



**Barcelona
Supercomputing
Center**

Centro Nacional de Supercomputación

Efficient Host-Device Data Transfers

Marc Jordà, Antonio J. Peña

Based on material from NVIDIA's GPU Teaching Kit

Montevideo, 21-25 October 2019



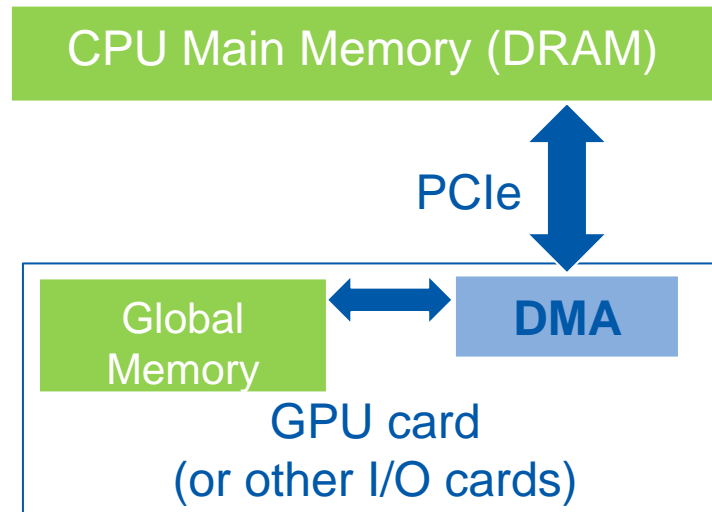
**Barcelona
Supercomputing
Center**
Centro Nacional de Supercomputación

PINNED HOST MEMORY

CPU-GPU Data Transfer using DMA

DMA (Direct Memory Access) hardware is used for the data copies to obtain better efficiency

- Frees CPU for other tasks
- Hardware unit specialized to transfer a number of bytes requested by OS
- Between physical memory address space regions (some can be mapped I/O memory locations)
- Uses system interconnect, typically PCIe in today's systems (NVLINK in some new machines)



Virtual Memory Management

Modern computers use **virtual memory** management

- Many virtual memory spaces mapped into a single physical memory
- Virtual addresses (pointer values) are translated into physical addresses

Not all variables and data structures are always in the physical memory

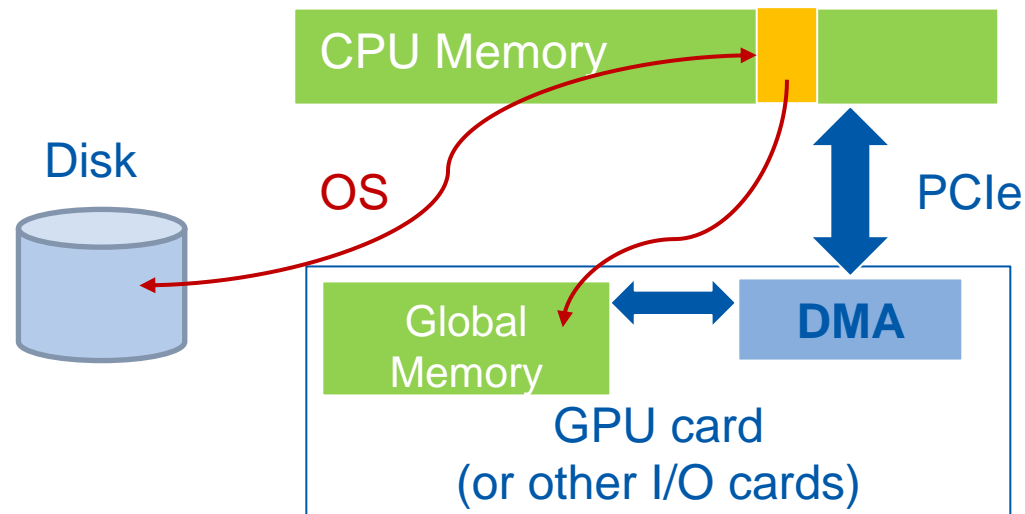
- Each virtual address space is divided into pages that are mapped into the physical memory
- Memory pages can be paged out to disk to make room in the DRAM
- Whether or not a variable is in the physical memory is checked at address translation time

Data Transfer and Virtual Memory

DMA uses physical addresses

- When `cudaMemcpy()` copies an array, it is implemented as one or more DMA transfers
- Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
- No address translation for the rest of the same DMA transfer so that high efficiency can be achieved

The OS could potentially page-out the data that is being read or written by a DMA and page-in another virtual page into the same physical location



Pinned Memory and DMA Data Transfer

- Pinned memory are virtual memory pages that are specially marked so that they cannot be paged-out to disk
- Allocated with a special system API function call
- a.k.a. Page Locked Memory, Locked Pages, etc.
- CPU memory that serves as the source or destination of a DMA transfer must be allocated as pinned memory

CUDA Data Transfer Uses Pinned Memory

If source or destination of a `cudaMemcpy()` in the host memory is not allocated in pinned memory, it needs to be first copied to a pinned memory buffer

- extra copy adds overhead

`cudaMemcpy()` is faster if the host memory (source or destination) is allocated in pinned memory since no extra copy is needed

Allocate/Free Pinned Memory

`cudaHostAlloc()`, three parameters

- Address of pointer to the allocated memory
- Size of the allocated memory in bytes
- Option – use `cudaHostAllocDefault` for now

`cudaFreeHost()`, one parameter

- Pointer to the memory to be freed

Using Pinned Memory in CUDA

Use the allocated pinned memory and its pointer the same way as those returned by `malloc()` ;

The only difference is that the allocated memory cannot be paged by the OS

The `cudaMemcpy()` function should be about 2x faster with pinned memory

Pinned memory is a limited resource

- over-subscription may affect system stability

Putting It Together - Vector Addition Host Code Example

```
int main()
{
    float *h_A, *h_B, *h_C;
    ...
    cudaHostAlloc((void **) &h_A, N*sizeof(float), cudaHostAllocDefault);
    cudaHostAlloc((void **) &h_B, N*sizeof(float), cudaHostAllocDefault);
    cudaHostAlloc((void **) &h_C, N*sizeof(float), cudaHostAllocDefault);
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

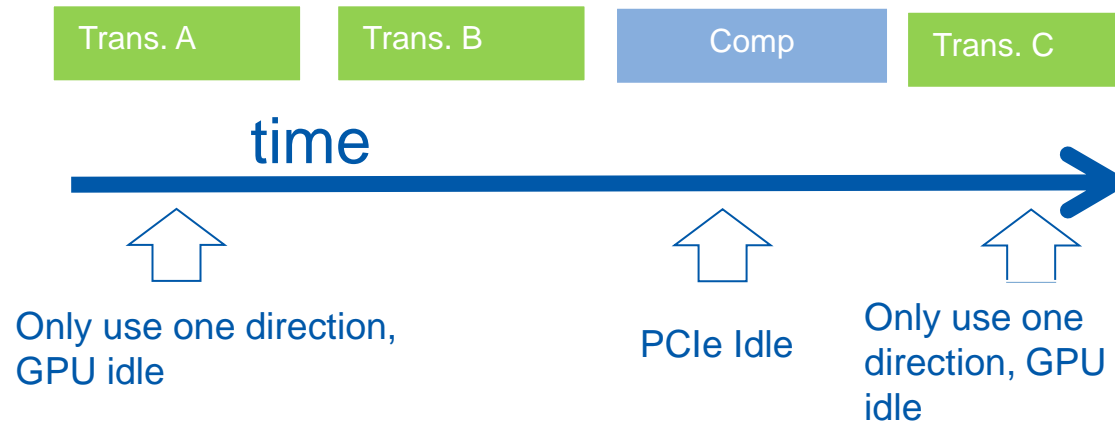


**Barcelona
Supercomputing
Center**
Centro Nacional de Supercomputación

TASK PARALLELISM IN CUDA

Serialized Data Transfer and Computation

- So far, the way we use `cudaMemcpy` serializes data transfer and GPU computation for `VecAddKernel()`



Device Overlap

Some CUDA devices support device overlap

- Simultaneously execute a kernel while copying data between device and host memory

Most GPUs in HPC servers/clusters support it and have 2 DMA engines

- Potential overlapping of:
 - Kernel execution
 - Copy from host to device
 - Copy from device to host

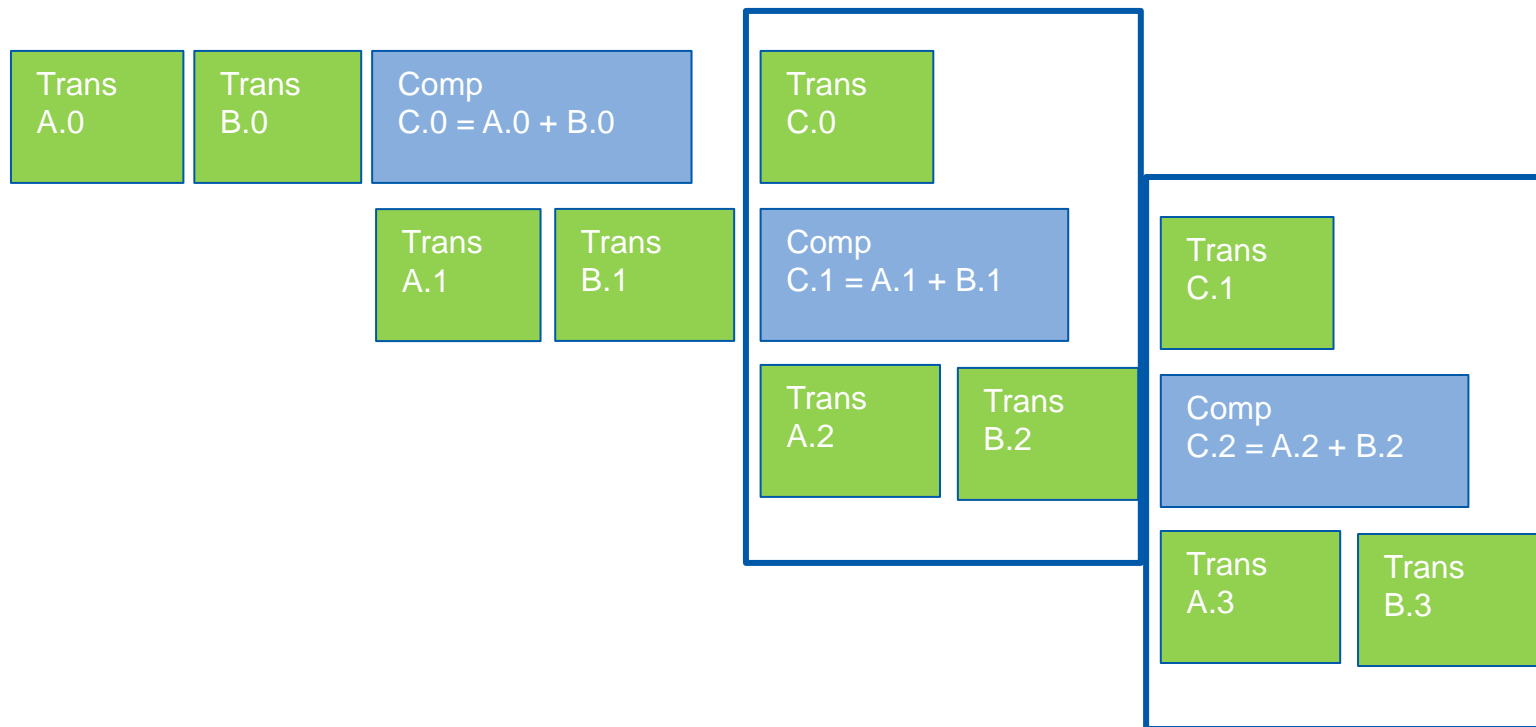
Code to check if execution/copy overlap is supported for a device:

```
int dev_count;
cudaDeviceProp prop;

cudaGetDeviceCount(&dev_count);
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&prop, i);
    if (prop.deviceOverlap)
    ...
}
```

Ideal, Pipelined Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments



CUDA Streams

CUDA supports parallel execution of kernels and data copies with “Streams”

Each stream is a queue of operations

- kernel launches
- `cudaMemcpyAsync ()` calls

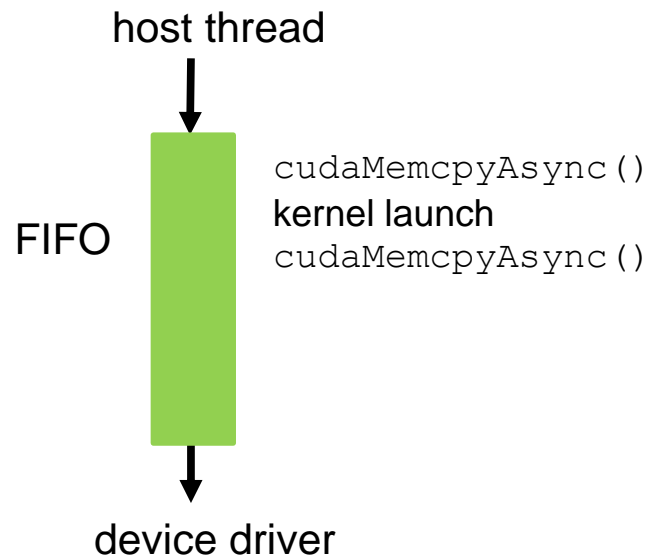
Operations (tasks) in different streams can go in parallel

- “Task parallelism”

Streams

Requests made from the host code are put into First-In-First-Out queues

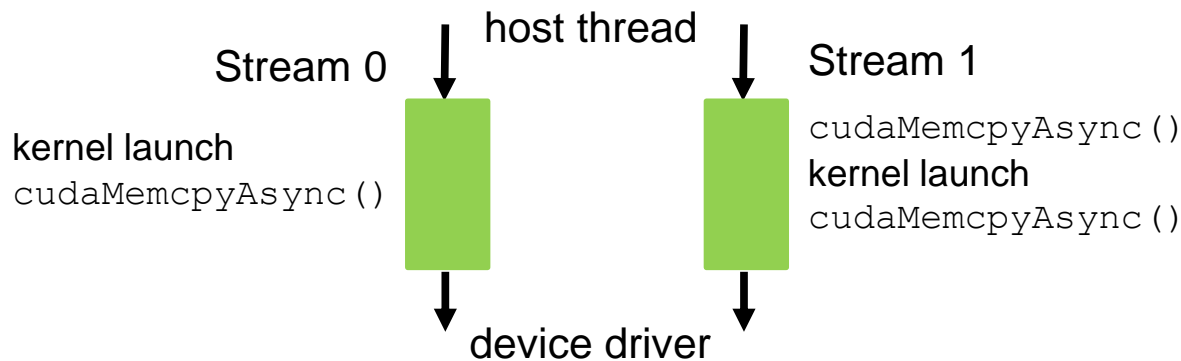
- Queues are read and processed asynchronously by the driver and device
- Driver ensures that commands in a queue are processed in sequence. E.g., Memory copies end before kernel launch, etc.
- The stream to use is specified as an extra argument



Streams cont.

To allow concurrent copying and kernel execution, use multiple streams

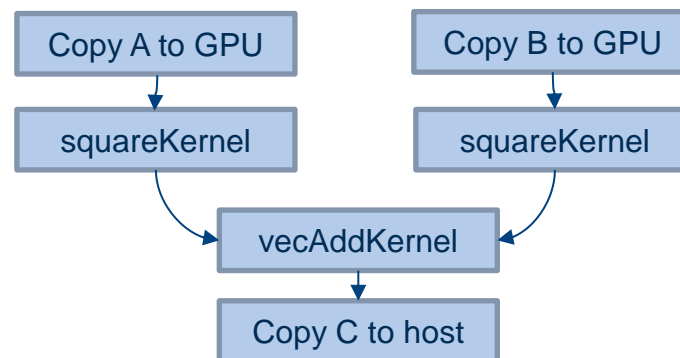
- Operations in the same stream will be executed in order
- Operations in different streams can potentially be executed at the same time
 - Depends on availability of corresponding hardware (gpu cores or DMA engines)



- Potential to use multiple streams depends on the algorithm

Example:

$$C = A * A + B * B$$



Pipelined vecAdd Host Code

```
cudaStream_t stream0, stream1;  
cudaStreamCreate(&stream0);  
cudaStreamCreate(&stream1);
```

```
float *d_A0, *d_B0, *d_C0; // device memory for stream 0  
float *d_A1, *d_B1, *d_C1; // device memory for stream 1
```

```
// cudaMalloc() calls for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go  
here
```

Pipelined vecAdd Host Code (Cont.)

```
for (int i=0; i<n; i += SegSize*2) {  
    cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float), ..., stream0);  
    cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float), ..., stream0);  
    vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0, ...);  
    cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float), ..., stream0);  
  
    cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float), ..., stream1);  
    cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float), ..., stream1);  
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);  
    cudaMemcpyAsync(d_C1, h_C+i+SegSize, SegSize*sizeof(float), ..., stream1);  
}
```

Wait Until All Tasks Have Completed

- `cudaStreamSynchronize (stream)`
 - Used in host code
 - Takes one parameter – stream identifier
 - Wait until all tasks in a stream have completed
 - E.g., `cudaStreamSynchronize (stream0)` in host code ensures that all tasks in `stream0` have completed
- **This is different from** `cudaDeviceSynchronize ()`
 - Also used in host code
 - No parameter
 - Waits until all tasks in all streams have completed for the current device
- `cudaEventSynchronize (event)`
 - Events can be inserted in a stream with `cudaEventRecord (...)`
 - Waits until all work in the stream previous to the event record has finished
- **Also** `cuda {Stream, Event} Query (...)` to check if job is done, but without waiting for it to finish

CUDA Graph API – Since CUDA 10

```
// Define graph of work + dependencies

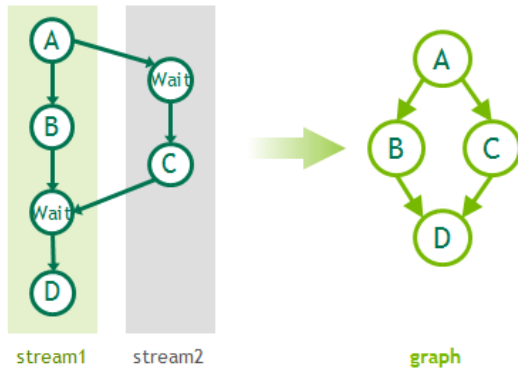
cudaGraphCreate(&graph);
cudaGraphAddNode(graph, kernel_a, {}, ...);
cudaGraphAddNode(graph, kernel_b, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_c, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_d, { kernel_b, kernel_c }, ...);

// Instantiate graph and apply optimizations

cudaGraphInstantiate(&instance, graph);

// Launch executable graph 100 times

for(int i=0; i<100; i++)
    cudaGraphLaunch(instance, stream);
```



```
// Start by initiating stream capture

cudaStreamBeginCapture(stream1);

// Build stream work as usual

A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph

cudaStreamEndCapture(stream1, &graph);
```



**Barcelona
Supercomputing
Center**

Centro Nacional de Supercomputación

QUIZ

Question 1

Which of the following is a correct CUDA API call that allocates 1,024 bytes of pinned memory for h_A?

- a) `cudaHostAlloc((void **) h_A, 1024, cudaHostAllocDefault);`
- b) `cudaPinnedAlloc((void **) h_A, 1024, cudaPinnedAllocDefault);`
- c) `cudaHostAlloc((void **) &h_A, 1024, cudaHostAllocDefault);`
- d) `cudaPinnedAlloc((void **) &h_A, 1024, cudaPinnedAllocDefault);`

Question 1 - Answer

Which of the following is a correct CUDA API call that allocates 1,024 bytes of pinned memory for h_A?

- a) `cudaHostAlloc((void **) h_A, 1024, cudaHostAllocDefault);`
- b) `cudaPinnedAlloc((void **) h_A, 1024, cudaPinnedAllocDefault);`
- c) `cudaHostAlloc((void **) &h_A, 1024, cudaHostAllocDefault);`**
- d) `cudaPinnedAlloc((void **) &h_A, 1024, cudaPinnedAllocDefault);`

Question 2

Which of the following statements is true?

- a) Data transfer between CUDA device and host is done by DMA hardware using virtual addresses.
- b) The OS always guarantees that any memory being used by DMA hardware is not swapped out.
- c) If a pageable data is to be transferred by `cudaMemcpy()`, it needs to be first copied to a pinned memory buffer before transferred.
- d) Pinned memory is allocated with `cudaMalloc()` function.

Question 2 - Answer

Which of the following statements is true?

- a) Data transfer between CUDA device and host is done by DMA hardware using virtual addresses.
- b) The OS always guarantees that any memory being used by DMA hardware is not swapped out.
- c) If a pageable data is to be transferred by cudaMemcpy(), it needs to be first copied to a pinned memory buffer before transferred.**
- d) Pinned memory is allocated with cudaMalloc() function.

Explanation: (A) is incorrect – DMA uses physical addresses. (B) OS does not guarantee so unless the memory is pinned. (D) Pinned memory is allocated with the cudaHostAlloc() function.

Question 3

- Which of the following CUDA API call can be used to perform an asynchronous data transfer?
- a) `cudaMemcpy();`
 - b) `cudaAsyncMemcpy();`
 - c) `cudaMemcpyAsync();`
 - d) `cudaDeviceSynchronize();`

Question 3 - Answer

Which of the following CUDA API call can be used to perform an asynchronous data transfer?

- a) `cudaMemcpy();`
- b) `cudaAsyncMemcpy();`
- c) `cudaMemcpyAsync();`**
- d) `cudaDeviceSynchronize();`

Question 4

- ⌘ What is the CUDA API call that makes sure that all previous kernel executions and memory copies in a device have been completed?
- a) `__syncthreads()`
 - b) `cudaDeviceSynchronize()`
 - c) `cudaStreamSynchronize()`
 - d) `__barrier()`

Question 4 - Answer

⌘ What is the CUDA API call that makes sure that all previous kernel executions and memory copies in a device have been completed?

- a) `__syncthreads()`
- b) `cudaDeviceSynchronize()`**
- c) `cudaStreamSynchronize()`
- d) `__barrier()`

www.bsc.es



**Barcelona
Supercomputing
Center**

Centro Nacional de Supercomputación

Thank you!

For further information please contact
marc.jorda@bsc.es, antonio.pena@bsc.es