CENG443
Heterogeneous Parallel Programming

Introduction

Course Overview

Instructor: **Işıl ÖZ**

isiloz@iyte.edu.tr, office:101
Office hours: Monday 13:30-15:30

TA: **Semih ORHAN**

Lectures: **Wednesday 09:45-12:30**

Course Overview

Textbook:

David B. Kirk, Wen-Mei W. Hwu: Programming Massively Parallel Processors : A Hands-on Approach, Morgan Kaufmann Publishers, 2017.

Slides:

On the CMS after the lecture

Grading:

20% Midterm

50% HW&Projects

30% Final

Course Overview

What will you learn?

- How to program heterogeneous parallel computing systems and achieve high performance
- CUDA parallel computing platform and API, tools and techniques
- Principles and patterns of parallel algorithms
- Processor architecture features and constraints

Parallel Computing

Using multiple processing units in parallel to solve problems more quickly than with a single processing unit

Applications in engineering and design (DNA sequence analysis)

Scientific applications (oceanography, astrophysics)

Commercial applications (data mining, transaction processing)

Why More Speed or Parallelism

Scientific applications

computational model to simulate the underlying molecular activities in molecular biology

Video and audio coding and manipulation

HD TV

User interfaces

Modern smart phone users enjoy a more natural interface with high-resolution touch screens

Single Processor Performance

From 1986-2002, microprocessors' performance was increasing an average of 50% per year

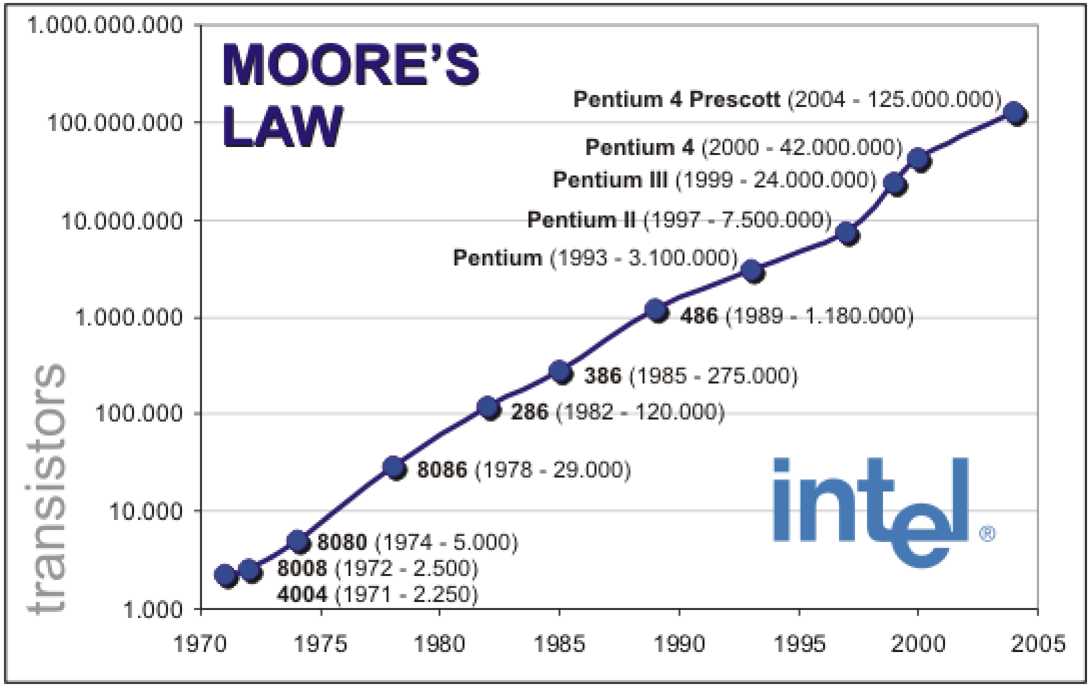
Then single-processor performance dropped to about 20% increase per year

Increase in single processor performance has been driven by the increasing *density*, the decreasing *size* of transistors

Moore's Law

An observation of Intel co-founder Gordon Moore in 1965, that the number of transistors in a dense integrated circuit doubles approximately every two years

It comes to end due to physical limitations, transistors approach the size of an atom, also power wall



Power Wall

Smaller transistors = faster processors

Faster processors = increased power consumption

Increased power consumption = increased heat

Increased heat = unreliable processors

Solution: Move away from single-core systems to multiprocessor systems

Multicore Systems

CPU: The "Central Processing Unit"

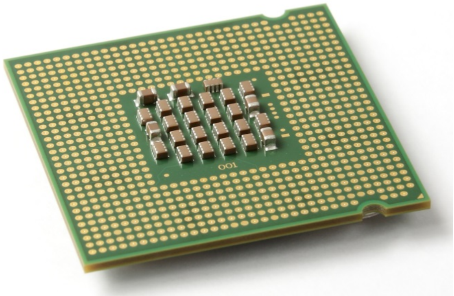
Traditionally, applications use CPU for primary calculations

General-purpose capabilities

Established technology

Usually equipped with 8 or less powerful cores

Optimal for concurrent processes but not large scale parallel computations



Manycore Systems

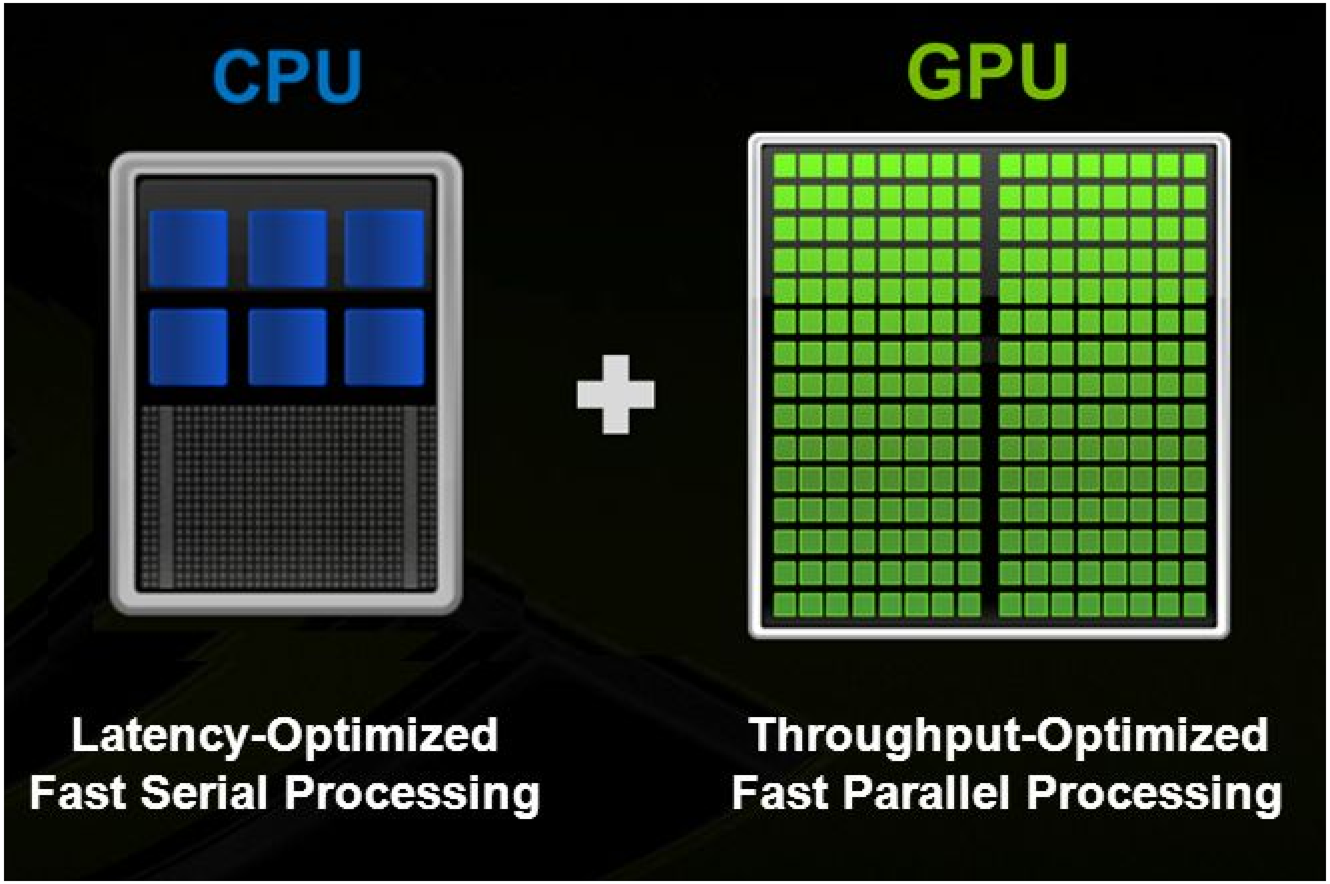
GPU: The "Graphics Processing Unit"

Relatively new technology designed for parallelizable problems

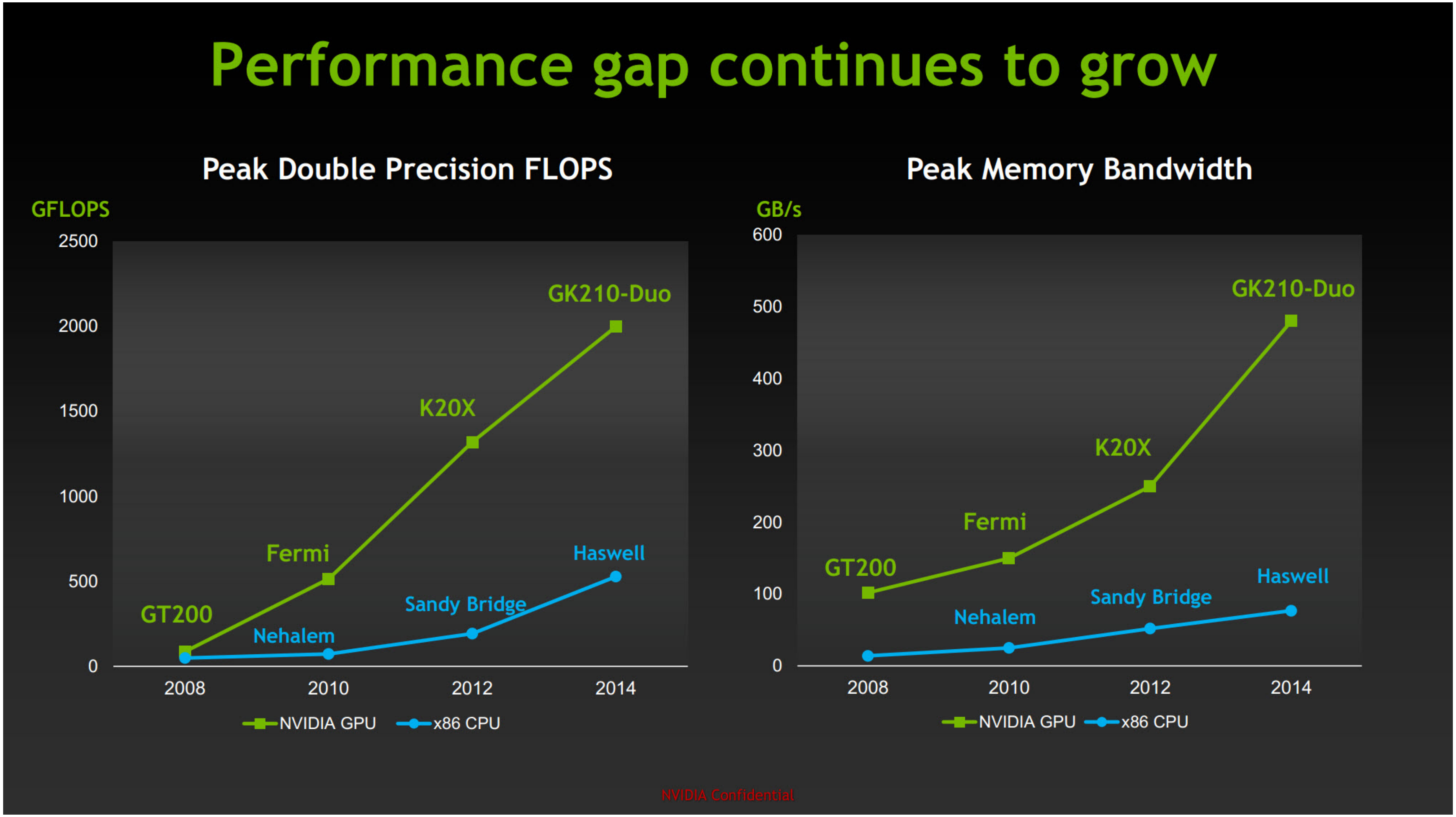
Initially created specifically for graphics

Became more capable of general computations

Heterogeneous Parallel Computing



CPU vs GPU



Performance gap continues to grow
Peak Double Precision FLOPS
GFLOPS
2500
2000
1500
1000
500
0
GK210-Duo
K20X
Fermi
Haswell
Sandy Bridge
GT200
Nehalem
2008
2010
2012
2014
NVIDIA GPU
x86 CPU
Peak Memory Bandwidth
GB/s
600
500
400
300
200
100
0
GK210-Duo
K20X
Fermi
GT200
Haswell
Sandy Bridge
Nehalem
2008
2010
2012
2014
NVIDIA GPU
x86 CPU
NVIDIA Confidential

Graphics

Example: Raytracing

enables real-time light reflections and cinematic effects in games

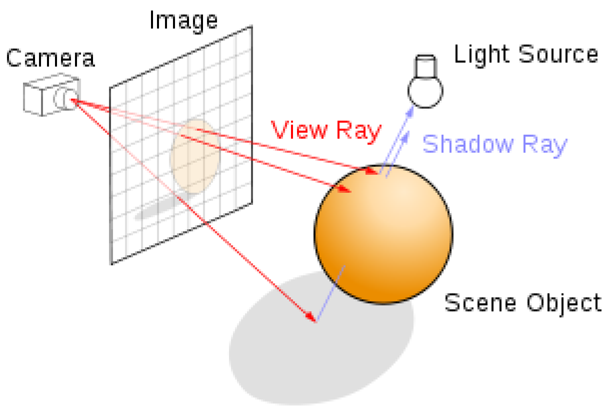
for all pixels (i,j):

    Calculate ray point and direction in 3d space

    if ray intersects object:

        calculate lighting at closest object ***

        store color of (i,j)



https://www.youtube.com/watch?v=lMSuGoYcT3s

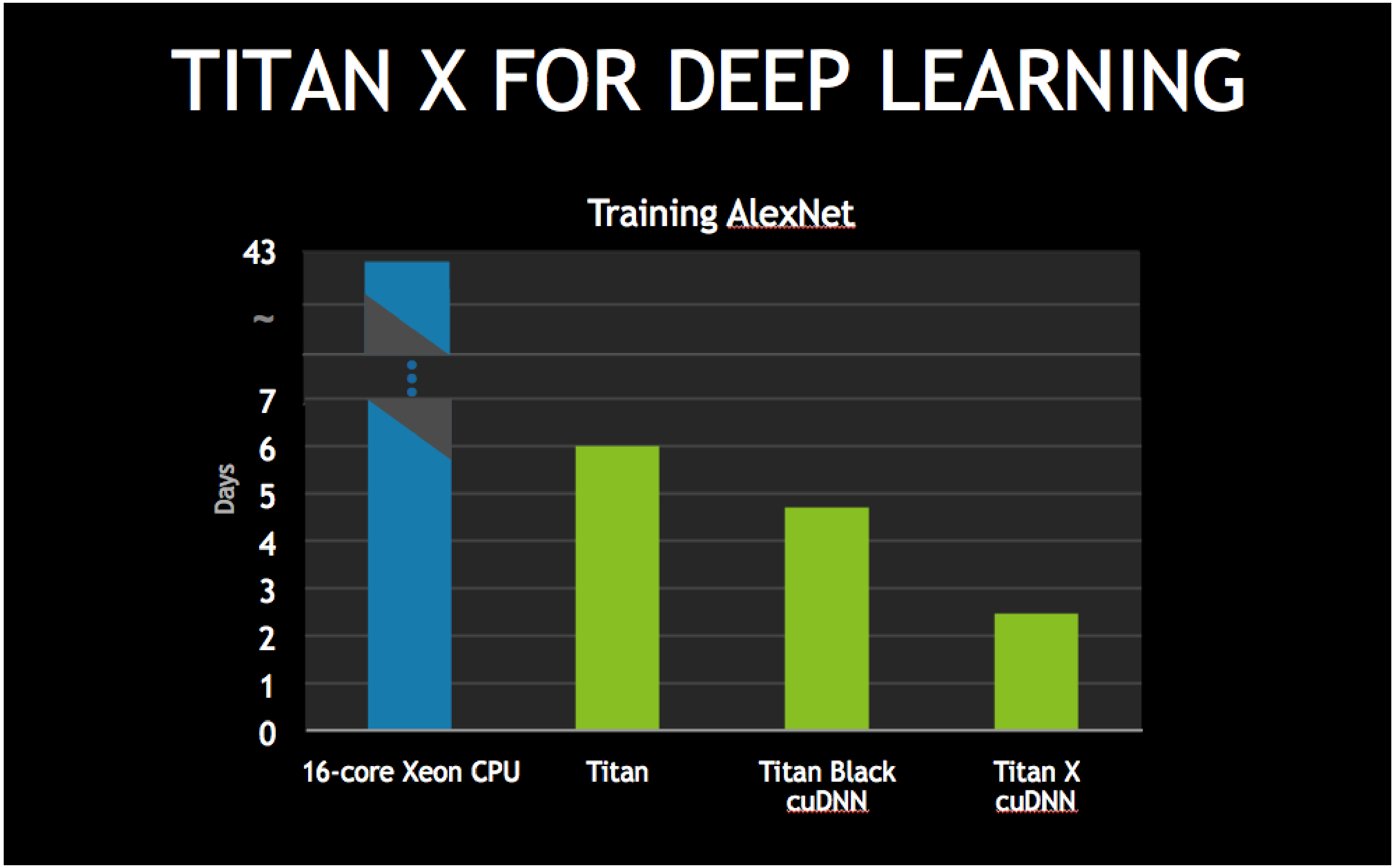
GPGPU

General-Purpose Computation on GPUs

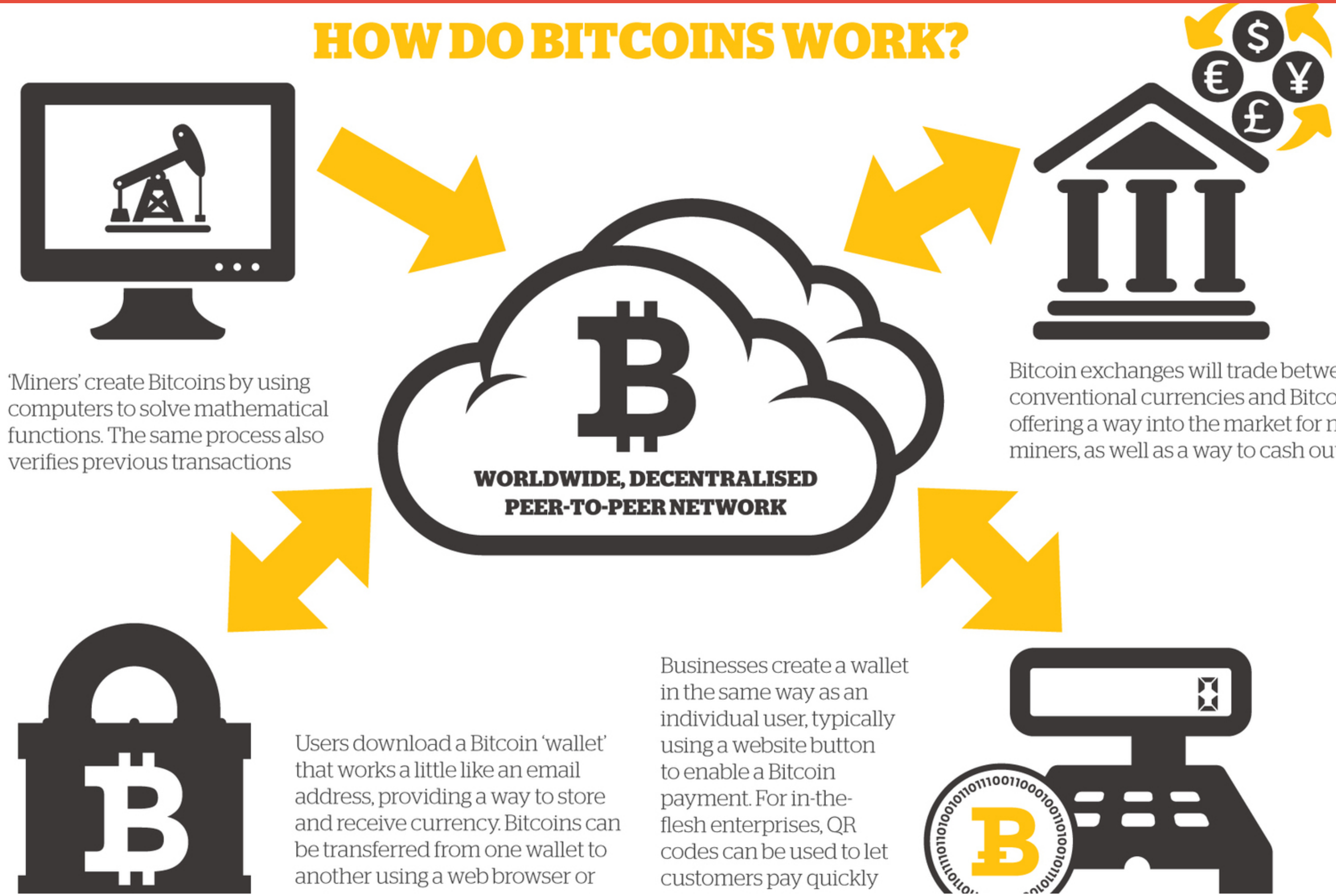
The GPU is no longer just for graphics

- Particle systems, collision detection
- Fluid dynamics
- Simulation

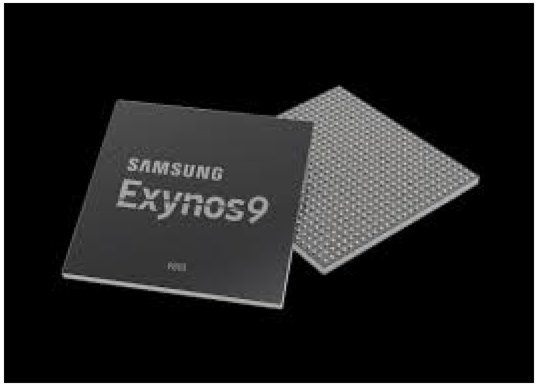
Deep Learning



Bitcoin Mining



Embedded Systems



Apple A11

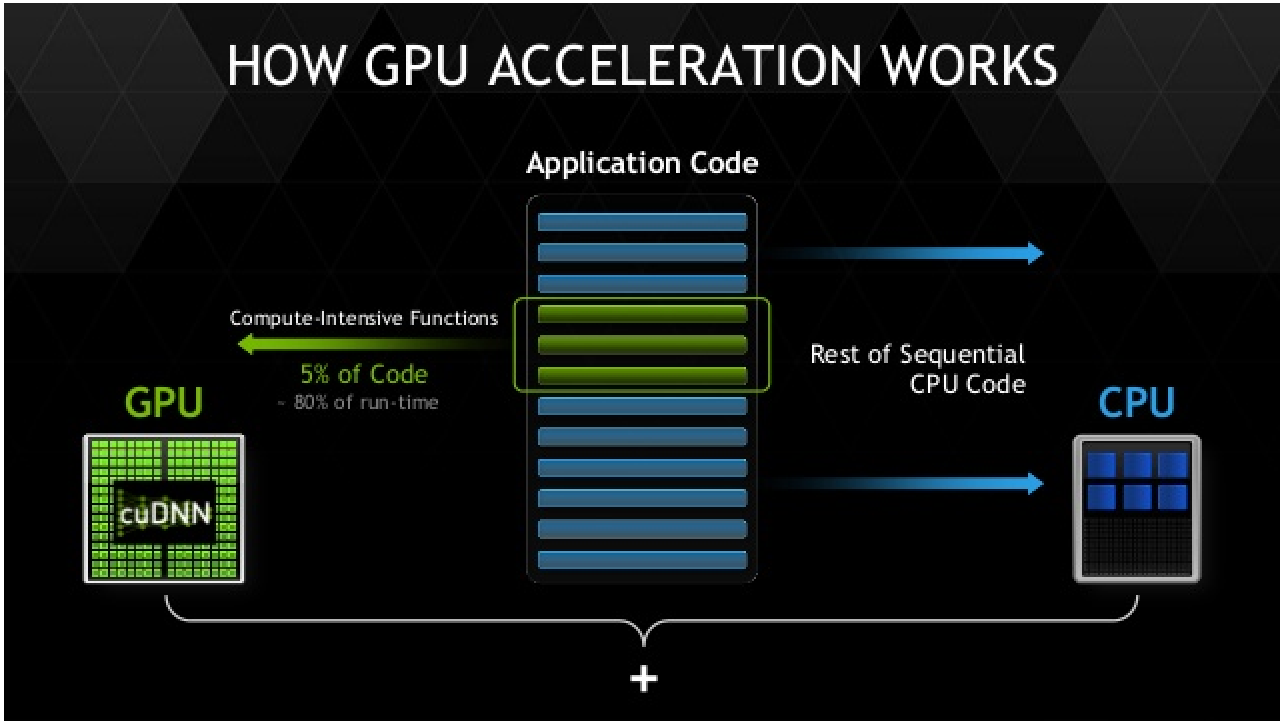
Mobile chip designed for IPhone 8, IPhone 8 Plus, and IPhone X

Hexa-core multicore

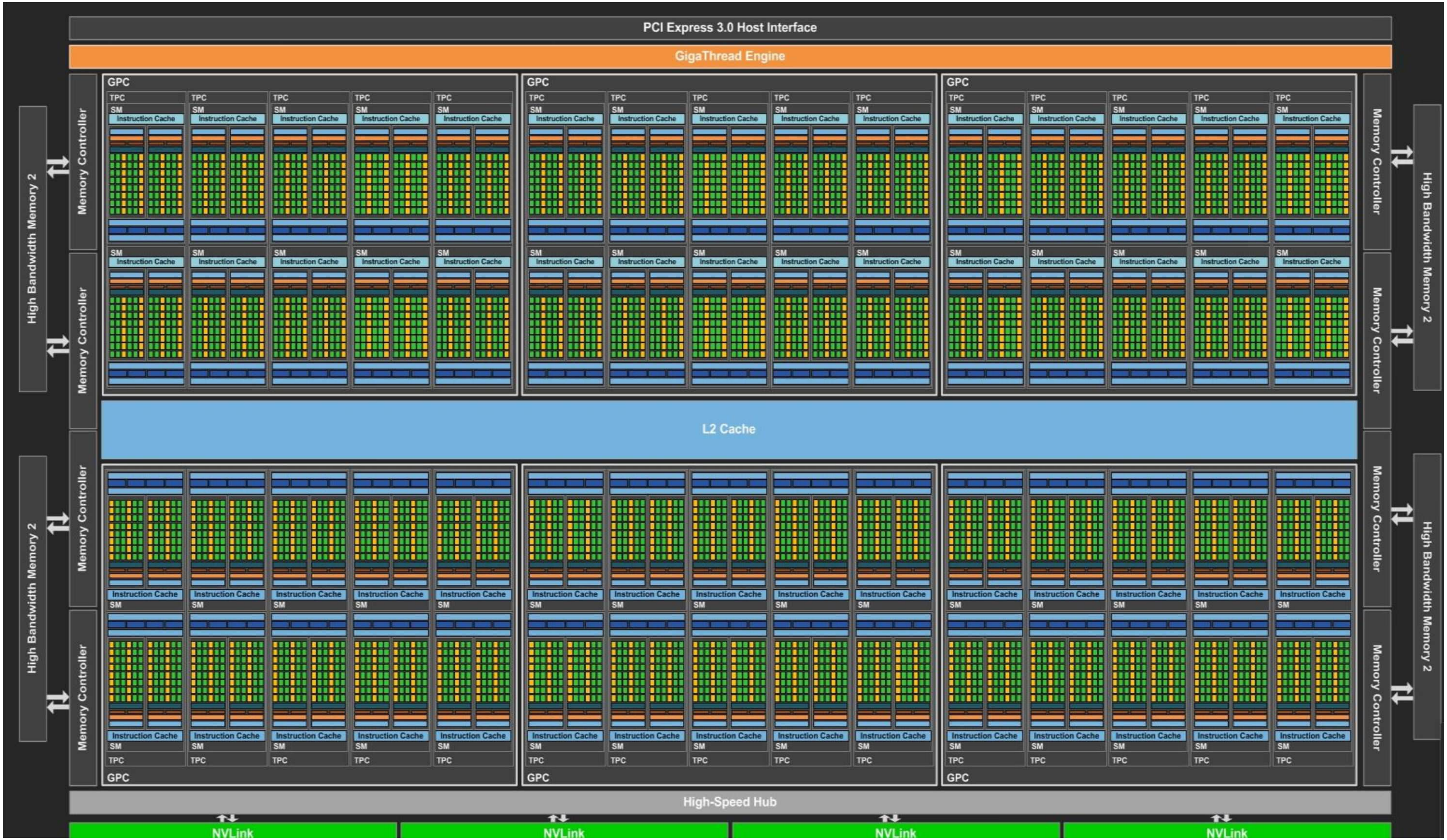
- Two high-performance cores called Monsoon
- Four low-power cores called Mistral

3-core GPU

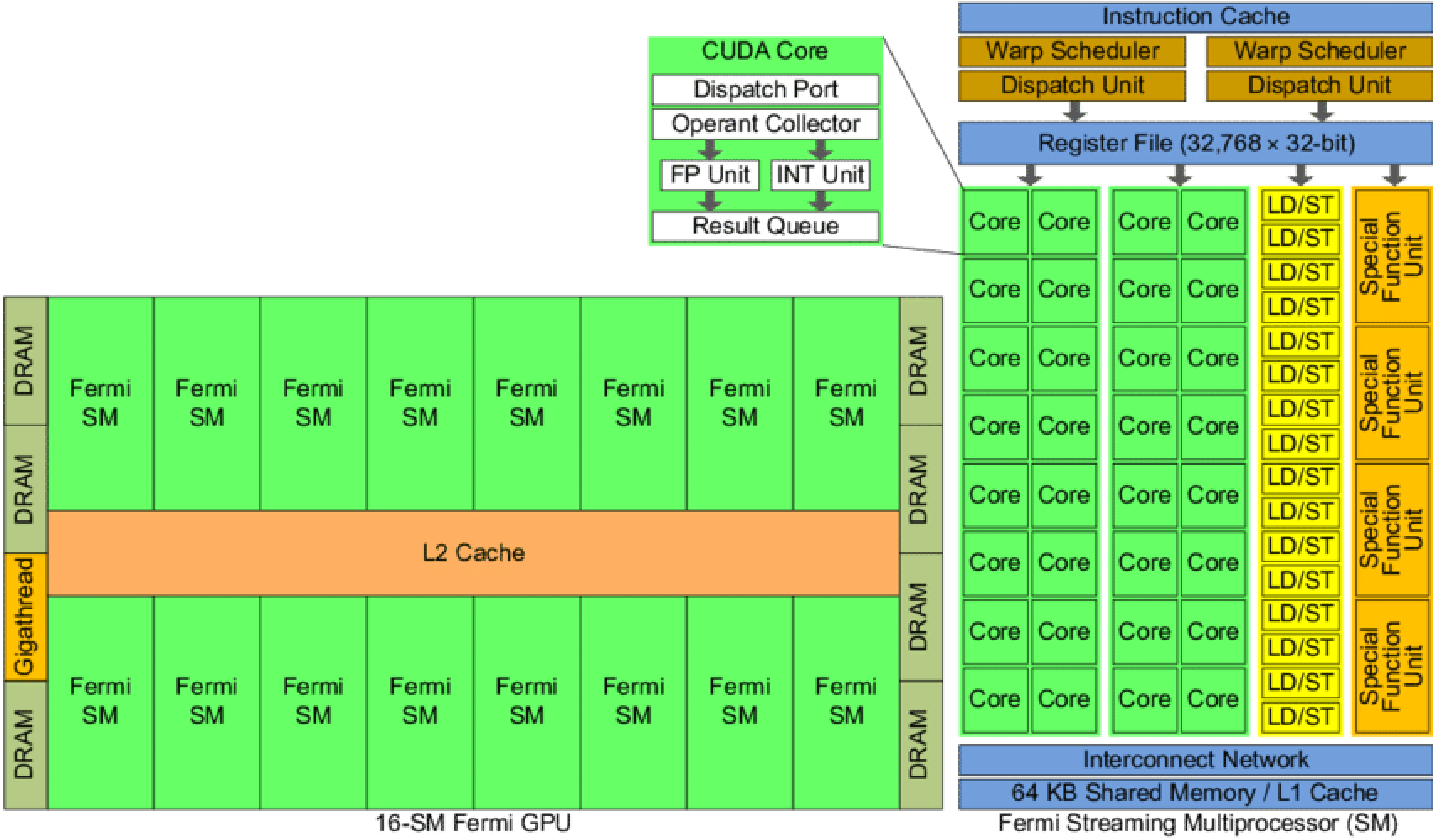
GPU Acceleration



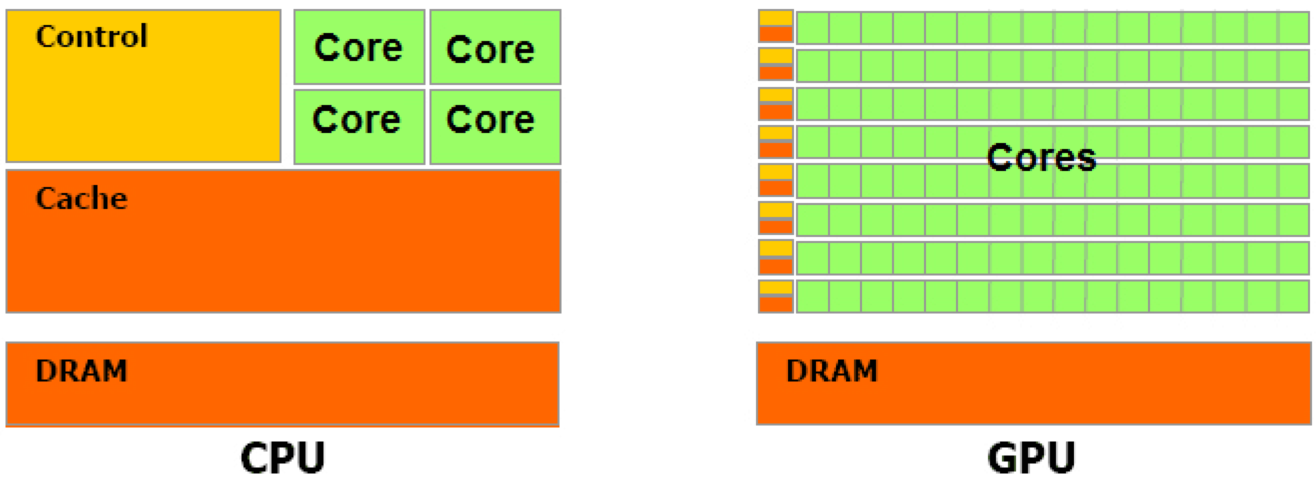
GPU Architecture



GPU Architecture Example



CPU vs GPU



Latency vs Throughput

Latency

Elapsed time

Throughput

Events per unit time

Example: move people 10 miles

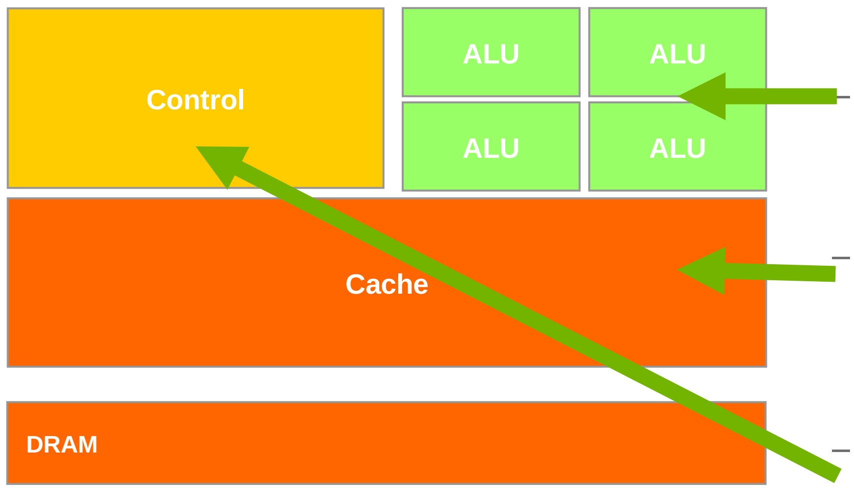
Car: capacity=5, speed=60 miles/hour

Bus: capacity=60, speed=20 miles/hour

Latency: car=10 min, bus=30 min

Throughput: car=15 PPH (count return), bus=60 PPH

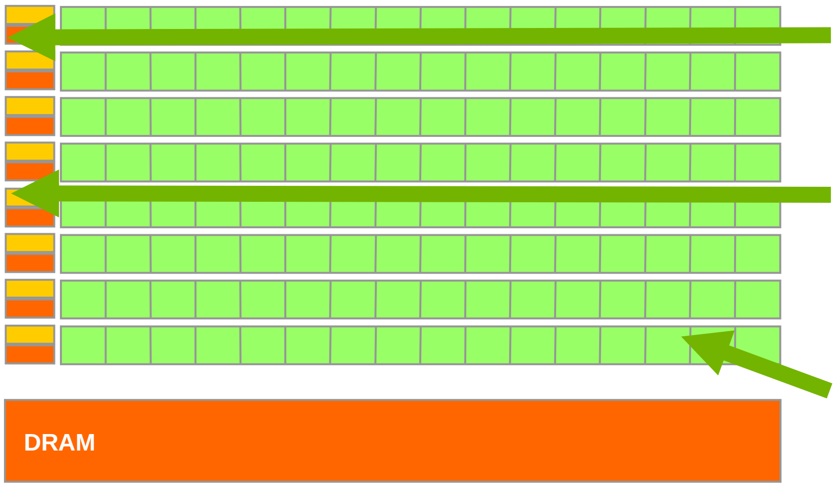
CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

minimize the execution latency of a single thread

GPUs: Throughput Oriented Design

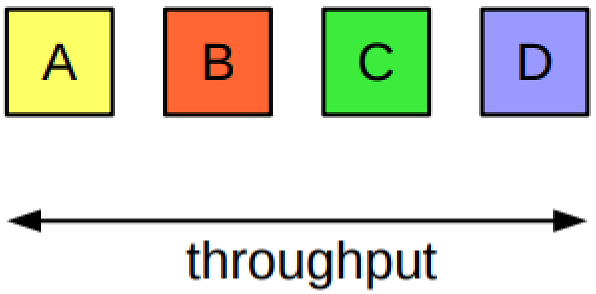


- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Uses of Parallelism

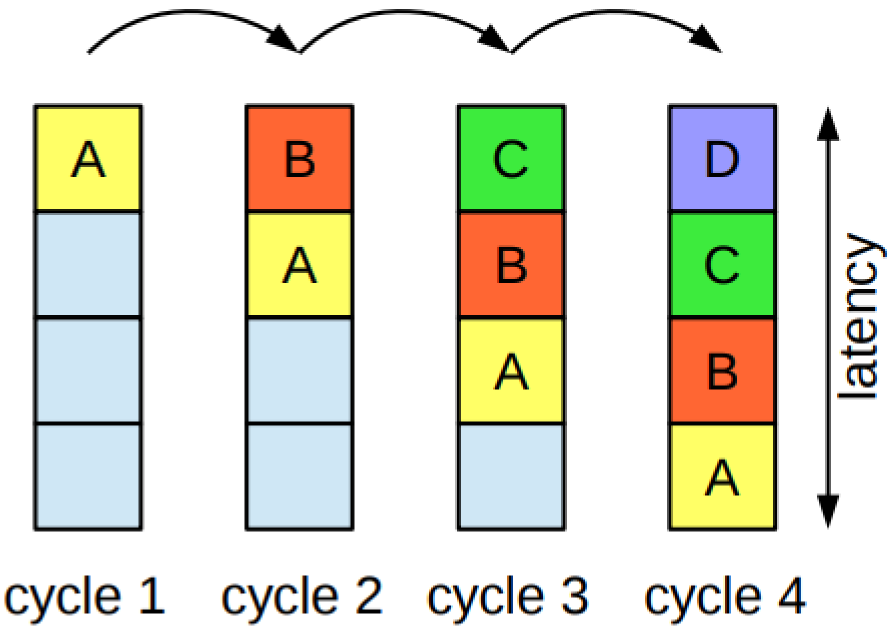
“Horizontal” parallelism for throughput

More units working in parallel



“Vertical” parallelism for latency hiding

Pipelining: keep units busy when waiting for dependencies, memory



Winning Applications with Both CPU and GPU

- **CPUs for sequential parts where latency matters**
  - CPUs can be 10X+ faster than GPUs for sequential code
- **GPUs for parallel parts where throughput wins**
  - GPUs can be 10X+ faster than CPUs for parallel code

GPU Advantages

a very large presence in the market place

Only a few elite applications funded by government and large corporations have been successfully developed on traditional parallel computing systems

GPUs have been sold by the hundreds of millions

practical form factors and easy accessibility

for medical imaging, fine to publish a paper based on a 64-node cluster machine

But real-world clinical applications on MRI machines utilize some combination of a PC and special hardware accelerators

GPU Advantages

executing numeric computing applications

- IEEE Floating-Point Standard was not strong in early GPUs
- Up to 2009, a major barrier was that the GPU floating-point arithmetic units were primarily single precision
- this has changed with the recent GPUs whose double precision execution speed approaches about half that of single precision, a level that only high-end CPU cores achieve

programming effort

- Until 2006, OpenGL or Direct3D techniques were needed
- Much easier with the release of CUDA

Simple Example

Add two arrays

A[ ] + B[ ] -> C[ ]

CPU:

```
float *C = malloc(N * sizeof(float));
for (int i = 0; i < N; i++)
C[i] = A[i] + B[i];
return C;
```

Parallel Version

(allocate memory for C)

Create # of threads equal to number of cores on processor (around 2, 4, perhaps 8)

(Indicate portions of A, B, C to each thread...)

In each thread,

    For (i from beginning region of thread)

        C[i] <- A[i] + B[i]

//lots of waiting involved for memory reads, writes, ...

Wait for threads to synchronize...

This is slightly faster – 2-8x (slightly more with other tricks)

Parallel Performance

How many threads? How does performance scale?

Context switching:

- The action of switching which thread is being processed
  - High penalty on the CPU
  - Not an issue on the GPU

GPU Version

(allocate memory for A, B, C on GPU)

Create the “kernel” – each thread will perform one (or a few) additions

Specify the following kernel operation:

    For all i‘s (indices) assigned to this thread:

        C[i] <- A[i] + B[i]

Start ~20000 (!) threads

Wait for threads to synchronize...

GPU Performance

We have lots of cores

This allows us to run many threads simultaneously with no context switches

In a typical system, thousands of threads are queued up for work. If the GPU must wait on one group of threads, it simply begins executing work on another.

Parallel Programming

design parallel algorithms with the same level of algorithmic (computational) complexity as sequential algorithms (large parallel overheads)

the execution speed of many applications is limited by memory access speed

the execution speed of parallel programs is often more sensitive to the input data characteristics than their sequential counter parts

Parallel Programming Models

Distributed-memory programming

- MPI, PVM

Shared-memory programming

- Pthreads, OpenMP

GPU programming

- CUDA, OpenCL

MapReduce programming

- Hadoop

CUDA

CUDA (Compute Unified Device Architecture)

Enables a general purpose programming model on NVIDIA GPUs

Enables explicit GPU memory management

GPU threads are extremely lightweight

- Very little creation overhead

GPU needs 1000s of threads for full efficiency

- Multi-core CPU needs only a few

CUDA Program

CUDA platform is accessible through CUDA-accelerated libraries and APIs

CUDA C is an extension of standard ANSI C with a handful of language extensions to enable heterogeneous programming, and also straightforward APIs to manage devices, memory, and other tasks

Integrated host+device app C program

- Serial or modestly parallel parts in host C code
- Highly parallel parts in device SPMD kernel C code