Ejecución de kernels

Streams

- Un *stream* es una secuencia de comandos para el GPU
- Normalmente los *kernels* se ejecutan en el *default stream* (número 0)
- Se puede especificar el *stream* con el cuarto argumento para lanzar el *kernel*:
  - kernel<<< grid_size, block_size, shared_memory, stream >>>

Streams

```
cudaStream_t stream;
cudaStreamCreate(&stream);
foo_kernel<<< grid_size, block_size, 0, stream >>>();
cudaStreamDestroy(stream);
```

Ejemplo: cuda_default_stream.cu

→ se puede analizar los *streams* en NVVP

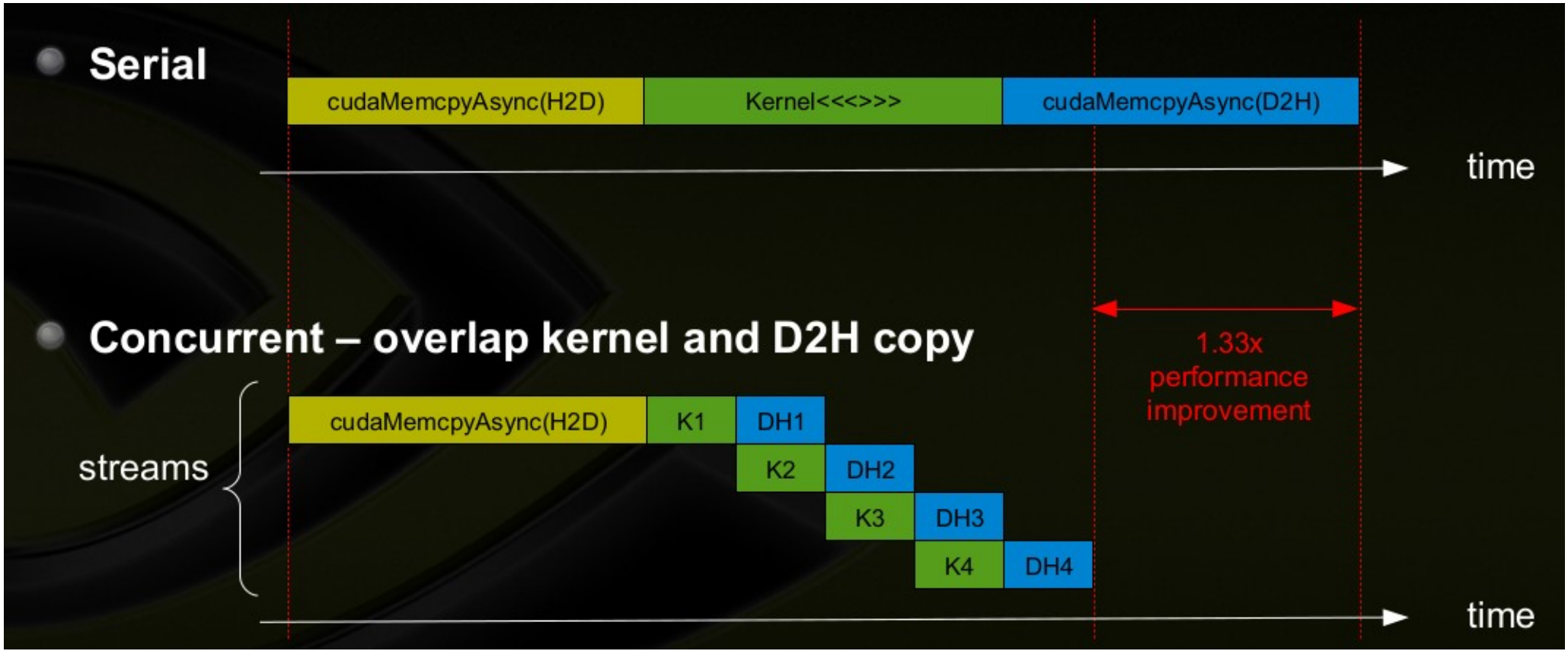
Streams

- Ejemplo: cuda_multi_stream.cu
  - Los *kernels* se ejecutan en una forma asincrónica con el *host*
  - Las operaciones de CUDA en *streams* diferentes son independientes
- Ejemplo: cuda_multi_stream_with_sync.cu
  - Se puede sincronizar los *streams* con: cudaStreamSynchronize(stream)
  - Esta función obliga que el *host* espere hasta que el *stream* termina.
- Ejemplo: cuda_multi_stream_with_default.cu
  - Todos los otros *streams* son sincrónicos con el *default stream*
  - Para tener *streams* operando en paralelo, mejor no usar el *default*

Streams: pipelining

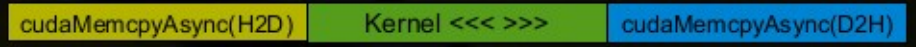
- Una aplicación de los *streams:*
  - operaciones de transferencia de datos en unos *streams* coinciden con computos en otros
  - otra manera de “esconder” el *latency*

Streams: pipelining



Streams: pipelining

- **Serial** (1x)

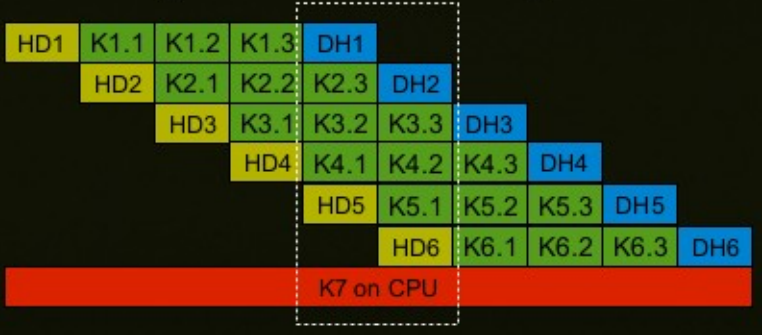


- **2-way concurrency** (up to 2x)

- **3-way concurrency** (up to 3x)

- **4-way concurrency** (3x+)

- **4+ way concurrency**



Streams: pipelining

- 3 requirimientos:
  - Memoria en el *host* debe ser “pinned”, cudaMallocHost()
  - Transferencia de datos debe ser asincrónica, cudaMemcpyAsync()
  - Las operaciones de CUDA deben estar en *streams* diferentes (y nada en el *default stream*)
- Ejemplo: cuda_pipelining.cu
  - programa escrito en C++, ocupa *object-oriented programming* para organizar la ejecución de los kernels

CUDA Callback

- Una función de *callback* es una función llamada por el *host* en algún momento durante la ejecución del stream
- Es útil para obtener información sobre el estatus de un *stream*
- Ejemplo: cuda_callback.cu
  - Para obtener información acerca del tiempo de ejecución de cada *stream*
- Según la documentación de CUDA:
  - el uso de *callbacks* es obsoleto
  - hay una alternativa que es cudaLaunchHostFunc que agrega una función del *host* en la “fila” del *stream.*

Stream priority

- Se puede asociar una prioridad a los *streams*
- Ejemplo: prioritized_cuda_stream.cu
  - cudaDeviceGetStreamPriorityRange()
  - cudaStreamCreateWithPriority()
    - *stream* nuevo es sincrónico con *default*: cudaStreamDefault
    - NO es sincrónico: cudaStreamNonBlocking

CUDA events

- Para grabar eventos en el lado del *device* (GPU) en el *stream*
- Ejemplo: cuda_event.cu
  - para determinar el tiempo de ejecución del *kernel*
  - ejemplo con *callback* tiene la desventaja de que el *callback* está llamado en el *host*, no el *device*
- Ejemplo: cuda_event_with_streams.cu
  - similar, pero con varios *streams*
  - se puede sincronizar los *streams* con *events*

Synchronization

- Sincronizar todo:
  - cudaDeviceSynchronize()
  - bloquea *host* hasta que todas las instrucciones de CUDA terminen.
- Sincronizar con respecto a un *stream*:
  - cudaStreamSynchronize(stream)
  - bloquea *host* hasta que todas las instrucciones de CUDA en el *stream* terminen
- Sincronizar con eventos
  - crear eventos dentro de los *streams* para sincronizar
  - cudaEventRecord(event, stream)
  - cudaEventSynchronize(event)
  - cudaStreamWaitEvent(stream, event)
  - cudaEventQuery(event)

CUDA Dynamic Parallelism

- Lanzar *kernels* dentro de un *kernel*
  - Permite el uso de *child grids*
  - Algoritmos recursivos
  - *Grids* adaptativos (simulaciones!)
- Ejemplo:
  - dynamic_parallelism.cu
  - recursion.cu

CUDA/OpenMP

- Se puede combinar CUDA con OpenMP:
  - lanzar *kernels* de distintos *threads* de OpenMP
- Ejemplo: openmp.cu
  - cada *stream* corresponde a un *thread* de OpenMP
- Ejemplo: openmp_default_stream.cu
  - cada *thread* de OpenMP lanza el *default stream*

MPS

- Se puede ejecutar *kernels* de procesos distintos
- Dado la manera en que los procesos interactuan con el GPU en la práctica los kernels ejecutan en una forma secuencial
- Con el modo de *Multi-Process Service* (MPS) se puede tener ejecución simultanea de los kernels de procesos distintos
  - así podemos combinar MPI con CUDA
- MPS está disponible solamente en Linux

MPS

- Funciona como un *daemon*: un programa que corre en el *background*
  - todos los procesos mandan sus comandos al daemon de MPS
  - MPS manda los comandos de CUDA al GPU
  - Para el GPU, hay solamente un proceso ocupando sus recursos (MPS)

MPS

- Ejemplo: simpleMPI.cu
  - modificación del programa con OpenMP
  - cada proceso de MPI lanza varios *threads* de OpenMP, y cada *thread* lanza un *kernel* en el GPU!
- Primero, sin MPS...

MPS

- Ahora inicializamos MPS:
  - export CUDA_VISIBLE_DEVICES=0
  - sudo nvidia-smi -i 0 -c 3
  - sudo nvidia-cuda-mps-control -d
- Programa debería (en principio) ejecutar más rápido ya que puede aprovechar de uso concurrente del GPU en cada proceso de MPI

MPS

- Deactivamos MPS con:
  - echo “quit” | sudo nvidia-cuda-mps-control
  - sudo nvidia-smi -i 0 -c 0

Kernel execution overhead

- Ejemplo: cuda_kernel.cu
- Usamos operación de SAXPY

  1) llamar un *kernel* dentro de un ciclo (simple_saxpy_kernel)

  2) poner el ciclo dentro del *kernel* (iterative_saxpy_kernel)

  3) un *kernel* recursivo (recursive_saxpy_kernel)

Kernel execution overhead

- Hay *overhead* para ejecutar un *kernel* dentro de un loop (asignación de recursos, etc.)
- También para recursión (típicamente más)
- El caso de tener el ciclo dentro del *kernel* es lo más rápido