Heterogeneous Parallel Programming COMP4901D

Host-Device Data Transfer

Overview

- CUDA memory copy between host and device
- Pinned memory
- CUDA streams
- Overlapping memory copy and computation

Why

- Bandwidth between GPU and device memory: 178GB/second on M2090
- Bandwidth between host memory and device memory: 8 GB/second on the PCIe x16 Gen2
- Host-device data transfer often the bottleneck in the overall performance

Simple Ways of Optimizing Host-Device Data Transfer

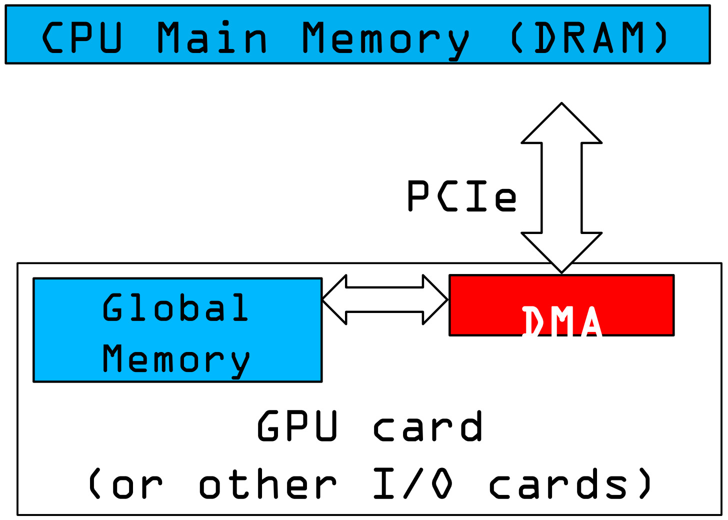
- Minimize transfers
  - Intermediate data can be allocated, operated on, and deallocated without ever copying to host memory
  - Run most of the code on the GPU
- Group transfers
  - One large transfer much better than many small ones; needs a large buffer for batching

More Advanced Ways of Optimizing Host-Device Data Transfer

- Pinned memory (page-locked memory)
- Asynchronous and overlapping transfers

Behind cudaMemcpy ()

- DMA (Direct Memory Access) hardware is used for cudaMemcpy() for efficiency
  - Frees CPU for other tasks
  - Transfers a number of bytes requested by OS
  - Uses PCI-e bus



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 when mapped into physical memory
  - Memory pages can be paged out to make room
  - Whether a variable is in the physical memory is checked at address translation time

DMA 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 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

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

Data Transfer Using 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

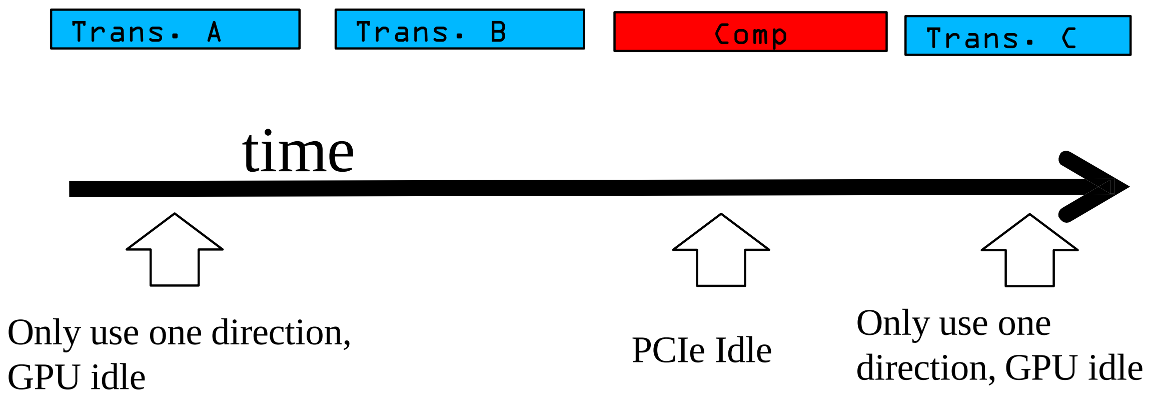
- Use the allocated pinned memory and its pointer the same way as those returned by malloc();
- The only difference is that the allocated memory cannot be paged by the OS
- The cudaMemcpy() function is about 2X faster with pinned memory
- Pinned memory is a limited resource
  - over-subscription can have serious consequences

Vector Add Example with Pinned Memory

```
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);
}
```

Data Transfer and Computation Serialized

- So far, the way we use cudaMemcpy serializes data transfer and GPU computation


Trans. A
Trans. B
Comp
Trans. C
time
Only use one direction, GPU idle
PCIe Idle
Only use one direction, GPU idle

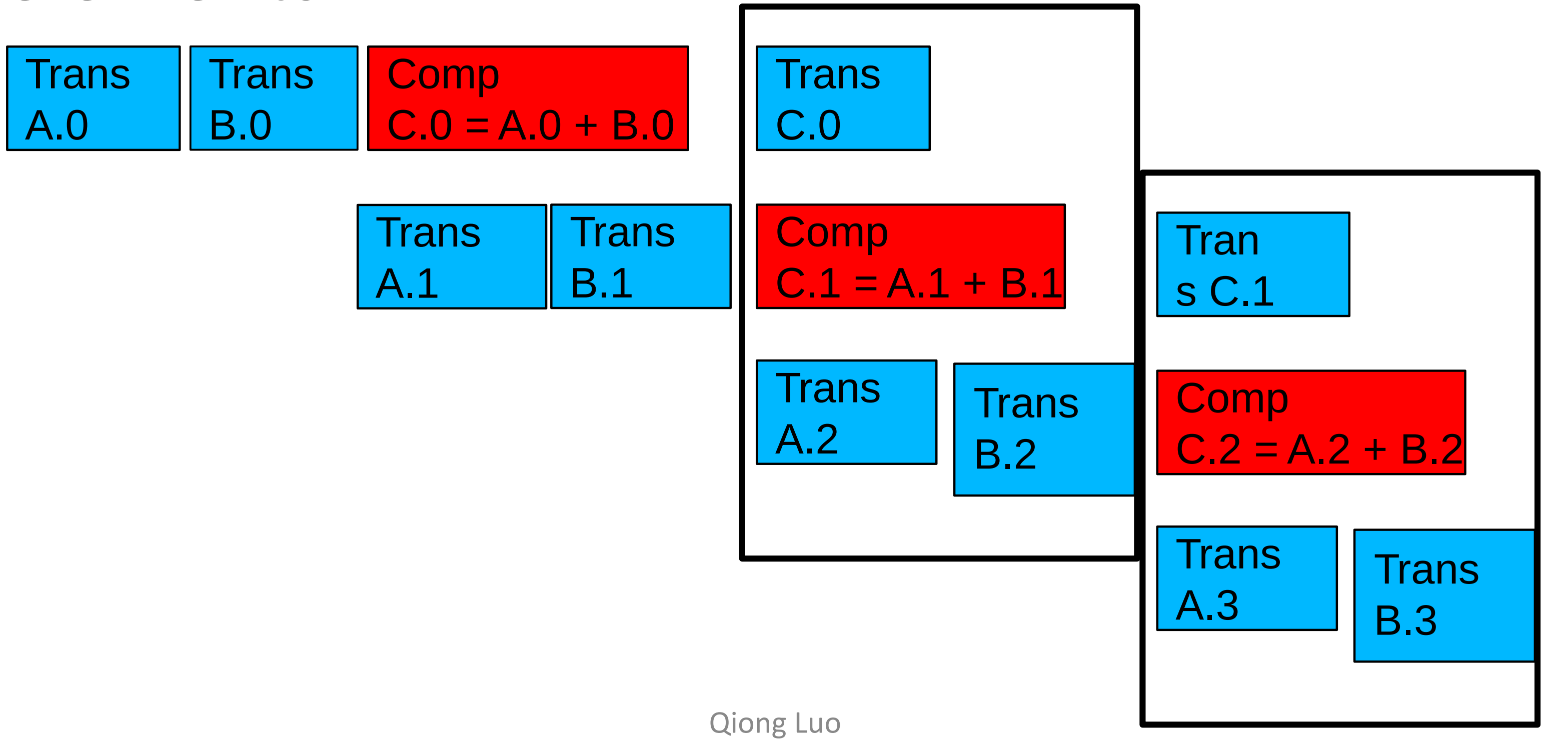
Device Capability of Overlapping Data Transfer and Execution

- 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) …
```

Ideal Timing Pipeline

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

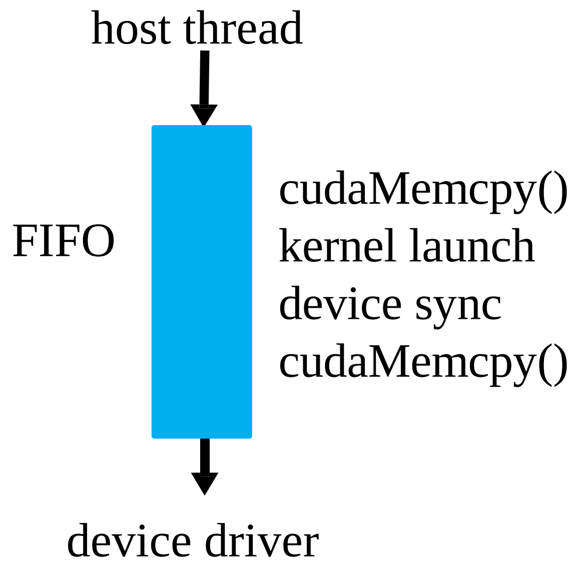


CUDA Streams

- CUDA supports parallel execution of kernels and Memcpy with “Streams”
  - Each stream is a queue of operations (kernel launches and Memcpy’s)
  - Operations (tasks) in different streams can go in parallel: “Task parallelism”

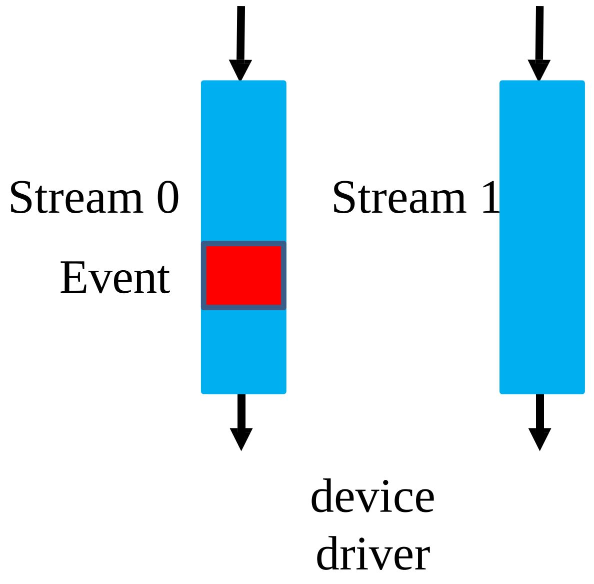
Queues Behind CUDA Streams

- 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 executes

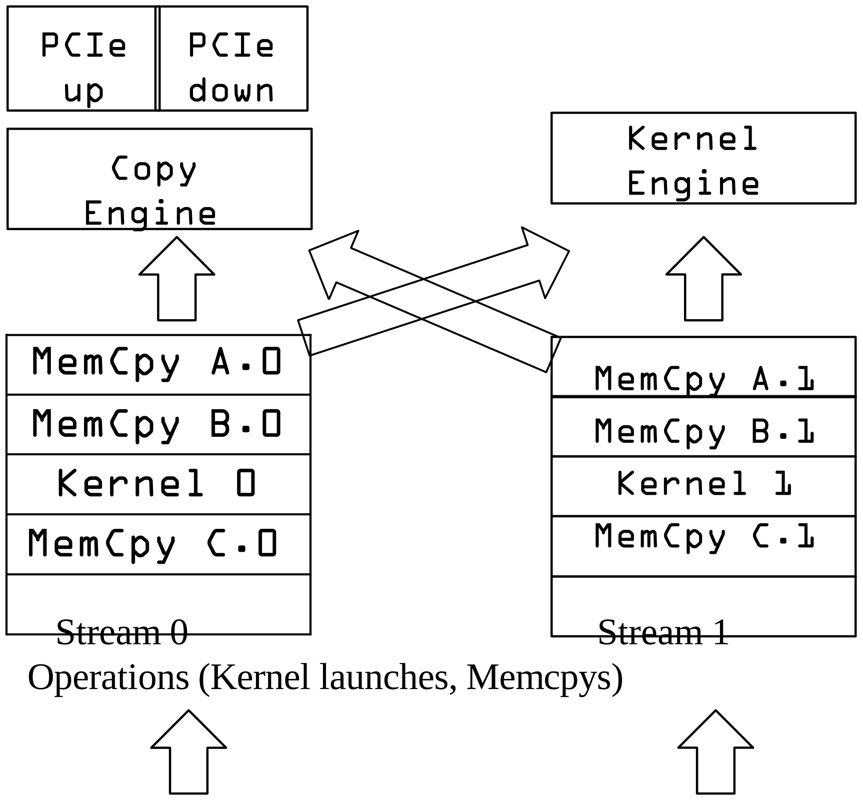


Multiple Streams for Concurrent Copying and Kernel Execution

- To allow concurrent copying and kernel execution, use multiple CUDA streams
  - CUDA "events" allow the host thread to query and synchronize with individual streams.



Conceptual View of Multiple Streams



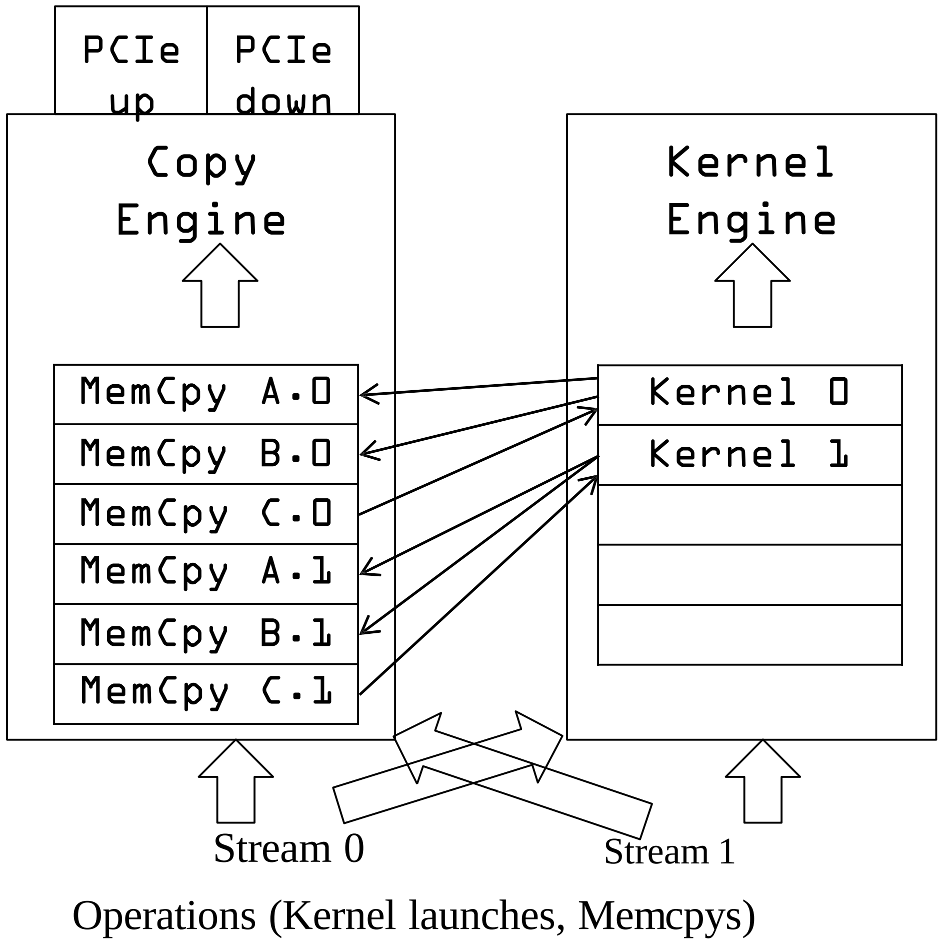
Example of Multiple CUDA Streams

```
1.cudaStream_t stream0, stream1;
2.cudaStreamCreate(&stream0);
3.cudaStreamCreate(&stream1);

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

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

Example of Multiple CUDA Streams (cont.)

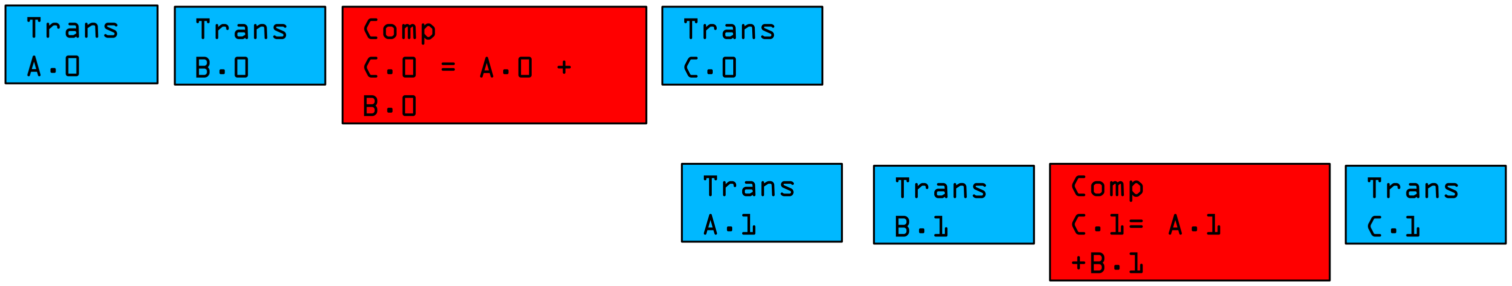
```
6.for (int i=0; i<n; i+=SegSize*2) {
7.  cudaMemcpyAsync(d_A0, h_A+i, SegSize*sizeof(float),…, stream0);
8.  cudaMemcpyAsync(d_B0, h_B+i, SegSize*sizeof(float),…, stream0);
9.  vecAdd<<<SegSize/256, 256, 0, stream0>>>(d_A0, d_B0,…);
10. cudaMemcpyAsync(h_C+i, d_C0, SegSize*sizeof(float),…, stream0);
11. cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),…,
stream1);
12. cudaMemcpyAsync(d_B1, h_B+i+SegSize, SegSize*sizeof(float) ,…,
stream1);
13. vecAdd<<<SegSize/256, 256, 0, stream1>>>(d_A1, d_B1, …);
14.  cudaMemcpyAsync(d_C1, h_C+i+SegSize, SegSize*sizeof(float),…,
           stream1);
}
```

Conceptual View



We Want Finer Grain Concurrency

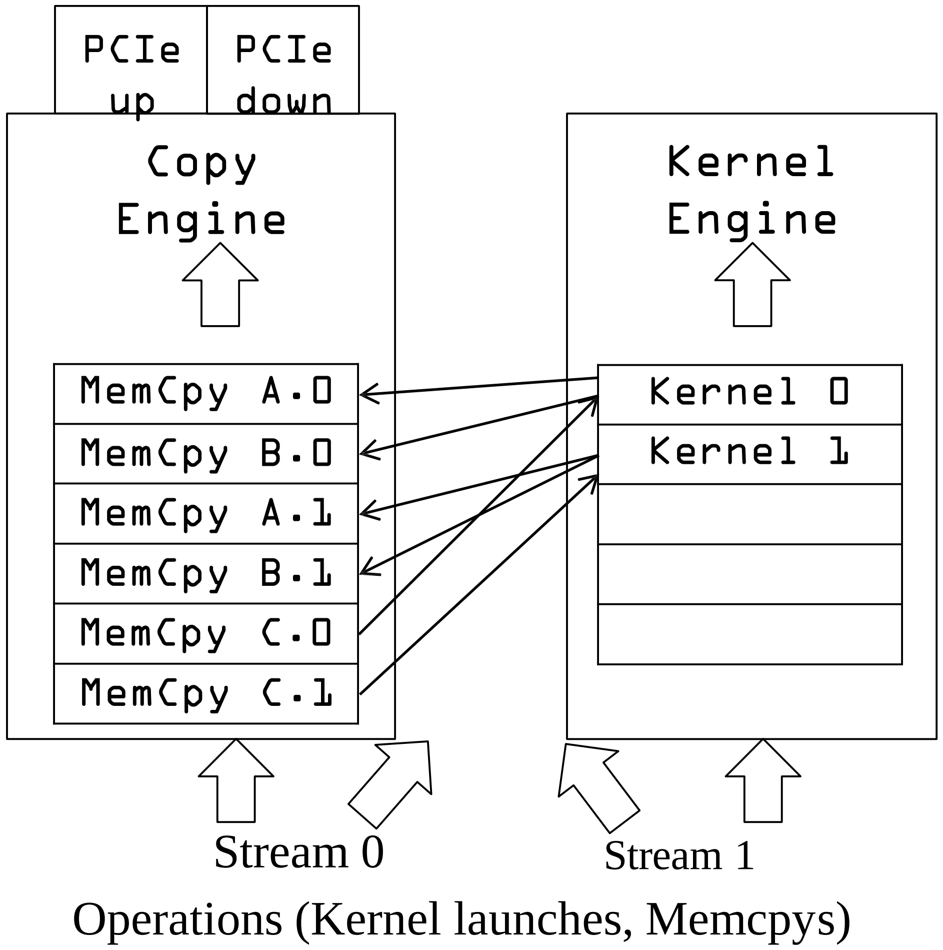
- C.0 blocks A.1 and B.1 in the copy engine queue



Better Concurrency with Streams

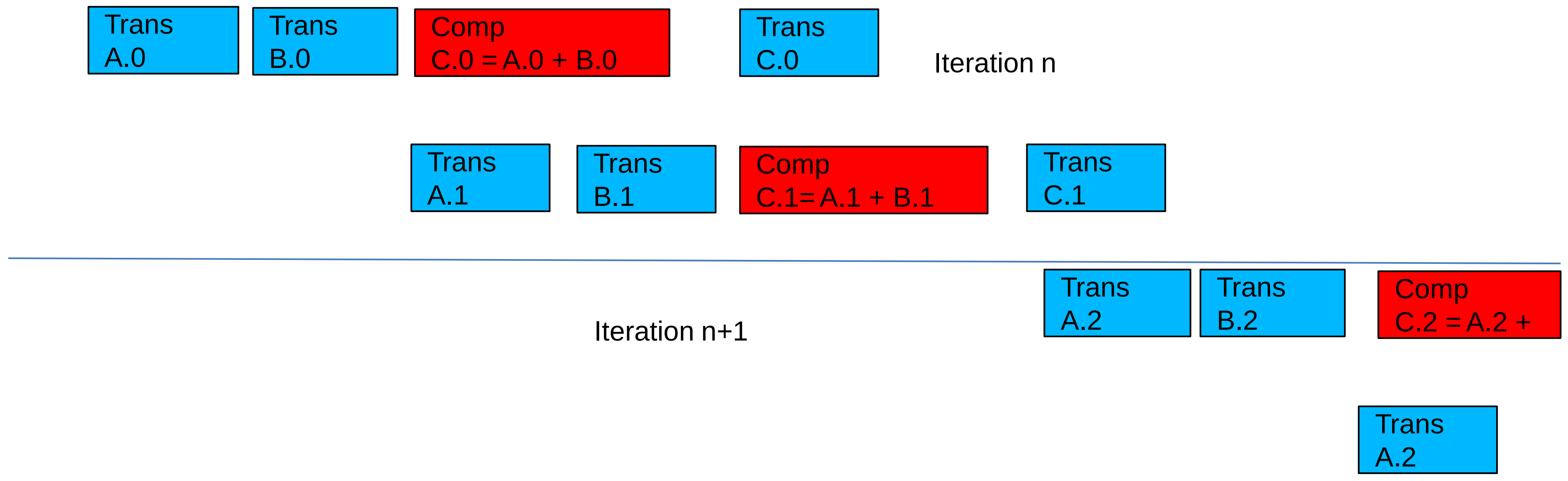
```
        for (int i=0; i<n; i+=SegSize*2) {
cudaMemcpyAsync(d_A0,    h_A+i,    SegSize*sizeof(float),…,    stream0);
cudaMemcpyAsync(d_B0,    h_B+i,    SegSize*sizeof(float),…,    stream0);
cudaMemcpyAsync(d_A1, h_A+i+SegSize, SegSize*sizeof(float),…,
                stream1);
cudaMemcpyAsync(d_B1, 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 or B.1



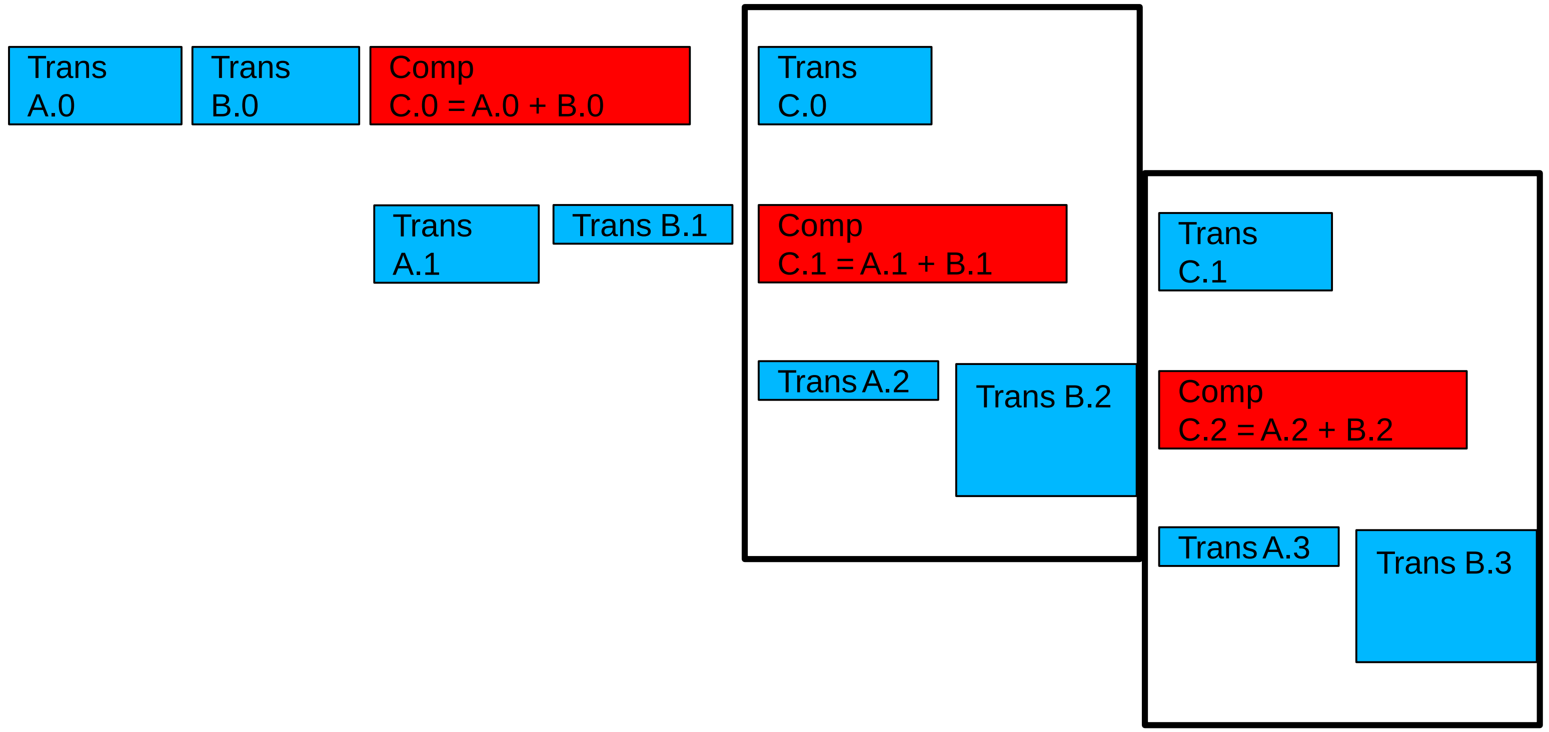
Not the Best Overlap Yet

- C.1 blocks the next iteration's 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
- cudaDeviceSynchronize()
  - Also used in host code
  - No parameter
  - Wait until all tasks in all streams have completed for the current device

Summary

- Memory transfer between host and device is usually the performance bottleneck.
- General ways of optimizing memory transfer
  - Minimize transfer
  - Batch transfer
- Two more advanced optimizations in CUDA
  - Pinned memory
  - Overlapping transfer and computation