Parallel Programming

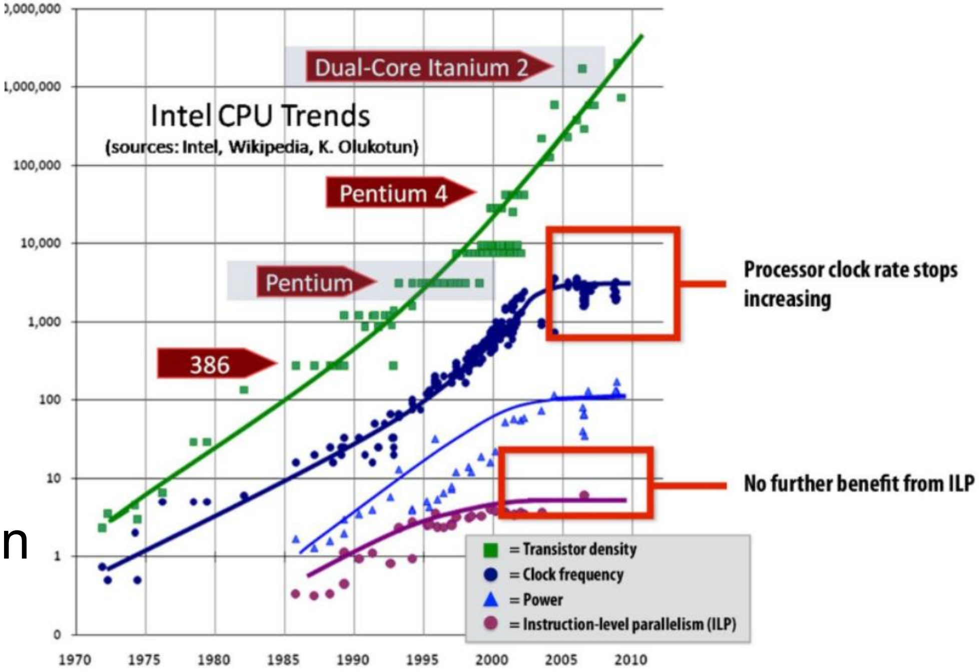
Course Introduction

Phạm Trọng Nghĩa

ptnghia@fit.hcmus.edu.vn

Why need parallel?

- Many applications have demanded **more execution speed** and resources
- The rate of single-instruction stream performance scaling has decreased
  - Frequency scaling limited by power
  - ILP scaling tapped out
- Architects are now building faster processors by adding more execution units that run in parallel
- Software must be **written to be parallel** to see performance gains



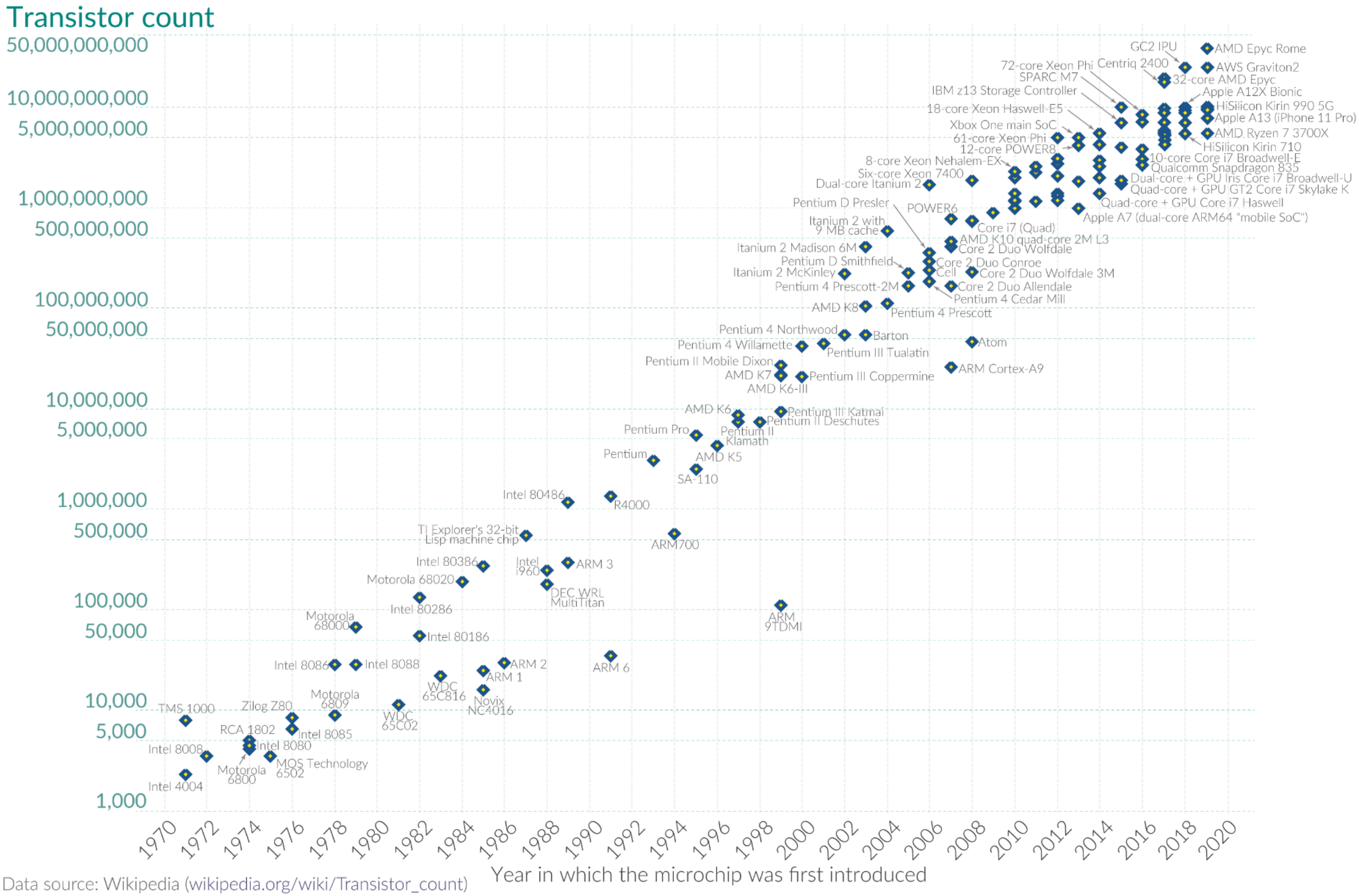
- *Transistor count(k)*
- *Clock Speed (Mhz)*
- *Power consumption (W)*
- *Instruction-level parallelism (ILP)*

Moore Law

Moore’s Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Our World in Data



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

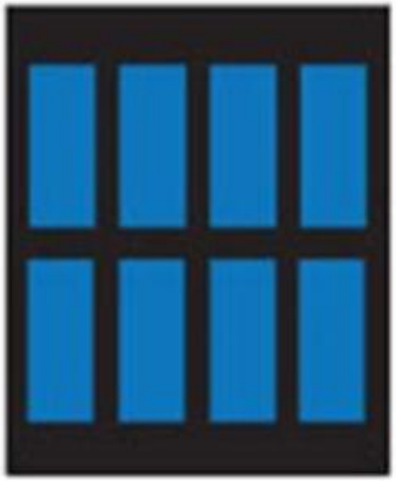
OurWorldinData.org – Research and data to make progress against the world’s largest problems.

CPU vs GPU

CPU vs GPU

CPU - Multicore

- Have a few cores, each core is powerful and complex
- Focus on execution speed



GPU – Many core

- Have many many cores, each core is weak and simple
- Focus on throughput

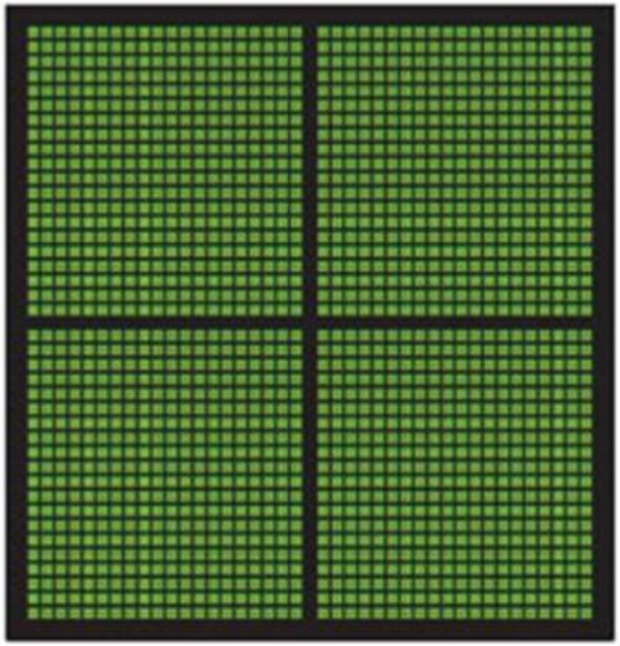


Image source: http://www.nvidia.com/object/what-is-gpu-computing.html

CPU vs GPU

CPU

Have a few cores, each core is powerful and complex

Focus on optimizing latency; latency = an amount of time to complete a task

GPU

Have many many cores, each core is weak and simple

Focus on optimizing throughput; throughput = # tasks completed in a time unit

Example: the task is transporting a person from location A to B, the distance from A to B: 4500 km

Car: 2 people, 200 km/h
Latency = ? h
Throughput = ? people/h

Bus: 40 people, 50 km/h
Latency = ? h
Throughput = ? people/h

CPU vs GPU

CPU

Have a few cores, each core is powerful and complex

Focus on optimizing latency; latency = an amount of time to complete a task

GPU

Have many many cores, each core is weak and simple

Focus on optimizing throughput; throughput = # tasks completed in a time unit

Example: the task is transporting a person from location A to B, the distance from A to B: 4500 km

Car: 2 people, 200 km/h
Latency = **22.5** h
Throughput = **0.09** people/h

Bus: 40 people, 50 km/h
Latency = **90** h
Throughput = **0.44** people/h

So, is car or bus better?

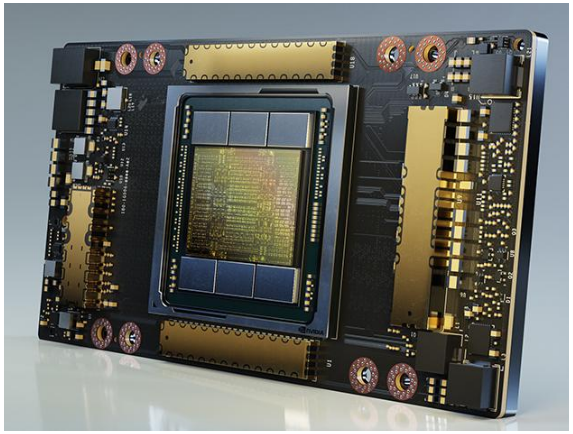
CPU vs GPU

CPU

- 24 core Intel multicore server microprocessor
- **0.33** TLOPS for double-precision and **0.66** TFLOPS for single precision

GPU - NVIDIA Tesla A100

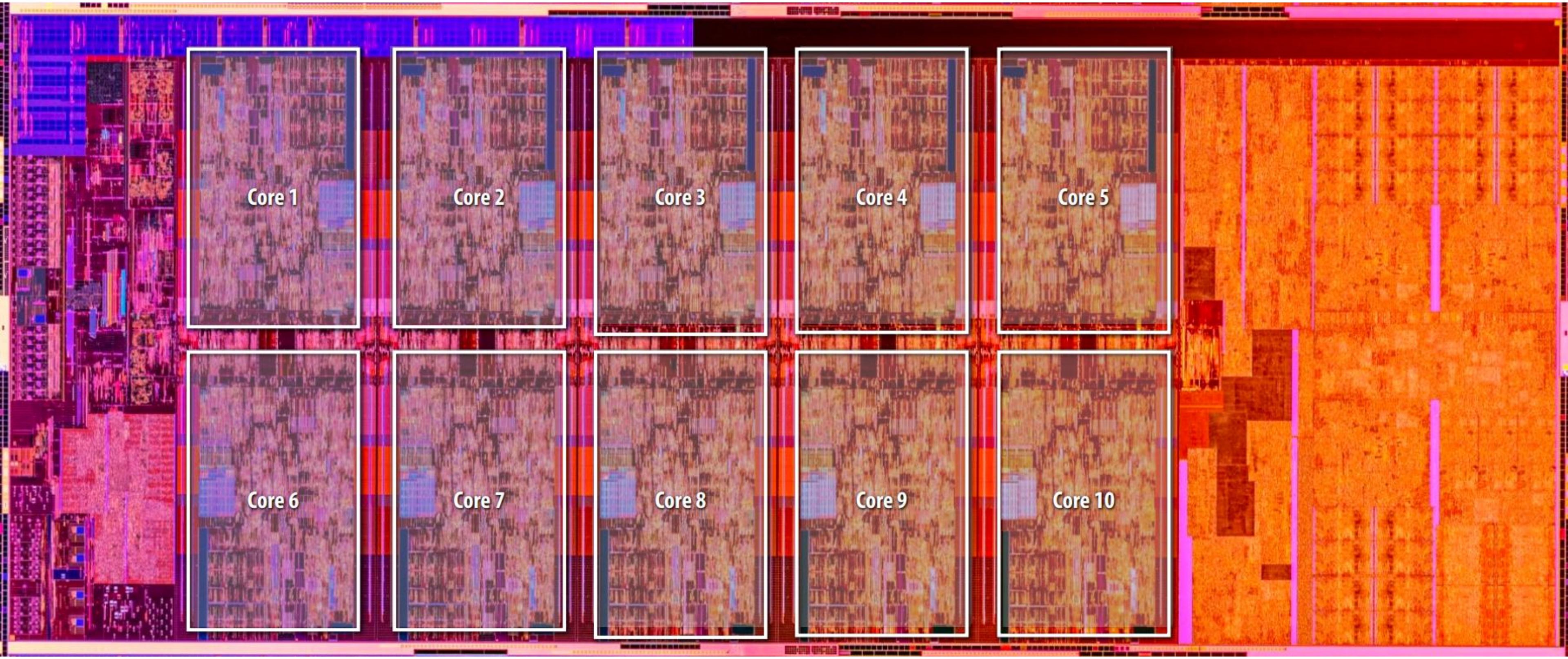
- 108SM, 6912 CUDA cores and 432 Tensor cores
- **9.7** TFLOPS for 64-bit double-precision, **156** TFLOPS for 32-bit single-precision, and **312** TFLOPS for 16-bit half-precision



FLOPS (FLoating-point Operations Per Second)
TFLOPS (TeraFLOPS)

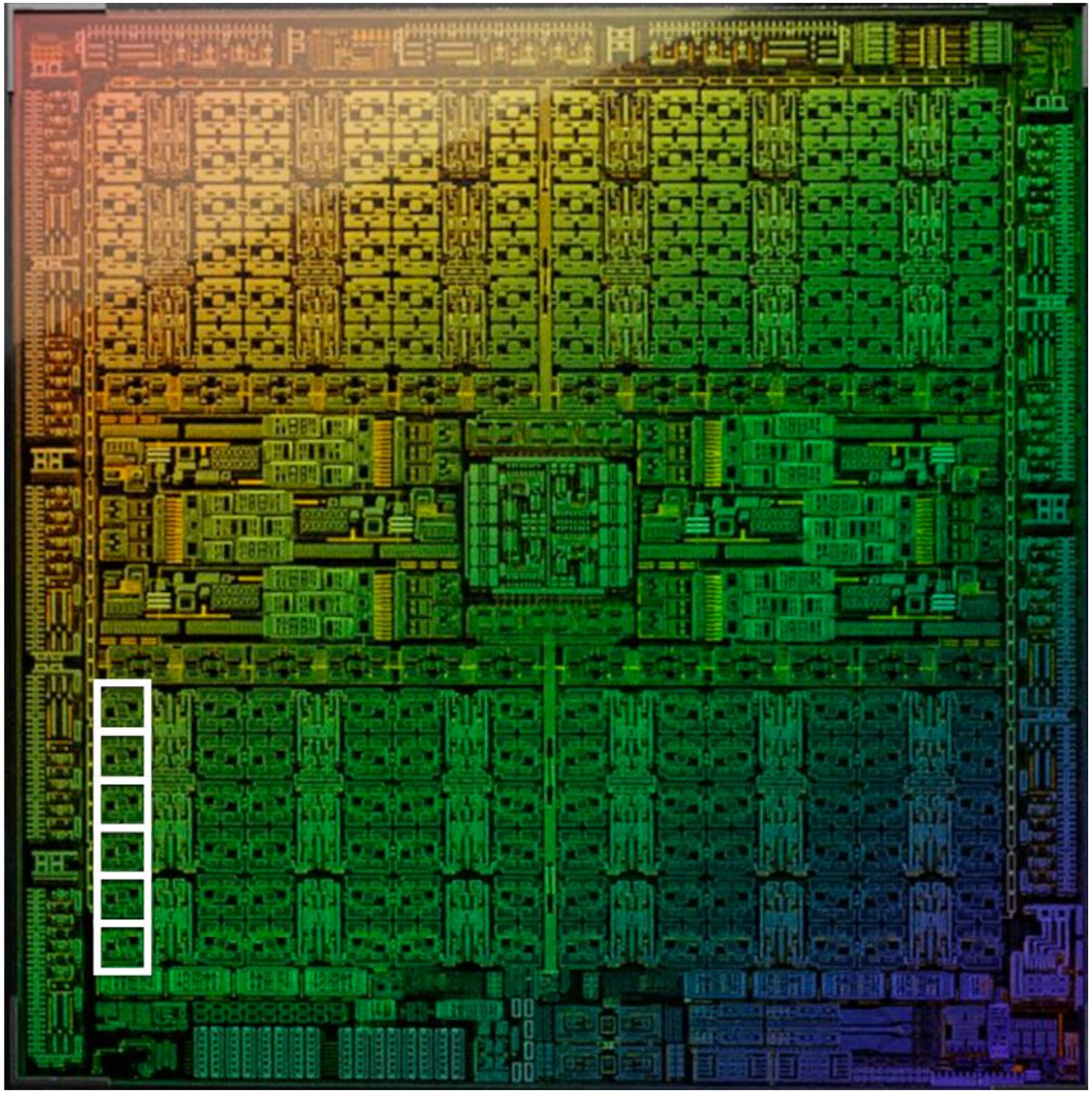
CPU vs GPU

- Intel “Comet Lake” 10th Generation Core i9 10-core CPU

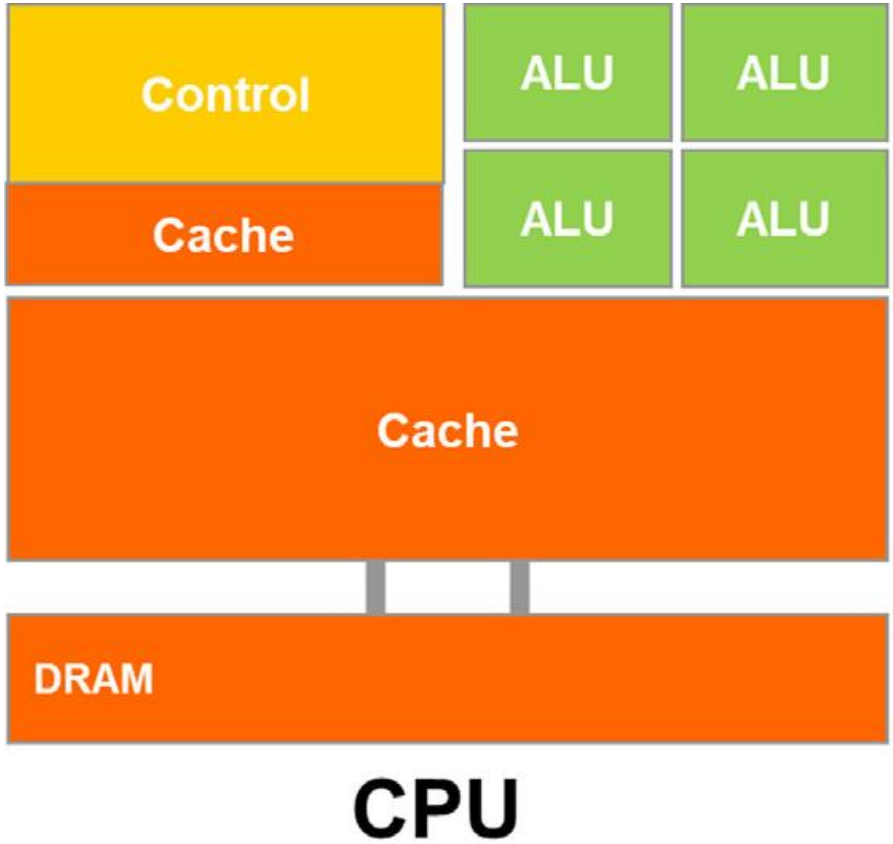


CPU vs GPU

- NVIDIA AD102 GPU
- 18,432 fp32 multipliers
- Organized in 144 processing blocks (called SMs)



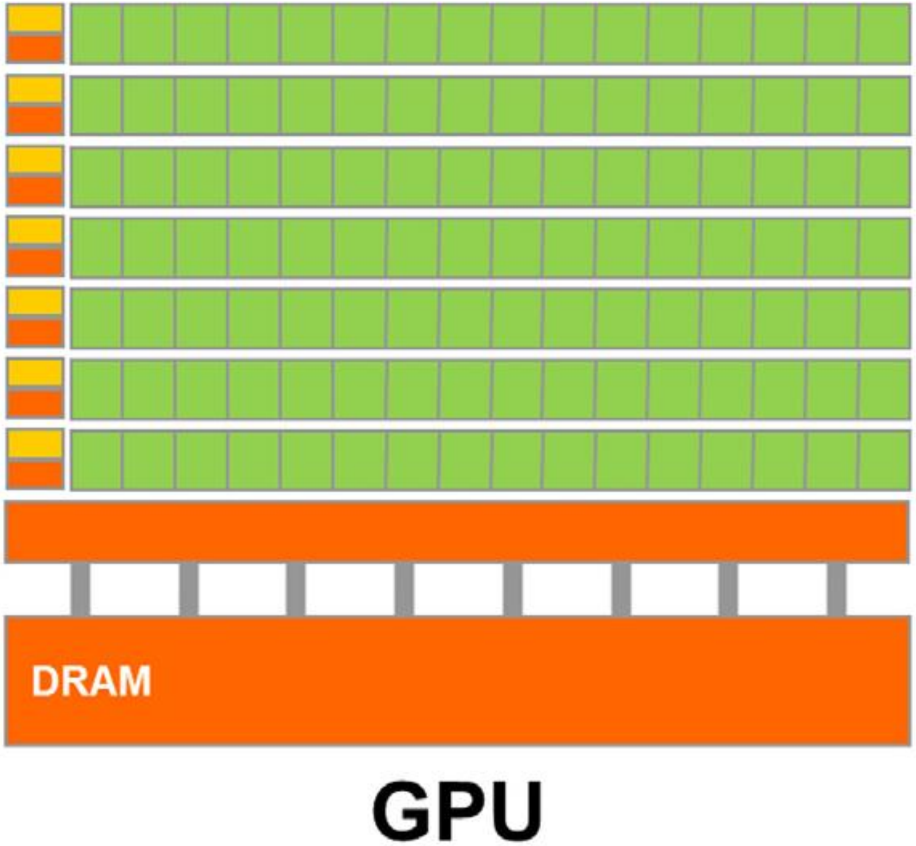
CPU: Latency-oriented design



- Powerful ALU
  - Reduce operation latency.
  - Increased chip area and power
- Large caches:
  - Convert long-latency memory accesses into short-latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

Reduces the execution latency of each individual thread

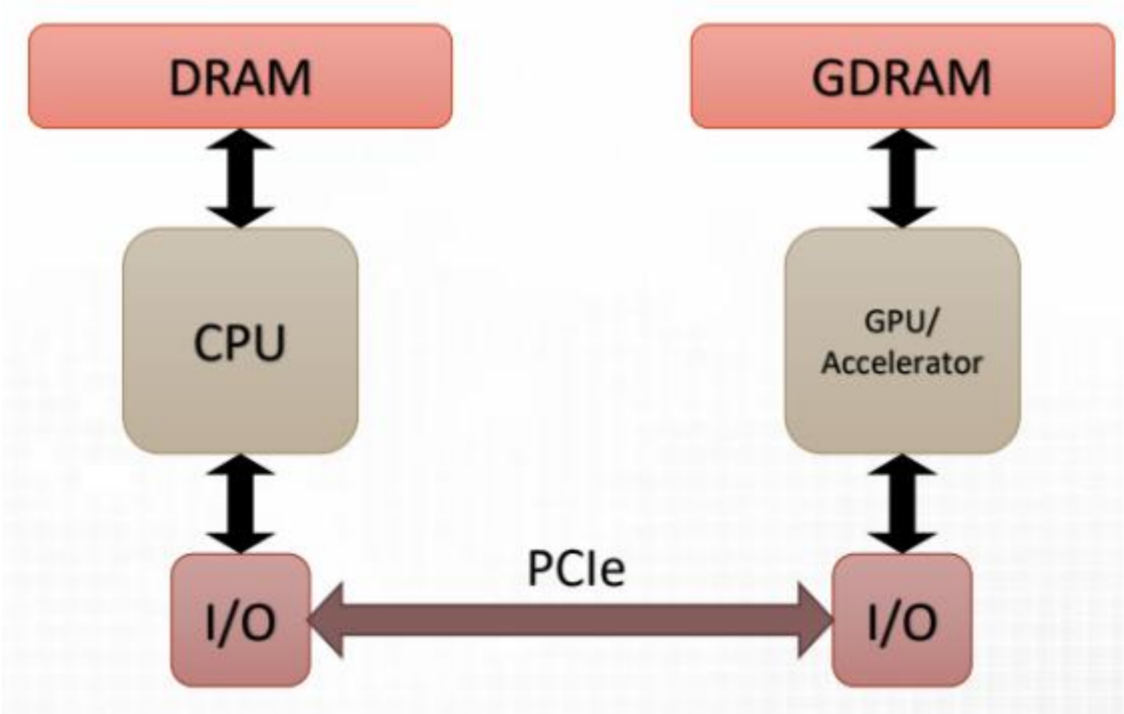
GPU: Throughput-oriented design



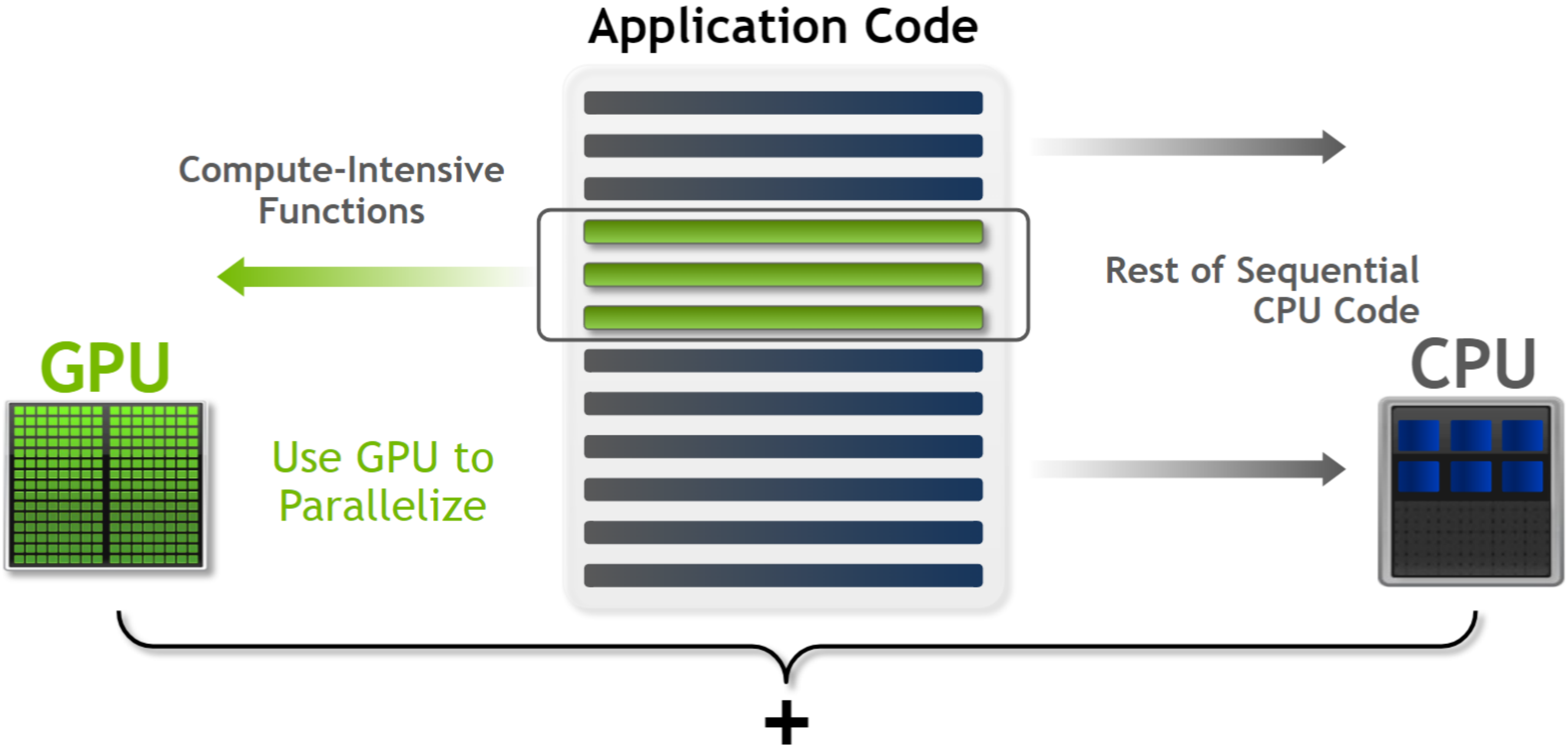
- Small caches
  - To boost memory throughput
- Simple control:
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughtput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread size

Strategies: use Both CPU & GPU

- Problem: Still require OS, IO and scheduling
- Solution: "Hybrid System"
  - CPU provides management and
  - "Accelerators" (or co-processors) such as GPUs provide compute power



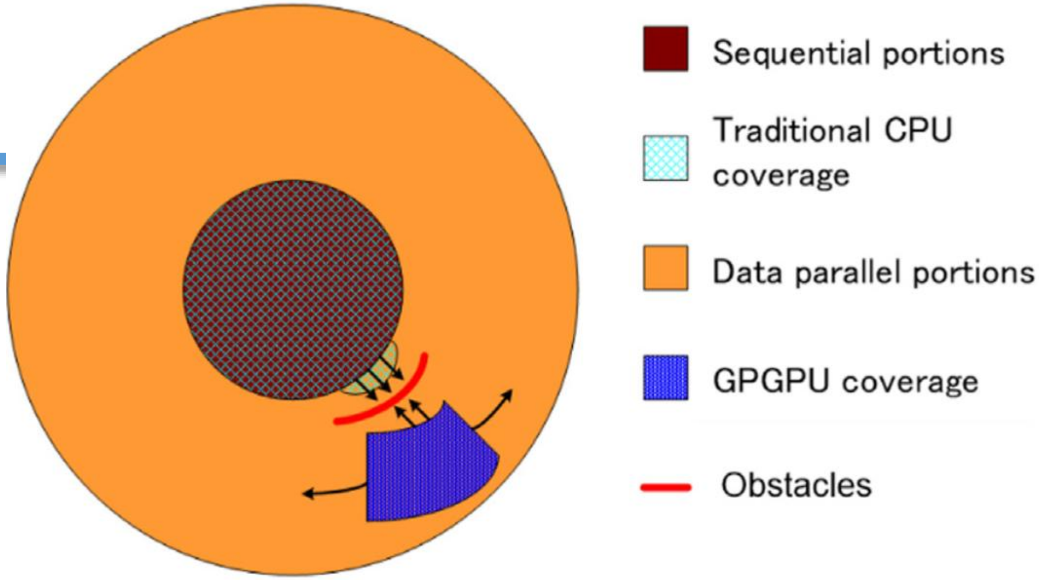
CPU + GPU



CUDA (Compute Unified Device Architecture) **C/C++** is extended-C/C++, allows us to write a program taking advantage of both CPU and GPU (NVIDIA): sequential parts will run on CPU, <u>massively parallel parts will run on GPU</u>

Image source: John Cheng et al. Professional CUDA C Programming. 2014

CPU + GPU



- Core area: sequential code.
  - These portions are very hard to parallelize.
  - CPUs tend to do a very good job on these portions.
  - Take up a large portion of the code, but only a small portion of the execution time
- "Peach flesh" portions:
  - Easy to parallelize.
  - Parallel programming in heterogeneous computing systems can drastically improve the speed of these applications.

Applications of parallel programming on GPU

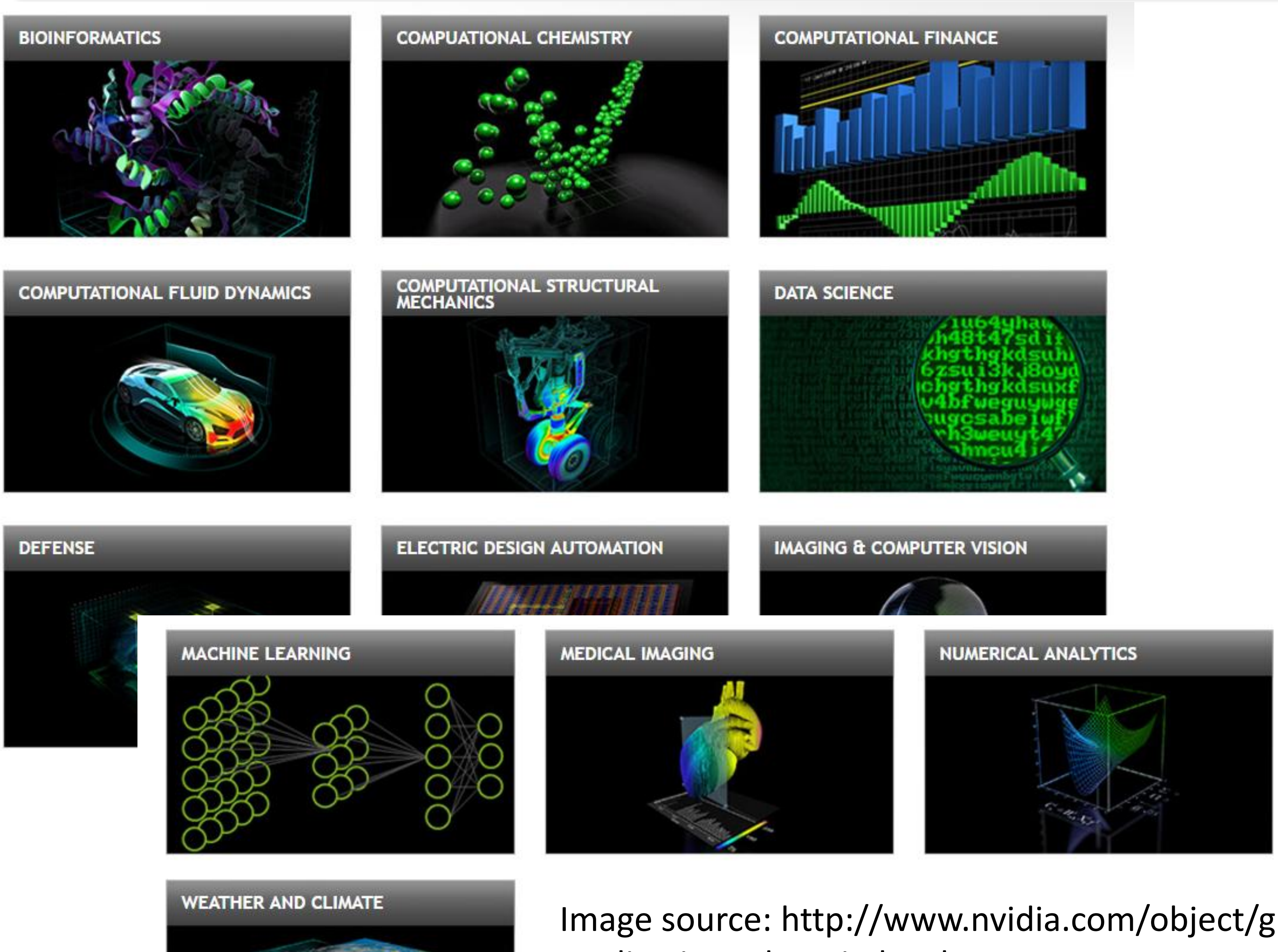
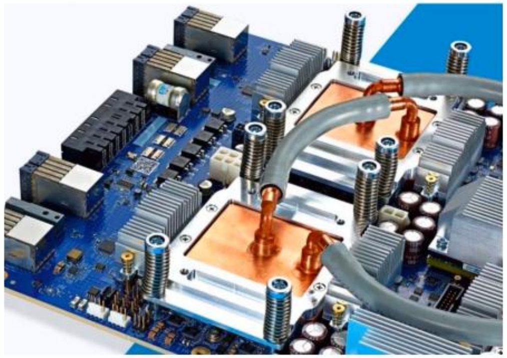


Image source: http://www.nvidia.com/object/gpu-

Specialized hardware to accelerate DNN inference/training

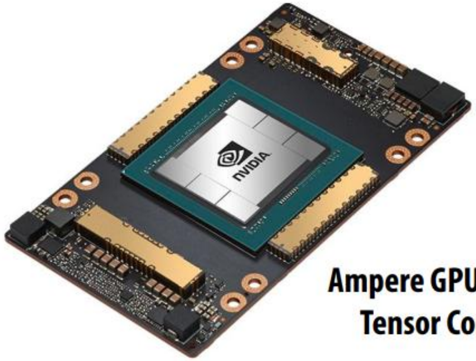


Google TPU3

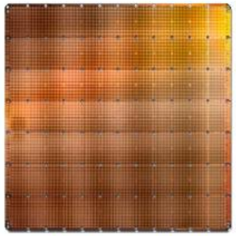
GraphCore IPU

Apple Neural Engine

AWS Trainium



Ampere GPU with Tensor Cores



Cerebras Wafer Scale Engine

SambaNova Cardinal SN10

Challenges in parallel programming

- Question: Is parallel programming easier or hard?
- Answer:
  - Easy: Do not care about performance, just want it able to run.
  - Hard: when you want optimize, get higher performance

Challenges in parallel programming

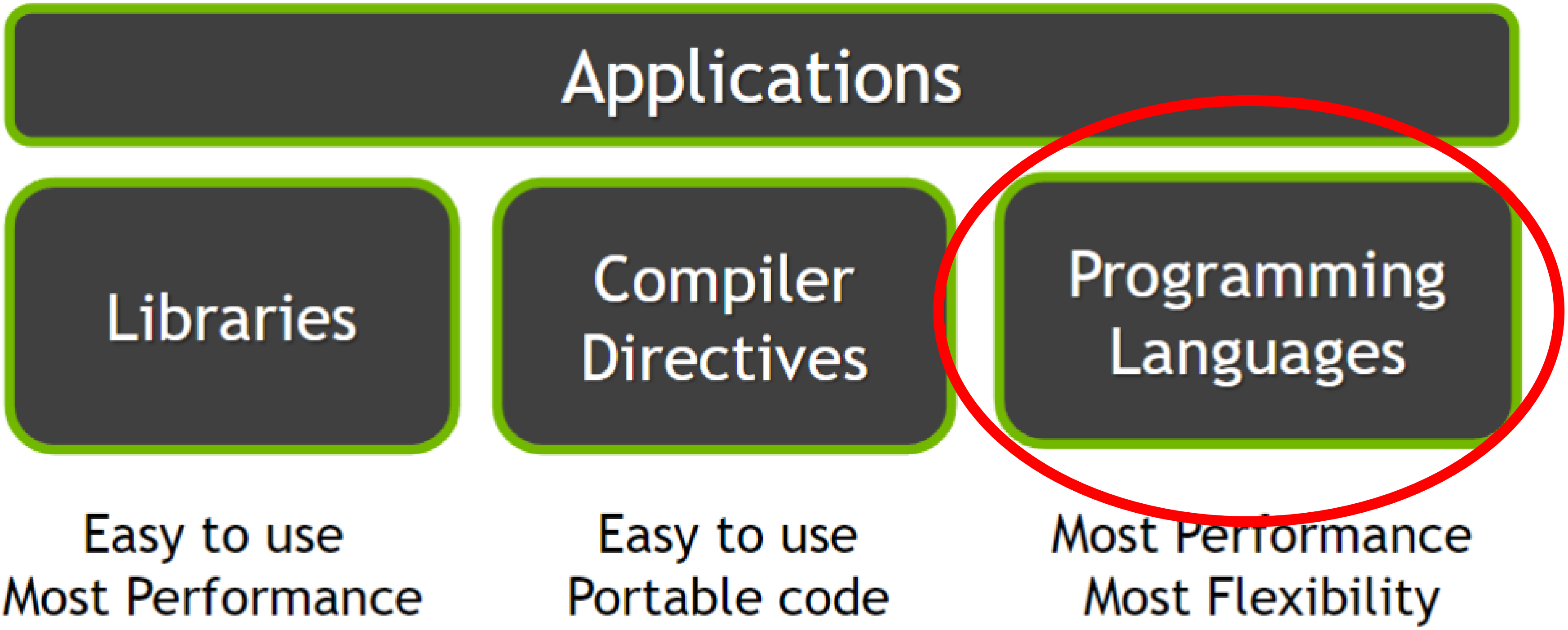
- Challenging to design parallel algorithms with the same level of algorithmic (computational) **complexity** as that of sequential algorithms
  - Some parallel algorithms do more work than their sequential counterparts
  - Parallelizing often requires **non-intuitive** ways of thinking about the problem and may **require redundant work** during execution
- The execution speed of many applications is limited by **memory access** latency and/or throughput
  - Requires methods for improving memory access speed

Challenges in parallel programming

- Execution speed of parallel programs is often more sensitive to the **input data characteristics** than is the case for their sequential counterparts
  - Unpredictable data sizes and uneven data distributions
- Require threads to collaborate with each other
  - Using synchronization operations such as barriers or atomic operations

Most of these challenges have been addressed by researchers

3 Ways to Accelerate Applications



Applications

Libraries

Compiler Directives

Programming Languages

Easy to use
Most Performance

Easy to use
Portable code

Most Performance
Most Flexibility

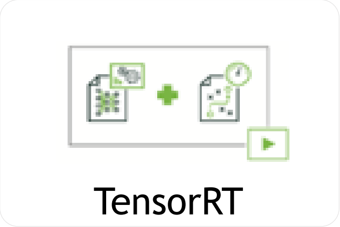
Libraries: Easy, High-Quality

- **Ease of use**: enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”**: Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality**: Libraries offer high-quality implementations of functions encountered in a broad range of applications

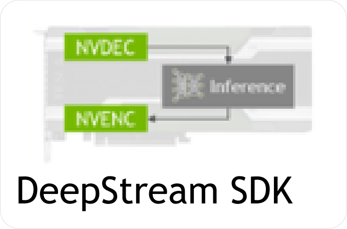
NVIDIA GPU Accelerated Libraries

DEEP LEARNING

cuDNN



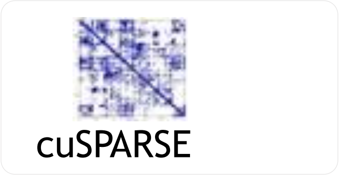
TensorRT



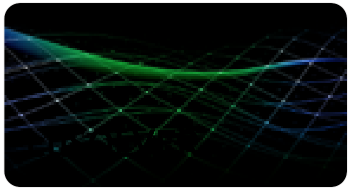
DeepStream SDK

LINEAR ALGEBRA

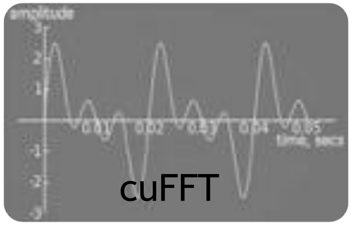
LAS



cuSPARSE



SIGNAL, IMAGE, VIDEO

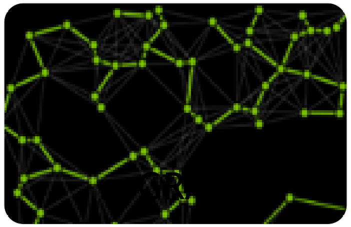


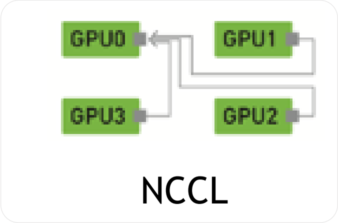
cuFFT

NVIDIA NPP

CODEC SDK

PARALLEL ALGORITHMS





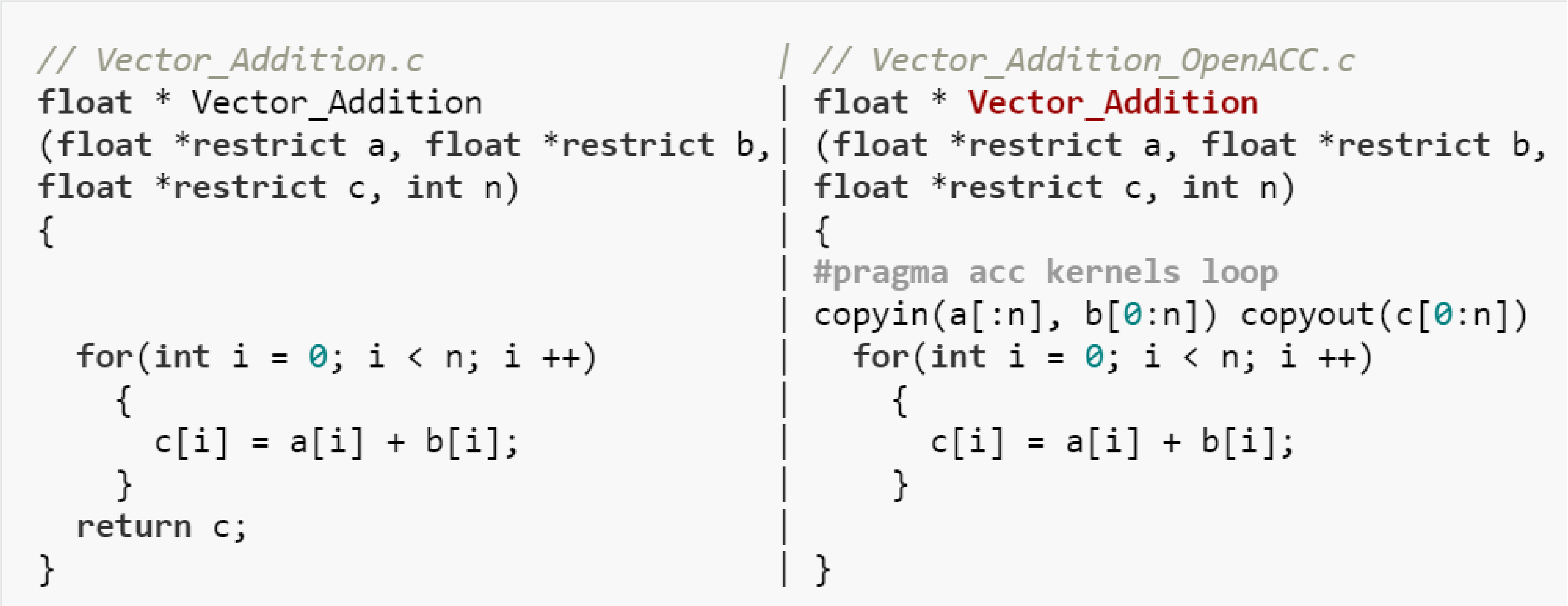
NCCL

https://developer.nvidia.com/gpu-accelerated-libraries

Compiler Directives: Easy, Portable

- **Ease of use**: Compiler takes care of details of parallelism management and data movement
- **Portable**: The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- **Uncertain**: Performance of code can vary across compiler versions

Compiler Directives: OpenACC



```
// Vector_Addition.c                  | // Vector_Addition_OpenACC.c
float * Vector_Addition               | float * Vector_Addition
(float *restrict a, float *restrict b,| (float *restrict a, float *restrict b,
float *restrict c, int n)             | float *restrict c, int n)
{                                     | {
                                      | #pragma acc kernels loop
                                      | copyin(a[:n], b[0:n]) copyout(c[0:n])
  for(int i = 0; i < n; i ++)         |   for(int i = 0; i < n; i ++)
    {                                 |     {
      c[i] = a[i] + b[i];             |       c[i] = a[i] + b[i];
    }                                 |     }
  return c;                           |
}                                     | }
```

https://ulhpc-tutorials.readthedocs.io/en/latest/gpu/openacc/basics/

Compiler Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems

5x in 40 Hours

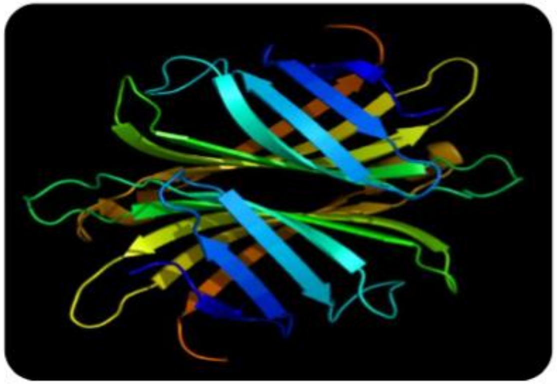
Valuation of Stock Portfolios using Monte Carlo

Global Technology Consulting Company

2x in 4 Hours

Interaction of Solvents and Biomolecules

University of Texas at San Antonio

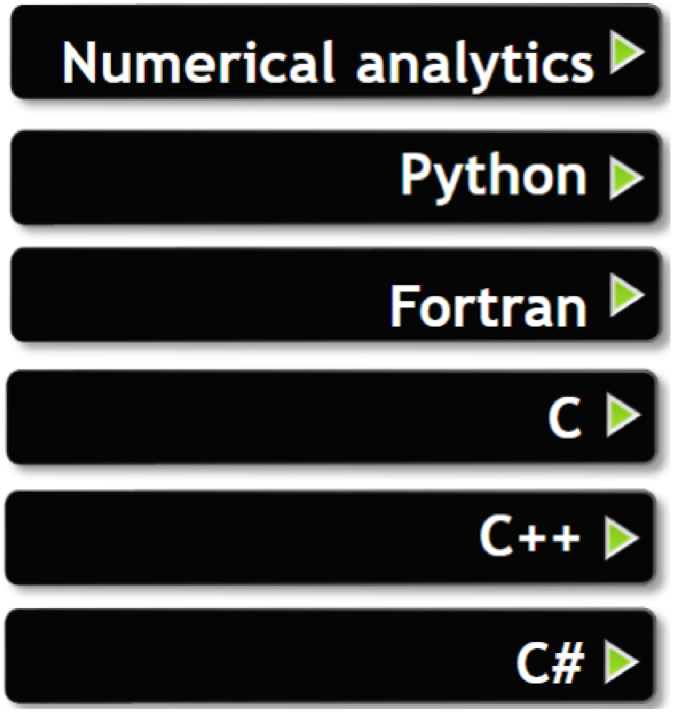


5x in 8 Hours

Programming Languages: Most Performance and Flexible

- **Performance**: Programmer has best control of parallelism and data movement
- **Flexible**: The computation does not need to fit into a limited set of library patterns or directive types
- **Verbose**: The programmer often needs to express more details

Programming Languages: Most Performance and Flexible



Numerical analytics ▶	• MATLAB, Mathematica, LabVIEW
Python ▶	• PyCUDA, Numba
Fortran ▶	• CUDA Fortran, OpenACC
C ▶	• **CUDA C**, OpenACC
C++ ▶	• CUDA C++, Thrust
C# ▶	• Hybridizer

Accelerated Computing is not all about performance

It’s all about ENERGY

It’s about Performance per Watt

Thinking about efficiency

- FAST != EFFICIENT
- Just because your program runs faster on a parallel computer, it does not mean it is using the hardware efficiently
  - Is 2x speedup on computer with 10 processors a good result?
- **Programmer's perspective**: make use of provided machine capabilities
- **HW designer's perspective**: choosing the right capabilities to put insystem (performance/cost, cost = silicon area?, power?, etc.)

Shift in CPU Design Philosophy

- Before 2004: Within the chip area budget, maximize performance
  - Increasingly aggressive speculative execution for ILP
- After 2004:
  - Area within the chip matters (limits # of cores/chip):maximize performance per area
  - Power consumption is critical (battery life, data centers): maximize performance per Watt
  - Major focus on efficiency of cores

Course topics:

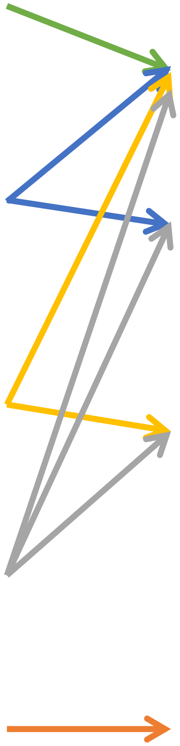
□ Introduction to CUDA; example: vector addition, convolution, … (2 weeks)

□ GPU parallel execution in CUDA; example: reduction, … (3 weeks)

□ Types of GPU memories in CUDA; example: reduction, convolution, … (3 weeks)

□ Example: scan, histogram, sort (3 weeks)

□ Optimizing a CUDA program; additional topics in parallel programming (0-1 week)



After successful completing the course, the student will be able to:

- Parallelize common tasks to run on GPU using CUDA
- Apply knowledge of GPU parallel execution in CUDA to speed up a CUDA program
- Apply knowledge of GPU memories in CUDA to speed up a CUDA program
- Apply the optimization process to optimize a CUDA program
- Apply teamwork skills to complete final project

Course assessment

- **Small quiz** throughout the course: 10% of the grade
- **Lab exercises** throughout the course: 40% of the grade
- **Group final project**: 50% of the grade, 2 students / group

Course assessment

Remember: the main goal is to learn, truly learn

You can discuss ideas with others as well as consult Internet sources, but your writing and code must be your own, based on your own understanding

If you violate this rule, you will get 0 score for the course

Advices

- In this course, we will focus on parallel programming on GPU (Graphics Processing Unit)
- Don’t worry if you don’t have GPU ;-)
- We will use Google Colab for this course.

Setup coding environment

- Where to find a machine with CUDA-enabled GPU?
  - Google Colab, it's free and ready to run CUDA programs ☺
  - Even if you have your own GPU, you should use Google Colab because teacher will use it to run and grade your programs
- Code, compile, and run:
  - Write and save code (.cu file) in your local machine by your favorite editor (with editors not recognizing .cu file automatically and not highlighting syntax with colors, the simple way is to set language/syntax as C/C++)
  - Open a notebook in Colab (you must sign in to your gmail), select "Runtime, Change runtime type" and set "Hardware accelerator" as GPU, upload .cu file
  - In a Colab cell, compile: `!nvcc file-name.cu -o run-file-name`

    If we don't specify run-file-name, it will default to a.out
  - In a Colab cell, run: `!./run-file-name`
- Demo …

RESOURCES

- Wen-Mei, W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.
- Cheng John, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014
- NVIDIA, NVidia CUDA C Programming Guide, version 12.5 or later
- Lê Hoài Bắc, Vũ Thanh Hưng, Trần Trung Kiên. *Lập trình song song trên GPU*. NXB KH & KT, 2015
- NVIDIA. *Intro to Parallel Programming*. Udacity
- NVIDIA. *CUDA Toolkit Documentation*

Reference

- [1] *Illinois-NVIDIA GPU Teaching Kit*
- [2] Wen-Mei, W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022

THE END