

UCR



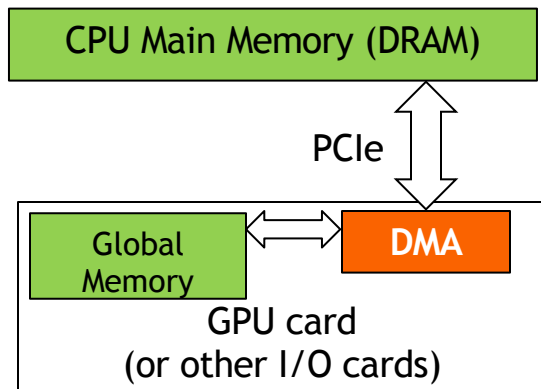
CUDA Pinned Memory & CUDA Streams

UNIVERSITY OF CALIFORNIA, RIVERSIDE

PINNED HOST MEMORY

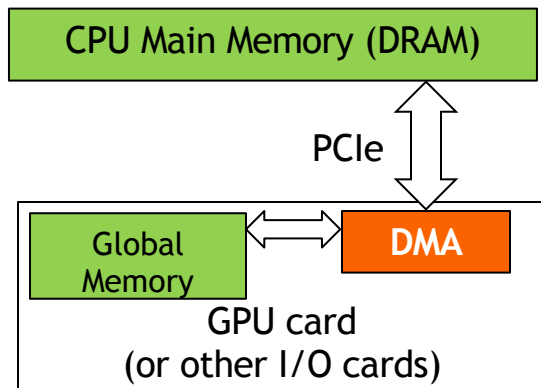
CPU-GPU Data Transfer using DMA

- DMA (Direct Memory Access) hardware is used by `cudaMemcpy()` for 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



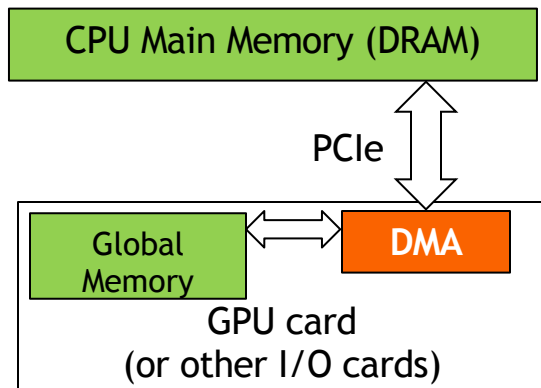
Virtual Memory Management

- 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**



Virtual Memory Management

- Not all variables and data structures are always in the physical memory!
 - Each virtual address space is divided into pages that are mapped into and out of the physical memory
 - **Virtual memory pages can be paged out** of the physical memory to make room
 - Whether or not a variable is in the physical memory is checked at address translation time



Data Transfer and Virtual Memory

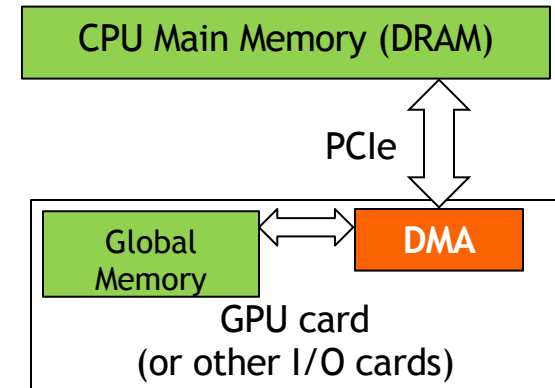
- DMA uses physical addresses
 - When `cudaMemcpy()` copies an array, it is implemented as one or more DMA transfers
 - Address is translated and page presence checked for the entire source and destination regions at the beginning of each DMA transfer
 - No address translation for the rest of the same DMA transfer so that high efficiency can be achieved
- The OS could accidentally 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
- a.k.a. Page Locked Memory, Locked Pages, etc.
- Allocated with a special system API function call
- CPU memory that serve as the source or destination of a DMA transfer must be allocated as pinned memory

CUDA data transfer uses pinned memory.

- The DMA used by `cudaMemcpy()` requires that any source or destination in the host memory is allocated as pinned memory
- If a 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 – extra 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 can have serious consequences

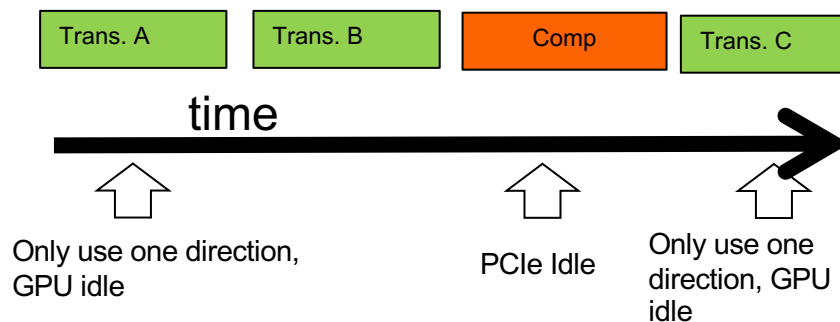
Putting It Together - Vector Addition Host Code Example

```
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);
    ...
    // cudaMemcpy() runs 2X faster
}
```

CUDA STREAMS

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

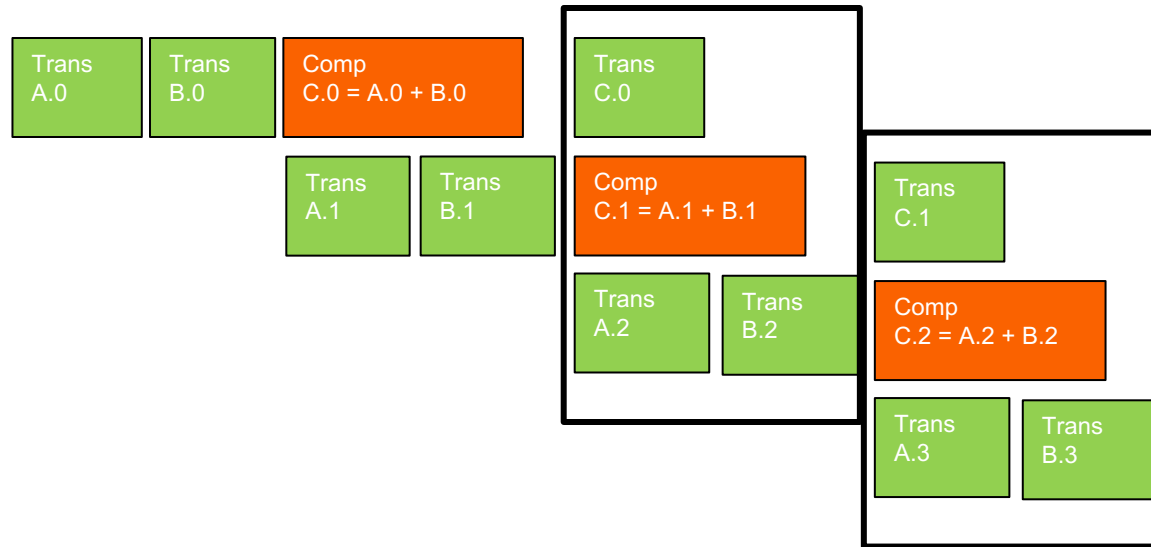
```
int dev_count;
cudaDeviceProp prop;

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

(All modern GPUs support this.)

Ideal, Pipelined Timing

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

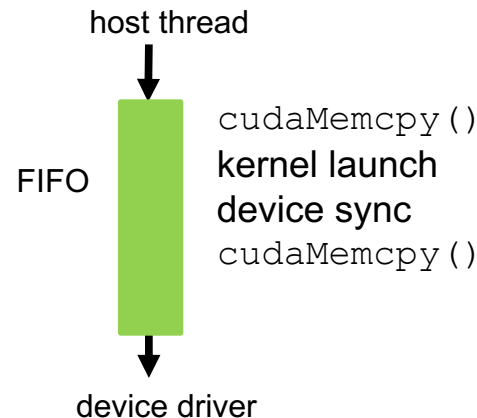


CUDA Streams

- CUDA supports parallel execution of kernels and `cudaMemcpy()` with “Streams”
- Each stream is a *queue of operations* (operations = kernel launches and `cudaMemcpy()` calls)
- Operations (tasks) in different streams can go in parallel
 - “Task parallelism”

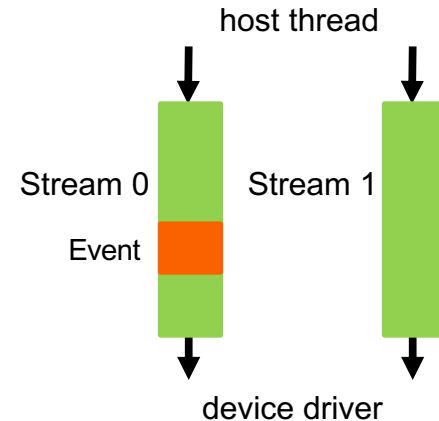
Streams

- 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.

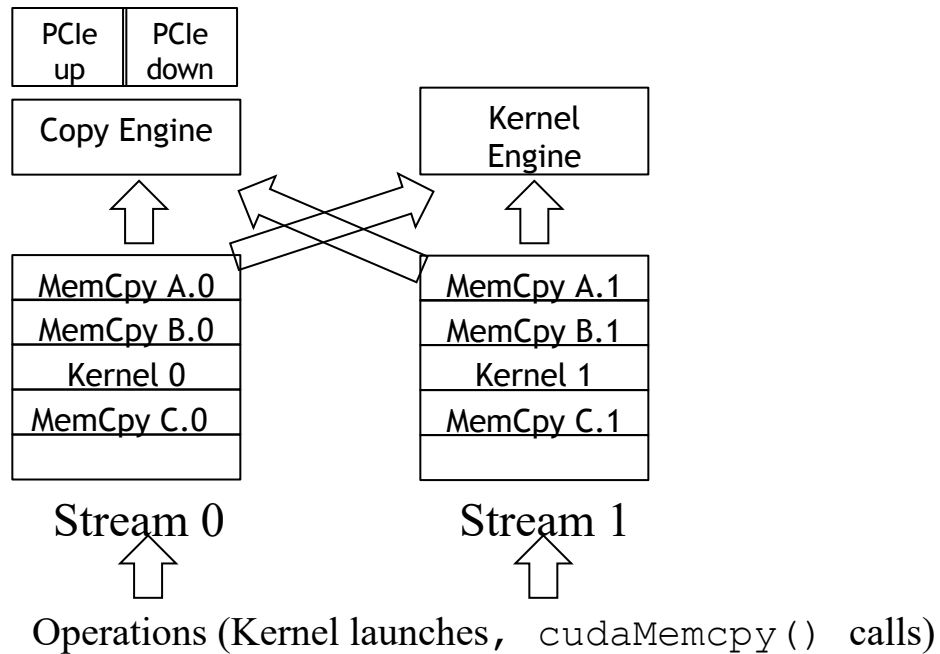


Streams

- To allow concurrent copying and kernel execution, use multiple queues, called “streams”
- CUDA “events” allow the host thread to query and synchronize with individual queues (i.e. streams).



Conceptual View of Streams



OVERLAPPING DATA TRANSFER W/ COMPUTATION

Simple Multi-Stream Host Code

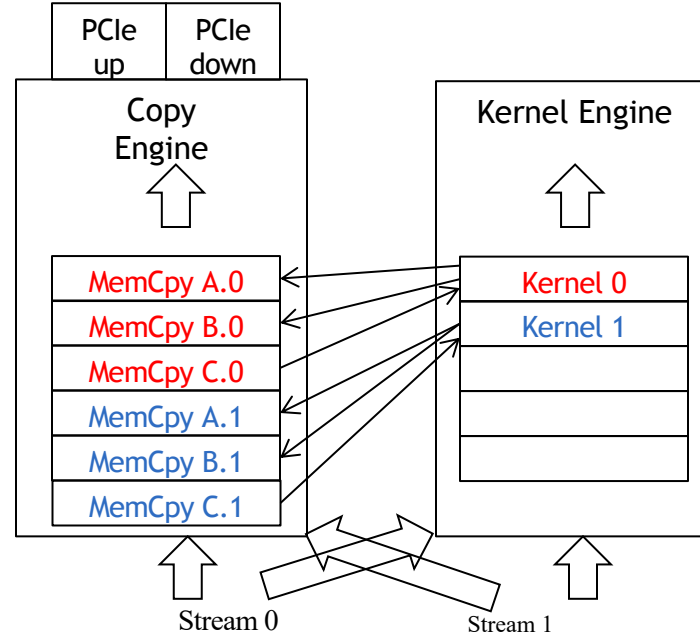
```
cudaStream_t stream0, stream1;  
cudaStreamCreate(&stream0);  
cudaStreamCreate(&stream1);  
  
float *d_A0, *d_B0, *d_C0; // device memory for stream 0  
float *d_A1, *d_B1, *d_C1; // device memory for stream 1  
  
// cudaMalloc() calls for d_A0, d_B0, d_C0, d_A1, d_B1, d_C1 go here
```

Simple Multi-Stream 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);  
}
```

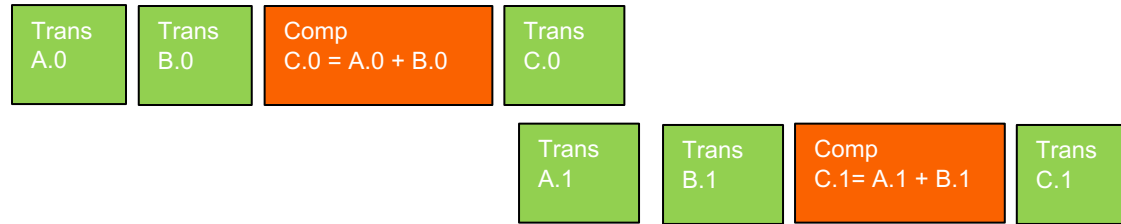
A View Closer to *Reality*



Operations (Kernel launches, `cudaMemcpy()` calls)

Not quite the overlap we want in some GPUs

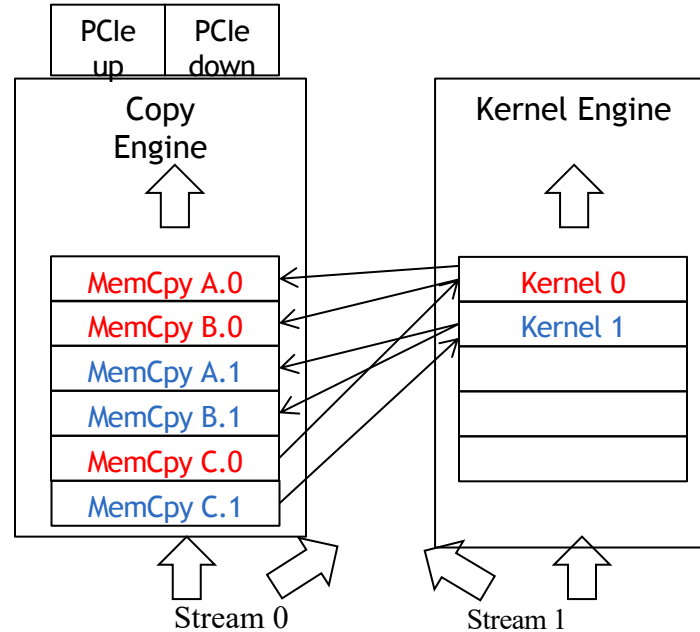
- C.0 blocks A.1 and B.1 in the copy engine queue



A *Better* Multi-Stream Host Code

```
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);  
    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, stream0>>>(d_A0, d_B0, ...);  
    vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, ...);  
  
    cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),..., stream0);  
    cudaMemcpyAsync(h_C+i+SegSize, d_C1, SegSize*sizeof(float),..., stream1);  
}
```

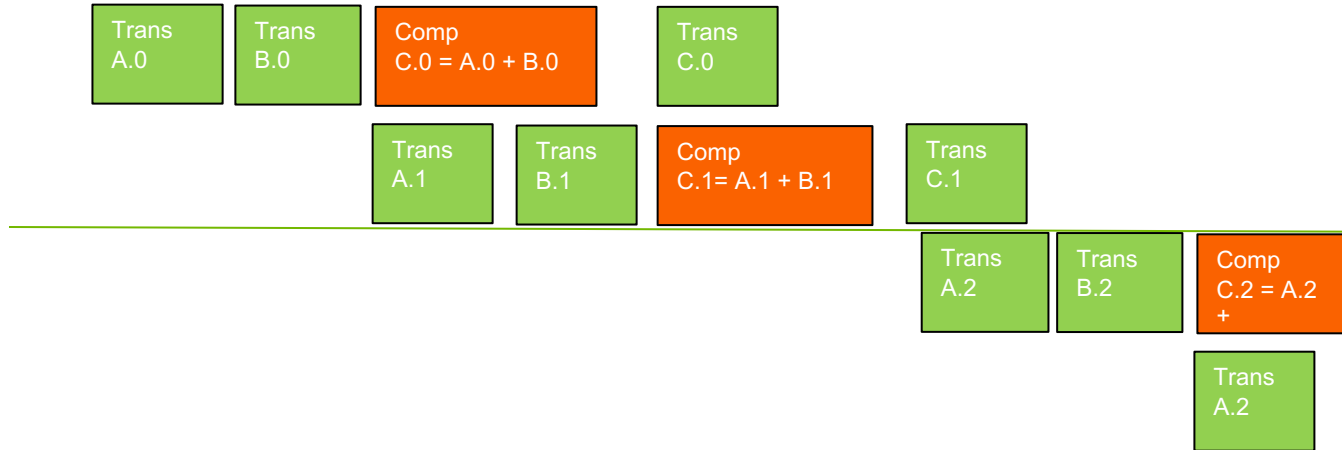
C.0 no longer blocks A.1 and B.1



Operations (Kernel launches, cudaMemcpy() calls)

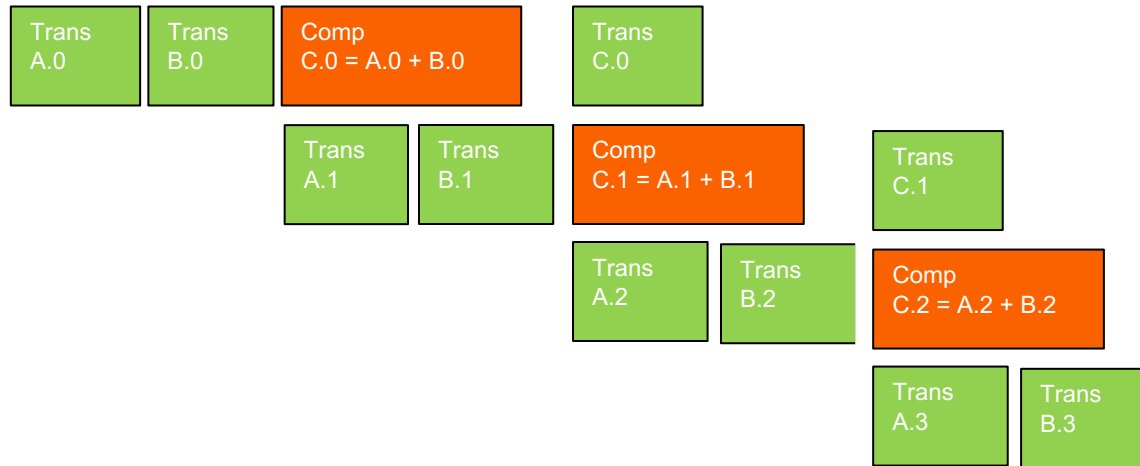
Better, not quite the best overlap

- C.1 blocks next iteration A.0 and B.0 in the copy engine queue



Ideal, Pipelined Timing

- Will need **at least three buffers** for each original A, B, and C, code is more complicated



Wait until all tasks have completed

- `cudaStreamSynchronize(stream_id)`
 - 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 the queues of `stream0` have completed

- This is different from `cudaDeviceSynchronize()`
 - Also used in host code
 - No parameter
 - `cudaDeviceSynchronize()` waits until **all tasks in all streams** have completed for the current device