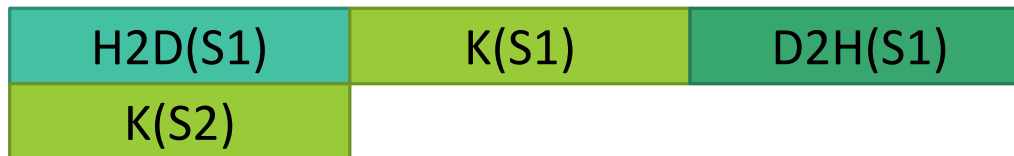


tiempo

Ejemplos

Bloqueo por stream:

```
kernel<<<grid_size, block_size, 0, stream2>>>(..., dev1, ...);  
cudaMemcpyAsync(dev2, host2, size, cudaMemcpyHostToDevice, stream1);  
kernel<<<grid_size, block_size, 0, stream1>>>(..., dev2, ...);  
cudaMemcpyAsync(host2, dev2, size, cudaMemcpyDeviceToHost, stream1);
```

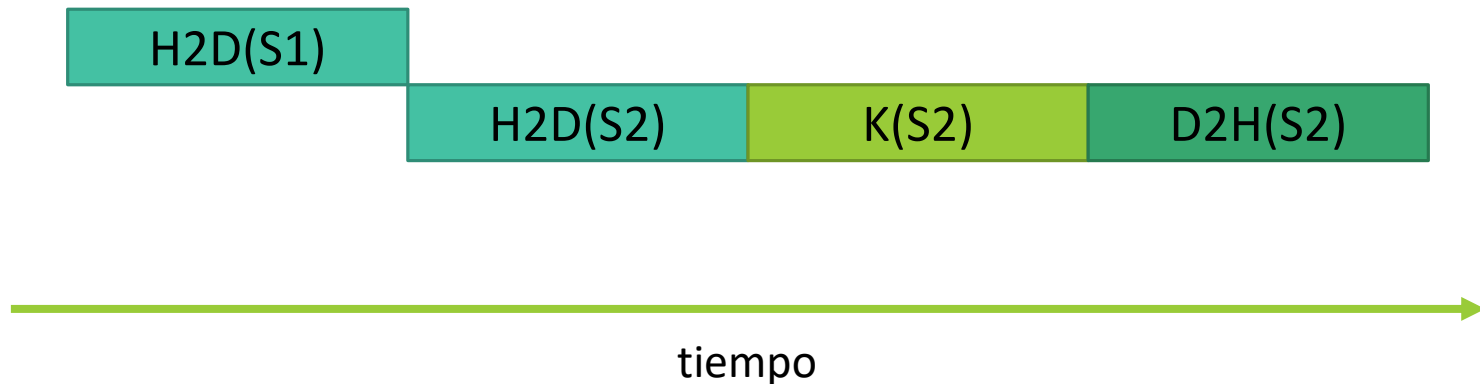


tiempo

Ejemplos

Bloqueo por cola:

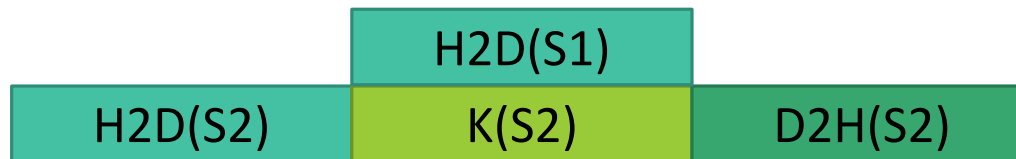
```
cudaMemcpyAsync(dev1, host1, size, cudaMemcpyHostToDevice, stream1);  
cudaMemcpyAsync(dev2, host2, size, cudaMemcpyHostToDevice, stream2);  
kernel<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...);  
cudaMemcpyAsync(host2, dev2, size, cudaMemcpyDeviceToHost, stream2);
```



Ejemplos

Bloqueo por cola:

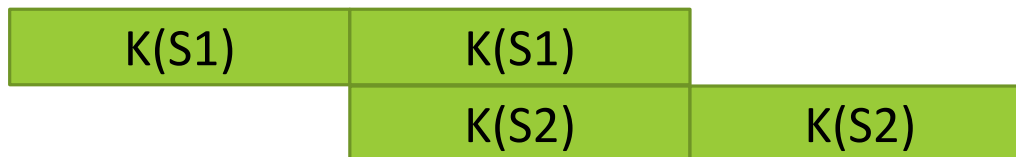
```
↪ cudaMemcpyAsync(dev2, host2, size, cudaMemcpyHostToDevice, stream2);  
↪ cudaMemcpyAsync(dev1, host1, size, cudaMemcpyHostToDevice, stream1);  
  kernel<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...);  
  cudaMemcpyAsync(host2, dev2, size, cudaMemcpyDeviceToHost, stream2);
```



Ejemplos

Bloqueo mixto (stream y cola):

```
kernel<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...);  
kernel<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...);  
kernel<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...);  
kernel<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...);
```




→
tiempo

Ejemplos

Bloqueo mixto (stream y cola):

```
kernel<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...);  
kernel<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...);  
kernel<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...);  
kernel<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...);
```



K(S1)	K(S1)
K(S2)	K(S2)

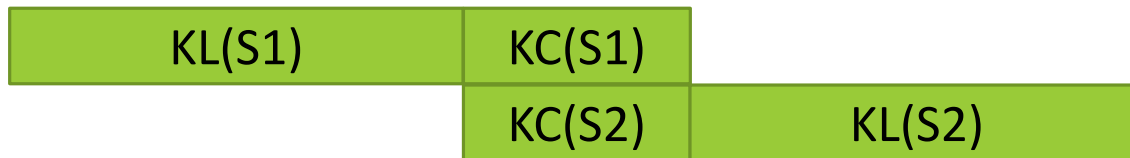


tiempo

Ejemplos

Bloqueo mixto (stream, cola y duración):

```
kernelL<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...); // Larga duración  
kernelC<<<grid_size, block_size, 0, stream1>>>(..., dev2, ...); // Corta duración  
kernelC<<<grid_size, block_size, 0, stream2>>>(..., dev3, ...); // Corta duración  
kernelL<<<grid_size, block_size, 0, stream2>>>(..., dev4, ...); // Larga duración
```



tiempo

Ejemplos

Bloqueo mixto (stream, cola y duración):

```
kernelL<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...); // Larga duración  
kernelC<<<grid_size, block_size, 0, stream2>>>(..., dev3, ...); // Corta duración  
kernelC<<<grid_size, block_size, 0, stream1>>>(..., dev2, ...); // Corta duración  
kernelL<<<grid_size, block_size, 0, stream2>>>(..., dev4, ...); // Larga duración
```

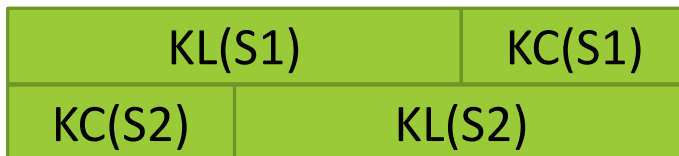


tiempo

Ejemplos

Bloqueo mixto (stream, cola y duración):

```
kernelL<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...); // Larga duración  
kernelC<<<grid_size, block_size, 0, stream2>>>(..., dev3, ...); // Corta duración  
kernelL<<<grid_size, block_size, 0, stream2>>>(..., dev4, ...); // Corta duración  
kernelC<<<grid_size, block_size, 0, stream1>>>(..., dev2, ...); // Larga duración
```

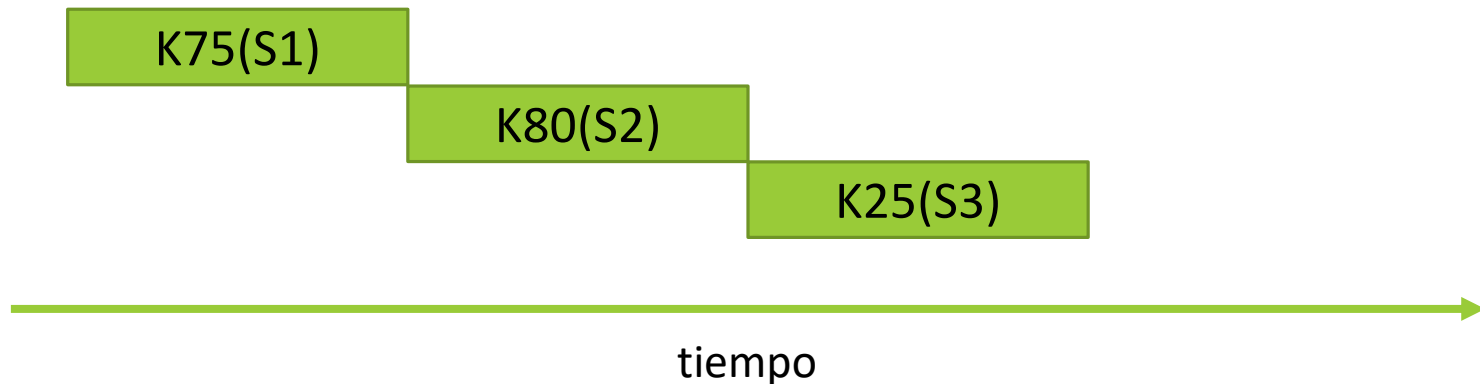


→
tiempo

Ejemplos

Bloqueo por recursos:

```
kernel75<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...); // 75% de los recursos  
kernel80<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...); // 80% de los recursos  
kernel25<<<grid_size, block_size, 0, stream3>>>(..., dev3, ...); // 25% de los recursos
```



Ejemplos

Bloqueo por recursos:

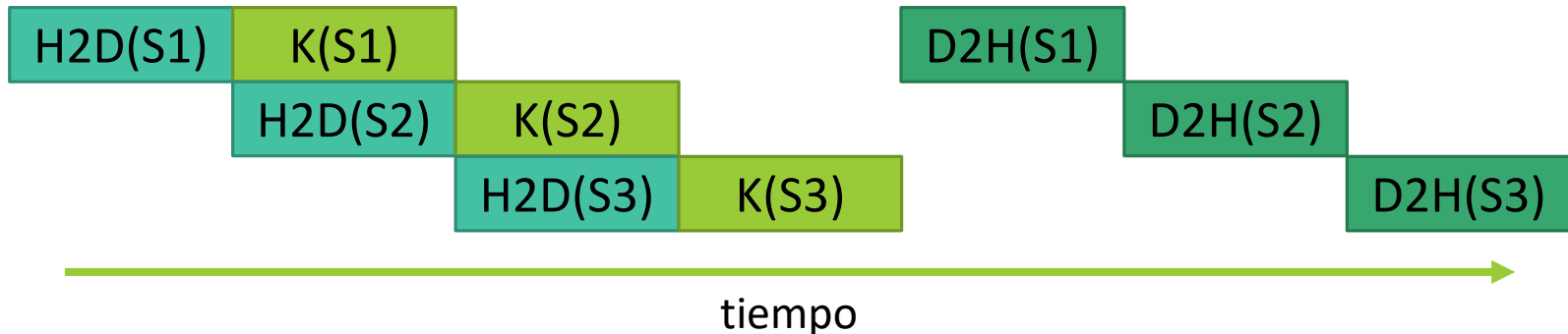
```
kernel75<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...); // 75% de los recursos  
kernel25<<<grid_size, block_size, 0, stream3>>>(..., dev3, ...); // 25% de los recursos  
kernel80<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...); // 80% de los recursos
```



Ejemplos

Bloqueo por kernels secuenciales:

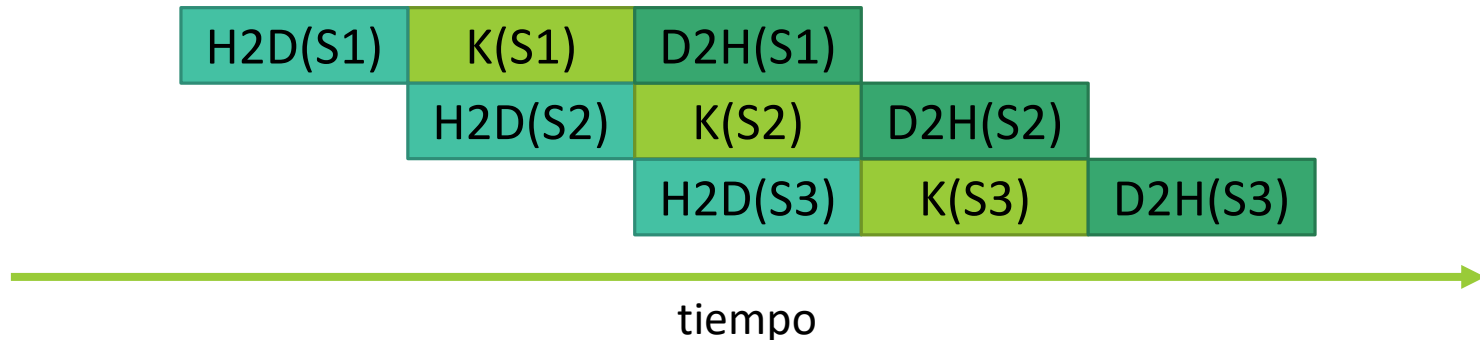
```
cudaMemcpyAsync(dev1, host1, size, cudaMemcpyHostToDevice, stream1);
cudaMemcpyAsync(dev2, host2, size, cudaMemcpyHostToDevice, stream2);
cudaMemcpyAsync(dev3, host3, size, cudaMemcpyHostToDevice, stream3);
kernel<<grid_size, block_size, 0, stream1>>>(..., dev1, ...);
kernel<<grid_size, block_size, 0, stream2>>>(..., dev2, ...);
kernel<<grid_size, block_size, 0, stream3>>>(..., dev3, ...);
cudaMemcpyAsync(host1, dev1, size, cudaMemcpyDeviceToHost, stream1);
cudaMemcpyAsync(host2, dev2, size, cudaMemcpyDeviceToHost, stream2);
cudaMemcpyAsync(host3, dev3, size, cudaMemcpyDeviceToHost, stream3);
```



Ejemplos

Bloqueo por kernels secuenciales:

```
cudaMemcpyAsync(dev1, host1, size, cudaMemcpyHostToDevice, stream1);  
kernel<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...);  
cudaMemcpyAsync(host1, dev1, size, cudaMemcpyDeviceToHost, stream1);  
cudaMemcpyAsync(dev2, host2, size, cudaMemcpyHostToDevice, stream2);  
kernel<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...);  
cudaMemcpyAsync(host2, dev2, size, cudaMemcpyDeviceToHost, stream2);  
cudaMemcpyAsync(dev3, host3, size, cudaMemcpyHostToDevice, stream3);  
kernel<<<grid_size, block_size, 0, stream3>>>(..., dev3, ...);  
cudaMemcpyAsync(host3, dev3, size, cudaMemcpyDeviceToHost, stream3);
```



Ejemplos

Pero hay que tener cuidado. ¿Qué pasa si solo tenemos un copy engine?

```
cudaMemcpyAsync(dev1, host1, size, cudaMemcpyHostToDevice, stream1);  
kernel<<<grid_size, block_size, 0, stream1>>>(..., dev1, ...);  
cudaMemcpyAsync(host1, dev1, size, cudaMemcpyDeviceToHost, stream1);  
cudaMemcpyAsync(dev2, host2, size, cudaMemcpyHostToDevice, stream2);  
kernel<<<grid_size, block_size, 0, stream2>>>(..., dev2, ...);  
cudaMemcpyAsync(host2, dev2, size, cudaMemcpyDeviceToHost, stream2);  
cudaMemcpyAsync(dev3, host3, size, cudaMemcpyHostToDevice, stream3);  
kernel<<<grid_size, block_size, 0, stream3>>>(..., dev3, ...);  
cudaMemcpyAsync(host3, dev3, size, cudaMemcpyDeviceToHost, stream3);
```

