Lec1: Introduction

2025 Spring, COSC4397

Based on “Programming Massively Parallel Processors, 4th Ed”
Chapter 1.

Outline

- Historic events that pushed heterogeneous parallel computing into the mainstream.
- Latency oriented device (CPU) vs throughput oriented device (GPU)
- Amdahl’s law
- Parallel algorithms and parallel programming
- CUDA and other programming interfaces

History

- Computer processors have enjoyed exponential performance improvements for many decades even for sequential programs up until 2003.
  - Moore's law: exponentially more stuff on a chip
  - Faster clock time, better microarchitecture of higher IPC, more cache, etc
  - Implicit parallelization of sequential program via pipelining, OOO, scheduling, …
  - Result: your program runs faster with every generation of processors, sometimes without even recompiling!
  - But it hits diminishing return around 2003…

History

- The trend slowed or stopped around 2003, largely due to energy consumption and heat dissipation issues.
  - Increasing frequency is too expensive in terms of power & energy (much faster than proportional, empirically more than quadratic and less than cubic)
  - On the other hand, implicit parallelization also hits a wall due to limited parallelism in sequential program and again, high complexity and energy consumption in analyzing and extracting the instruction level parallelism
  - Result? Instead of increasing performance of single core, processors have more and more cores, and each core's performance stays relatively the same or improves slowly over time
  - Corollary: without explicit parallel program your application will not run faster on newer hardware (and sometimes slower!)

Explicit parallel programming

- How to effectively use multi-cores?
  - **Multi-program**: for example, you may be running several applications on the computer simultaneously, one music play, one browser, and one program to download some big file
  - **Multi-processing**: a program might spawn multiple processes for explicit parallelism reasons (such as MPI programs) or security reasons (browser tabs) or both. Such program can make use of multi cores at the same time.
  - **Multi-threading**: a program might spawn only one process, but within the process spawn multiple threads that can be executed on different cores.

Multi-core enough?

- Things become more heterogeneous:
  - We have big/small cores on a single chips now (performance cores or efficiency cores)
  - We have more modules on the chip/in a computer that specializes in different things
    - GPU: high throughput computing, graphics
    - DSP: sound?
    - Neural engines: essentially matrix multiplication accelerator
    - FPGA?
  - For example, high end games were traditionally one of the driving force of performance improvement, and the solution has long been using a heterogeneous system that consists of CPU (general logic) + GPU (graphics, rendering)

Multi-core vs Many-threads

- Multi-core processor:
  - Relatively small number of cores, 2-128 cores, each optimized for executing sequential programs, 2-256 threads (counting hyperthreads)
  - Floating point performance (FLOPS) e.g.
    Recent 24 core Intel processor: 0.33TFLOPS double precision (DP), 0.66 TFLOPS single precision (SP).
- Many-threads processor:
  - In contrast to multi-core processor, it optimizes throughput of all cores by over-specifying parallelism in threads
  - Recent example NVIDIA A100 GPU: ~10,000 threads (cores? Not really, more like SIMD pipelines, will discuss later in arch of GPU)
  - 9.7TFLOPS DP, 156 TFLOPS SP, 312 TFLOPS HP (Half precision, 16bit)

Big gap in performance

- GPUs seems very appealing in certain applications (especially those heavy in number crunching, usually floating point)
- Two questions:
  - Why the gap?
  - How to program GPU (not for graphics)?

Latency optimized vs throughput optimized

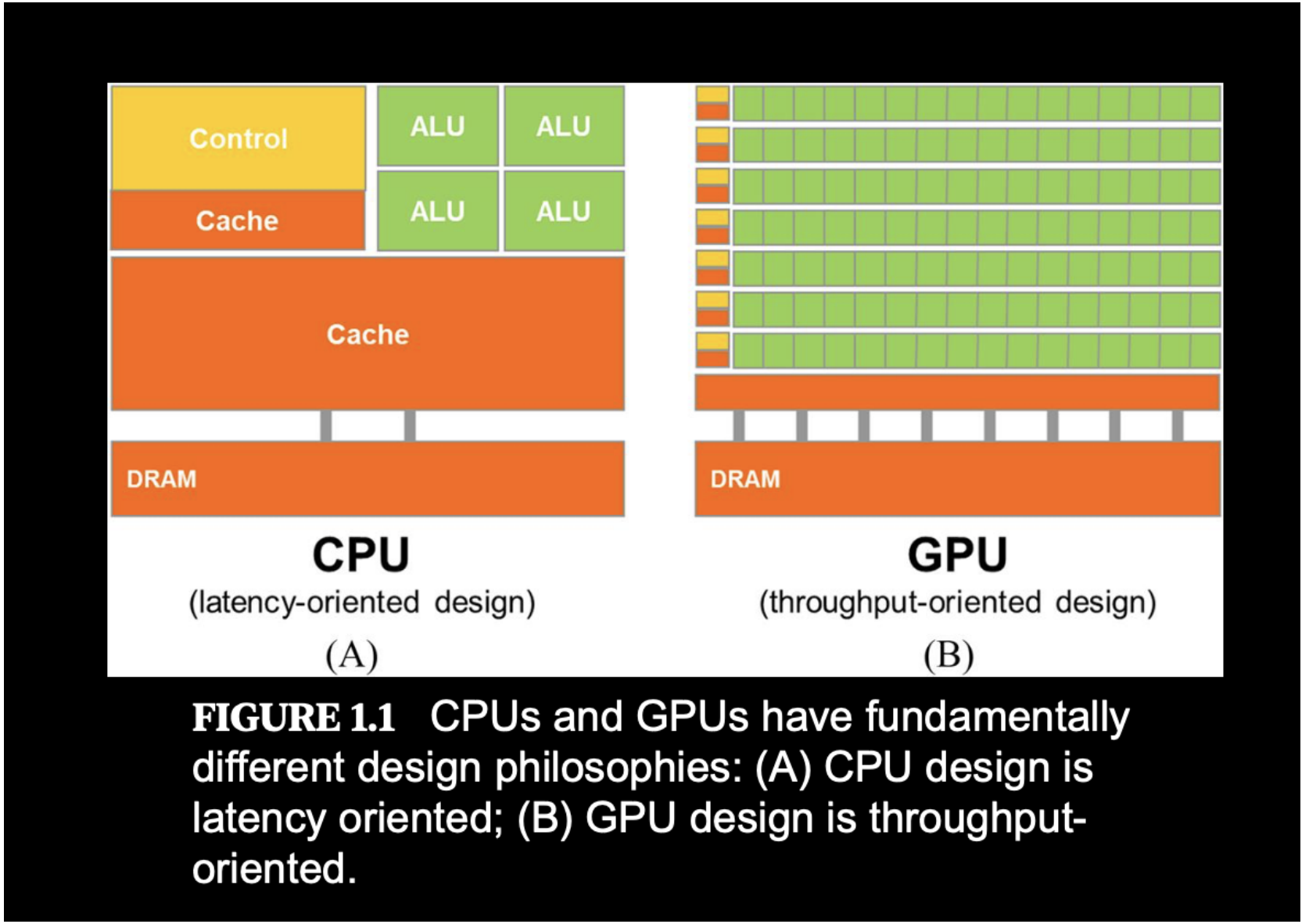

FIGURE 1.1 CPUs and GPUs have fundamentally different design philosophies: (A) CPU design is latency oriented; (B) GPU design is throughput-oriented.

Latency is harder to optimize than throughput

- Scaling throughput: 2x ALU => 2x throughput
- Reducing latency is much harder, 2x lower latency might require 2x current, and 4x more chip area & power.

(Reducing latency also improves throughput, it’s just much more costly than just adding functional units/channels/lanes etc)

How to program GPU?

- Prior to 2007 when CUDA was released, General Purpose GPU (GPGPU) programming was very awkward, essentially representing your program as a graphics pipeline and write shaders.
- In 2007 when CUDA was released and the NVIDIA G80 GPU, a compute pipeline was added to facilitate GPGPU.
- GPGPU requires massive amount of parallelism—CUDA/C provides extension to the C/C++ language to program in what NVIDIA called SIMT (single instruction multiple threads) manner. This name is quite confusing, but it really is a form of SPMD (single program multiple data).

Aside: what applications require such extreme performance?

- Traditionally games are at the frontiers of what hardware could do, pushing the envelope over each generations.
- Scientific computing—mostly simulations (solving equations)---require very large amount of compute because of discretization which results in large number of variables.
- More recently, AI/ML applications, especially deep neural networks, and particularly the training of it, consumes an astonishing amount of compute (frontiers models routinely costs tens of millions dollars even with GPUs and TPUs, it would be much higher and takes longer without GPUs/TPUs)
- Industrial applications, CAD, imaging, EDA, …

Speedups & Amdahl’s law

- How much speedup can we expect from parallelizing an application?
- For example, a serial application takes 10s on single core CPU. 80% of it can be parallelized on GPU and we achieved 100x speed up. Overall, the speed up is:
  10s / (2s + 8s/100) = 5x
- ”only 5x” speedup overall. This is Amdahl’s law—parallelization hits diminishing returns. Both “parallelize well” and “parallelize more” are needed to achieve good overall speedup.

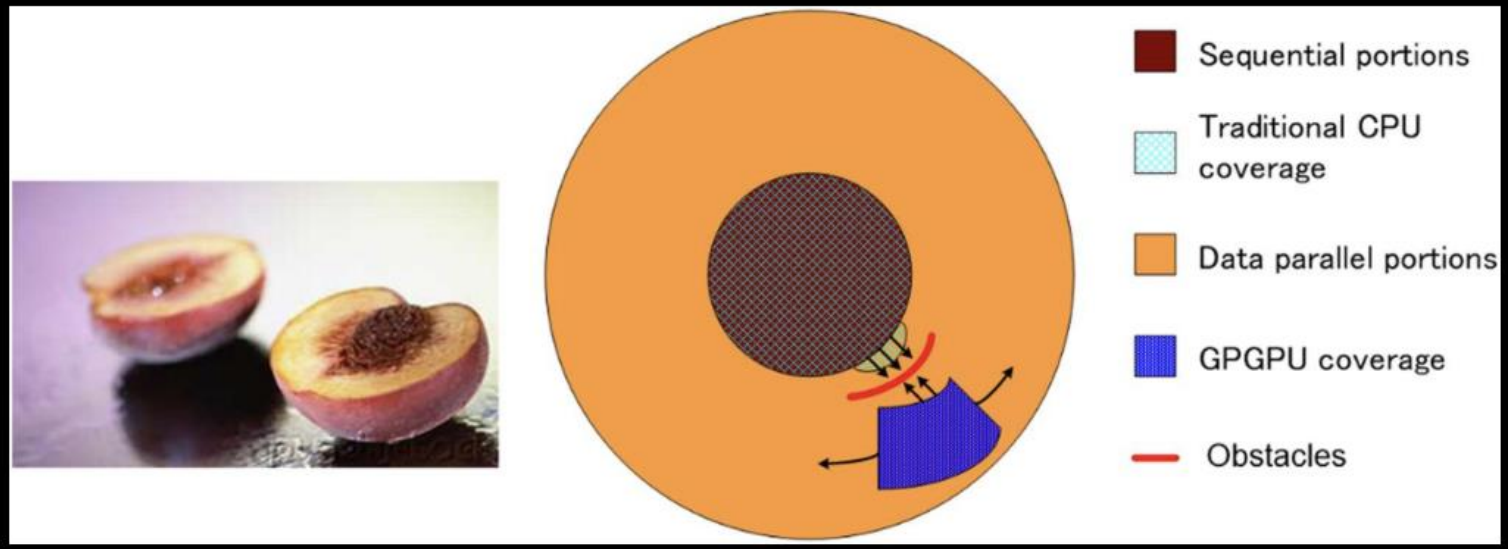

FIGURE 1.2 Coverage of sequential and parallel application portions. The sequential portions and the traditional (single-core) CPU coverage portions overlap with each other. The previous GPGPU technique offers very limited coverage of the data parallel portions, since it is limited to computations that can be formulated into painting pixels. The obstacles refer to the power constraints that make it hard to extend single-core CPUs to cover more of the data parallel portions.

Challenges:

- You do need to care about performance (otherwise why would you bother doing more complex parallel programming?)
  - This is challenging on its own right on a serial program on CPU
  - Sadly, in recent decades except the gaming industry, most modern software are not efficient (relative to its potential on very capable modern hardware)
- Parallel algorithms: certain algorithms are not amenable to parallelization (recursive algorithms for example, such as quick sort, or say Prim's algorithm for MST).
- Unique challenges for parallel programs:
  - Load balance
  - Synchronization
  - Work decomposition and mapping to different execution units

Goals of this course

- Teach you how to program massively parallel processor to achieve high performance.
  - Need a bit of understanding of hardware and architecture. Will discuss when needed. Generally, more knowledge of computer architecture helps with getting performance.
- Parallel programming with correct functionality and reliability
  - Managing parallelization
- Scalability and performance portability to future generations of hardware
  - Key: sufficient parallelism; regularize and localize memory access; minimize critical resources and conflicts

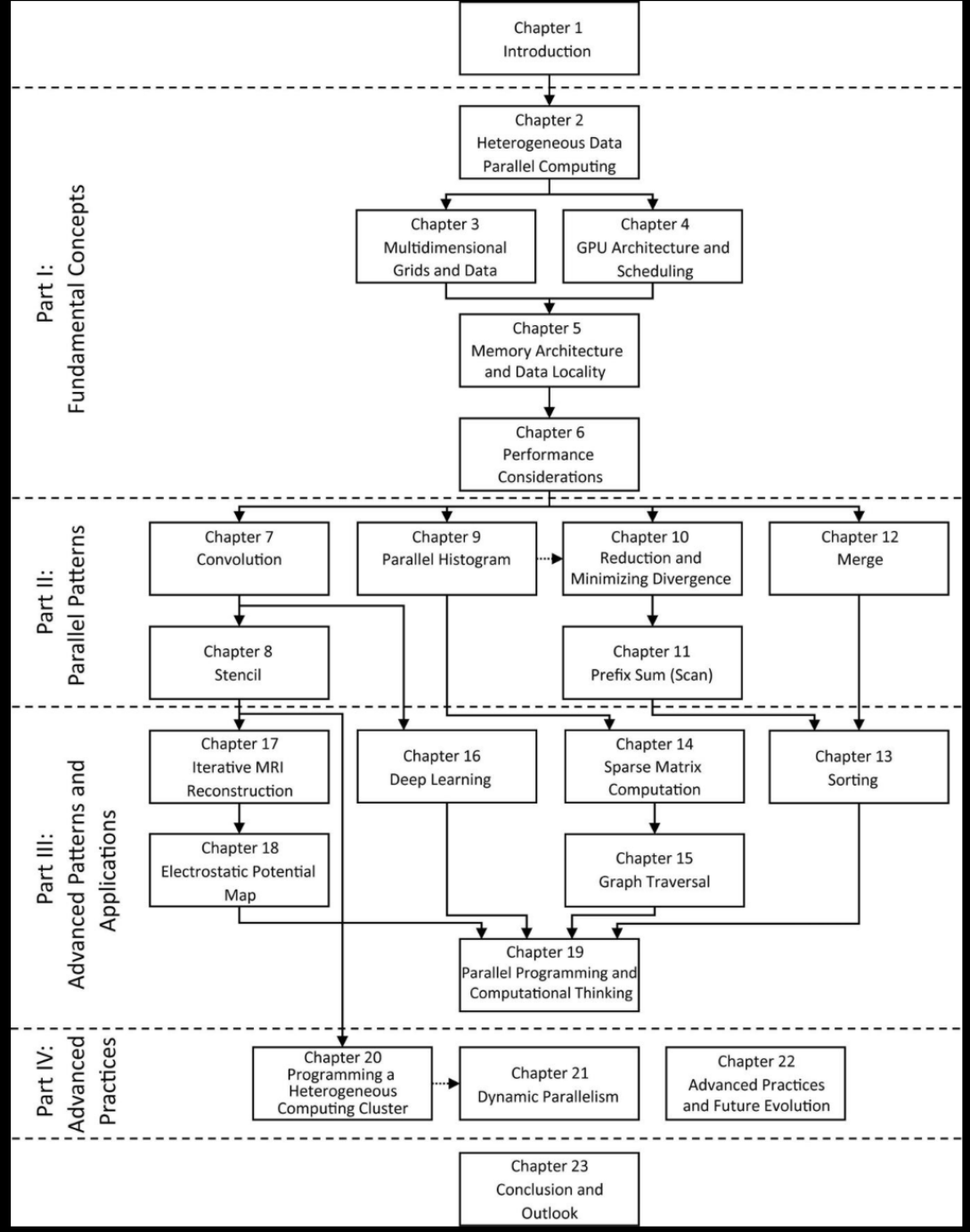

FIGURE P.1 Organization of the book.