CS 179 Lecture 13

Host-Device Data Transfer

Moving data is slow

So far we've only considered performance when the data is already on the GPU

This neglects the slowest part of GPU programming: getting data on and off of GPU

Moving data is important

Intelligently moving data allows processing data larger than GPU global memory (~6GB)

Absolutely critical for real-time or streaming applications (common in computer vision, data analytics, control systems)

VRAM

Max VRAM on a GPU (Nvidia Maxwell): 12 GB
Max RAM on a CPU (Xeon E7): 1536 GB

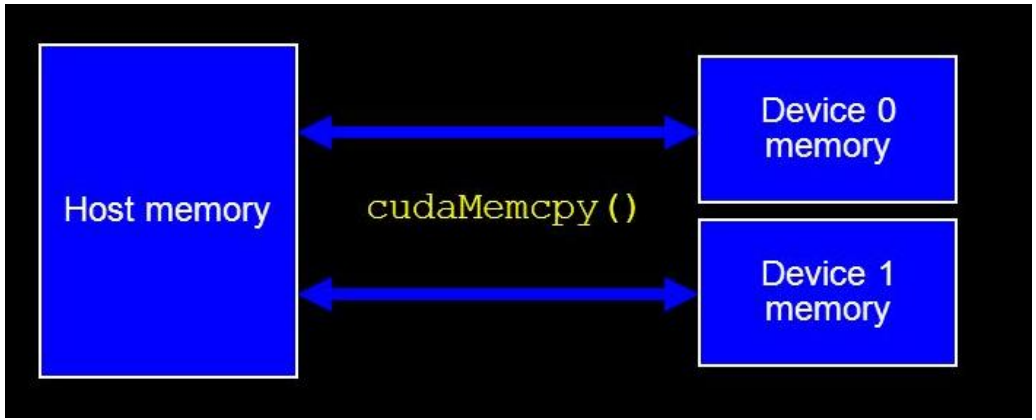
1. VRAM is soldered directly onto the graphics card, allowing the VRAM modules to sit closely to the GPU, allowing quicker transfer between VRAM and GPU, increasing graphics performance. Secondly, sockets and circuits most likely would cause GPU prices to go up.

2. The next generation of Nviida GPUs (Pascal) will supposedly feature 32 GB of memory in the top of the line versions.

Matrix transpose: another look

```
Time(%)        Time    Calls  Avg        Name
49.35%  29.581ms    1     29.581ms   [CUDA memcpy DtoH]
47.48%  28.462ms    1     28.462ms   [CUDA memcpy HtoD]
3.17%   1.9000ms    1     1.9000ms   naiveTransposeKernel
```

Only 3% of time spent in kernel! 97% of time spent moving data onto and off GPU! Copying between host and GPU is SLOW.



Lecture Outline

- IO strategy
- CUDA streams
- CUDA events
- How it all works: virtual memory, command buffers
- Pinned host memory
- Managed memory

A common pattern

```
while (1) {
      cudaMemcpy(d_input, h_input, input_size)
      kernel<<<grid, block>>>(d_input, d_output)
      cudaMemcpy(output, d_output, output_size)
}
```

Throughput limited by IO!
How can we hide the latency?

Dreams & Reality

Reality

HD 0		
	kernel 0	
		DH 0
HD 1		
	kernel 1	
		DH 1
HD 2		
	kernel 2	

time

Dreams

HD 0		
HD 1	kernel 0	
HD 2	kernel 1	DH 0
HD 3	kernel 2	DH 1
HD 4	kernel 3	DH 2
HD 5	kernel 4	DH 3
HD 6	kernel 5	DH 4
HD 7	kernel 6	DH 5

Turning dreams into reality

What do we need to make the dream happen?

- hardware to run 2 transfers and 1 kernel in parallel
- 2 input buffers
- 2 output buffers

- asynchronous memcpy & kernel invocation

Latency hiding checklist

Hardware:

- maximum of 4, 16, or 32 concurrent kernels (depending on hardware) on CC >= 2.0
- 1 device→host copy engine
- 1 host→device copy engine

(2 copy engines only on newer hardware, some hardware has single copy engine shared for both directions)

Asynchrony

An *asynchronous* function returns as soon is it called.

There is generally an interface to check if the function is done and to wait for completion.

Kernel launches are asynchronous.
`cudaMemcpy` is not.

cudaMemcpyAsync

Convenient asynchronous memcpy! Similar arguments to normal `cudaMemcpy`.

```
while (1) {
      cudaMemcpyAsync(d_in, h_in, in_size)
      kernel<<<grid, block>>>(d_in, d_out)
      cudaMemcpyAsync(out, d_out, out_size)
}
```

Can anyone think of any issues with this code?

CUDA Streams

In previous example, need `cudaMemcpyAsync` to finish before kernel starts. Luckily, CUDA already does this.

Streams let us enforce ordering of operations and express dependencies.

Useful blog post describing streams

The null / default stream

When stream is not specified, operation only starts after all other GPU operations have finished.
CPU code can run concurrently with default stream.

Stream example

```
cudaStream_t s[2];
cudaStreamCreate(&s[0]); cudaStreamCreate(&s[1]);
for (int i = 0; i < 2; i++) {
    kernel<<<grid, block, shmem, s[i]>(d_outs[i], d_ins[i]);
    cudaMemcpyAsync(h_outs[i], d_outs[i], size, dir, s[i]);
}
for (int i = 0; i < 2; i++) {
    cudaStreamSynchronize(s[i]);
    cudaStreamDestroy(s[i]);
}
```

kernels run in parallel!

CUDA events

Streams synchronize the GPU (but can synchronize CPU/GPU with `cudaStreamSynchronize`)

Events are simpler way to enforce CPU/GPU synchronization.

Also useful for timing!

Events example

```
#define START_TIMER() {                          \
          gpuErrChk(cudaEventCreate(&start));            \
          gpuErrChk(cudaEventCreate(&stop));             \
          gpuErrChk(cudaEventRecord(start));             \
          }
#define STOP_RECORD_TIMER(name) {                                  \
          gpuErrChk(cudaEventRecord(stop));                        \
          gpuErrChk(cudaEventSynchronize(stop));                   \
          gpuErrChk(cudaEventElapsedTime(&name, start, stop));  \
          gpuErrChk(cudaEventDestroy(start));                      \
          gpuErrChk(cudaEventDestroy(stop));                       \
          }
```

Events methods

`cudaEventRecord` - records that an event has occurred. Recording happens not at time of call but after all preceding operations on GPU have finished

`cudaEventSynchronize` - CPU waits for event to be recorded

`cudaEventElapsedTime` - compute time between recording of events

Other stream/event methods

- `cudaStreamAddCallback` `<- Add host function to stream`
- `cudaStreamQuery` <- check if stream completed
- `cudaEventQuery` `<- check if event completed`
- `cudaDeviceSynchronize` `<- Wait on all streams`

Can also parameterize event recording to happen only after all preceding operations complete in a given stream (rather than in all streams)

CPU/GPU communication

How exactly do the CPU and GPU communicate?

Virtual Memory

Could give a week of lectures on virtual memory…

Key idea: The memory addresses used in programs do not correspond to physical locations in memory. A program deals solely in virtual addresses. There is a table that maps (process id, address) to physical address.

What does virtual memory give us?

Each process can act like it is the only process running.
The same virtual address in different processes can point to different physical addresses (and values).

Each process can use more than the total system memory.
Store *pages* of data on disc if there is no room in physical memory.
Operating system can move pages around physical memory and disc as needed.

Unified Virtual Addressing

On 64-bit OS with GPU of CC >= 2.0, GPU pointers live in disjoint address space from CPU. Makes it possible to figure out which memory an address lives on at runtime.

NVIDIA calls it unified virtual addressing (UVA)

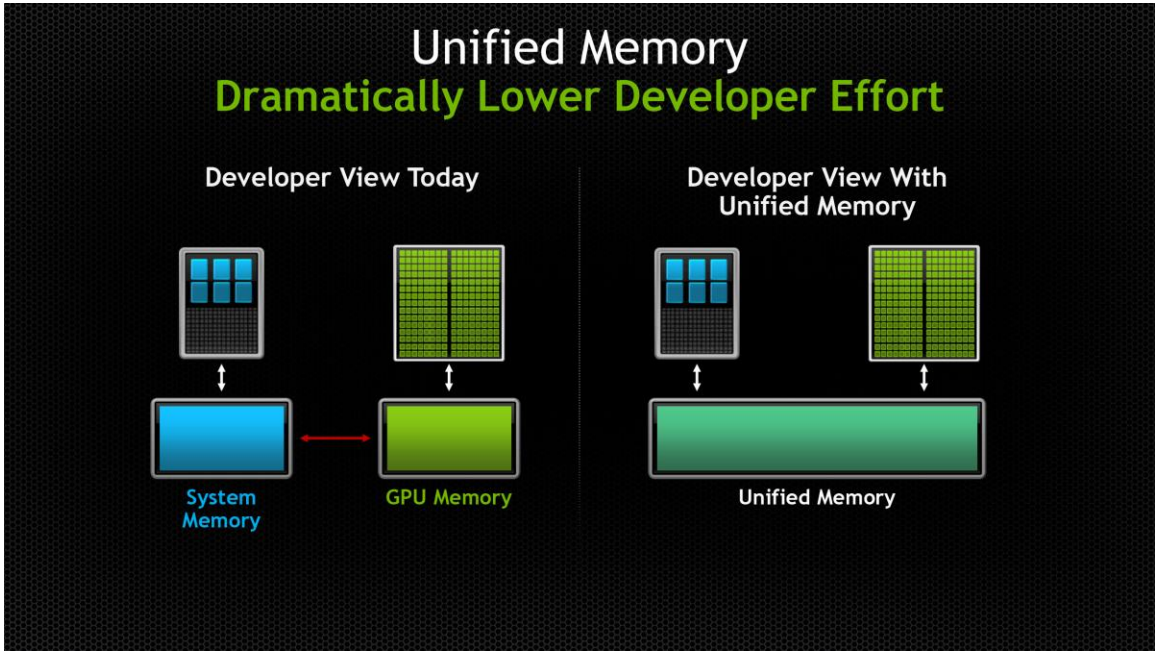
`cudaMemcpy(dst, src, size, cudaMemcpyDefault)`, no need to specify `cudaMemcpyHostToDevice` or etc.

Unified Virtual Addressing/Memory

You can think of unified memory as “smart pinned memory”. Driver is allowed to cache memory on host or any GPU.

Available on CC >= 3.0

cudaMallocManaged/
cudaFree



Virtual memory and GPU

To move data from CPU to GPU, the GPU must access data on host. GPU is given virtual address.

2 options:

(1) for each word, have the CPU look up physical address and then perform copy. Discontiguous physical memory. slow!

(2) tell the OS to keep a page at a fixed location (*pinning*). Directly access physical memory on host from GPU (*direct memory access a.k.a. DMA)*. fast!

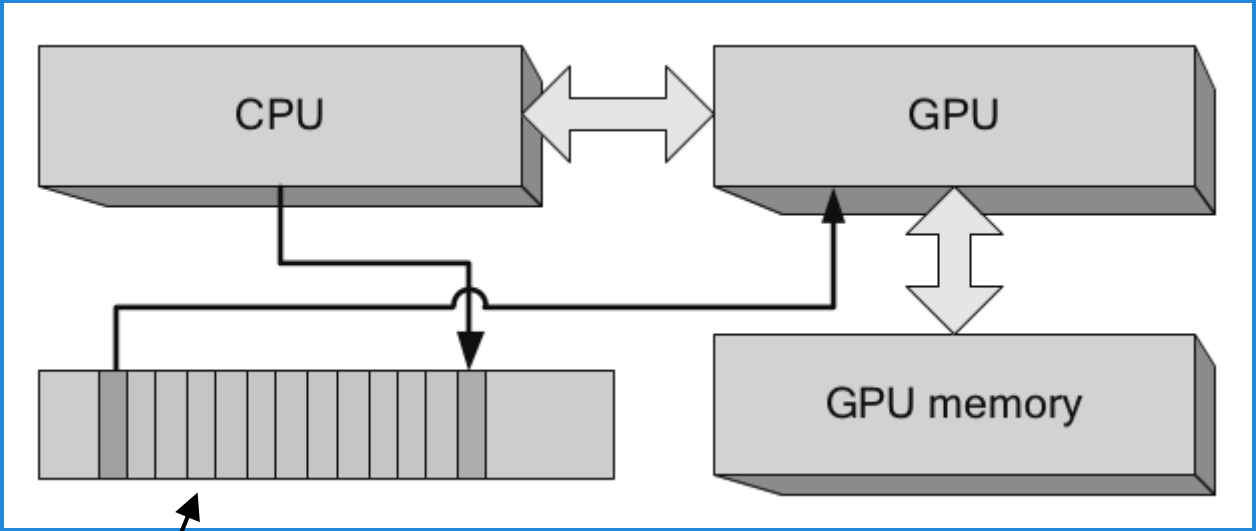
Memcpy

`cudaMemcpy(Async)`:

Pin a host buffer in the driver.

Copy data from user array into pinned buffer.

Copy data from pinned buffer to GPU.



Command buffers (diagram courtesy of CUDA Handbook)

Taking advantage of pinning

`cudaMallocHost` allocates pinned memory on the host.
`cudaFreeHost` to free.

Advantages:

(1) can dereference pointer to pinned host buffers on device! Lots of PCI-Express (PCI-E) traffic :(

(2) `cudaMemcpy` is considerably faster when copying to/from pinned host memory.

Pinned host memory use cases

- self-referential data structures that are not easy to copy (such as a linked list)
- deliver output as soon as possible (rather than waiting for kernel completion and memcpy)

Must synchronize and wait for kernel to finish before accessing kernel result on host.

Disadvantages of pinning

Pinned pages limit freedom of OS memory management. `cudaMallocHost` will fail (due to no memory available) long before `malloc`.

Coalesced accesses are extra important while accessing pinned host memory.

Potentially tricky concurrency issues.