

POLITECNICO
MILANO 1863

GPUs and Heterogeneous Systems
(programming models and architectures)

CUDA concurrency and streams

Introduction

- NVIDIA GPUs support two types of parallelism:
 - Data parallelism
 - Task parallelism
- Up to now we have investigated **data parallelism** by means of mechanisms for **kernel level concurrency**
 - Single kernel implementation is optimized w.r.t. HW mechanisms to minimize execution time

But ...

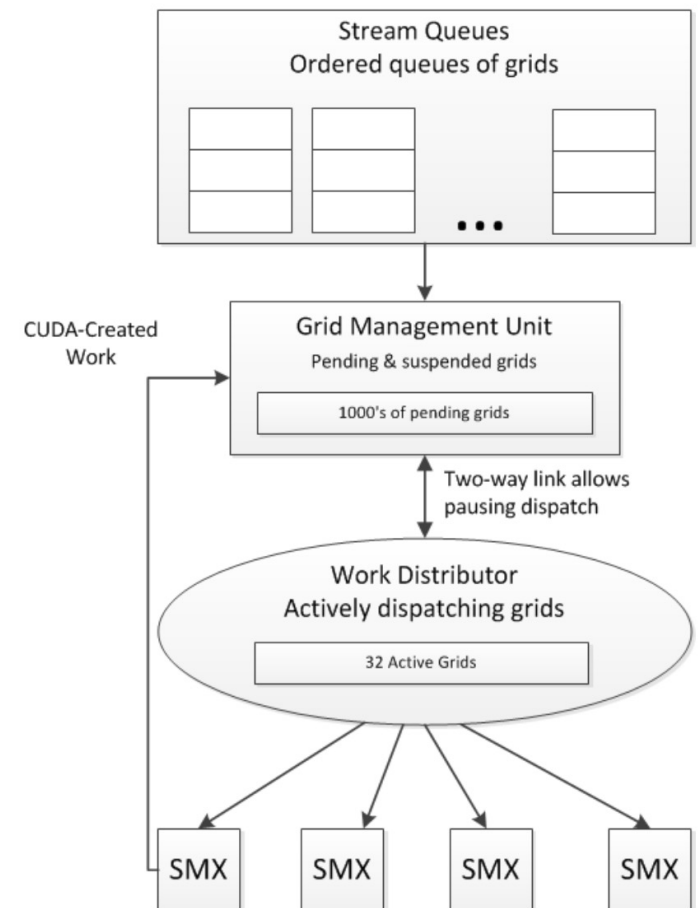
- Data transfers and kernels are executed sequentially

Recap on the execution model

- The host program used a **single stream** to submit CUDA commands
 - The GPU executes commands in the **FIFO order**
 - Commands are executed strictly in a **sequence without overlapping**
- From the host side, CUDA commands may be:
 - **Blocking**
 - The host program is blocked until the function returns
 - E.g.: `cudaMemcpy`
 - **Non-blocking**
 - The host program continues asynchronously its execution
 - E.g.: `kernel launch`

GPU mechanisms for task parallelism

- NVIDIA GPUs (\geq Kepler) provide mechanisms for the concurrent executions of different tasks
 - **Multiple HW queues** where **tasks** can be placed by:
 - The **program** on the **host**
 - The **kernel** on the **device**
 - A **grid management unit** performing the **scheduling** of tasks
 - Thus, enabling **grid level concurrency**
- **CPU and GPU work in parallel** as well!



CUDA mechanisms for task parallelism

- On top of GPU architecture, CUDA provides:
 - **Multiple streams**
 - **Asynchronous** (i.e., non-blocking) **memory transfer** commands
- Such mechanisms enable **task parallelism** by overlapping:
 - Host computation
 - Device computation
 - Host-device (device-host) data transfer
- Execution of multiple kernels may be overlapped on both the host and device sides
- Do remember that concurrency is at a logical point of view
 - **Actual concurrent execution depends on the available HW resources**

CPU/GPU concurrent execution

- CPU tasks can be executed concurrently to GPU kernels

```
cudaMemcpy(..., cudaMemcpyHostToDevice);  
foo<<<blocks, threads>>>();  
cpuFoo();  
cudaMemcpy(..., cudaMemcpyDeviceToHost);
```

Case 1: CPU task is longer than GPU one



Case 2: GPU task is longer than CPU one



Memory transfers
block the host

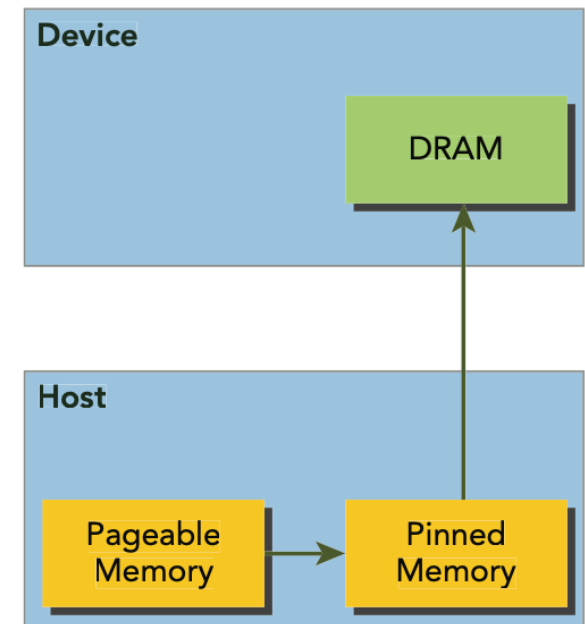
We have a single thread in
the host submitting
commands and executing
CPU tasks

■ Command submit
■ Task
■ Memory transfer

Synchronous vs. asynchronous memory data transfer

- Host memory is managed by the OS based on **virtual memory mechanism**:
 - Allocated host memory is **pageable**
 - Data can be **paged** in and out of the main memory by the OS
 - GPU cannot safely access **pageable memory**
- To transfer data to the device memory, **CUDA driver**
 1. **Allocates page-locked (pinned) host memory**
 2. **Copies data to the pinned memory**
 3. **Transmits data from the pinned memory to the device one**

Pageable Data Transfer

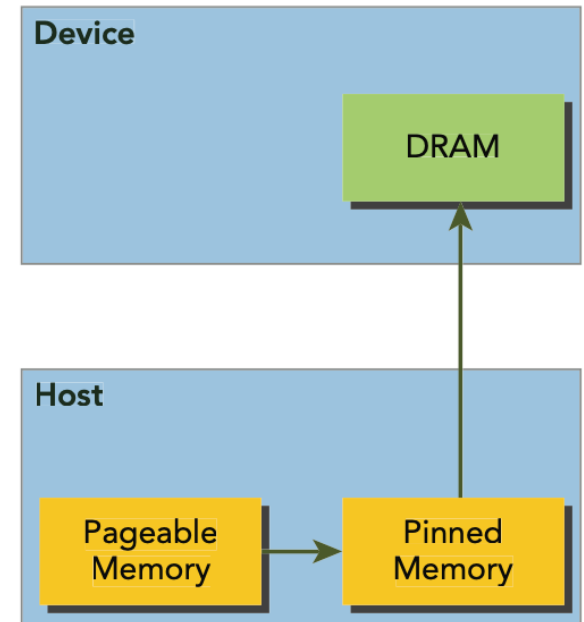


Synchronous vs. asynchronous memory data transfer

- **CUDA supports only blocking (synchronous) data transfer API functions on pageable memory!**

```
/*...*/  
int main(){  
    int *h_va, *d_va;  
    h_va = malloc(N*sizeof(int));  
    cudaMalloc(&d_va, N*sizeof(int));  
    /*...*/  
    cudaMemcpy(d_va, h_va, N*sizeof(int),  
               cudaMemcpyHostToDevice);  
  
    /*...*/  
    free(h_va);  
    cudaFree(d_va);  
}
```

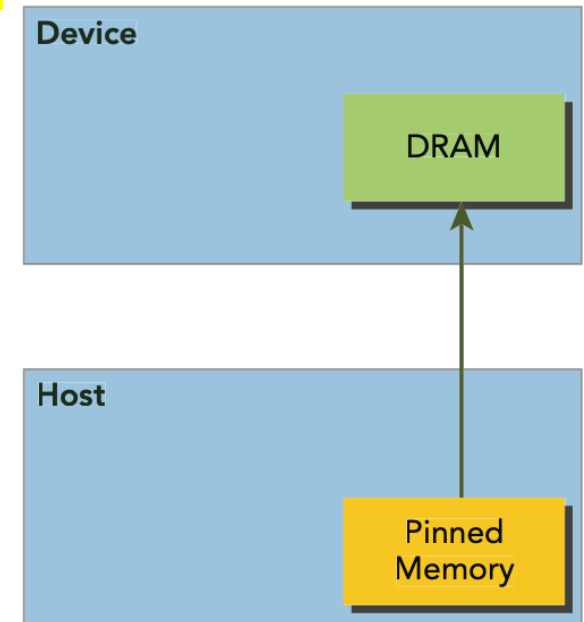
Pageable Data Transfer



Synchronous vs. asynchronous memory data transfer

- **Pinned memory** can be explicitly allocated in the code
 - To avoid such a “two-hop” process
 - To allow the usage of **asynchronous data transfer** API functions
- **Do note:** an excessive allocation of pinned memory might degrade host system performance
 - It reduces the amount of pageable memory for virtual memory data

Pinned Data Transfer



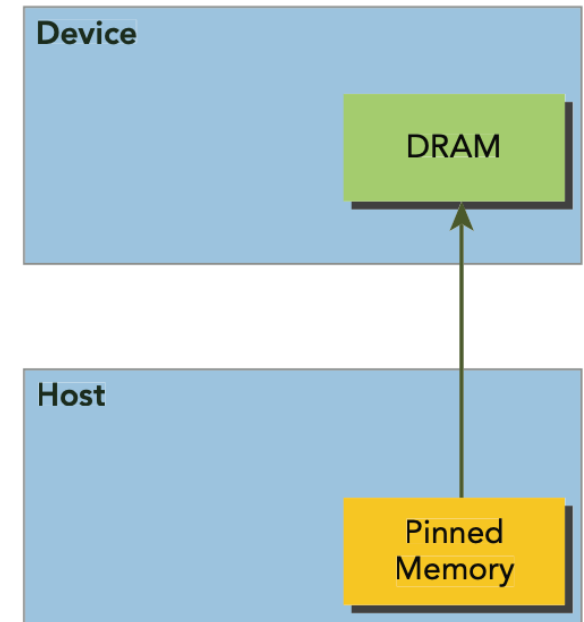
Synchronous vs. asynchronous memory data transfer

- How to transfer asynchronously data to/from pinned memory:

```
/*...*/
int main(){
    int *h_va, *d_va;
    cudaMallocHost(&h_va, N*sizeof(int));
    cudaMalloc(&d_va, N*sizeof(int));
    /*...*/
    cudaMemcpyAsync(d_va, h_va, N*sizeof(int),
                   cudaMemcpyHostToDevice);

    /*...*/
    cudaDeviceSynchronize();
    cudaFreeHost(h_va);
    cudaFree(d_va);
}
```

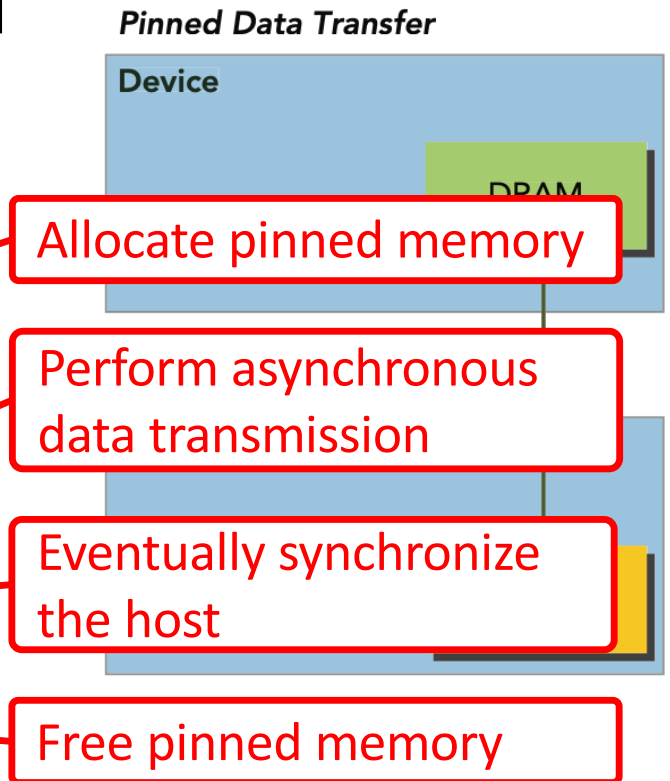
Pinned Data Transfer



Synchronous vs. asynchronous memory data transfer

- How to transfer asynchronously data to/from pinned memory:

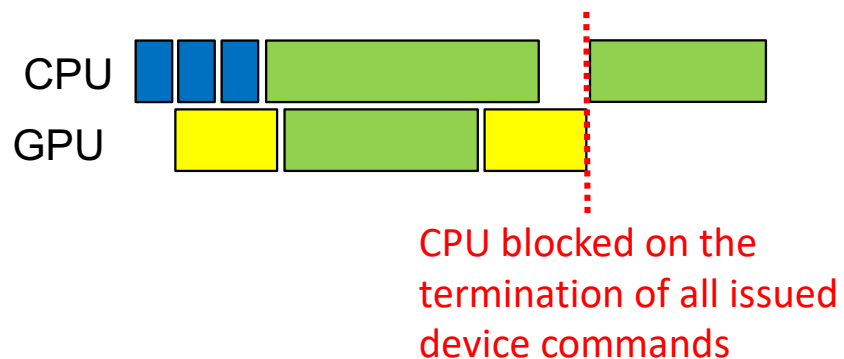
```
/*...*/  
int main(){  
    int *h_va, *d_va;  
    cudaMallocHost(&h_va, N*sizeof(int));  
    cudaMalloc(&d_va, N*sizeof(int));  
    /*...*/  
    cudaMemcpyAsync(d_va, h_va, N*sizeof(int),  
                   cudaMemcpyHostToDevice);  
  
    /*...*/  
    cudaDeviceSynchronize();  
    cudaFreeHost(h_va);  
    cudaFree(d_va);  
}
```



CPU/GPU concurrent execution

- Tasks can be executed concurrently to GPU kernels

```
cudaMemcpyAsync(..., cudaMemcpyHostToDevice);  
foo<<<blocks, threads>>>();  
cudaMemcpyAsync(..., cudaMemcpyDeviceToHost);  
cpuFoo();  
cudaDeviceSynchronize();  
doOther();
```



CUDA streams

- A **stream** is a queue of device work
 - The host pushes CUDA commands (memory transfers or kernel launches) in the stream
 - Device pops and executes commands from streams when resources are free
- Two types of streams:
 - **Default stream** (or NULL stream)
 - The implicit stream used in a program
 - **Non-default stream**
 - Streams explicitly declared and used by the programmer

Managing a CUDA stream

```
/*...*/

int main(){
    /*...*/
    float *h_A, *d_A;
    cudaMallocHost(&h_A, nbytes);
    cudaMalloc(&d_A, nbytes);

    cudaStream_t stream1;
    cudaStreamCreate(&stream1);

    /*...*/
}
```


Managing a CUDA stream

```
/*...*/
```

```
int main(){
```

```
/*...*/
```

```
float *h_A, *d_A;
```

```
cudaMallocHost(&h_A, nbytes);
```

```
cudaMalloc(&d_A, nbytes);
```

```
cudaStream_t stream1;
```

```
cudaStreamCreate(&stream1);
```

```
/*...*/
```

Pinned memory has to be allocated since asynchronous memory transfers are used with streams

Allocate device memory

Declare a stream variable

Create a stream variable

Managing a CUDA stream

```
/*...*/
cudaMemcpyAsync(d_A, h_A, nbytes,
                cudaMemcpyHostToDevice, stream1);

kernel<<<blocks, threads, 0, stream1>>>(d_A, d_B);

cudaMemcpyAsync(h_B, d_B, nbytes,
                cudaMemcpyDeviceToHost, stream1);

cudaStreamSynchronize(stream1);
/*...*/
```

Managing a CUDA stream

```
/*...*/  
cudaMemcpyAsync(d_A, h_A, nbytes,  
                cudaMemcpyHostToDevice, stream1);  
  
kernel<<<blocks, threads, 0, stream1>>>(d_A, d_B);  
  
cudaMemcpyAsync(h_B, d_B, nbytes,  
                cudaMemcpyDeviceToHost, stream1);  
  
cudaStreamSynchronize(stream1);  
/*...*/
```

- The stream is passed as argument to commands
- When the stream is not specified, the command is placed in the default stream!

Block the host waiting for termination of all commands in the stream

Managing a CUDA stream

```
/*...*/  
  
cudaStreamDestroy(stream1);  
  
cudaFreeHost(h_A);  
  
/*...*/  
}
```

Managing a CUDA stream

```
/*...*/
```

```
cudaStreamDestroy(stream1);
```

Destroy the stream



```
cudaFreeHost(h_A);
```

Pinned memory has to
be freed



```
/*...*/
```

```
}
```

Streams and concurrency

- Host side:
 - If command is blocking the host gets blocked until command termination
 - If command is non-blocking the host continues the execution immediately
- Device side:
 - Operations within the **same stream** are **executed in order (FIFO) and cannot overlap**
 - Operations in **different streams** are **unordered** and can **overlap**
- Multiple streams are used to enable task level parallelism
 - Default stream has special synchronization rules
 - Concurrent execution is actually possible if necessary HW resources are available, otherwise execution is serialized

Default stream

- Default stream is synchronous with all other streams
 - Commands in the default stream cannot overlap with other streams
 - Workflow:
 1. All commands in other streams called before the command in the default stream are executed
 2. The command in the default stream is executed
 3. Subsequent commands in other streams called after the command in the default stream are executed

Default stream

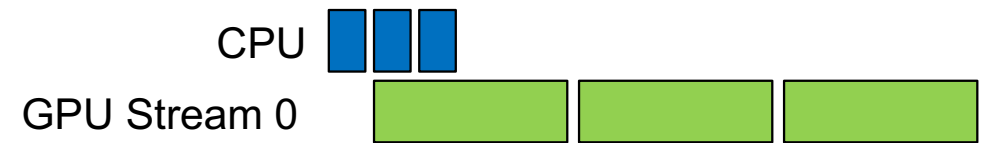
- There is a single default stream in the host program
 - CUDA<=7 – it is possible to declare a default stream per host thread
- Stream declared with non-blocking flag are not synchronous with the default stream

```
cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking);
```

Examples of kernel concurrency

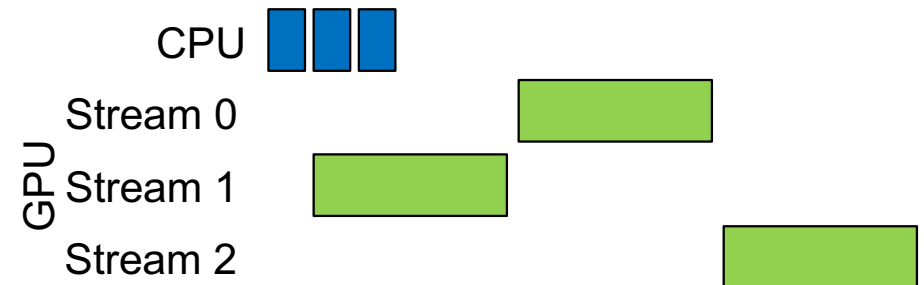
- Default stream:

```
foo<<<blocks, threads>>>();  
foo<<<blocks, threads>>>();  
foo<<<blocks, threads>>>();
```



- Default and user streams:

```
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
foo<<<blocks, threads, 0, stream1>>>();  
foo<<<blocks, threads>>>();  
foo<<<blocks, threads, 0, stream2>>>();
```



Assuming to have enough resources

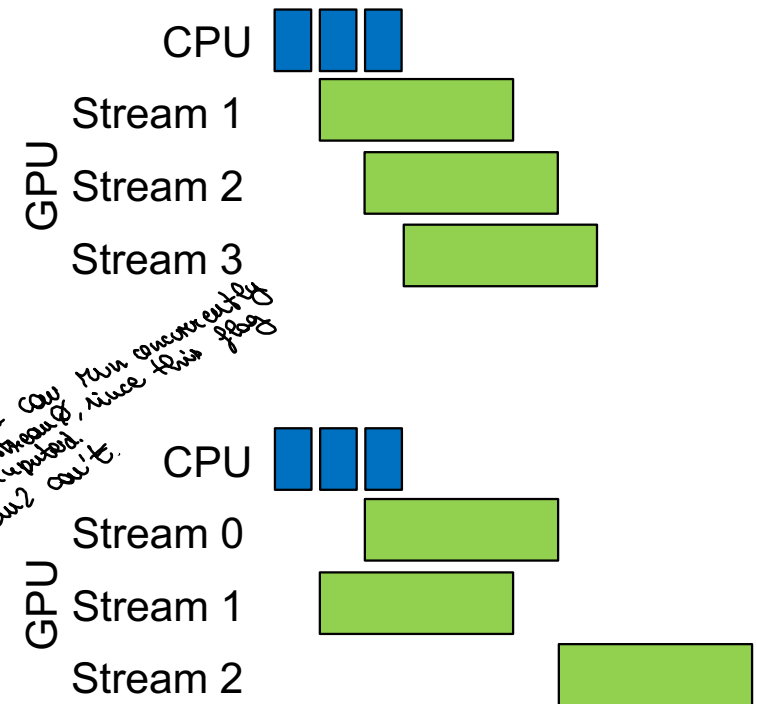
Examples of kernel concurrency

- User streams:

```
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
cudaStreamCreate(&stream3);  
foo<<<blocks, threads, 0, stream1>>>();  
foo<<<blocks, threads, 0, stream2>>>();  
foo<<<blocks, threads, 0, stream3>>>();
```

- Default and user streams:

```
cudaStreamCreateWithFlags(&stream1,  
                           cudaStreamNonBlocking);  
cudaStreamCreate(&stream2);  
foo<<<blocks, threads, 0, stream1>>>();  
foo<<<blocks, threads>>>();  
foo<<<blocks, threads, 0, stream2>>>();
```



Assuming to have enough resources

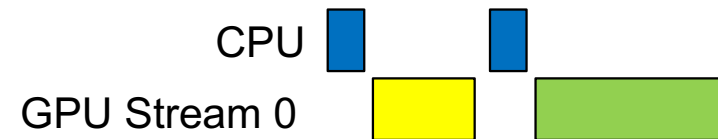
Kernel execution and memory transfer concurrency

- Memory copies can execute concurrently if (and only if):
 - Different non-default streams are used
 - The copy uses pinned memory and is performed by means of asynchronous API
- Resource limitations: PCIe can execute a single transfer per direction at a time
 - Multiple concurrent memory copies in the same direction are serialized

Examples of kernel and data transfer concurrency

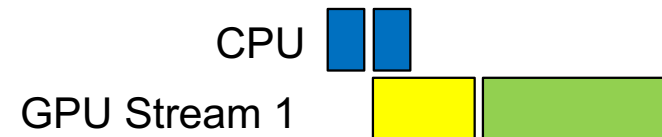
- Synchronous commands:**

```
cudaMemcpy(...);  
foo<<<...>>>();
```



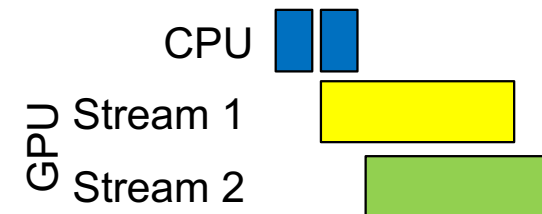
- Asynchronous commands in the same stream:**

```
cudaMemcpyAsync(..., stream1);  
foo<<<..., stream1>>>();
```



- Asynchronous commands in different streams:**

```
cudaMemcpyAsync(..., stream1);  
foo<<<..., stream2>>>();
```

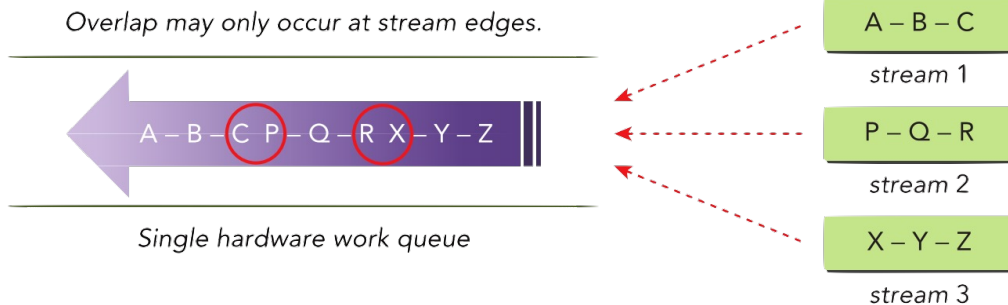


Assuming to have enough resources

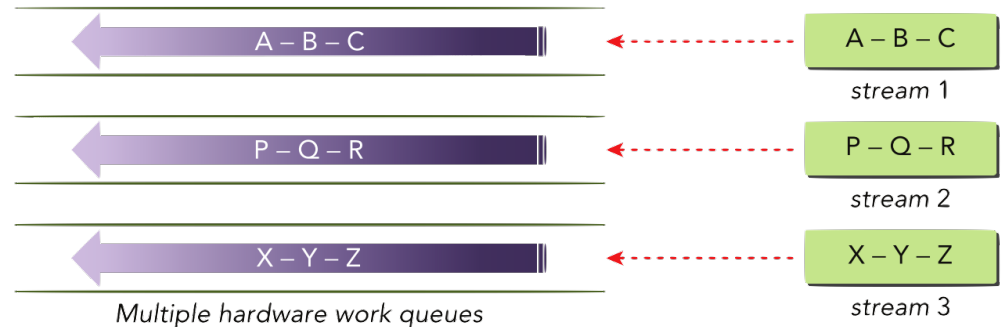
HW mechanisms to support multiple streams

- NVIDIA GPUs (\geq Kepler) are provided with multiple HW queues (32 ones) where streams are separately mapped
 - Fermi GPU was provided with a single HW queue causing false dependency issues
- `CUDA_DEVICE_MAX_CONNECTIONS` OS environmental variable can be set to vary the actual number of HW queues to be used

Fermi GPU



Kepler GPU



Host-device synchronization

- Host execution is implicitly synchronized with device one on the following CUDA function calls:
 - `cudaMallocHost()`
 - `cudaHostAlloc()`
 - `cudaMalloc()`
 - `cudaMemcpy*()` – **non** Async
 - `cudaMemset*()` – **non** Async
 - `cudaDeviceSetCacheConfig()`

Host-device synchronization

- Host execution can be explicitly synchronized with device one in different ways:
 - `cudaDeviceSynchronize()`
 - Blocks host until all issued CUDA commands are completed
 - `cudaStreamSynchronize()`
 - Blocks host until all issued CUDA commands in a specific stream are completed
 - Events
 - Allow fine-grained synchronization
 - Between CUDA commands on device, or
 - With the host

Events

- A **CUDA event is a marker** that can be pushed in the stream
 - It can be used to mark to a specific point in the sequence of submitted commands
 - Used for **synchronization** and for **profiling** (as already seen)
- Events have a **Boolean state**:
 - **Occurred**
 - When created
 - When reached in the stream (also annotated with the timestamp)
 - **Not occurred**
 - When pushed in the stream

Managing events

```
/*...*/  
int main() {  
    /*...*/  
    cudaEvent_t event1;  
    cudaEventCreate(&event1);  
  
    vsumKernel<<<N/256, 256>>>(d_va, d_vb, d_vc);  
    cudaEventRecord(event1);  
    cudaEventSynchronize(event1);  
    /*...*/  
    cudaEventDestroy(event1);  
}
```

Managing events

```
/*...*/  
int main() {  
    /*...*/  
    cudaEvent_t event1;  
    cudaEventCreate(&event1);  
  
    vsumKernel<<<N/256, 256>>>(d_va, d_vb, d_vc);  
    cudaEventRecord(event1);  
    cudaEventSynchronize(event1);  
    /*...*/  
    cudaEventDestroy(event1);  
}
```

Event variable

Create event

Push the event in the desired
synchronization point w.r.t. the
device

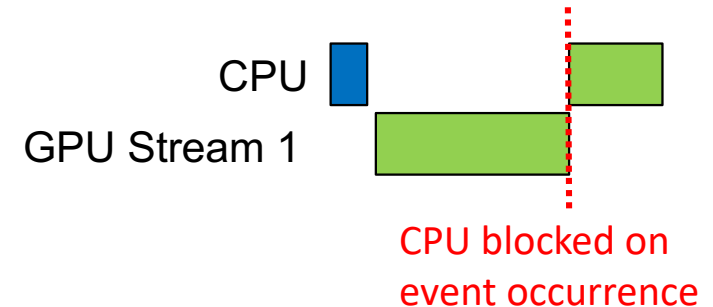
Enforce host synchronization
in the desired point

Destroy event

Examples of event usage

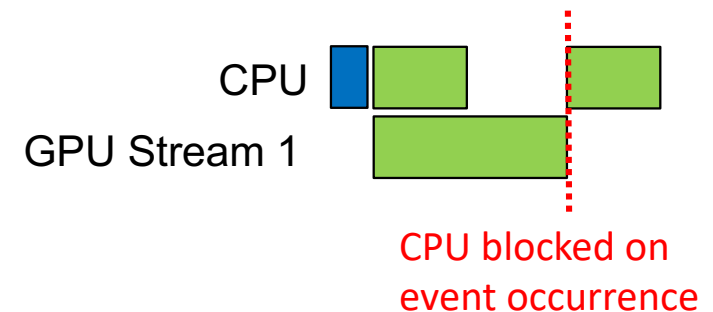
- Host-device synchronization:

```
foo<<<blocks, threads, 0, stream1>>>();  
cudaEventRecord(event1);  
cudaEventSynchronize(event1);  
cpuFoo();
```



- Host-device synchronization:

```
foo<<<blocks, threads, 0, stream1>>>();  
cudaEventRecord(event1);  
cpuFoo();  
cudaEventSynchronize(event1);  
cpuFoo();
```



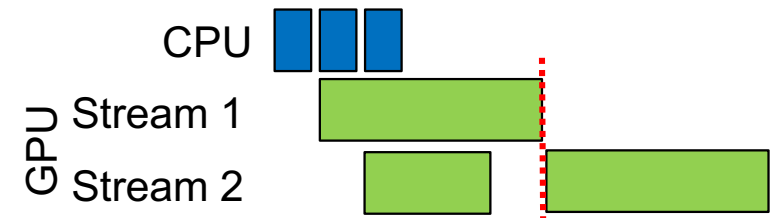
Assuming to have enough resources

Examples of event usage

- Intra-device synchronization:

```
cudaStreamCreate(&stream1);  
cudaStreamCreate(&stream2);  
foo<<<blocks, threads, 0, stream1>>>();  
cudaEventRecord(event1, stream1);  
goo<<<blocks, threads, 0, stream2>>>();  
cudaStreamWaitEvent(stream2, event1);  
foo<<<blocks, threads, 0, stream2>>>();
```

Stream2 needs to wait for event1. Since event1 is recorded at the end of Stream1, Stream2 will wait for Stream1



Stream2 blocked on event occurrence

`cudaStreamWaitEvent` add a synchronization point in a stream (1st parameter) on a specified event (2nd parameter)

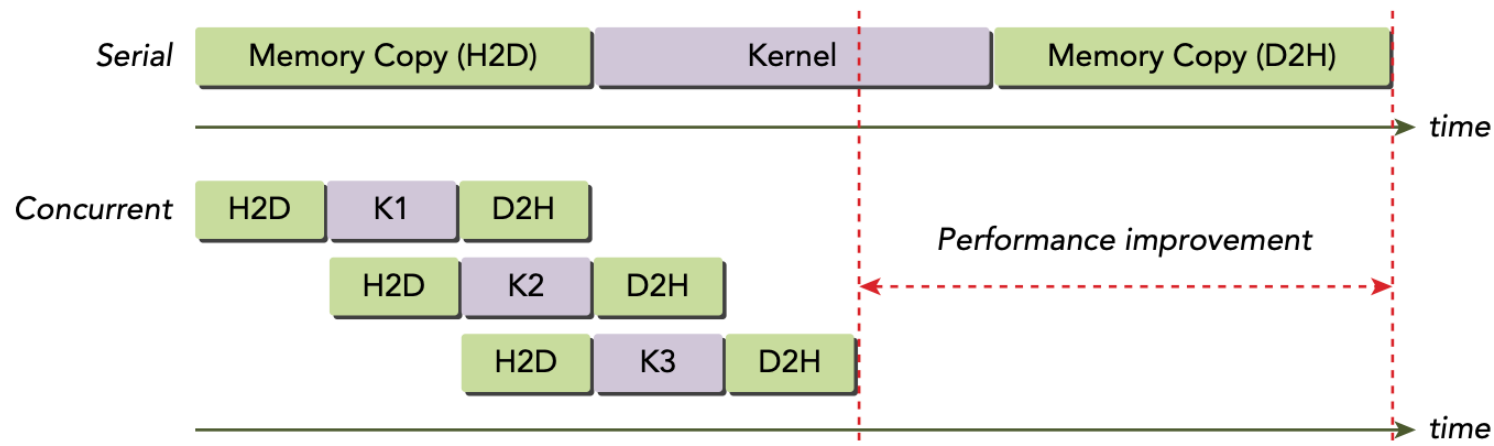
Assuming to have enough resources

Non-blocking synchronization functions

- The `synchronization functions` have `non-blocking counterparts`:
 - `cudaStreamQuery(stream);`
 - `cudaEventQuery(event);`
- They returns:
 - `cudaSuccess` (completed)
 - `cudaErrorNotReady` (not completed)
 - `cudaErrorInvalidResourceHandle` (error)

Practical exploitation of task level parallelism

- Parallelize data transfer and kernel execution on different streams
 - Spit the overall work in chunks
 - Limitation: 1 single data transfer can be executed per direction



- And... parallelize multiple kernels and host tasks!

An example: vector sum

```
/*...*/
#define N ...
#define NSTREAM ...
#define BDIM ...

int main(){
    /*...*/
    float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
    cudaHostAlloc(&h_A, N*sizeof(float));
    cudaHostAlloc(&h_B, N*sizeof(float));
    cudaHostAlloc(&h_d, N*sizeof(float));

    cudaMalloc(&d_A, N*sizeof(float));
    cudaMalloc(&d_B, N*sizeof(float));
    cudaMalloc(&d_C, N*sizeof(float));
    /*...*/
}
```

An example: vector sum

```
/*...*/
```

```
#define N ...
```

```
#define NSTREAMS ...
```

```
#define BDIM ...
```

Macros for data size,
number of streams and
block size

```
int main(){
```

```
/*...*/
```

```
float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
```

```
cudaHostAlloc(&h_A, N*sizeof(float));
```

```
cudaHostAlloc(&h_B, N*sizeof(float));
```

```
cudaHostAlloc(&h_d, N*sizeof(float));
```

Declare pointers to host
and device memories

Allocate host pinned
memory

```
cudaMalloc(&d_A, N*sizeof(float));
```

```
cudaMalloc(&d_B, N*sizeof(float));
```

```
cudaMalloc(&d_C, N*sizeof(float));
```

```
/*...*/
```

Allocate device memory

An example: vector sum

```
/*...*/  
int iElem = N / NSTREAMS;  
size_t iBytes = iElem * sizeof(float);  
dim3 block (BDIM);  
dim3 grid ((iElem + block.x - 1) / block.x);  
  
cudaStream_t stream[NSTREAMS];  
  
for (int i = 0; i < NSTREAMS; i++)  
    cudaStreamCreate(&stream[i]);  
  
/*...*/
```

An example: vector sum

```
/*...*/  
int iElem = N / NSTREAMS;  
size_t iBytes = iElem * sizeof(float);  
dim3 block (BDIM);  
dim3 grid ((iElem + block.x - 1) / block.x);  
  
cudaStream_t stream[NSTREAMS];  
  
for (int i = 0; i < NSTREAMS; i++)  
    cudaStreamCreate(&stream[i]);  
  
/*...*/
```

Compute the amount of data to be processed by a kernel (for simplicity, $N\%NSTREAM==0$)

Compute grid size

Declare array of streams

Create the streams

An example: vector sum

```
/*...*/  
for (int i = 0; i < NSTREAMS; i++) {  
    int ioffset = i * iElem;  
    cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes,  
                   cudaMemcpyHostToDevice, stream[i]);  
    cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes,  
                   cudaMemcpyHostToDevice, stream[i]);  
    sumArrays<<<grid, block, 0, stream[i]>>>  
        (&d_A[ioffset], &d_B[ioffset], &d_C[ioffset], iElem);  
    cudaMemcpyAsync(&h_C[ioffset], &d_C[ioffset], iBytes,  
                   cudaMemcpyDeviceToHost, stream[i]);  
}  
cudaDeviceSynchronize();  
/*...*/
```

An example: vector sum

```
/*...*/
for (int i = 0; i < NSTREAMS; i++) {
    int ioffset = i * iElem;
    cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes,
                   cudaMemcpyHostToDevice, stream[i]);
    cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes,
                   cudaMemcpyHostToDevice, stream[i]);
    sumArrays<<<grid, block, 0, stream[i]>>>
        (&d_A[ioffset], &d_B[ioffset], &d_C[ioffset], iElem);
    cudaMemcpyAsync(&h_C[ioffset], &d_C[ioffset], iBytes,
                   cudaMemcpyDeviceToHost, stream[i]);
}
cudaDeviceSynchronize();
/*...*/
```

- Submit in each stream
- 2x H2D memory transfers
 - kernel launch
 - D2H memory transfer


An example: vector sum

```
/*...*/
for (int i = 0; i < NSTREAMS; i++) {
    int ioffset = i * iElem;
    cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes,
                   cudaMemcpyHostToDevice, stream[i]);
    cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes,
                   cudaMemcpyHostToDevice, stream[i]);
    sumArrays<<<grid, block, 0, stream[i]>>>
        (&d_A[ioffset], &d_B[ioffset], &d_C[ioffset], iElem);
    cudaMemcpyAsync(&h_C[ioffset], &d_C[ioffset], iBytes,
                   cudaMemcpyDeviceToHost, stream[i]);
}
cudaDeviceSynchronize();
/*...*/
```

Compute and use offset to
the data chunk for the
current kernel

An example: vector sum

```
/*...*/  
for (int i = 0; i < NSTREAMS; i++) {  
    int ioffset = i * iElem;  
    cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes,  
                   cudaMemcpyHostToDevice, stream[i]);  
    cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes,  
                   cudaMemcpyHostToDevice, stream[i]);  
    sumArrays<<<grid, block, 0, stream[i]>>>  
        (&d_A[ioffset], &d_B[ioffset], &d_C[ioffset], iElem);  
    cudaMemcpyAsync(&h_C[ioffset], &d_C[ioffset], iBytes,  
                   cudaMemcpyDeviceToHost, stream[i]);  
}  
cudaDeviceSynchronize();  
/*...*/
```



Synchronize host at the end

An example: vector sum

```
/*...*/  
cudaFree(d_A);  
cudaFree(d_A);  
cudaFree(d_A);  
  
cudaFreeHost(h_A);  
cudaFreeHost(h_B);  
cudaFreeHost(h_C);  
  
for (int i = 0; i < NSTREAMS; ++i)  
    cudaStreamDestroy(stream[i]);  
  
return 0;  
}
```

An example: vector sum

```
/*...*/
cudaFree(d_A);
cudaFree(d_A);
cudaFree(d_A);

cudaFreeHost(h_A);
cudaFreeHost(h_B);
cudaFreeHost(h_C);

for (int i = 0; i < NSTREAMS; ++i)
    cudaStreamDestroy(stream[i]);

return 0;
}
```

Release all resources

References

- Slides mainly based on:
 - J. Chen, M. Grossman, T. Mckercher, **Professional Cuda C Programming, Chapter 6**