CSE 4373/5373 - General Purpose GPU Programming

Course Introduction

Alex Dillhoff

University of Texas at Arlington

Course Overview

Goals

- Learn to program GPUs.
- Learn to think in parallel.
- Utilize profiling tools to optimize code.
- Study complex problems that introduce new parallel patterns.

Course Overview

This course is split into three sections.

1. Fundamentals
2. Parallel Patterns
3. Case Studies

Fundamentals

- Basic CUDA C++ Programming
- Introductory parallel patterns
- Familiarity with profiling tools

Parallel Patterns

- Convolution
- Stencil
- Reduction
- Prefix Sum
- Histogram
- and more...

Case Studies

- MRI Reconstruction
- Deep Learning
- Electrostatic Potential Map
- Improvements in Attention Mechanisms

Heterogeneous Parallel Computing

Heterogeneous Parallel Computing

Heterogeneous computing refers to systems that use more than one kind of processor or cores.

In this class, we will use both CPUs and GPUs.

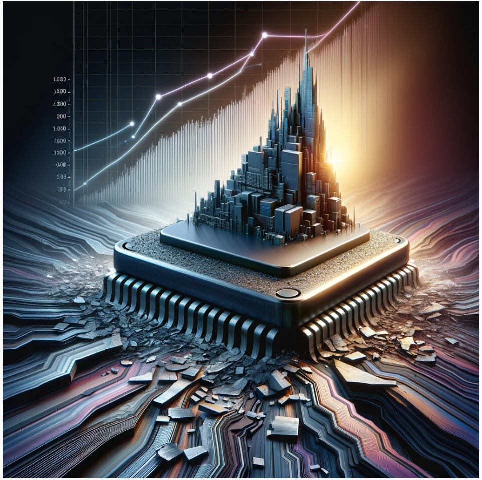
Heterogeneous Parallel Computing

Not every task can be parallelized.

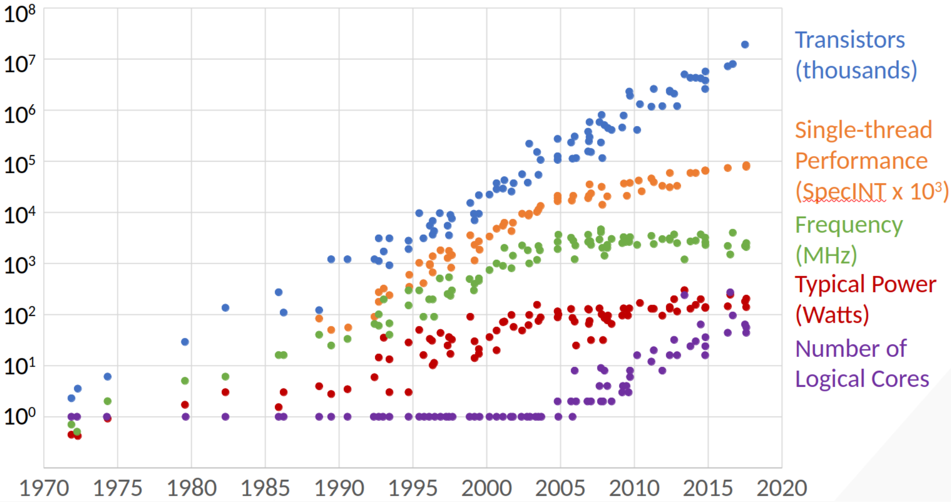
- Some tasks are inherently sequential.
- Parallelism could refer to the data *or* the task.
- By studying patterns and problems, you will be able to identify opportunities for parallelism.

Heterogeneous Parallel Computing

- 30 years ago, most devices were single-core.
- Performance increases on a single core hit a physical limit.
- Multi-core CPUs broke the barrier and continued Moore's Law.



Processor Trends



Source: M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, C. Batten (1970-2010). K. Rupp (2010-2017).

Heterogeneous Parallel Computing

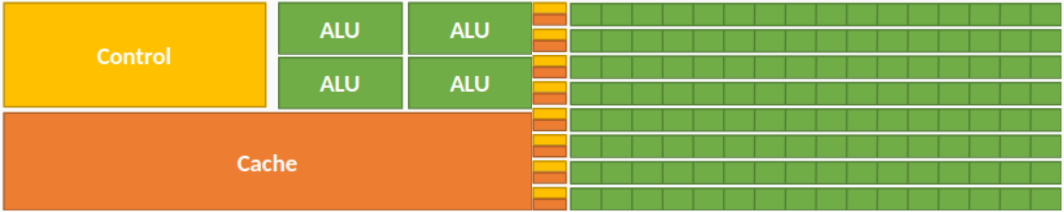
- Today, transistors sizes are approaching the atomic scale.
- Single chips are again hitting a physical limit.
- Multi-core CPUs are still improving, but not as fast as before.
- The solution? Horizontal scaling

Heterogeneous Parallel Computing

- Our focus is *not* on distributed computing.
- We will focus on parallelism within a single machine.
- This will require harmony between the CPU and GPU.
- First, we must understand the differences between the two.

Latency versus Throughput

CPU: Latency-Oriented **GPU: Throughput-Oriented**



Latency versus Throughput

- CPUs follow a latency-first design.
- Much of the space is dedicated to caching and branch prediction.
- This allows for general purpose computing and fast context switching.

Latency versus Throughput

- GPUs follow a throughput-first design.
- Much of the space is dedicated to arithmetic units.
- This allows for fast parallel computation.
- The high latency of memory access is hidden by the large number of threads.

GPUs and Supercomputers

GPUs are featured in many of the top 500 supercomputers.

Name	CPUs	GPUs	Peak PFLOP/s
El Capitan	1,051,392 cores	43,808 AMD Instinct MI300A	2,746.38
Frontier (Oak Ridge NL)	606,208 cores	37,888 AMD MI250X	2,055.72
Aurora (Argonne NL)	1,100,000 cores (est.)	63,744 Intel GPU Max	1,980.01
Eagle (Microsoft Azure)	1,123,200 (combined)	Unknown Split (NVIDIA H100)	846.74
HPC6	222,208 cores	13,888 AMD MI250X	606.97

A Brief History of GPU Programming

- GPUs were originally designed for graphics.
- Many vertices and pixels can be processed in parallel in a straightforward way.
- The first GPU programming languages were shader languages (OpenGL, DirectX).
- These languages were designed for graphics, not general purpose computing.
- Everything was accomplished through hacking the pixel shaders.

A Brief History of GPU Programming

- In 2006, NVIDIA released the GeForce 8800 GTX, the first CUDA-capable GPU.
- CUDA refers to the architecture as well as the programming model.
- CUDA allows for general-purpose GPU programming via the unified shader pipeline.
- This means that each ALU can be utilized for any task.

A Brief History of GPU Programming

- The ALUs have access to global memory as well as shared memory, managed by software.
- The hardware itself has gone through many iterations since 2006.
- We will study the architecture, as its understanding is critical in optimizing code.

A Brief History of GPU Programming

- OpenCL, an open standard for heterogeneous computing, was released in 2009.
- It is supported by NVIDIA, AMD, and Intel.
- It is similar to CUDA, but is not tied to a specific hardware architecture.

Applications

Applications

- We are in the midst of a data explosion.
- Many firms are collecting data at an unprecedented rate.
- This data must be processed and analyzed.
- Scaling up to large datasets requires parallelism.

Linear Algebra Libraries

- Many linear algebra libraries are GPU-accelerated.
- cuBLAS, cuSPARSE, cuSOLVER, cuRAND, cuFFT, etc.
- These libraries are highly optimized and can be used in your code.
- We will study the algorithms behind these libraries.

Machine Learning

- The success of Deep Learning has been driven by GPUs.
- Model training and optimization is a perfect candidate for data parallelism.
- Large models require a massive amount of data.
- Specialized optimization algorithms can execute functions in parallel.
- We will study cuDNN, a GPU-accelerated deep learning library.

Computer Vision

- Computer vision is a field that has been revolutionized by GPUs and deep learning.
- Convolutional Neural Networks (CNNs) are used to process images.
- CNNs are highly parallelizable.
- Studying the core operation behind them will unlock new parallel patterns.

Other Applications

- Computational Chemistry
- Financial Analysis
- Medical Imaging
- Digital Audio/Video Processing
- Statistical Modeling
- Numerical Methods
- Ray Tracing
- Interactive Physics
- and more...

What to expect from this course

What to expect from this course

- This course is hands-on.

What to expect from this course

- This course is hands-on.
- Almost every topic will be accompanied by a programming exercise.

What to expect from this course

- This course is hands-on.
- Almost every topic will be accompanied by a programming exercise.
- Most of these exercises will be included in the assignments.

What to expect from this course

- This course is hands-on.
- Almost every topic will be accompanied by a programming exercise.
- Most of these exercises will be included in the assignments.
- Questions and quizzes will help focus your learning.

What to expect from this course

By the end of this course, you should have the following skills:

- Familiarity with the CUDA programming model.

What to expect from this course

By the end of this course, you should have the following skills:

- Familiarity with the CUDA programming model.
- Ability to think in parallel.

What to expect from this course

By the end of this course, you should have the following skills:

- Familiarity with the CUDA programming model.
- Ability to think in parallel.
- Identify sections of code that can be parallelized.

What to expect from this course

By the end of this course, you should have the following skills:

- Familiarity with the CUDA programming model.
- Ability to think in parallel.
- Identify sections of code that can be parallelized.
- Implementation of parallel solutions.

What to expect from this course

By the end of this course, you should have the following skills:

- Familiarity with the CUDA programming model.
- Ability to think in parallel.
- Identify sections of code that can be parallelized.
- Implementation of parallel solutions.
- Debugging and profiling of parallel code.

What to expect from this course

By the end of this course, you should have the following skills:

- Familiarity with the CUDA programming model.
- Ability to think in parallel.
- Identify sections of code that can be parallelized.
- Implementation of parallel solutions.
- Debugging and profiling of parallel code.
- Measure and analyze performance.

First Lesson: Measuring Speedup

- System A takes T_A time to complete a task.
- System B takes T_B time to complete the same task.
- The speedup of system B over system A is defined as:

$$S = \frac{T_A}{T_B}$$

- If $S > 1$, then system B is faster.
- If $S < 1$, then system A is faster.

Adding in parallelization

Consider an application where

- t is the sequential execution time,
- p is the fraction of execution that is parallelizable, and
- s is the speedup of the parallelized portion.

Adding in parallelization

What is the overall speedup of the application?

Adding in parallelization

What is the overall speedup of the application?

$$
\begin{aligned}
t_{parallel} &= (1 - p) * t + \frac{p * t}{s} \\
&= \left(1 - p + \frac{p}{s}\right) * t \\
speedup &= \frac{t_{sequential}}{t_{parallel}} \\
&= \frac{t}{1 - p + \frac{p}{s} * t}
\end{aligned}
$$

Adding in parallelization

Simplified...

$$
\begin{aligned}
speedup &= \frac{t}{1 - p + \frac{p}{s} * t} \\
&= \frac{1}{1 - p + \frac{p}{s}}
\end{aligned}
$$

Adding in parallelization

As the speedup of the parallel portion approaches infinity...

$$\mathit{speedup} = \frac{1}{1 - p + \frac{p}{s}} \xrightarrow[s \to \infty]{} \frac{1}{1 - p}$$

Amdahl's law revealed

Amdahl's law states that the maximum speedup of an application is limited by the sequential portion.

$$\mathit{speedup} < \frac{1}{1 - p}$$

If $p = 0.9$, then the maximum speedup is 10x, no matter how many processors are used.