Introduction to Computing With Graphics Processors

Cris Cecka
Computational Mathematics
Stanford University

Lecture 1: Introduction to Massively Parallel Computing
Tutorial Goals

- Learn architecture and computational environment of GPU computing
  - Massively Parallel
  - Hierarchical threading and memory space
  - Principles and patterns of parallel programming
  - Processor architecture features and constraints
  - Scalability across future generations

- Introduction to programming in CUDA
  - Programming API, tools and techniques
  - Functionality and maintainability
Moore’s Law (paraphrased)

“The number of transistors on an integrated circuit doubles every two years.”

– Gordon E. Moore
Moore's Law (Visualized)

Transistor Count

GF100

Atom

Pentium

Intel 4004


Serial Performance Scaling is Over

- **Cannot** continue to scale processor frequencies
  - no 10 GHz chips

- **Cannot** continue to increase power consumption
  - can’t melt chip

- **Can** continue to increase transistor density
  - as per Moore’s Law
Why Massively Parallel Processing?

- A quiet revolution and potential build-up

Computation: TFLOPs vs. 100 GFLOPs

- GPU in every PC – massive volume & potential impact
Why Massively Parallel Processing?

- A quiet revolution and potential build-up
  - Bandwidth: ~10x
  - GPU in every PC – massive volume & potential impact

![Graph](chart.png)

- NVIDIA GPU
- Intel CPU

---

- 3GHz Dual Core P4
- 3GHz Core2 Duo
- 3GHz Xeon Quad
- T12
- Westmere

---

The “New” Moore’s Law

• Computers no longer get faster, just wider

• You must re-think your algorithms to be parallel!
  - Not only parallel, but hierarchically parallel...

• Data-parallel computing is most scalable solution
  - Otherwise: refactor code for 2 cores
  - You will always have more data than cores – build the computation around the data
Enter the GPU

- Highly Scalable
- Massively parallel
  - Hundreds of cores
  - Thousands of threads
- Cheap
- Available
- Programmable
GPU Evolution

- **High throughput** computation
  - GeForce GTX 280: 933 GFLOP/s
- **High bandwidth** memory
  - GeForce GTX 280: 140 GB/s
- **High availability** to all
  - 180+ million CUDA-capable GPUs in the wild
Graphics Legacy

• **Throughput** is paramount
  – Must paint every pixel within frame time
  – Scalability

• Create, run, & retire **lots of threads** very rapidly
  – Measured 14.8 Gthread/s on `increment()` kernel

• Use **multithreading** to hide latency
  – 1 stalled thread is OK if 100 are ready to run
Why is this different from a CPU?

- **CPU:** *minimize latency* experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic

- **GPU:** *maximize throughput* of all threads
  - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency => skip the big caches

- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not
SM Multiprocessor

- 32 CUDA Cores per SM (512 total)
- 8x peak FP64 performance
  - 50% of peak FP32 performance
- Direct load/store to memory
  - Usual linear sequence of bytes
  - High bandwidth (Hundreds GB/sec)
- 64KB of fast, on-chip RAM
  - Software or hardware-managed
  - Shared amongst CUDA cores
  - Enables thread communication
Key Architectural Ideas

- GPU serves as a coprocessor to the CPU
  - has its own device memory on the card

- **SIMT** (Single Instruction Multiple Thread) execution
  - threads run in groups of 32 called **warps**
  - threads in a warp share instruction unit (IU)
  - HW automatically handles divergence

- Hardware multithreading
  - HW resource allocation & thread scheduling
  - HW relies on threads to hide latency

- Threads have all resources needed to run
  - any warp not waiting for something can run
  - context switching is (basically) free
Enter CUDA

- A compiler and toolkit for programming NVIDIA GPUs
- Minimal extensions to familiar C/C++ environment
  - let programmers focus on parallel algorithms
- Scalable parallel programming model
  - Express parallelism and control hierarchy of memory spaces
  - But also uses a high level abstraction from hardware
- Provide straightforward mapping onto hardware
  - good fit to GPU architecture
  - maps well to multi-core CPUs too
Motivation
Compute Environment

- **Threads** executed by SP
  - Executing the same sequential **kernel**
  - On-chip registers, Off-ship local memory

- **Thread blocks** executed by SM
  - Threads in the same block can cooperate
  - On-ship shared memory
  - Synchronization

- **Grids of blocks** executed by device
  - Off-chip global memory
  - No synchronization
CUDA Model of Parallelism

• CUDA virtualizes the physical hardware
  – thread is a virtualized scalar processor (registers, PC, state)
  – block is a virtualized multiprocessor (threads, shared mem.)

• Scheduled onto physical hardware without pre-emption
  – threads/blocks launch & run to completion
  – blocks should be independent
NOT: Flat Multiprocessor

- Global synchronization isn’t cheap
- Global memory access times are expensive

- cf. PRAM (Parallel Random Access Machine) model
NOT: Distributed Processors

- Distributed computing is a different setting
- cf. BSP (Bulk Synchronous Parallel) model, MPI
A Common Programming Strategy

- Global memory resides in device memory (DRAM)
  - Much slower access than shared memory
- Tile data to take advantage of fast shared memory:
  - Generalize from adjacent_difference example
  - Divide and conquer
A Common Programming Strategy

- Global memory resides in device memory (DRAM)
  - Much slower access than shared memory
- Tile data to take advantage of fast shared memory:
  - Generalize from adjacent_difference example
  - Divide and conquer
A Common Programming Strategy

Partition data into subsets that fit into shared memory
A Common Programming Strategy

Handle each data subset with one **thread block**
A Common Programming Strategy

Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
Perform the computation on the subset from *shared* memory.
A Common Programming Strategy

Copy the result from *shared memory* back to global memory