VSCSE summer school - short course

Introduction to CUDA

Lecture 5
Basic Performance Considerations

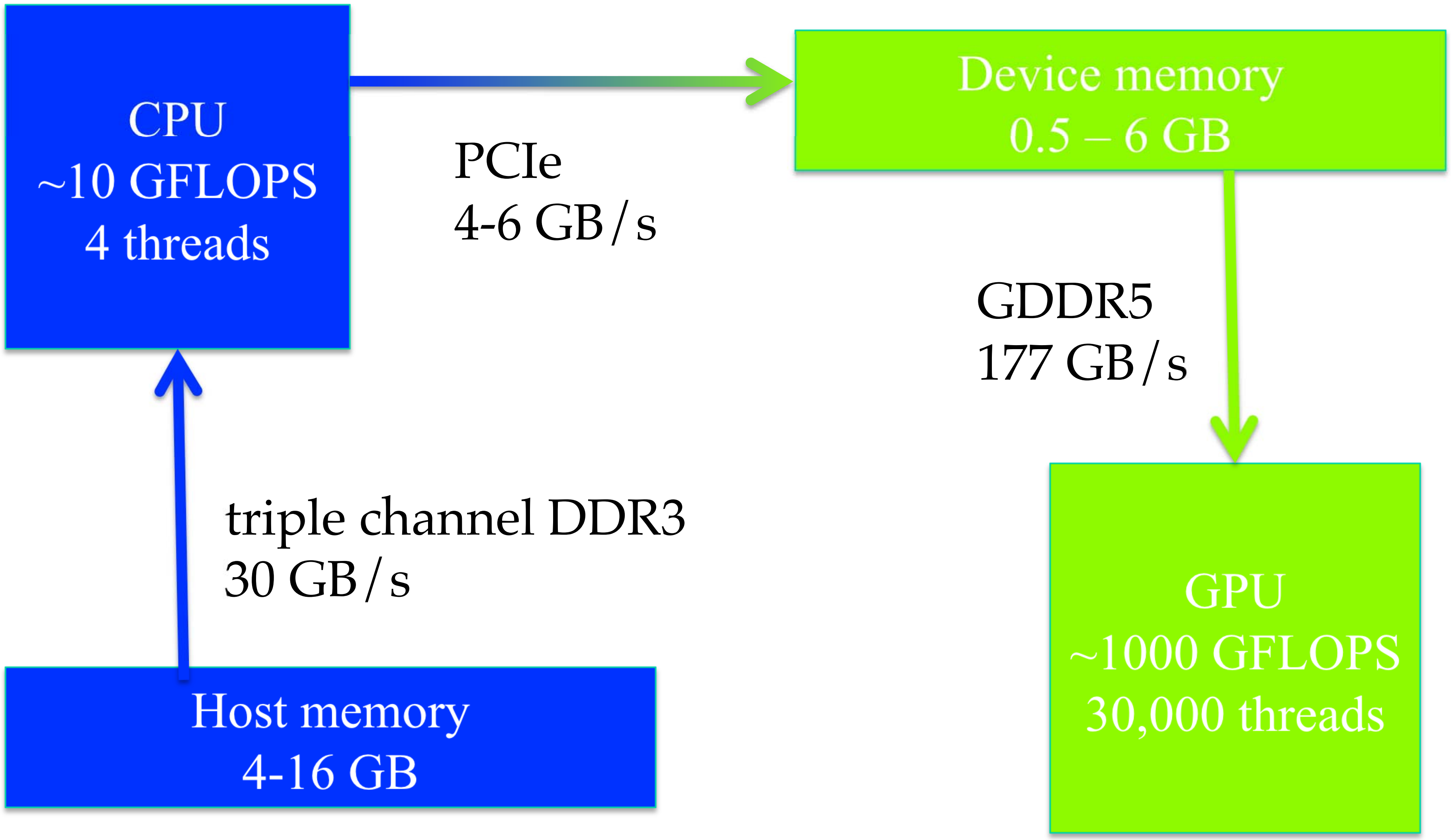
*Based on the NVIDIA CUDA Best Practice Guide

Joshua A. Anderson

Objective

- To learn the basic practices needed to obtain the best performing CUDA code possible.
- To understand the hardware specifications and how they relate to designing optimal algorithms.
- To learn how to model and optimize whole-application performance
- To understand proper methods to benchmark CUDA code

Overview


CPU
~10 GFLOPS
4 threads
PCIe
4-6 GB/s
Device memory
0.5 – 6 GB
GDDR5
177 GB/s
triple channel DDR3
30 GB/s
Host memory
4-16 GB
GPU
~1000 GFLOPS
30,000 threads

Host / device differences

- Host
  - Multicore – capable of several threads of simultaneous execution
  - Thread context switching is expensive
- Device
  - ***Many***core – smallest unit of execution is a warp of 32 threads
  - Over 30,000 threads needed to fully saturate the device
  - Thread context switching is free
- Host and device have separate RAM

Memory copies

- PCI-express is extremely slow (4-6 GB/s) compared to both host and device memory
  - use `cudaHostAlloc()` to attain 4-6 GB/s
- Minimize HtoD and DtoH mem copies
- Must include memcpy times in an analysis of the expected runtime
- Keep data on the device as long as possible
- Executing a non-optimal computation on the GPU may still be faster than copying back to the CPU, performing the operation fast and copying the results back

Maximum possible performance benefit

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

- Amdahl's law
  - P is the fraction of the application parallelized
  - Not really directly applicable to GPUs. N processors are not N times faster than 1 processor
  - Simplify by taking the limit as N tends toward infinity

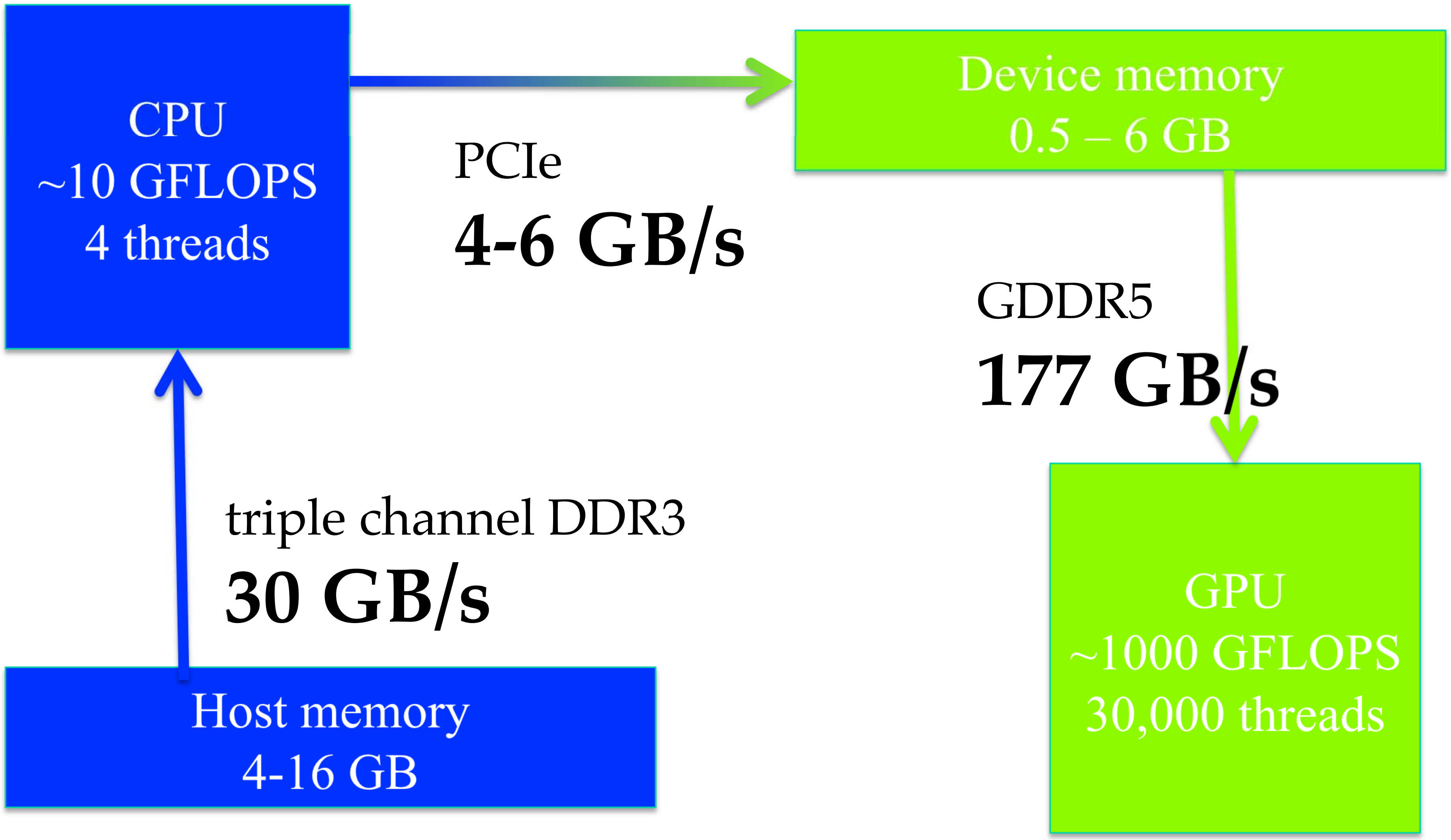
$$S = 1 / 1 - P.$$

  - Example: P=90% => S = 10. Yes, *only 10!*
  - Best possible application of development time is to increase P

Measuring performance

- cudaprof (more in lecture 8) is great for fine tuning individual kernels
- It does little to help you understand how much wall time is spent in various portions of an application
- Measure wall-clock time using
  - `gettimeofday()` in linux/mac
  - `GetSystemTimeAsFileTime()` in windows
- Kernel launches are asynchronous, call **`cudaThreadSynchronize()`** before every wall clock time measurement

Overview


CPU
~10 GFLOPS
4 threads
PCIe
4-6 GB/s
Device memory
0.5 – 6 GB
GDDR5
177 GB/s
triple channel DDR3
30 GB/s
Host memory
4-16 GB
GPU
~1000 GFLOPS
30,000 threads

Bandwidth

- The single most important performance consideration
- Always keep bandwidth in mind with every change made to CUDA code
- Know the theoretical peak bandwidth of the various data links
- Count bytes read/written and compare to the theoretical peak
  - Example: Each thread reads 100 floats and writes 2. 100,000 threads execute in 1ms.
  - (100+2)*sizeof(float)*100000 / 1e-3 = 38 GB/s

Bandwidth - continued

- Utilize the various memory spaces to your advantage
  - Constant, texture, shared, global
- When values are used multiple times in a thread, read once into a register and use multiple times
  - `int val = d_data[idx];`
  - `a += val;`
  - `b -= val;`

Bandwidth - continued

- Coalesce memory reads/writes

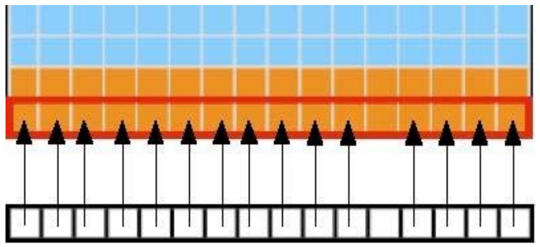


Figure 3.4 Coalesced access in which all threads but one access the corresponding word in a segr

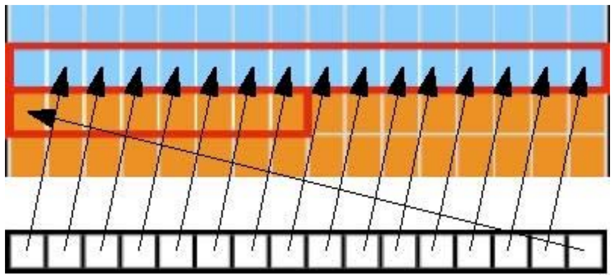
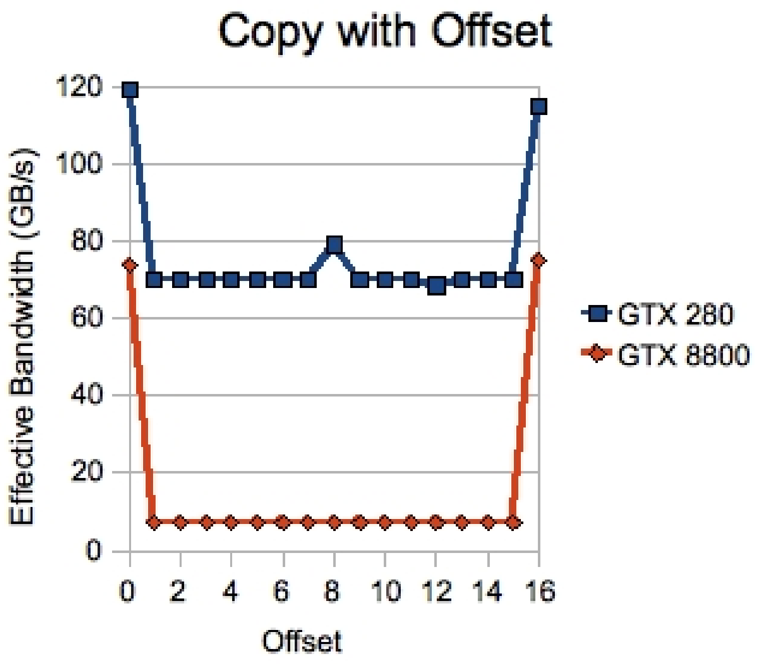


Figure 3.6 Misaligned sequential addresses that fall within two 128-byte segments



*Images from the NVIDIA CUDA Best Practice Guide

When to stop optimizing

- Compare actual memory bandwidth to theoretical (see previous slide)
- Compare actual floating point operation throughput to theoretical (more info in next lecture)
- Determine whether you are bandwidth or computation bound and optimize that portion until you attain near peak levels

(often) lower priority considerations

- Shared memory bank conflicts
- Divergent warps
- Occupancy
- .... though, in some kernels these can be important – examples in the next lecture

Conclusion