Tema Seminario

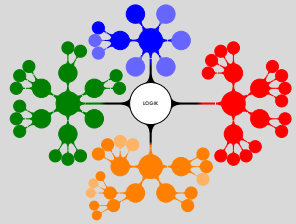
Programación manycore: streams, concurrencia y CUDA

Sistemas Concurrentes y Distribuidos (SCD)- Doble Grado Ingeniería Informática Matemáticas

Asignatura *de GIIM (3er curso)*

Fecha 28 septiembre 2023

Manuel I. Capel
manuelcapel@ugr.es
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada



Introducción

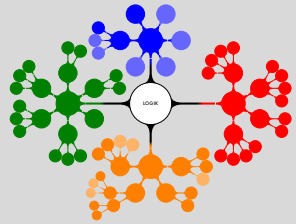
Bibliografía

Concurrencia programando con CUDA

Además del paralelismo multi-hebra de la CPU, se pueden realizar múltiples operaciones CUDA simultáneamente

1. CUDA Kernels `<<< ... >>>`
2. `cudaMemcpyAsync(H2D)` //Host To Device
3. `cudaMemcpyAsync(D2H)` //Device To Host

Manuel I. Capel
manuelcapel@ugr.es



Introducción
Bibliografía

Programación con Streams

Stream:

Una secuencia de operaciones que se ejecutan en la GPU en orden de emisión

Modelo de programación utilizado para llevar a cabo la Concurrencia

- Operaciones CUDA asignadas a diferentes *streams* pueden ejecutarse concurrentemente
- Operaciones CUDA emitidas desde distintos *streams* pueden ejecutarse entrelazadas

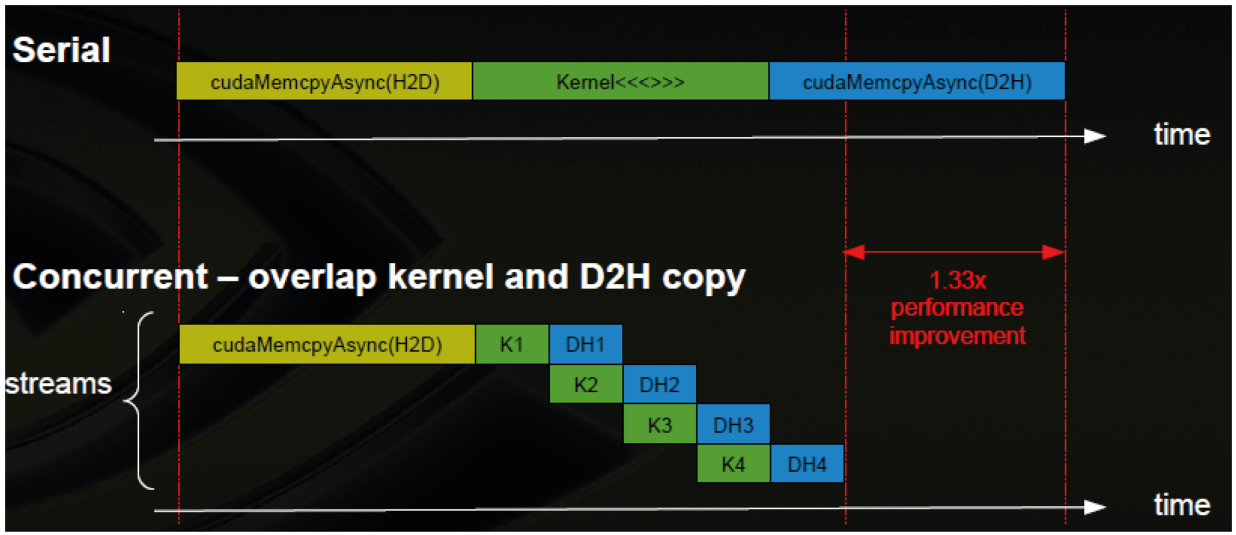
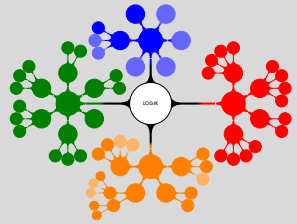


Figure: Mejora del rendimiento usando varios streams

Manuel I. Capel
manuelcapel@ugr.es



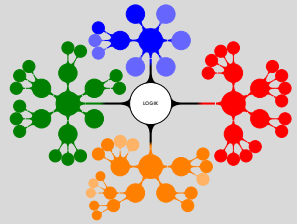
Introducción
Bibliografía

Streams y emisión de instrucciones

La concurrencia efectiva depende también del orden de emisión y del número de instrucciones paralelizables en el código

Figure: Paralelización de código y streams

Manuel I. Capel
manuelcapel@ugr.es



Introducción

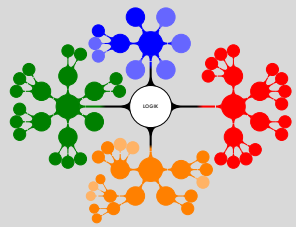
Bibliografía

Default Stream

`aka Stream '0'` se usa cuando no se declara ningún stream en el programa, como consecuencias:

- Ejecución del código completamente síncrona con respecto a la CPU y a la GPU
- Equivalente a un código que tuviera programada la instrucción `cudaDeviceSynchronize()` antes y después de cada instrucción
- Excepcionalmente se pueden ejecutar algunas operaciones asíncronamente en la CPU:
  - Lanzamientos de kernel en el stream por defecto
  - `cudaMemcpy*Async`
  - `cudaMemset*Async`
  - `cudaMemcpy` en la misma GPU
  - Copia `cudaMemcpy` H2D (“Host To Device") hasta un determinado tamaño de KB

Manuel I. Capel
manuelcapel@ugr.es



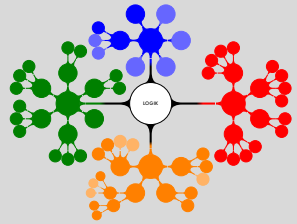
Introducción
Bibliografía

Requisitos para la concurrencia

programando con CUDA/GPU se ha de cumplir:

- Las operaciones CUDA se han haber asignado a distintos streams, que no sean el `Stream 0`
- Utilizar memoria de *paginas coloreadas* en CPU para ejecutar `cudaMemcpy*Async`:
  - Las páginas de memoria se encuentran bloqueadas
  - Se asignan con: `cudaMallocHost` o `cudaHostAlloc`
- Se ha de disponer de suficientes recursos:
  - Se hayan emitido operaciones `cudaMemcpy*Async` de distinto sentido
  - Suficientes recursos (SMEM, registros, bloques, etc.)

Manuel I. Capel
manuelcapel@ugr.es



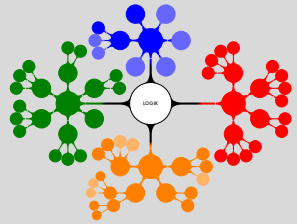
Introducción
Bibliografía

Ejemplo de código totalmente síncrono

- Se ha de evitar programar códigos similares al siguiente porque todas las operaciones que se emitan del stream por defecto serán síncronas (no hay concurrencia)

```
cudaMalloc(&dev1, tamaño);
double* host1= (double*)malloc(&host1,tamaño);
...
cudaMemcpy(dev1,host1,tamaño,H2D);
kernel2 <<<grid,bloque,0>>>(...,dev2,...);
kernel3 <<<grid,bloque,0>>>(...,dev3,...);
...
```


Manuel I. Capel
manuelcapel@ugr.es



Introducción
Bibliografía

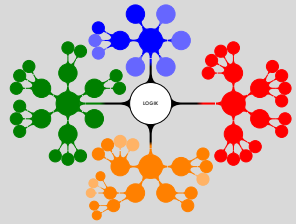
Ejemplo de código asíncrono con streams

- Totalmente asíncrono y, por tanto, ejecución concurrente de instrucciones CUDA
- Los datos utilizados por instrucciones concurrentes deben ser independientes

```
cudaStream_t stream1, stream2, stream3, stream4;
...
cudaMalloc(&dev1, tamaño);
cudaMallocHost(&host1,tamaño);
...
cudaMemcpyAsync(dev1,host1,tamaño,H2D,stream1);
kernel2 <<<grid,bloque,0,stream2>>>(...,dev2,...);
kernel3 <<<grid,bloque,0,stream3>>>(...,dev3,...);
cudaMemcpyAsync(host4,dev4,tamaño,D2H,stream4);

...
```


Manuel I. Capel
manuelcapel@ugr.es



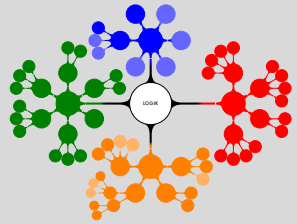
Introducción
Bibliografía

Sincronización Explícita

Instrucciones de sincronización en CUDA:

- Sincronizarlo “todo" :
  - `cudaDeviceSynchronize()`
  - Se bloquea el código en la CPU hasta que todas llamadas CUDA emitidas se hayan completado
- Sincronizarse con un stream específico:
  - `cudaStreamSynchronize(streamId)`
  - Se bloquea el código en la CPU hasta que todas las llamadas CUDA asignadas a `streamId` se hayan completado
- Sincronizar utilizando eventos:
  - Se crean eventos específicos, dentro de los streams, para utilizarlos como un vehículo de sincronización
  - `cudaEventRecord(evento, streamId)`
  - `cudaEventSynchronize(evento)`
  - `cudaStreamWaitEvent(streamId,evento)`
  - `cudaEventQuery(evento)`

Manuel I. Capel
manuelcapel@ugr.es



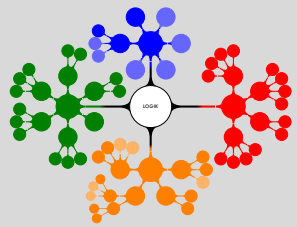
Introducción
Bibliografía

Sincronización Implícita

Estas operaciones sincronizan implícitamente a todas las demás operaciones CUDA:

- Asignación de páginas de memoria *coloreadas* (page-locked memory)
  - `cudaMallocHost`
  - `cudaHostAlloc`
- Asignación de memoria al *dispositivo* (código ejecutándose en la GPU)
  - `cudaMalloc`
- Versión no asíncrona de las operaciones de memoria:
  - `cudaMemcpy*`(sin sufijo `Async`)
  - `cudaMemset*`(sin sufijo `Async`)
- Cambio a la configuración de memoria `L1/shared)` `(acceso al nivel L1 de cache de la CPU)`
  - `cudaDeviceSetCacheConfig`

Manuel I. Capel
manuelcapel@ugr.es



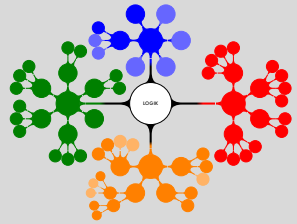
Introducción
Bibliografía

Planificación de los streams en una GPU

Depende de la arquitectura de la GPU, pero al menos se cuenta con:

- 3 colas /GPU
  - cola “Execution Engine" (EE)
  - 2 colas “Copy Engine" (CE) , una por cada sentido de la copia: H2D, D2H
- Las operaciones CUDA se emiten a la GPU en el orden en que se emitieron en el código de un núcleo lanzado:
  - Se ubican en la cola (EE,CE) correspondiente
  - Se mantienen las dependencias referidas a streams mientras estén ubicadas en las colas de ejecución, pero tales dependencias se pierden si las operaciones se ubican en la EE
- Una operación CUDA se despacha de la cola EE si:
  - Las llamadas que le preceden para el mismo stream se han completado
  - Las llamadas que le preceden en la cola de ejecución se han completado
  - Hay recursos disponibles para atenderla

Manuel I. Capel
manuelcapel@ugr.es



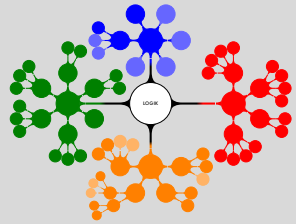
Introducción
Bibliografía

Planificación de los streams en una GPU-II

Depende de la arquitectura de la GPU, pero al menos se cuenta con:

- Los kernels CUDA se pueden ejecutar concurrentemente si se asignan a diferentes streams
  - Los bloques de hebras de un núcleo determinado se despachan en una cola si se han despachado todos los bloques de hebras de los núcleos precedentes y aún quedan recursos SM disponibles.
- Una operación bloqueada bloquea a todas operaciones de la misma cola, incluso si se refieren a otros streams

Manuel I. Capel
manuelcapel@ugr.es

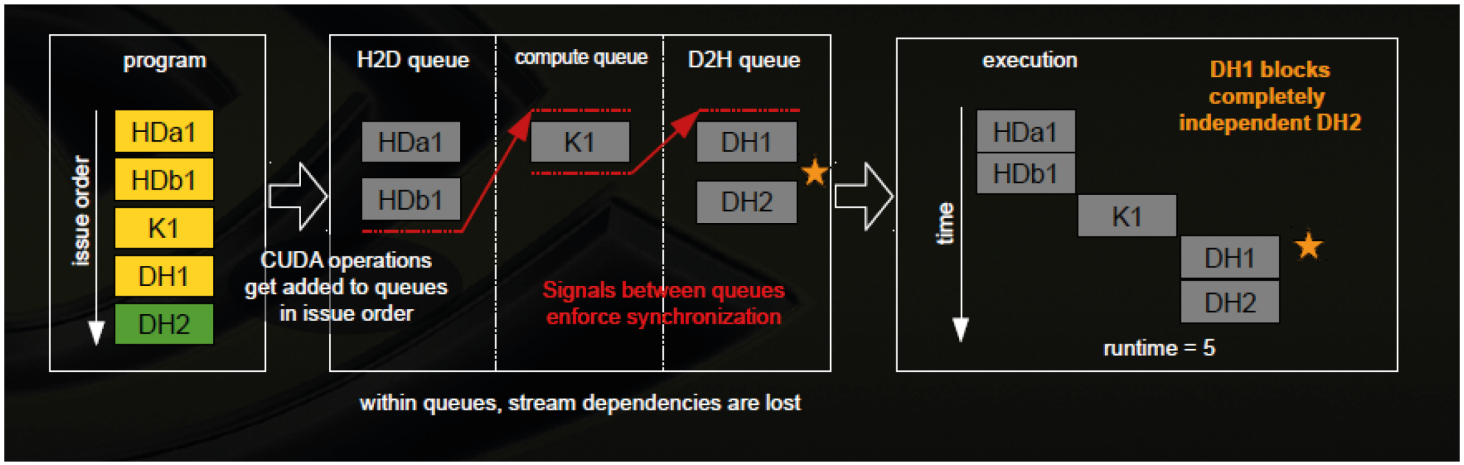


Introducción
Bibliografía

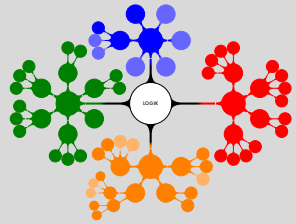
Ejemplo: cola bloqueada

2 streams, el stream 1 es emitido primero

- `Stream1: HDa1, HDb1, K1, DH1` (estas operaciones se emiten primero)
- `Stream2: DH2` (completamente independiente del stream1)



Manuel I. Capel
manuelcapel@ugr.es

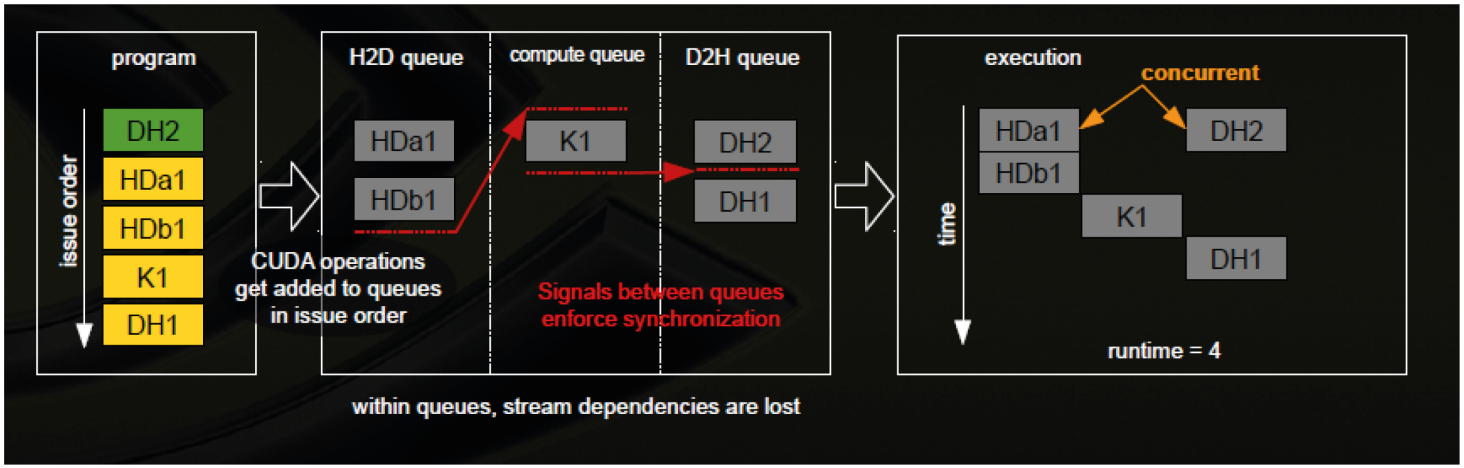


Introducción
Bibliografía

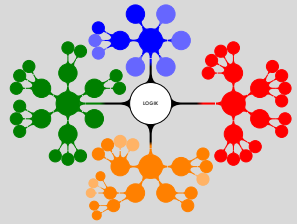
Ejemplo: cola bloqueada-II

2 streams, el stream 2 es emitido primero

- `Stream1: HDa1, HDb1, K1, DH1`
- `Stream2: DH2` (completamente independiente del stream1, se emite primero ahora)



Manuel I. Capel
manuelcapel@ugr.es

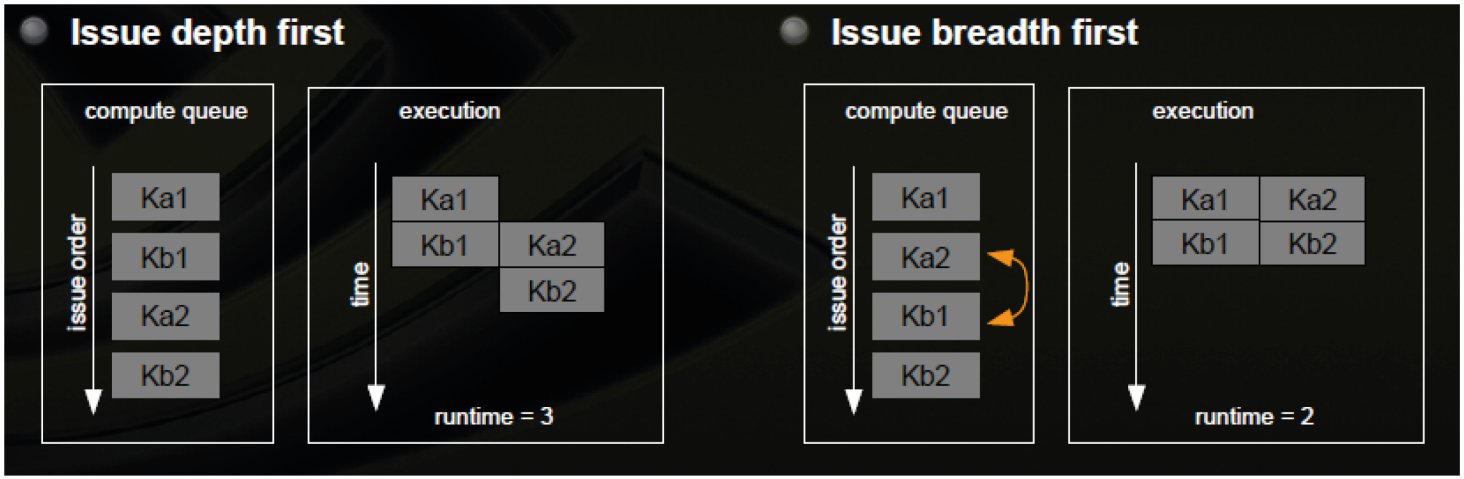


Introducción
Bibliografía

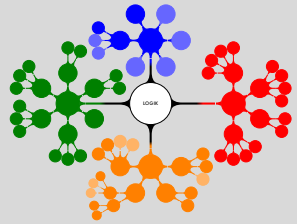
Ejemplo: kernel bloqueado

2 streams, que solo emiten kernels CUDA

- `Stream1: Ka1, Kb1`
- `Stream2: Ka2, Kb2`
- Los kernels son de tamaño similar y ocupan la mitad de un SM
- Hay que darse cuenta que el orden de emisión de los kernels afectará al tiempo de ejecución global



Manuel I. Capel
manuelcapel@ugr.es

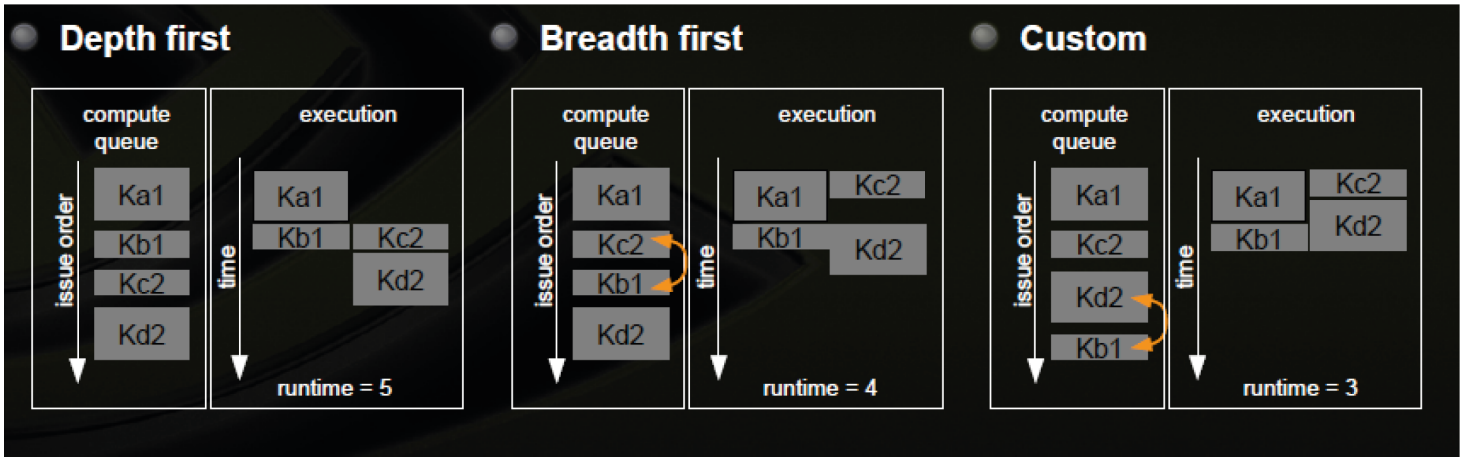


Introducción
Bibliografía

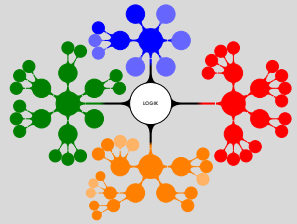
Ejemplo: kernel bloqueado-II

2 streams, que solo emiten kernels CUDA, pero ahora los kernels son de distinto tamaño

- `Stream1: Ka1{2}, Kb1{1}`
- `Stream2: Ka2{1}, Kb2{2}`
- Los kernels son de tamaño similar y ocupan la mitad de un SM
- Hay que darse cuenta que el orden de emisión de los kernels afectará al tiempo de ejecución global y también el tiempo {t} de ejecución de cada kernel



Manuel I. Capel
manuelcapel@ugr.es



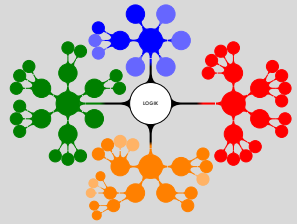
Introducción
Bibliografía

Planificación de kernels concurrentes

Planificación de los kernels concurrentes

- Se trata de kernels que programan y se lanzan dentro de otro *kernel-padre* (similar a un mecanismo *fork-join* de UNIX BSD)
- La planificación de los *kernels concurrentes* poseen características especiales
- Normalmente, se inserta una señal en las colas, después de la operación, para lanzar la siguiente operación en el mismo stream
- En la cola del motor de cálculo EE, para poder manejar kernel concurrentes, cuando los kernels regulares se emiten secuencialmente, se hace que la señal se retrase hasta después del último kernel secuencial
- Existen situaciones en que el retraso anterior de las señales pueden llegar a bloquear a otras colas

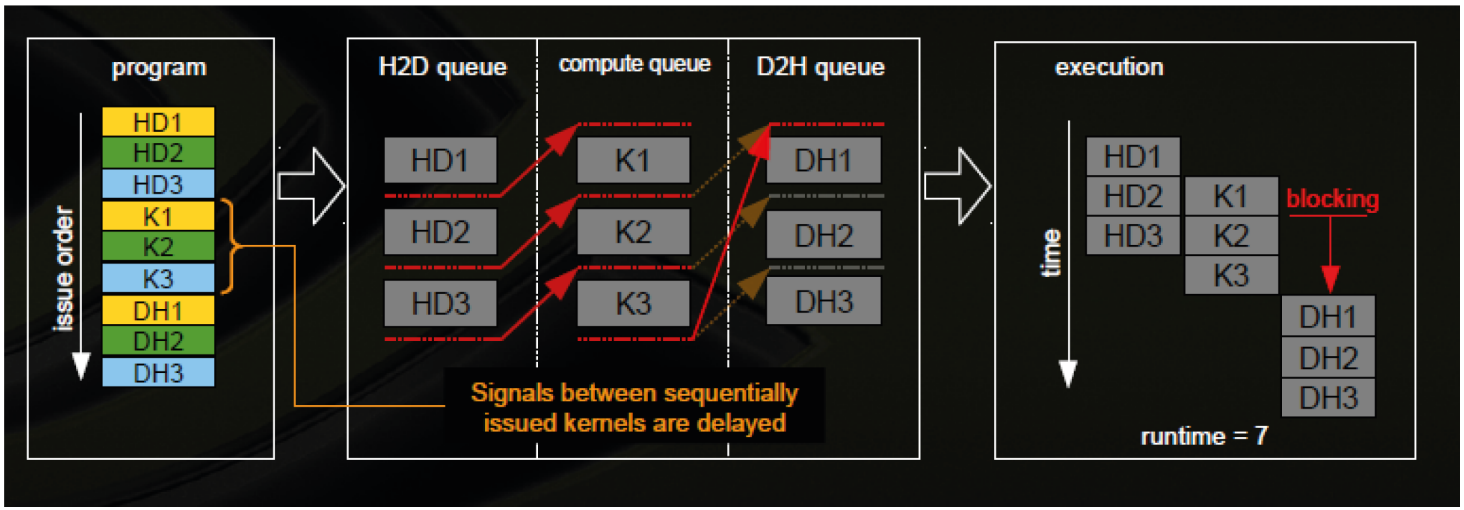
Manuel I. Capel
manuelcapel@ugr.es



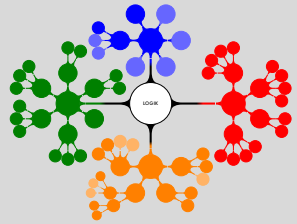
Introducción
Bibliografía

Kernels concurrentes

- 3 kernels, cada uno de ellos realiza las operaciones: HD (Host To Device), K, DH (Device To Host)
- Los kernels se lanzan siguiendo *primero en anchura*
  - Los kernels lanzados secuencialmente retrasan las señales y bloqueen en la operación `cudaMemcpy(D2H)`



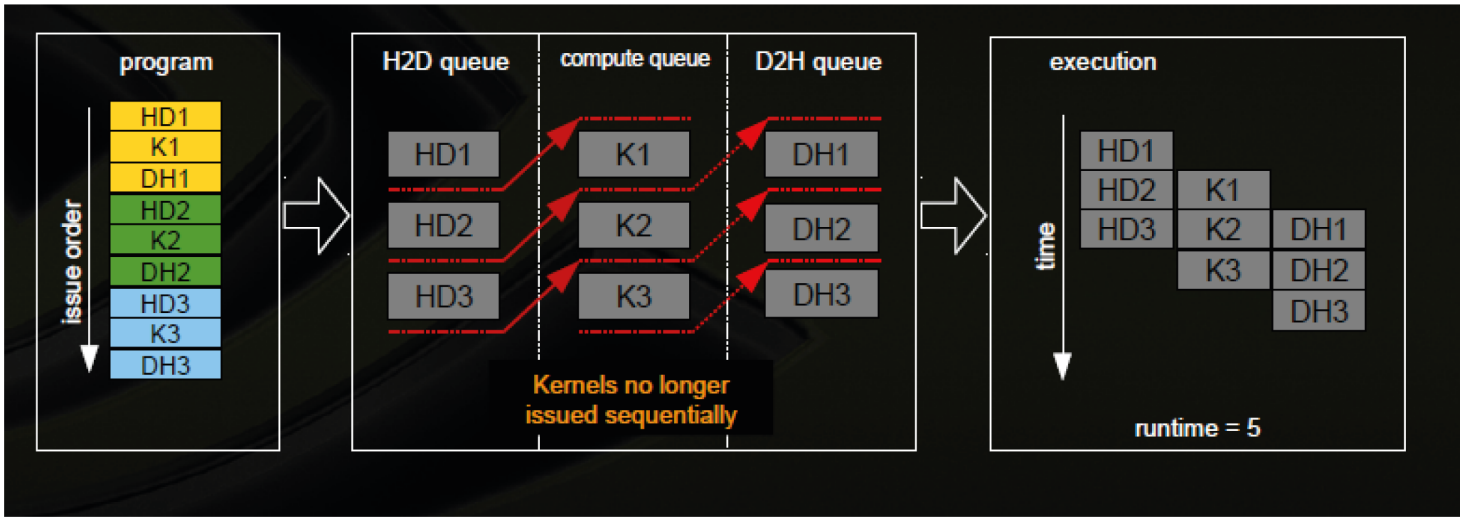
Manuel I. Capel
manuelcapel@ugr.es



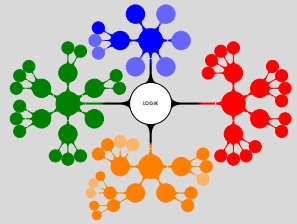
Introducción
Bibliografía

Kernels concurrentes-II

- 3 kernels, cada uno de ellos realiza las operaciones: HD (Host To Device), K, DH (Device To Host)
- Los kernels se lanzan siguiendo *primero en profundidad*



Manuel I. Capel
manuelcapel@ugr.es

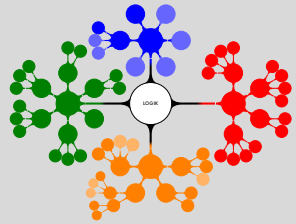


Introducción
Bibliografía

Bibliografía

- NVIDIA A100 Tensor Core GPU Architecture Unprecedented Acceleration at Every Scale (accedido 2023)
- Yurtsever, E., Lambert, J., Carballo, A., Takeda, K., 2020. A survey of autonomous driving: Common practices and emerging technologies. IEEE Access 8, 58443?58469. doi:10.1109/ACCESS.2020.2983149
- Amert, T., Otterness, N., Yang, M., Anderson, J.H., Smith, F.D., 2018. Gpu scheduling on the nvidia tx2: Hidden details revealed. Real-Time Systems Symposium January, 104?115. doi:10.1109/RTSS.2017.00017
- Singh, J., Olmedo, I.S., Capodieci, N., Marongiu, A., Caccamo, M., 2022. Reconciling QoS and concurrency in nvidia gpus via warp-level scheduling. 2022 Design, Automation and Test in Europe Conference and Exhibition , 1275?1280doi:10.23919/DATE54114.2022.9774761
- Calderón, A.J., Torres, C., Kosmidis, L., Fernando, C., Ramírez, N., Javier, F.,Almeida, C., 2022. Real-Time High-Performance Computing for Embedded Control Systems. doi:10.5821/dissertation-2117-371621
- Capodieci, N., Burgio, P., Cavicchioli, R., Olmedo, I.S., Solieri, M., Bertogna, M., 2022. Real-time requirements for adas platforms featuring shared memory hierarchies. IEEE Design and Test 39, 35?41. doi:10.1109/MDAT.2020.3013828

Manuel I. Capel
manuelcapel@ugr.es



Introducción
Bibliografía