GPU Programming and Parallel Computing

Guide to AI and Data Courses · Part VI: Systems and Applications

CPU and GPU | Thread model | Memory hierarchy | Parallel patterns | Measuring performance | Neighbouring courses

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/ai_courses/slides/Chapter_37_GPU_and_Parallel_Computing.html

INTRODUCTION

What Question Does It Ask?

> **How do thousands of cores share the same computation?**

GPU programming studies **data-parallel** computation, which applies the same operation to thousands of data elements at once. Using a model such as CUDA, the course teaches how to organize threads, use the memory hierarchy efficiently and measure speedup; this is also the hardware that makes deep learning practical.

At a glance

APPROACH data-parallel computation	**INPUT** parallelizable algorithm	**OUTPUT** accelerated program
SUCCESS MEASURE speedup, memory bandwidth	**MATH LOAD** low-medium	**DATA NEEDS** none

A Short History

1966
Flynn classifies computers as SISD, SIMD, MISD and MIMD.

1967
Amdahl shows that the remaining sequential part limits parallel speedup (Amdahl's law).

1999
NVIDIA GeForce 256 is marketed under the name "GPU".

2007
NVIDIA releases CUDA: the GPU becomes general-purpose programmable in a C-like language.

2012
AlexNet is trained on two GPUs and wins ImageNet; the GPU becomes the engine of deep learning.

2017
The Volta architecture introduces tensor cores: hardware units dedicated to matrix multiplication.

THE COURSE

Core Concepts

SIMT and warps
32 threads execute the same instruction together.

Thread organization
The grid, block and thread hierarchy.

Memory hierarchy
Registers, shared memory, global memory.

Coalesced access
Neighbouring threads should access neighbouring addresses.

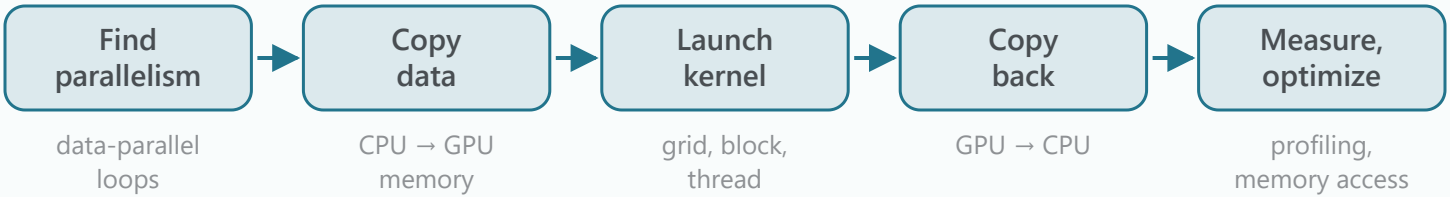
Synchronization
Barriers, atomic operations and race conditions.

Amdahl's law
The sequential part caps the achievable speedup.

Occupancy
Enough threads to keep the cores busy.

Parallel patterns
Map, reduction, prefix sum, stencil.

A Typical Workflow



Example: To add two vectors of two million elements, one thread is assigned to each element: the arrays are copied to GPU memory, the kernel is launched with 256-thread blocks and the result is copied back. Measurement often shows that the time goes into copying data, not into computing.

THE COURSE

What Does the Course Cover?

1. Parallel architectures: Flynn's taxonomy, CPU vs GPU
2. The CUDA model: grids, blocks and threads
3. Memory hierarchy and CPU-GPU data transfer
4. Coalesced access and shared memory
5. Warp execution, branch divergence, occupancy
6. Synchronization, atomics and race conditions
7. Parallel patterns: reduction and prefix sum
8. Stencils, histograms and sparse matrix operations
9. Tiled matrix multiplication and tensor cores
10. Streams, asynchronous copies and multiple GPUs
11. Profiling: Nsight and the roofline model
12. OpenMP on the CPU, MPI on clusters

Skills you will gain

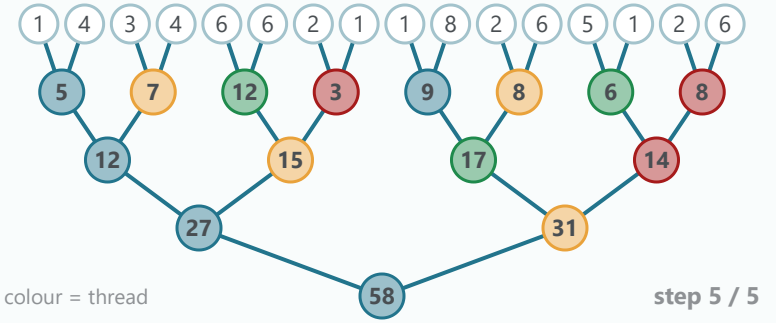
writing CUDA C kernels | finding bottlenecks with Nsight | Numba and CuPy in Python | CPU parallelism with OpenMP | measuring speedup correctly

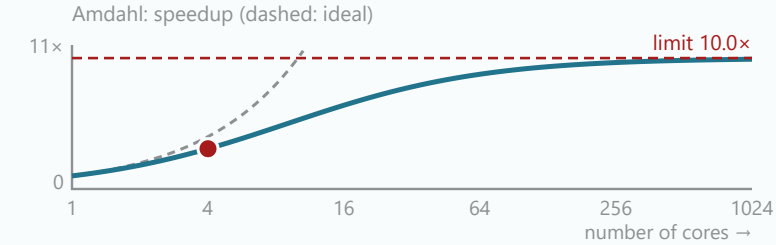
A typical semester; the order and coverage vary between universities.

TRY IT

Add Thousands of Numbers at Once

Computing a network's loss means adding millions of numbers; a GPU does it as a tree, adding pairs at the same time. Change the thread count and watch step by step; Amdahl's law shows the limit.





Number of threads p = **4**

Serial part of program = **10%**

Play step by step

16 numbers, **4** threads: **5 steps** (15 with one thread). Speedup 3.00×.
Amdahl: with a 10% serial part, 4 cores give **3.08×**; even infinitely many give at most **10.0×**.

Parallel programming teaches splitting work into **independent pieces**: the sum of n numbers finishes in log n steps. But even a small serial part sets a ceiling on the speedup.

CONTEXT

Prerequisites

Courses worth knowing before this one, and why they are needed.

Course	Why it is needed
Programming with C	pointers, arrays and memory allocation: the basis of CUDA C
Operating Systems	threads, synchronization and race conditions
Microprocessors	registers, caches and the memory hierarchy

CONTEXT

Are You Ready for This Course?

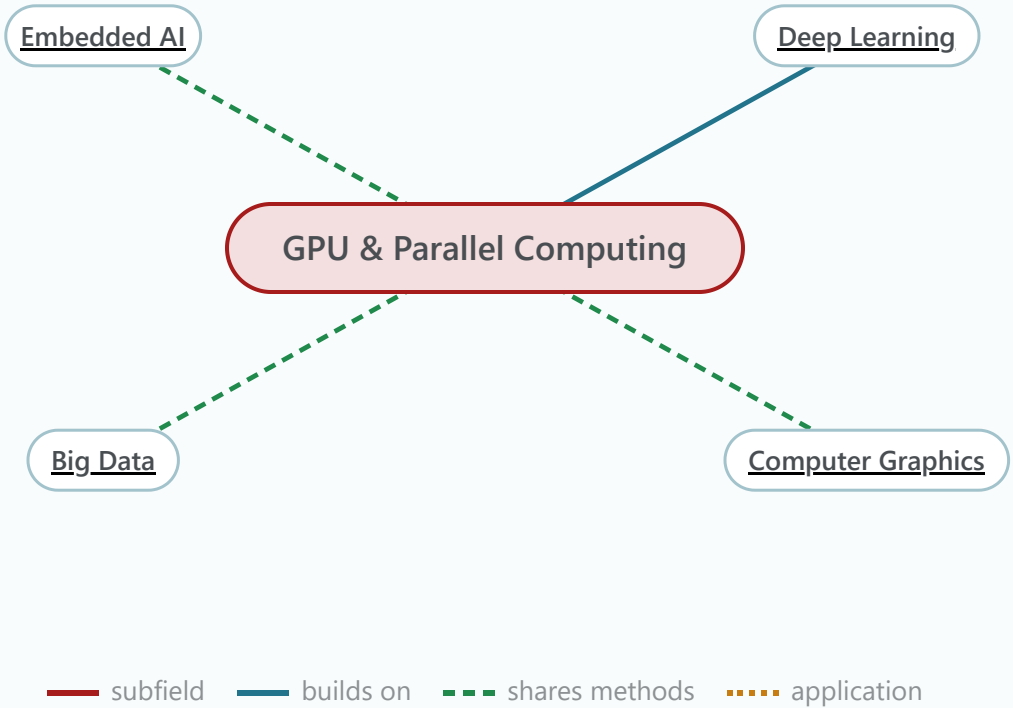
Ask yourself; for every "no", the right-hand column shows where to start.

Ask yourself	If not, start with
Can you manage a dynamic array in C with pointers and malloc?	A C programming course
Can you explain the race when two threads write the same variable?	Threads and synchronization from operating systems
Can you explain the speed gap between cache and main memory?	A microprocessors or computer architecture course
Can you write the product of two matrices with nested loops?	Linear Algebra

If you can say "yes" to most of these, you are ready; one or two gaps can be closed during the semester.

CONTEXT

Neighbouring Courses



Overlapping Courses

Relation	Course	Shared ground
builds on	Deep Learning builds on this course	parallel training of large networks, tensor operations
shares methods	Shares methods with Computer Graphics	graphics pipeline, shaders
shares methods	Shares methods with Big Data Systems	parallel and distributed computing
shares methods	Shares methods with Embedded AI	hardware accelerators

DIFFERENCES

Where It Differs (1/2)

Criterion	GPU & Parallel Computing	Deep Learning	Big Data
Core question	How do thousands of cores share the same computation?	How do many-layered networks learn useful representations from raw data?	How is data that does not fit on one machine stored and processed?
Approach	data-parallel computation	multilayer representation learning	distributed storage and computation
Input	parallelizable algorithm	raw data: images, text, audio	huge, fast-flowing data
Output	accelerated program	learned representation and prediction	processed data, query results
Success measure	speedup, memory bandwidth	accuracy, loss, scaling	scalability, latency, cost
Math load	low-medium	high: derivatives, matrices, probability	low
Data needs	none	large; needs GPUs	very large

- **Deep Learning:** Deep learning uses the GPU through ready-made libraries; this course writes the kernels and memory access beneath them.
- **Big Data:** Big data systems spread work across many machines; this course makes efficient use of thousands of cores on one device.

DIFFERENCES

Where It Differs (2/2)

Criterion	GPU & Parallel Computing	Computer Graphics	Embedded AI
Core question	How do thousands of cores share the same computation?	How is a realistic image produced from the description of a scene?	How do AI models run on small, low-power devices?
Approach	data-parallel computation	geometry and light simulation	resource-constrained inference
Input	parallelizable algorithm	scene: models, lights, camera	trained model, device constraints
Output	accelerated program	image or animation	small, fast model
Success measure	speedup, memory bandwidth	realism, frame rate	latency, energy, memory
Math load	low-medium	high: linear algebra, geometry	medium
Data needs	none	none	training data already exists

- **Computer Graphics:** Computer graphics programs the GPU with shaders to draw images; this course uses it for general-purpose computation.
- **Embedded AI:** Embedded AI teaches making do with little power; this course teaches getting peak performance from the largest accelerator.

PRACTICE

Applications

Deep learning
Matrix multiplication and convolution on tensor cores.

Scientific simulation
Fluids, climate and molecular dynamics.

Image and video
Real-time filters, encoding and decoding.

Finance
Monte Carlo risk and pricing.

Bioinformatics
Sequence alignment and protein structure computation.

Data analytics
Table and database operations on the GPU.

PRACTICE

Risks and Ethics

Inflated speedups

Leaving out data transfer or comparing against a weak CPU baseline makes the speedup look far larger than it is.

Silent race bugs

A kernel with missing synchronization is right on most runs and wrong on some; the bug is hard to catch.

Floating-point differences

Parallel sums change the order of operations, so results differ slightly from the CPU and from run to run.

Energy and cost

GPU clusters that train large models draw a lot of power; inefficient code inflates that cost for nothing.

In depth: Explainable AI and AI Ethics.

PRACTICE

Tools and Datasets

Tools and libraries

CUDA | OpenCL | OpenMP | MPI | Numba | CuPy | NVIDIA Nsight

Classic example problems

Vector addition | Matrix multiplication | Parallel reduction | Prefix sum (scan) | Histogram | N-body simulation

The metric is **speedup**: a sequential CPU version and a parallel GPU version of the same problem are compared on the same input.

PRACTICE

Common Misconceptions

Common belief	In fact
A GPU speeds up any program.	Only parts with many independent, similar operations speed up; branching and sequential parts do not.
Doubling the cores halves the run time.	Amdahl's law: the sequential part sets an upper bound on the achievable speedup.
Computation is the most expensive part.	In most GPU programs the bottleneck is memory bandwidth and data transfer between CPU and GPU.
Parallel programming is only for deep learning.	Simulation, graphics and scientific computing used GPUs long before deep learning.

NEXT STEPS

After This Course

Courses that build on or extend this one

Deep Learning

Related careers

GPU software engineer

High-performance computing (HPC) specialist

ML infrastructure engineer

Game and graphics engine developer

Scientific computing researcher

GPU programming opens up the engine beneath deep learning and graphics. From here you can move to systems topics such as distributed training, compilers and accelerator design, or to neighbouring courses in the guide such as big data systems and embedded AI.

NEXT STEPS

Further Reading

- Kirk, Hwu and El Hajj, *Programming Massively Parallel Processors*, 4th ed., Morgan Kaufmann, 2022.
- Sanders and Kandrot, *CUDA by Example*, Addison-Wesley, 2010.
- Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., Morgan Kaufmann, 2017.
- Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *AFIPS Conference Proceedings* 30, 1967.

The End

Chapter 37: GPU Programming and Parallel Computing

← Previous chapter · Map of the field · Next chapter →