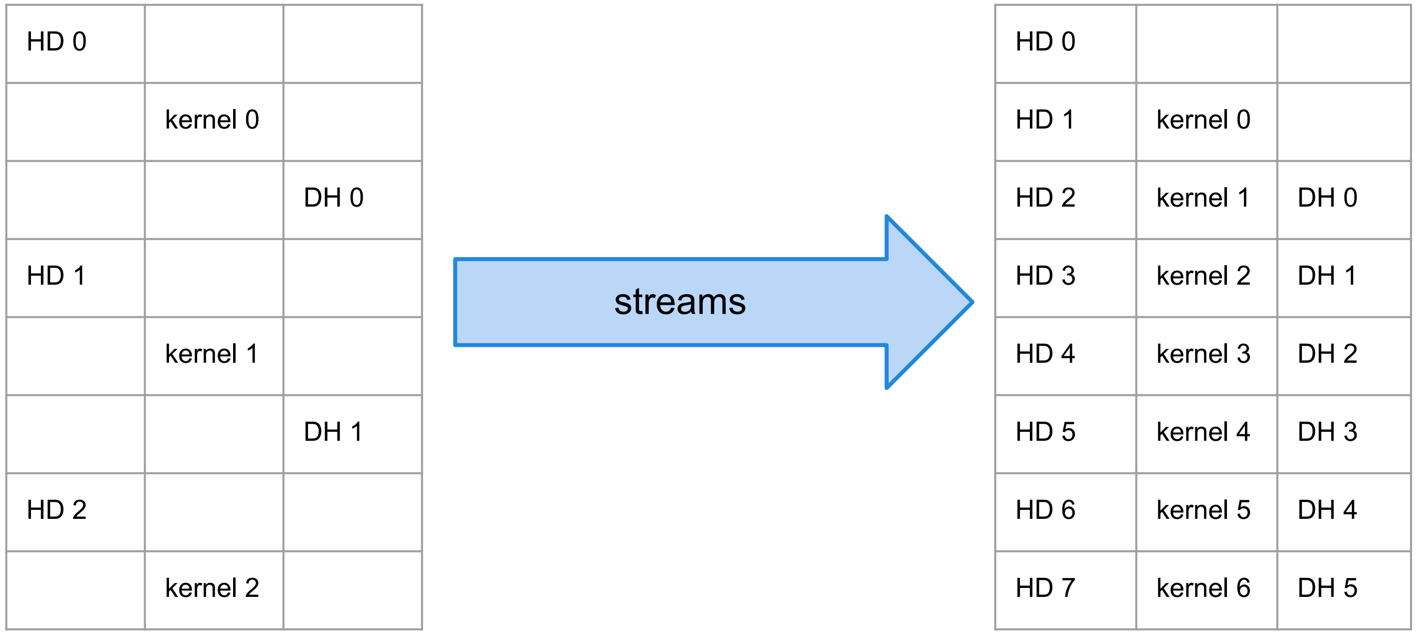
CS 179 Lecture 14

Last time



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

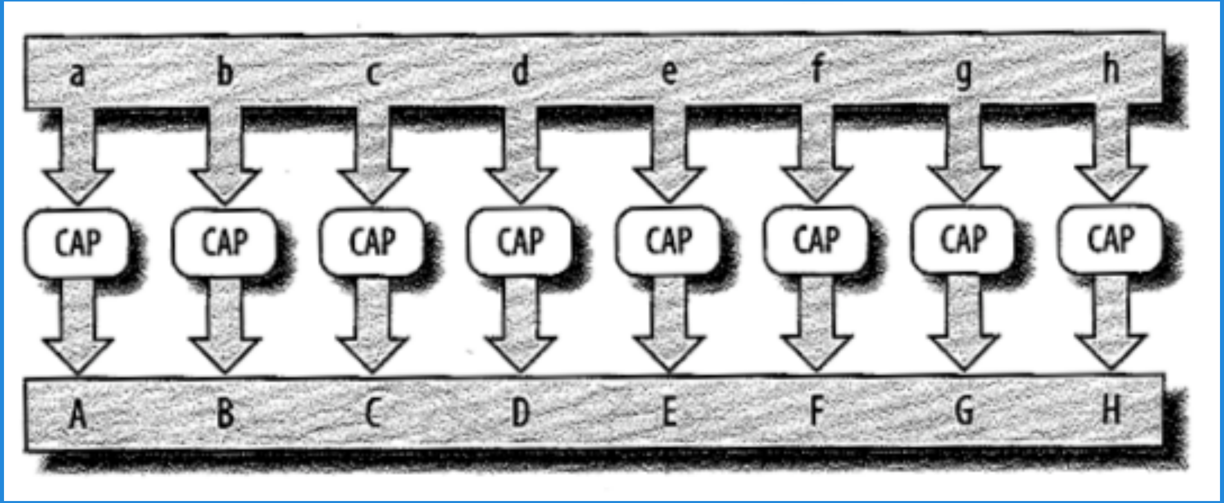
HD 0		
HD 1	kernel 0	
HD 2	kernel 1	DH 0
HD 3	kernel 2	DH 1
HD 4	kernel 3	DH 2
HD 5	kernel 4	DH 3
HD 6	kernel 5	DH 4
HD 7	kernel 6	DH 5

Today

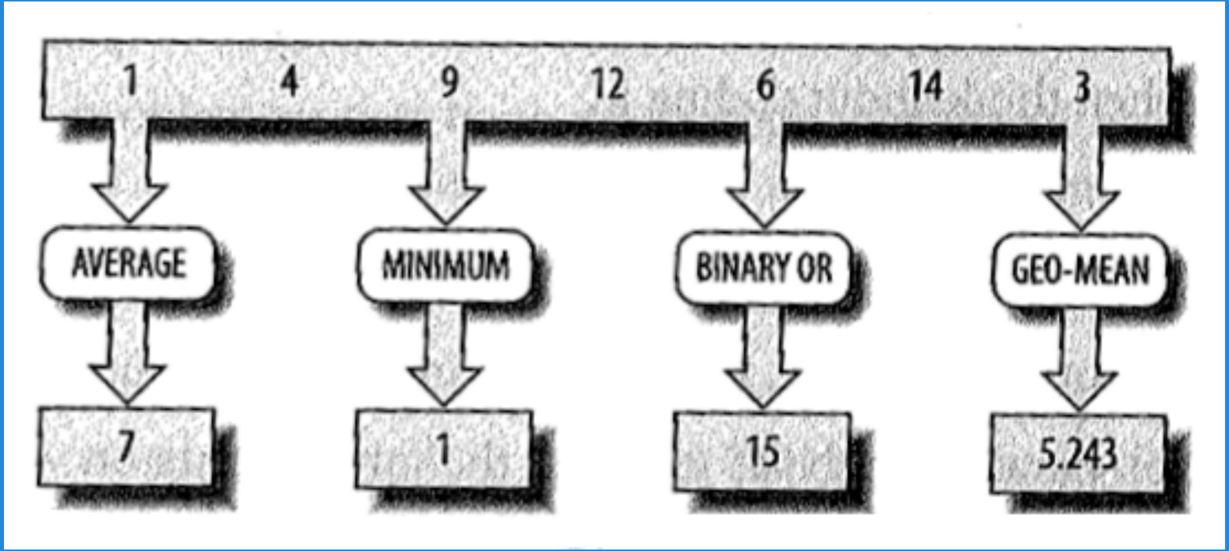
- Pipeline parallelism
- Programming multiple GPUs

General theme of week:

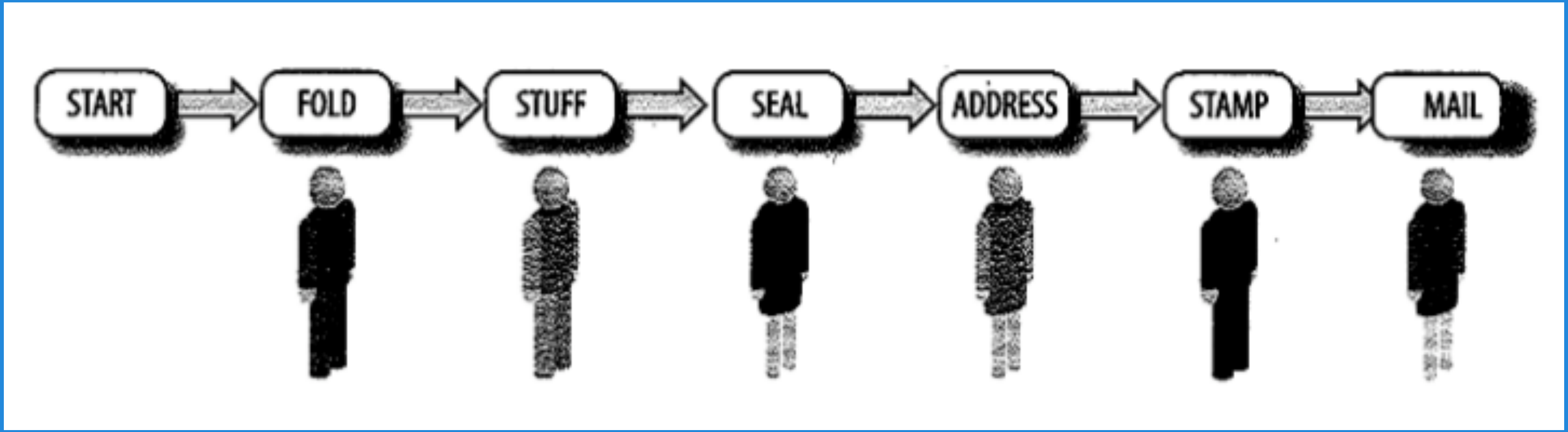
Using all of your computational resources in parallel.



Data Parallelism (figures from James Reinders)



Task Parallelism

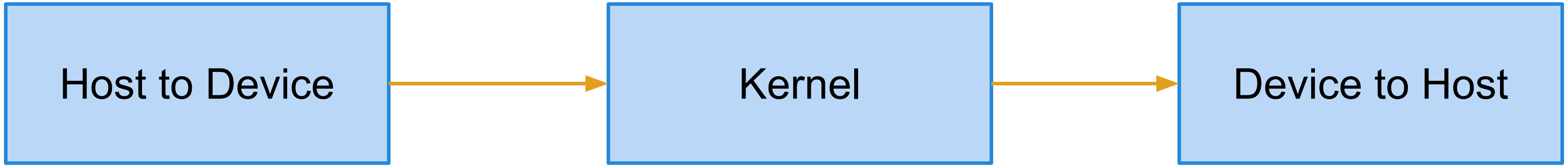


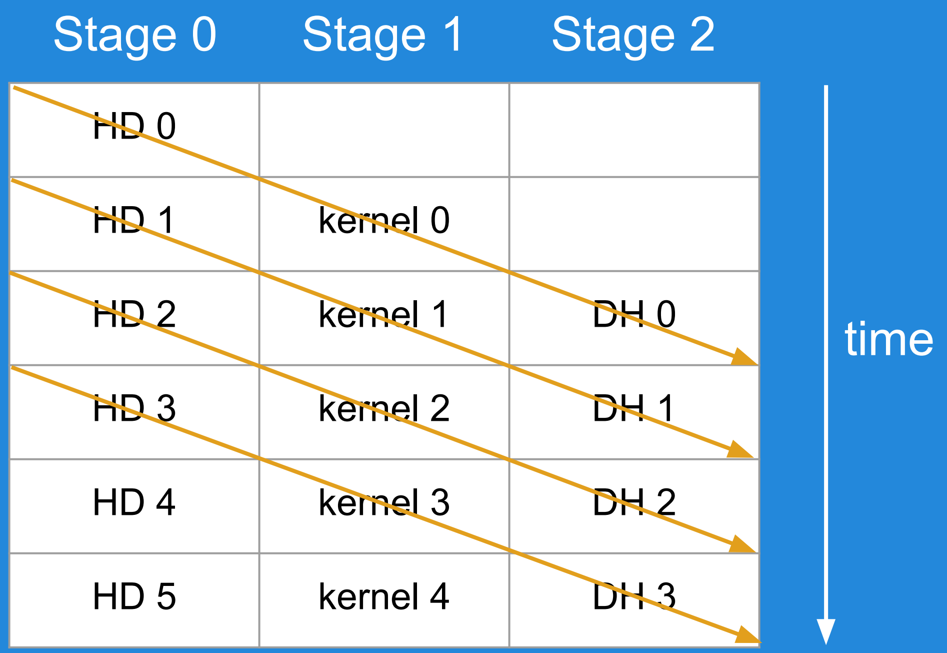
Pipeline Parallelism

Pipeline parallelism on GPU

```
while (1) {
    cudaMemcpy(d_in, h_in, input_size, cudaMemcpyHostToDevice);
    kernel<<<grid, block>>>(d_input, d_output);
    cudaMemcpy(h_out, d_out, output_size, cudaMemcpyDeviceToHost);
}
```

3 stage pipeline:





	Stage 0	Stage 1	Stage 2
	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
Stage computed by:	HD copy engine	SM’s	DH copy engine

Pipeline parallelism in action

Pipeline analysis

What are the latency and throughput of a pipeline?

(Hint: Analyze with with respect to latency and throughput of each stage of pipeline)

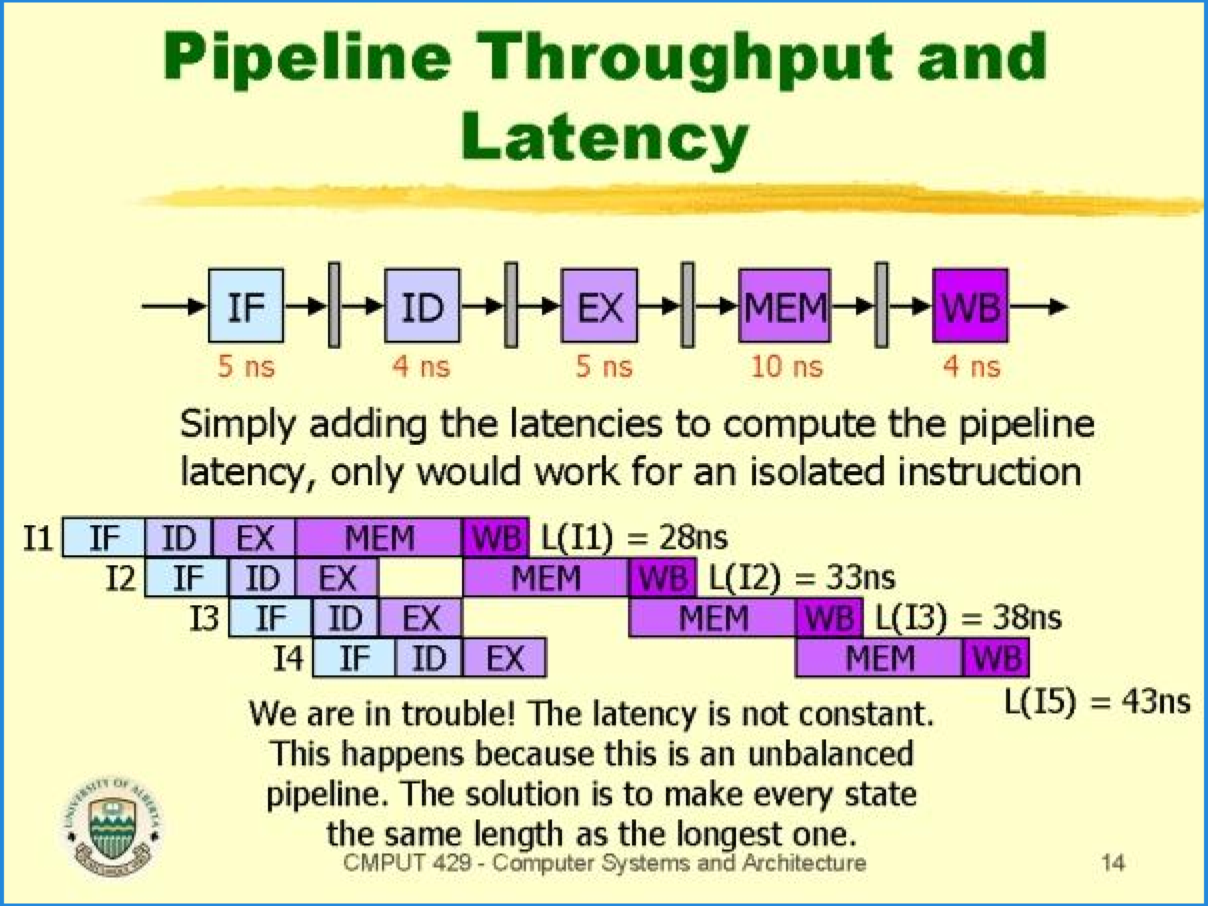
Pipeline analysis

Stage throughput = 1 / (stage latency)

Pipeline throughput = minimum of stage throughputs

Pipeline latency = sum of stage latencies

All of this assumes a stage can handle one packet of data at a time.



Pipeline Throughput and Latency

IF
5 ns
ID
4 ns
EX
5 ns
MEM
10 ns
WB
4 ns

Simply adding the latencies to compute the pipeline latency, only would work for an isolated instruction

I1 IF ID EX MEM WB L(I1) = 28ns
I2 IF ID EX MEM WB L(I2) = 33ns
I3 IF ID EX MEM WB L(I3) = 38ns
I4 IF ID EX MEM WB
L(I5) = 43ns

We are in trouble! The latency is not constant. This happens because this is an unbalanced pipeline. The solution is to make every state the same length as the longest one.

CMPUT 429 - Computer Systems and Architecture 14

Pipeline latency caveat (figure from University of Alberta)

Pipeline throughput analysis

Pipeline throughput

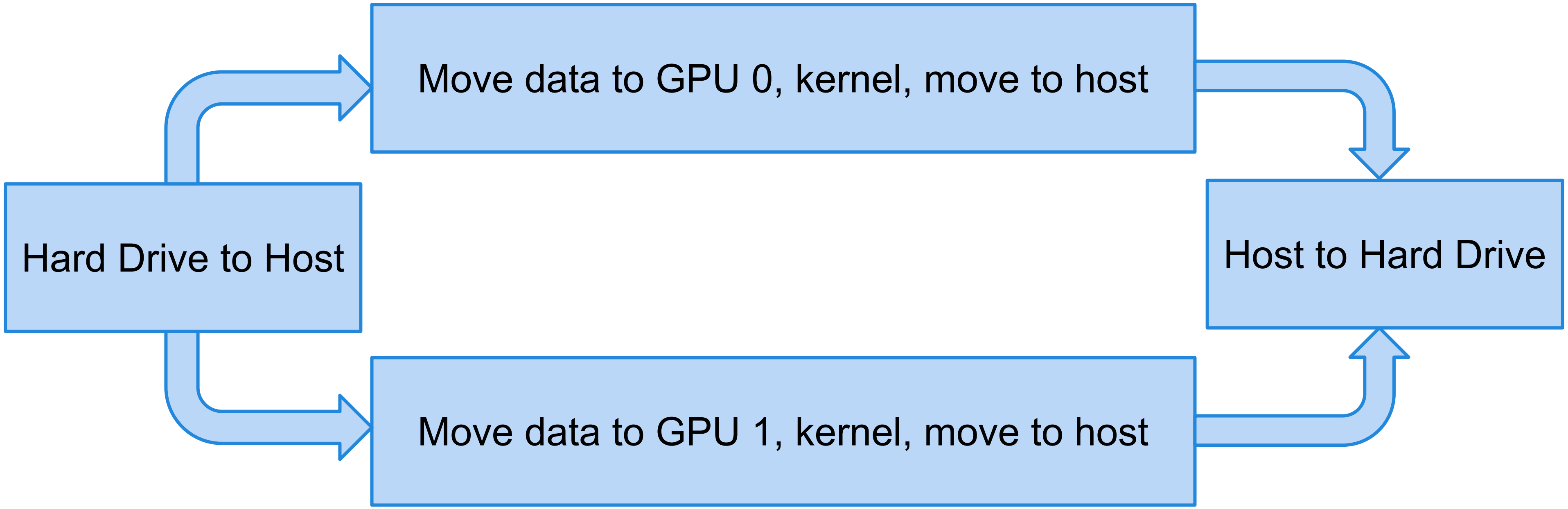
= minimum of stage throughputs

= minimum of (1 / stage latency)

= 1 / (maximum stage latency)

Equal because we assume each stage can only handle one packet of data at a time…

Cheating with data parallelism


Hard Drive to Host
Move data to GPU 0, kernel, move to host
Move data to GPU 1, kernel, move to host
Host to Hard Drive

If you have the hardware and independent computation, remove throughput bottlenecks with data parallelism!

Multiple GPUs

Can put multiple GPUs in a single computer.

CUDA provides interfaces to dispatch work to more than 1 GPU.

haru has 3x GTX 570

Simple interface

`cudaGetDeviceCount` -

how many CUDA capable GPUs?

`cudaSetDevice(int i)` -

execute future commands on GPU `i`

NVIDIA refers to multiple GPUs as “peers”

Data movement

Thanks to unified virtual addressing, you can just use `cudaMemcpy` with `cudaMemcpyDefault` to move data between GPUs.

Actually possible to DMA to one GPU from another and skip the host entirely.

Memcpy breaks concurrency on both GPUs.

Data access

Depending on hardware and motherboard layout, peers can have ability to directly access each other's memory over PCI-E.

`cudaDeviceCanAccessPeer` tells if access is possible

`cudaDeviceEnablePeerAccess` enables peer access.

Peer access example

Peer access is asymmetric

```
// allow device 0 to access device 1 memory
cudaSetDevice(0);
cudaDeviceEnablePeerAccess(1, 0);

// allow device 1 to access device 0 memory
cudaSetDevice(1);
cudaDeviceEnablePeerAccess(0, 0);
```

Peer access use cases & alternative

Peer access use cases are similar to using pinned host memory (both involve all accesses going over PCI-E).

Simpler alternative: use managed memory! Also accessible on host, `cudaMallocManaged`

GPU/GPU synchronization

Problem:

synchronize 2 GPUs without synchronizing full system (all GPUs + CPU)

Solution:

`cudaStreamWaitEvent`. Record an event one 1 GPU and have the other GPUs stream synchronize with it (but not with CPU).

Driving multiple GPUs

2 common options:

- single threaded process
- one thread per GPU

How many threads?

Single thread / process

Pros:

- simple

Cons:

- constantly have to call `cudaSetDevice`

One thread / GPU

Pros:

- call `cudaSetDevice` once per thread
- plays nice with MPI
- can use multiple CPU cores for computation

Cons:

- complex

cuBLAS-XT

NVIDIA’s cuBLAS-XT library takes advantage of the sort of full system parallelism we’ve been talking about.

Input: arbitrarily sized matrices in host memory

Output: matrix product in host memory

Programming multiple GPUs is almost like programming a distributed system. Want to minimize communication.

GPUs across multiple machines

More or less the same as doing scientific computation on a cluster without GPUs.

MPI commonly used.

For some networking hardware (such as Infiniband), it’s possible to DMA data straight from network adapter to GPU. This is expensive territory!

Conclusion

Pipeline parallelism is a great way to think about utilizing all available hardware.

Multiple GPUs can increase throughput through either data or pipeline parallelism.

Both parallelizing and distributing of algorithms requires careful thought about dependencies (or equivalently synchronization and communication).