CSE 599 I Accelerated Computing - Programming GPUS

Course Introduction

Administrivia

Office hours:

Tentatively Thursday 1-3, or by appointment

Grading:

50% programming assignments, 50% final project

Textbook (very optional):

Programming Massively Parallel Processors, Third Edition: A Hands-on Approach

David B. Kirk and Wen-mei W. Hwu.

I can provide students with a code for a 30% discount on the textbook from Elsevier.

Computing resources:

Students will need access to a CUDA-capable device (I can help with this)

GPU Teaching Kit

Accelerated Computing

Lecture 1.1 – Course Introduction

Course Introduction and Overview

Course Goals

- Learn how to program heterogeneous parallel computing systems and achieve
  - High performance and energy-efficiency
  - Functionality and maintainability
  - Scalability across future generations
  - Portability across vendor devices
- Technical subjects
  - Parallel programming API, tools and techniques
  - Principles and patterns of parallel algorithms
  - Processor architecture features and constraints

People

- Wen-mei Hwu (University of Illinois)
- David Kirk (NVIDIA)
- Joe Bungo (NVIDIA)
- Mark Ebersole (NVIDIA)
- Abdul Dakkak (University of Illinois)
- Izzat El Hajj (University of Illinois)
- Andy Schuh (University of Illinois)
- John Stratton (Colgate College)
- Isaac Gelado (NVIDIA)
- John Stone (University of Illinois)
- Javier Cabezas (NVIDIA)
- Michael Garland (NVIDIA)

Outline

- Course Introduction
- Intro to CUDA C
- CUDA parallelism model
- Memory and data locality
- Thread execution / computational efficiency
- Memory performance
- Parallel patterns:
  - Stencil (convolution)
  - Prefix sum (aka scan)
  - Histogram
  - Sparse matrices
  - Graph search
- Floating point considerations
- Dynamic parallelism / recursion
- GPU as part of a heterogeneous system
- Case studies
- ???

GPU Teaching Kit

Accelerated Computing

Lecture 1.2 – Course Introduction

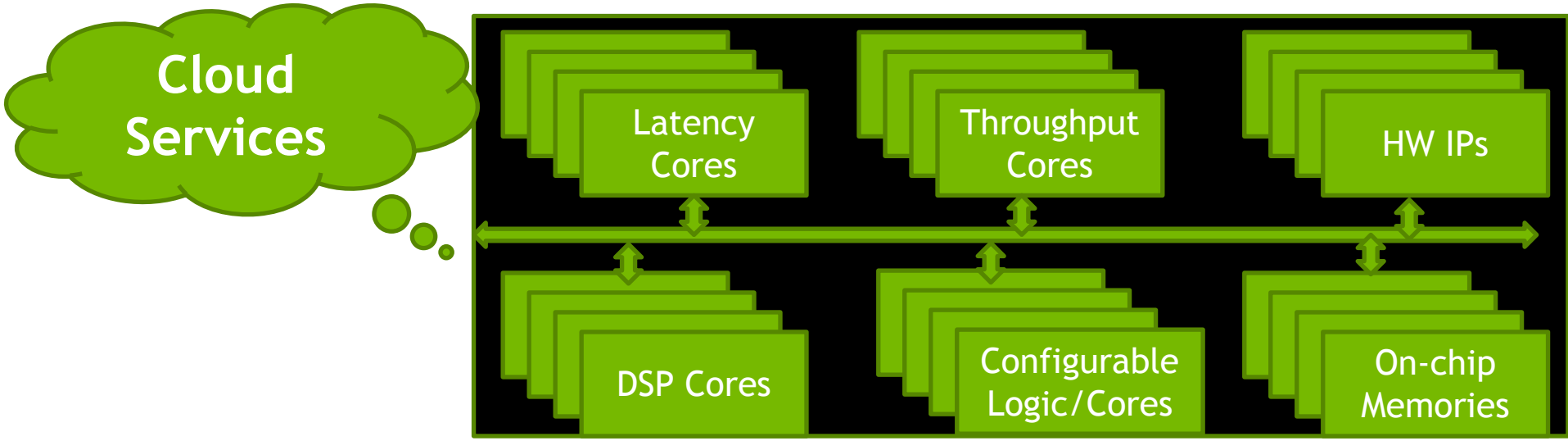
Introduction to Heterogeneous Parallel Computing

Objectives

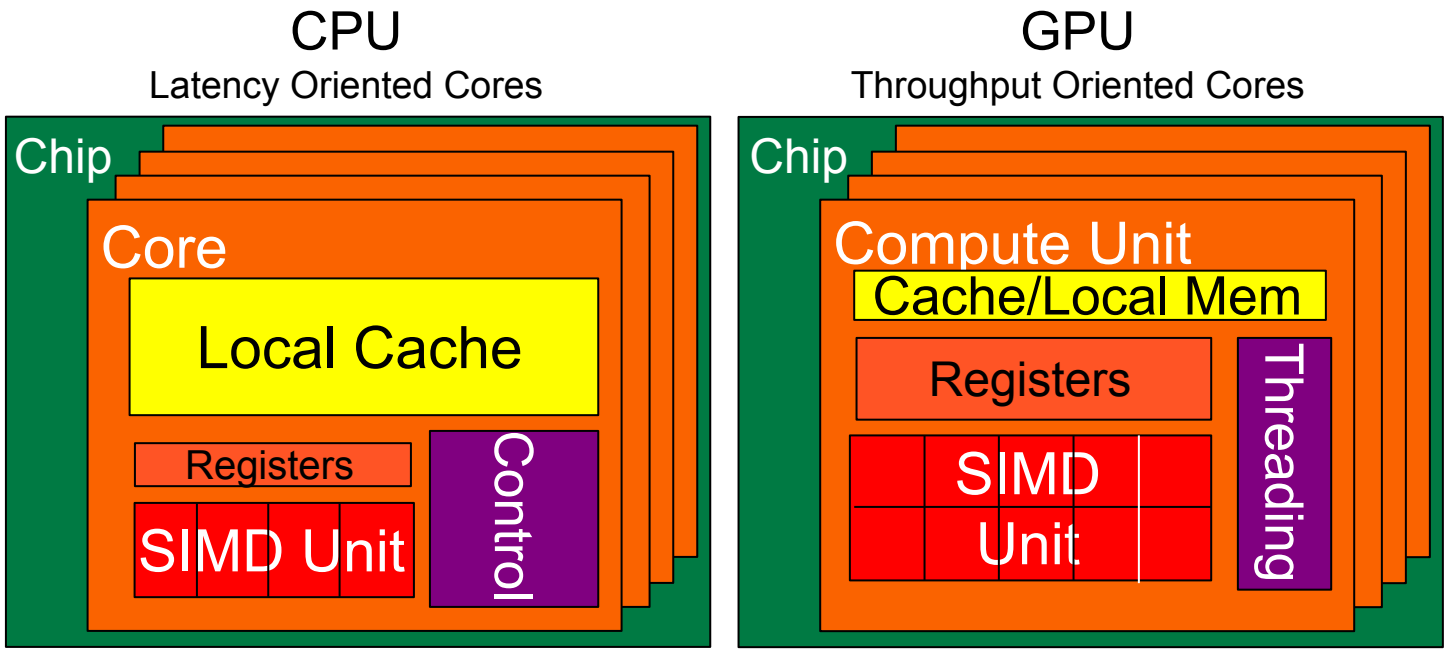
- To learn the major differences between latency devices (CPU cores) and throughput devices (GPU cores)
- To understand why winning applications increasingly use both types of devices

Heterogeneous Parallel Computing

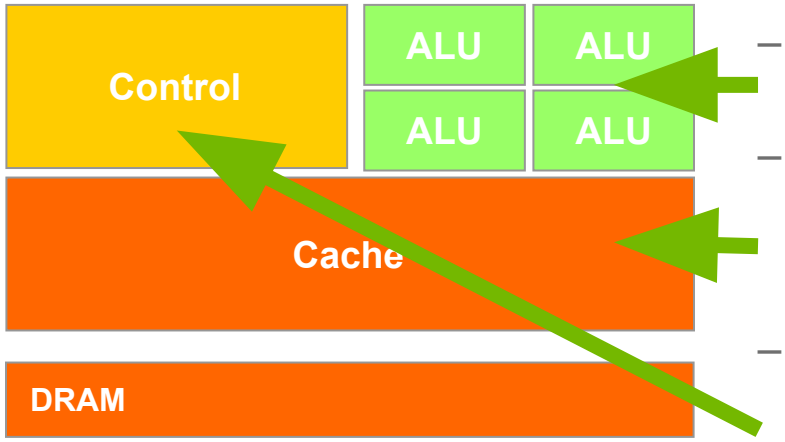
– Use the best match for the job (heterogeneity in mobile SOC)



CPU and GPU are designed very differently

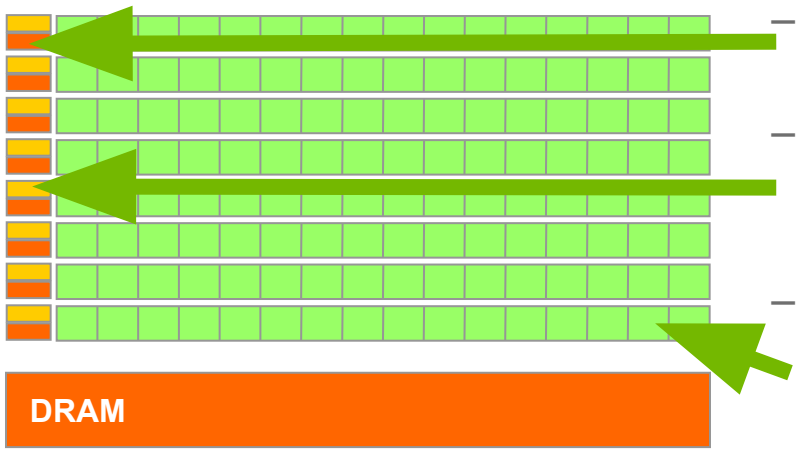


CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

GPUs: Throughput Oriented Design



- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10X+ faster than GPUs for sequential code

- GPUs for parallel parts where throughput wins
  - GPUs can be 10X+ faster than CPUs for parallel code

Core i7-6950X:
~300 GFLOPS

nVidia Titan X (Pascal):
~11,000 GFLOPS

GEFORCE GTX 1050

GPU Engine Specs:

	1050 Ti	1050
NVIDIA CUDA® Cores	**768**	**640**
Base Clock (MHz)	**1290**	**1354**
Boost Clock (MHz)	**1392**	**1455**

Memory Specs:

Memory Speed	**7 Gbps**	**7 Gbps**
Standard Memory Config	**4 GB GDDR5**	**2 GB GDDR5**
Memory Interface Width	**128-bit**	**128-bit**
Memory Bandwidth (GB/sec)	**112**	**112**

GEFORCE GTX 1080

GPU Engine Specs:

NVIDIA CUDA® Cores	**2560**
Base Clock (MHz)	**1607**
Boost Clock (MHz)	**1733**

Memory Specs:

Memory Speed	**10 Gbps**
Standard Memory Config	**8 GB GDDR5X**
Memory Interface Width	**256-bit**
Memory Bandwidth (GB/sec)	**320**

NVIDIA TITAN X

GPU Engine Specs:

NVIDIA CUDA® Cores	**3584**
Base Clock (MHz)	**1417**
Boost Clock (MHz)	**1531**

Memory Specs:

Memory Speed	**10 Gbps**
Standard Memory Config	**12 GB GDDR5X**
Memory Interface Width	**384-bit**
Memory Bandwidth (GB/sec)	**480**

Heterogeneous Parallel Computing in Many Disciplines

Financial Analysis

Scientific Simulation

Engineering Simulation

Data Intensive Analytics

Medical Imaging

Digital Audio Processing

Digital Video Processing

Computer Vision

Biomedical Informatics

Electronic Design Automation

Statistical Modeling

Numerical Methods

Ray Tracing Rendering

Interactive Physics

GPU Teaching Kit

Accelerated Computing

Lecture 1.3 – Course Introduction

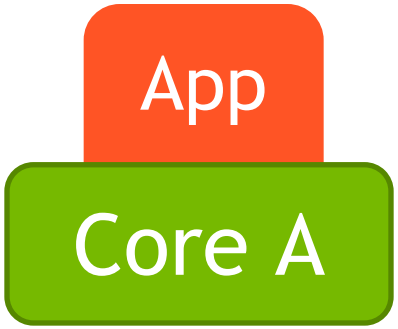
Scalability in Heterogeneous Parallel Computing

Objectives

– To understand the importance and nature of scalability in parallel programming

Keys to Software Cost Control

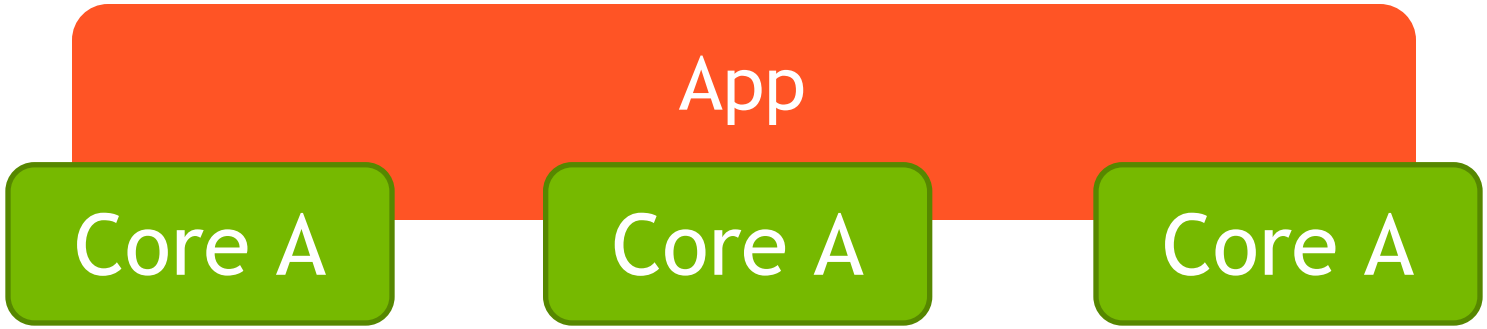
- Scalability



Keys to Software Cost Control

- Scalability
  - **The same application runs efficiently on new generations of cores**

Keys to Software Cost Control



- Scalability
  - The same application runs efficiently on new generations of cores
  - **The same application runs efficiently on more of the same cores**

More on Scalability

- Performance growth with HW generations
  - Increasing number of compute units (cores)
  - Increasing number of threads
  - Increasing vector length
  - Increasing pipeline depth
  - Increasing DRAM burst size
  - Increasing number of DRAM channels
  - Increasing data movement latency

GPU Teaching Kit

Accelerated Computing

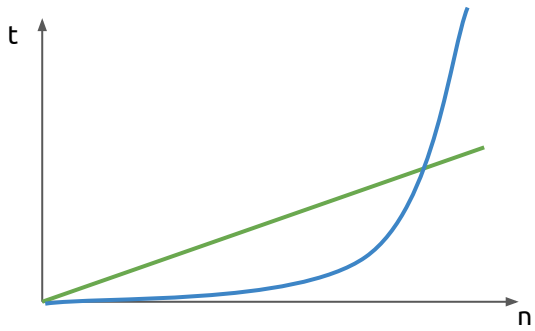
What is CUDA?

- A set of C language extensions
  - Requires a separate compiler (nvcc)
- A runtime API
- Development tools

When to use CUDA?

- When the program contains portions that are parallelizable
  - Task parallel or data parallel
- When parallelizable portions make up a significant amount of runtime:
  - If CUDA is used to achieve a 100X speedup of a portion of an application that accounted for 30% of runtime, the total applications speedup will be ~1.4X.
- When there is enough work to justify the overhead
  - Typically you want 5000+ active threads

Why is GPU programming hard?



- One must take care not to increase the computational complexity
- Algorithms are very often memory-bound rather than compute-bound
- Running times can be much more sensitive to dynamic input values
- Familiar sequential patterns such as recursion can map to very non-intuitive parallel patterns

Why learn CUDA?

Can’t I just use somebody else’s GPU-accelerated library?

- You may need features that the library doesn’t support
- It is still helpful to know what is happening at lower levels

Can’t I just wait until parallelization is handled automatically by the compiler?

- By some estimates, this could be ten years or more in the future

Why learn CUDA (specifically)?

nVidia has an installation base of about 1 billion CUDA-capable devices

Most concepts in this course generalize beyond CUDA (e.g. to OpcenCL)