Introduction to GPUs

Instructor and Course Introduction

Introduction

1. This module aims to introduce students to the fundamental principles of GPU architecture and programming. Additionally, it offers an opportunity for students to revisit processor design backgrounds as part of their learning experience.

Objectives

1. Describe the basic backgrounds of GPU
2. Explain the basic concept of data parallel architectures

Course Topics

1. GPU programming, hardware (architecture), compiler, and programming aspects
2. CUDA programming assignments
3. Programming assignments to help understand architecture and compiler issues
4. Reading assignments to learn the latest materials

Course Learning Objectives

1. Describe how GPU architecture works and be able to optimize GPU programs
2. Conduct GPU architecture or compiler research
3. Explain the latest advancements in hardware accelerators
4. Build a foundation for understanding the latest CUDA features

Research Areas

1. Heterogenous architecture:
    - CPU+GPU, CPU+PIM, security on GPU, ML optimizations on embedded computing, reliability of GPUs, processors for autonomous agents
2. Computer architecture and compiler, and systems together

Course Prerequisites

1. Basic 5-stage CPU pipeline
2. C++/Python programming skillset
3. Prior GPU programming experience is not necessary

What is a GPU?

1. 3D graphics accelerator
2. Massive parallel processors
3. ML accelerators
4. High performance processors

Differences Between CPUs and GPUs

	CPU	GPU
Target Applications	Latency sensitive applications	Throughput sensitive applications
Support precise exceptions?	Yes	No
Host-accelerator	Host	Accelerator
ISA	Public or open	Open/Closed

	CPU	GPU
Programming Model	SISD/SIMD	SPMD

Modern Processor Paradigms

Learning Objective

1. Describe CPU design and techniques to improve performance one CPUs

Review: Processor Design

1. 5-stage pipeline
    - Fetch stage: Instruction fetch (from I-cache)
    - Decode stage: Instruction becomes multiple uops; generates control signals for the rest of the pipelines
    - Schedule:
        - In-order scheduler: instructions are executed in program order
        - Out of order scheduler: Ready instructions are executed
    - Execution: Actual computation or access memory
    - Write-back: Update memory, register

Increasing Parallelism - Superscalar

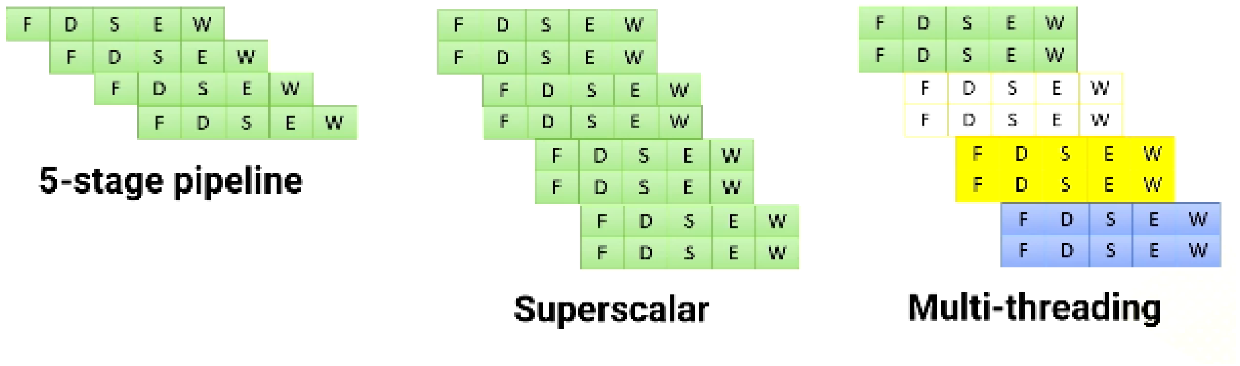
1. Superscalar: Each pipeline stage handles more than one instruction
    - Increasing instruction-level parallelism
    - Instruction per cycle (IPC) > 1
2. Instruction-level Parallelism
    - If a machine has 2 execution units, how many instructions can be executed simultaneously?
        - R1 = R2 + R3; R4 = R5 + R6;
            - Can be executed together
        - R1 = R2 + R3; R4 = R1 + R5;
            - Can't be executed together
    - Out of order processor helps to improve ILP -> find more independent instructions to be executed
    - Improving ILP has been one of the main focuses in CPU designs

Increasing CPU Performance

1. Deeper pipelines require a better branch predictor
2. Large caches: Caches reduce memory access latency
    - Larger caches reduce the number of cache misses

Increasing Parallelism - Multithreading

1. Multi-threading: Multiple HW threads, each with their own PC and register file



Superscalar Processor

Multi-Processors

1. Both GPUs and many-core CPUs aim to increase parallelism

Summary

1. CPUs increase performance by increasing ILP, frequency, and the number of cores in a system

How to Write Parallel Programs

Objectives

1. Describe different parallel programming paradigms
2. Recognize Flynn's classical taxonomy
3. Explore a CUDA program example

Amdahl's Law

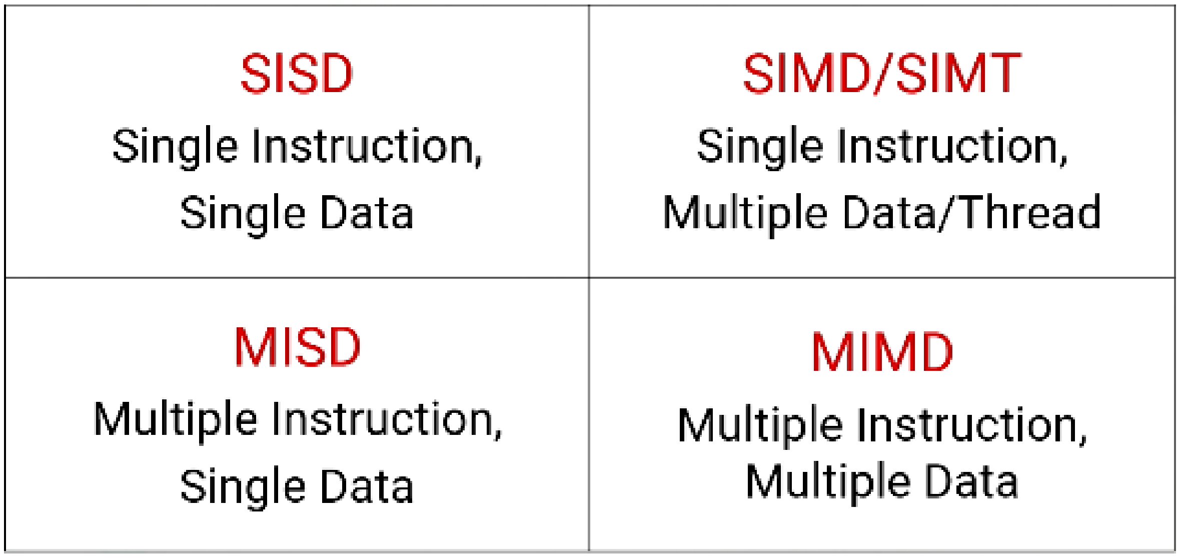
1. Speedup = Performance for the entire task using the enhancement when possible
    - Speedup = 1 / (P/N + S)
        - P = parallel fraction (1 - S)
        - N = number of processors (2, 4, 8…)
        - S = serial fraction
    - If serial fraction is small (S close to 0), performance improvement is proprtional to N

What to Consider for Parallel Programming

1. Increase parallelism by writing parallel programs
2. Let's think of an array addition and find min/max value
    - Can split tasks across processors (each processor gets A)
    - Can also split array in half (requires a reduction)

Flynn's Classical Taxonomy

1. CPUs can be SISD, SIMD, and MIMD
2. GPUs are typically SIMD



SISD Single Instruction, Single Data	SIMD/SIMT Single Instruction, Multiple Data/Thread
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

Flynn's Classical Taxonomy

SPMD Programming

1. Single Program Multiple Data programming
2. All cores (threads) are performing the same work (based on same program). But they are working on different data
3. Data decomposition is the typical programming pattern of SPMD
4. SPMD is typical GPU programming pattern and the hardware's execution model is SIMT

CUDA Program Example

1. All cores (threads) are executing the same vector_sum

```
vector_sum()
{
    index; //differentiate by thread ids
    sum += A[index];
|
```

Summary

1. Reviewed Flynn's classical taxonomy
2. GPU programming uses SPMD programming style which utilizes SIMT hardware

Introduction to GPU Architecture

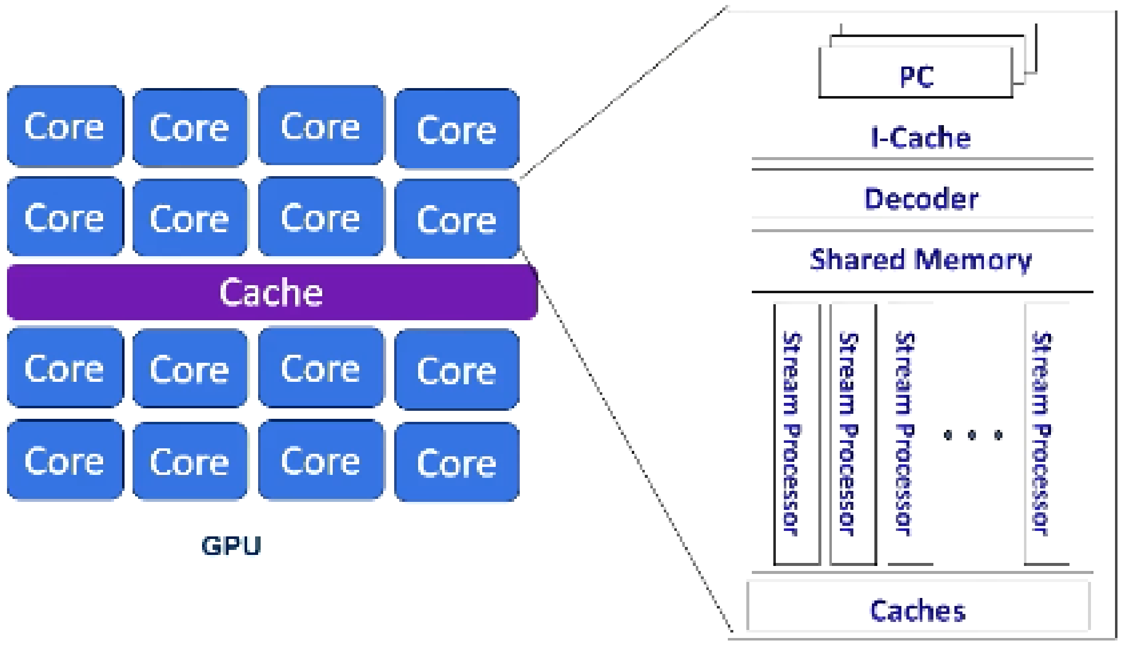
Objectives

1. Gain a high-level understanding of GPU architecture
2. Describe key terms including "streaming multiprocessors", "warp", and "wave-front"

GPU Architecture Overview

1. Each core can execute multiple threads
2. CPU cores are referred to as "streaming multiprocessors (SM)" in NVIDIA term or SIMT multiprocessors

3. Stream processors (SP) are ALU units, SIMT lane or cores on GPUs
4. In this course, core != stream processor



GPU Architecture

GPU Pipeline

1. Fetch
   - One instruction for each warp
   - Multiple PC registers exist to support multi-threaded architecture
   - Round-robin scheduler
   - Greedy scheduler: Switch warps on I-cache miss or branch
2. Decode
3. Register read
4. Scheduler (score boarding)
5. Execution (SIMD)
6. Writeback

Execution Unit: Warp/Wave-front

1. Warp/wave-front is the basic unit of execution
   - A warp/wave-front is a group of threads (e.g., 32 threads for the Tesla GPU architecture)
   - For a given instruction, when all of the data sources for an entire warp are ready, it is executed

Summary

1. Covered an overview of GPU architecture
2. Introduced terms like:
   - SM: Streaming Multiprocessor
   - SP: Streaming Processor

- Warp/wavefront

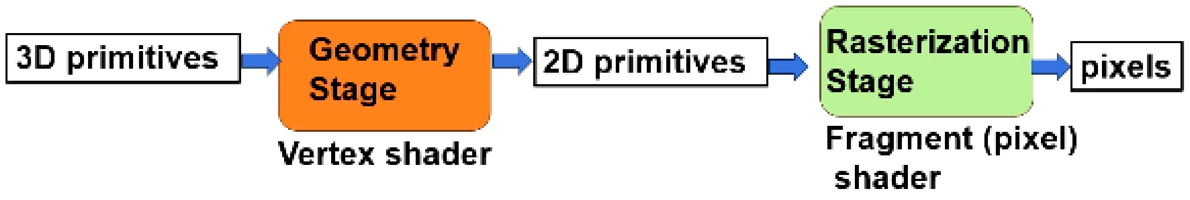
History/Trend of GPU Architectures

Objectives

1. Describe the evolution history of GPUs
2. Describe the evolution of traditional 3D graphics pipelines

Traditional 3D Graphics Pipeline

1. Traditional pipelined GPUs: separate vertex processing and pixel processing units
2. Programmable GPUs: Introduction of programmable units for vertex/pixel shader
3. Vertex/pixel shader progression to CUDA programmable cores (GPGPU)
4. Modern GPUs: Addition of ML accelerator components (e.g., tensor cores), programmable features



Graphics Pipeline

Programmable GPU Architecture Evolution

1. Cache hierarchies (L1, L2, etc.)
2. Extend FP 32 bits to FP 64 bits to support HPC applications
3. Integration of atomic and fast integer operations to support more diverse workloads
4. Utilization of high bandwidth memory
5. Addition of smaller floating point formats (FP16) to support ML workloads
6. Incorporation of tensor cores to support ML workloads
7. Integration of transformer cores to support transformer ML workloads

NVIDIA H100

1. FP32, FP64, FP16, INT8, and FP8 format
2. Tensor cores
3. Introduciton of new transformer engine
4. Increased capacity of large register files
5. Incorporation of a tensor memory accelerator
6. Continual increase in the number of SMs and FP units
7. NVIDIA NVLink switch system to connect multiple GPUs

Summary

1. GPU architectures started for 3D graphics pipeline acceleration
2. Now GPU architecture provides high data parallelism