CENG443
Heterogeneous Parallel Programming

Introduction

Işıl ÖZ, IZTECH, Fall 2023

06 October 2023

Programming Model

CUDA (Compute Unified Device Architecture) C

General purpose programming model for NVIDIA GPUs

Parallel computing platform and application programming interface (API)

Standard C Code

```c
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;


// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

C with CUDA extensions

```c
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```

Development Environment

CUDA Toolkit, nvcc compiler

Your PC with an NVIDIA GPU - Preferable

Google Colab

Amazon Cloud (AWS credit from NVIDIA)

- Contact me if you need this

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 impact of climate change

Video and audio coding and manipulation

UHD TV

User interfaces

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

Big data applications

Processing huge amount of data

Single Processor Performance

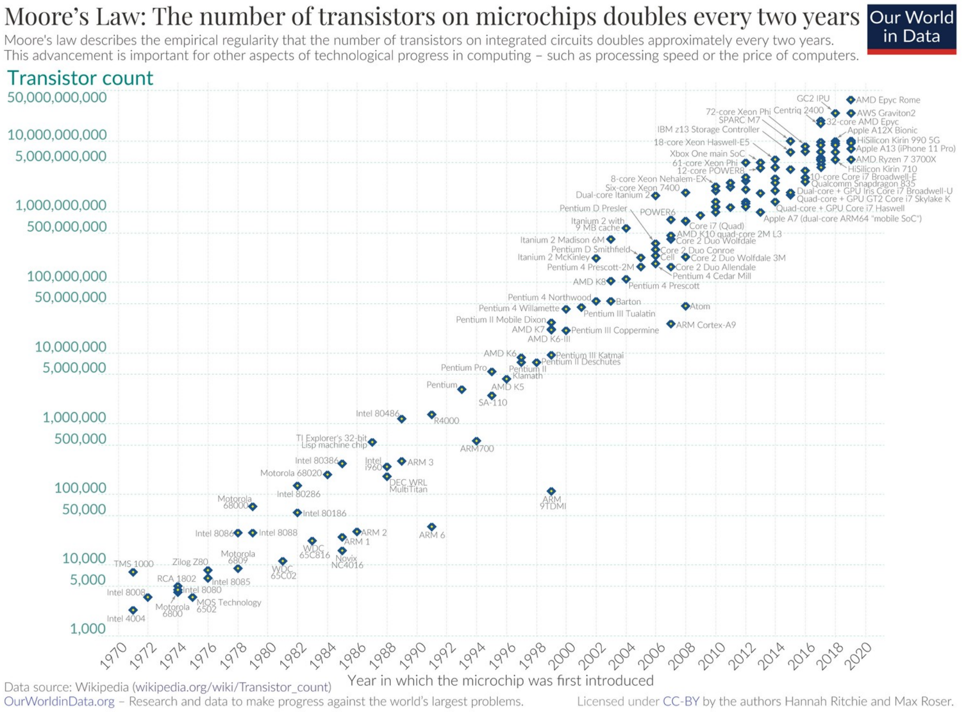
From 1986-2002, microprocessors' performance was increasing an average of 50% per year

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 Gordon Moore in 1965, that the number of transistors in a dense integrated circuit doubles approximately every two years



Cooking-Aware Computing

Power Wall

More/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"

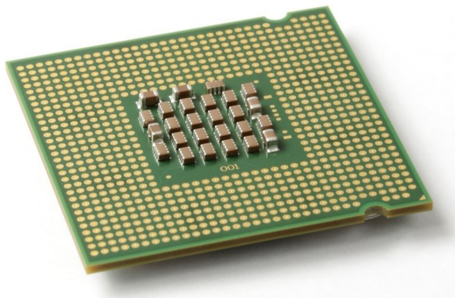
Traditionally, applications use CPU for primary calculations

General-purpose capabilities

Established technology

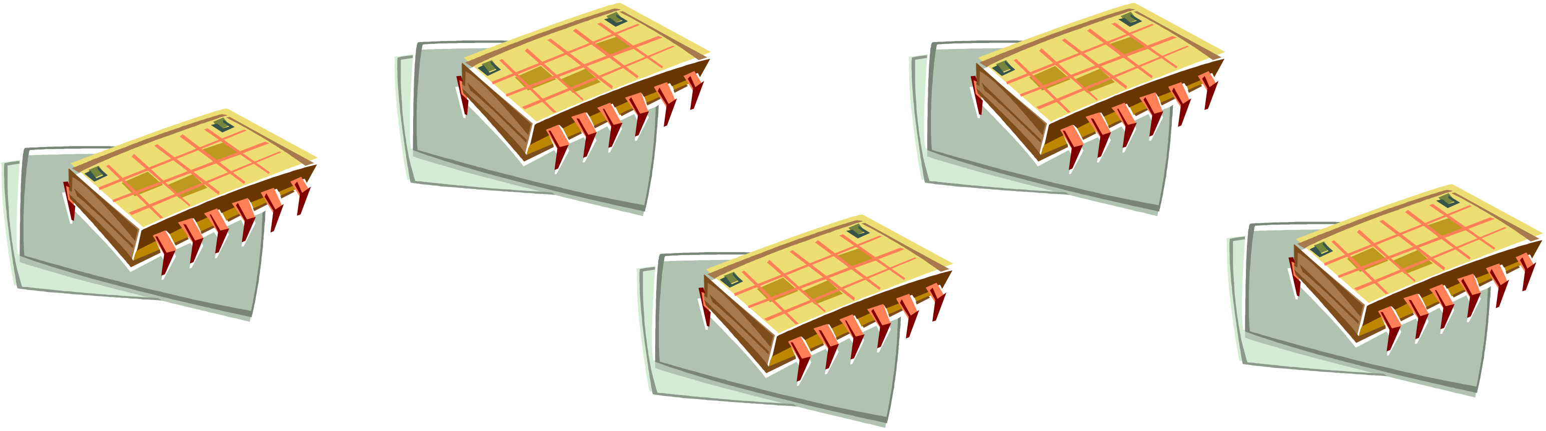
Usually equipped with 8 or less powerful cores

Optimal for concurrent processes but not large scale parallel computations



Multicore Systems

Instead of designing and building faster microprocessors, put multiple processors (each core is less powerful than the previous generation's single core design) on a single integrated circuit



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

GeForce 256, world's first GPU, in 1999 by NVIDIA

GPUs

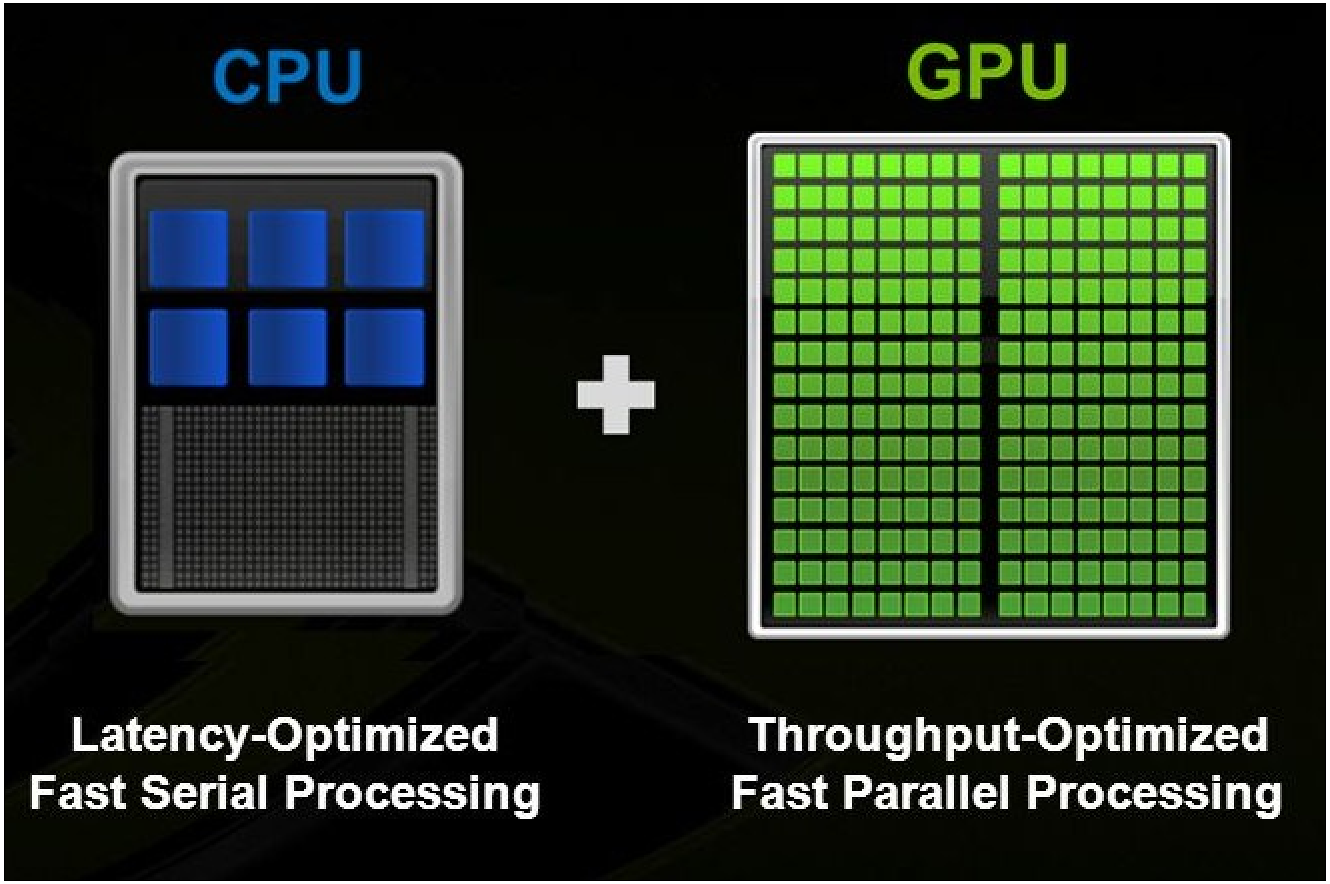
Everywhere from inside smartphones, laptops, datacenters, and all the way to supercomputers

Initially, real-time rendering with a focus on video games

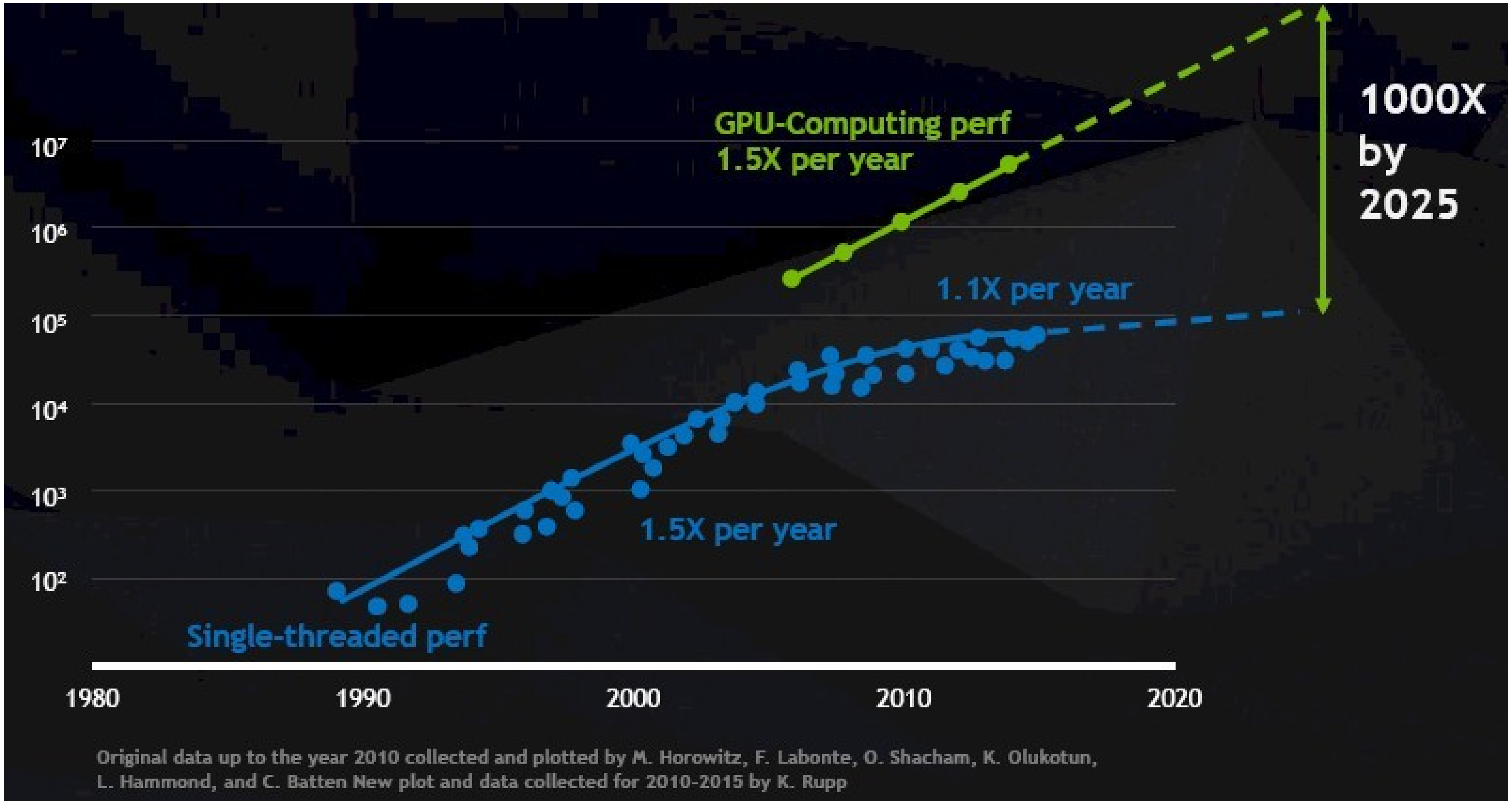
Increasingly support non-graphics computing

Refined GPU architectures and programming model to increase flexibility as well as energy efficiency

Heterogeneous Parallel Computing

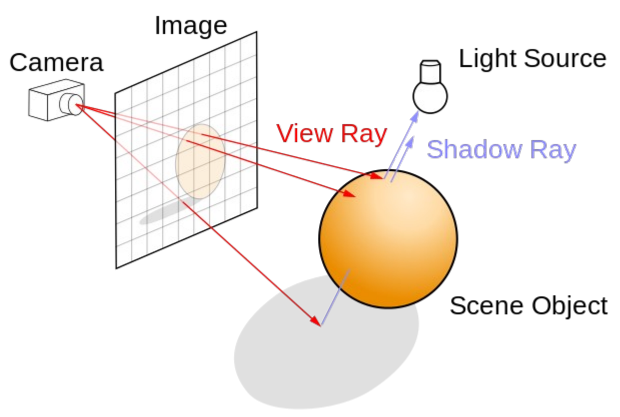


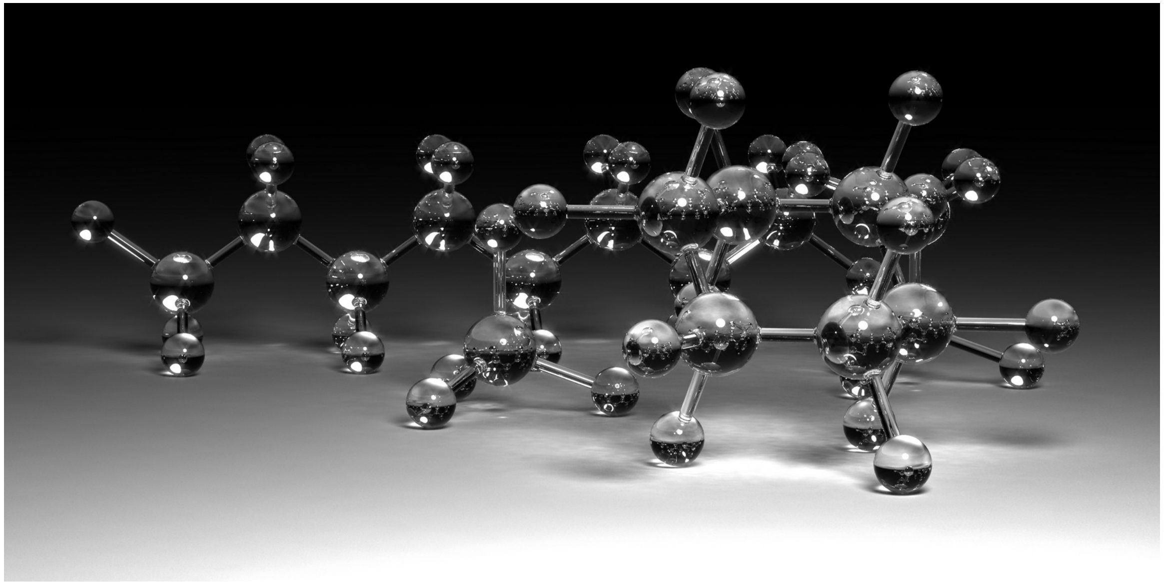
CPU vs GPU Performance



Example: Raytracing

Enables real-time light reflections and cinematic effects in games





https://www.youtube.com/watch?v=gq1klB4inoc

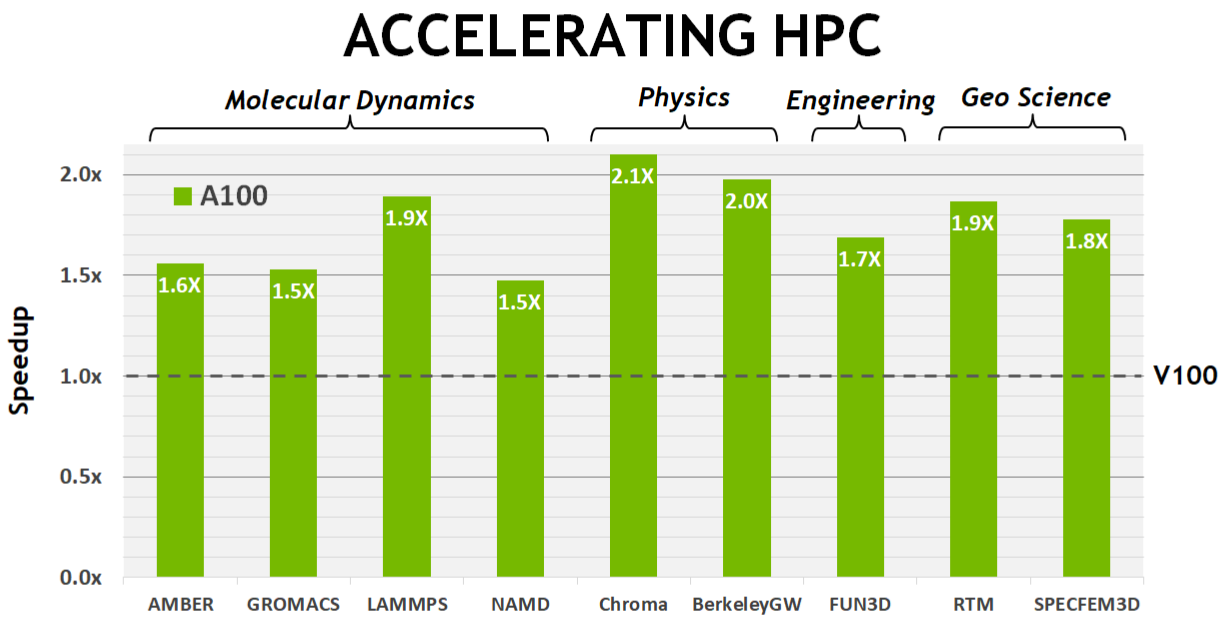
GPGPU

General-Purpose Computation on GPUs

The GPU is no longer just for graphics

- Particle systems, collision detection
- Fluid dynamics
- Simulation
- Neural network

High Performance Computing



All results are measured
Except BerkeleyGW, V100 used is single V100 SXM2. A100 used is single A100 SXM4
More apps detail: AMBER based on PME-Cellulose, GROMACS with STMV (h-bond), LAMMPS with Atomic Fluid LJ-2.5, NAMD with v3.0a1 STMV_NVE
Chroma with szscl21_24_128, FUN3D with dpw, RTM with Isotropic Radius 4 1024^3, SPECFEM3D with Cartesian four material model
BerkeleyGW based on Chi Sum and uses 8xV100 in DGX-1, vs 8xA100 in DGX A100

Deep Learning

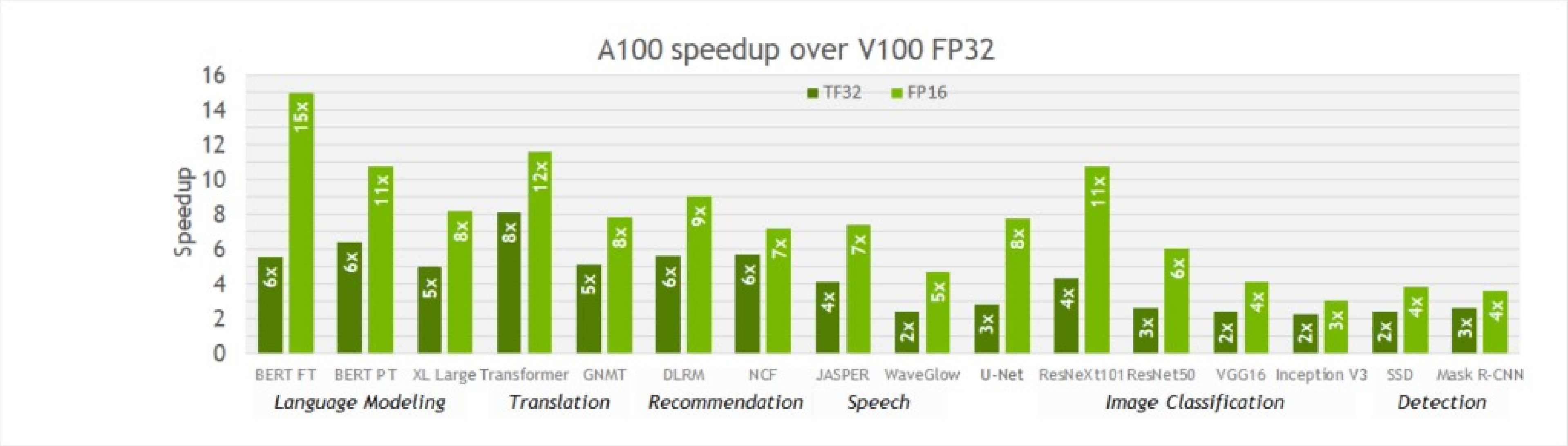
Why neural networks become very popular in recent years, though much theory was developed 20 years ago?

Huge amount of data

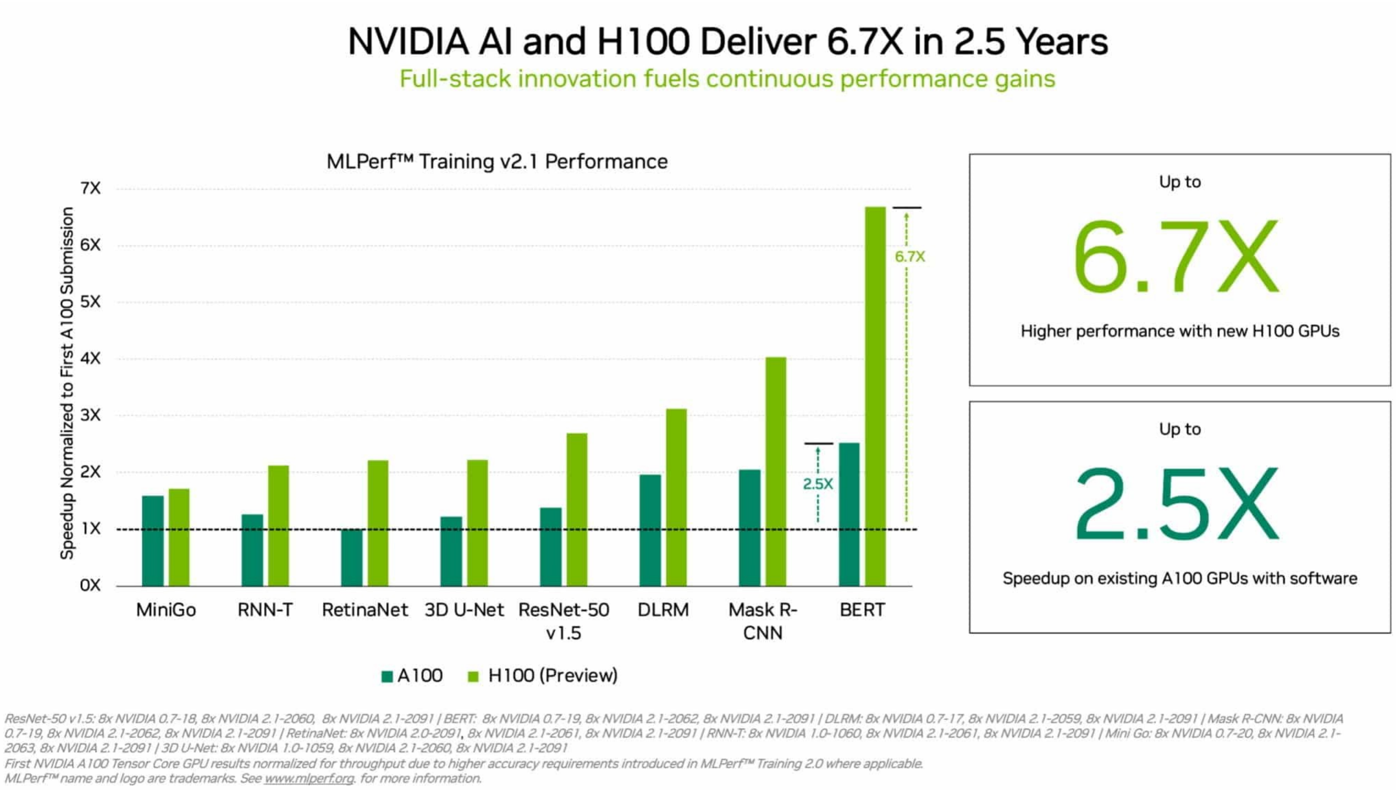
Computational power

https://www.youtube.com/watch?v=DiGB5uAYKAg

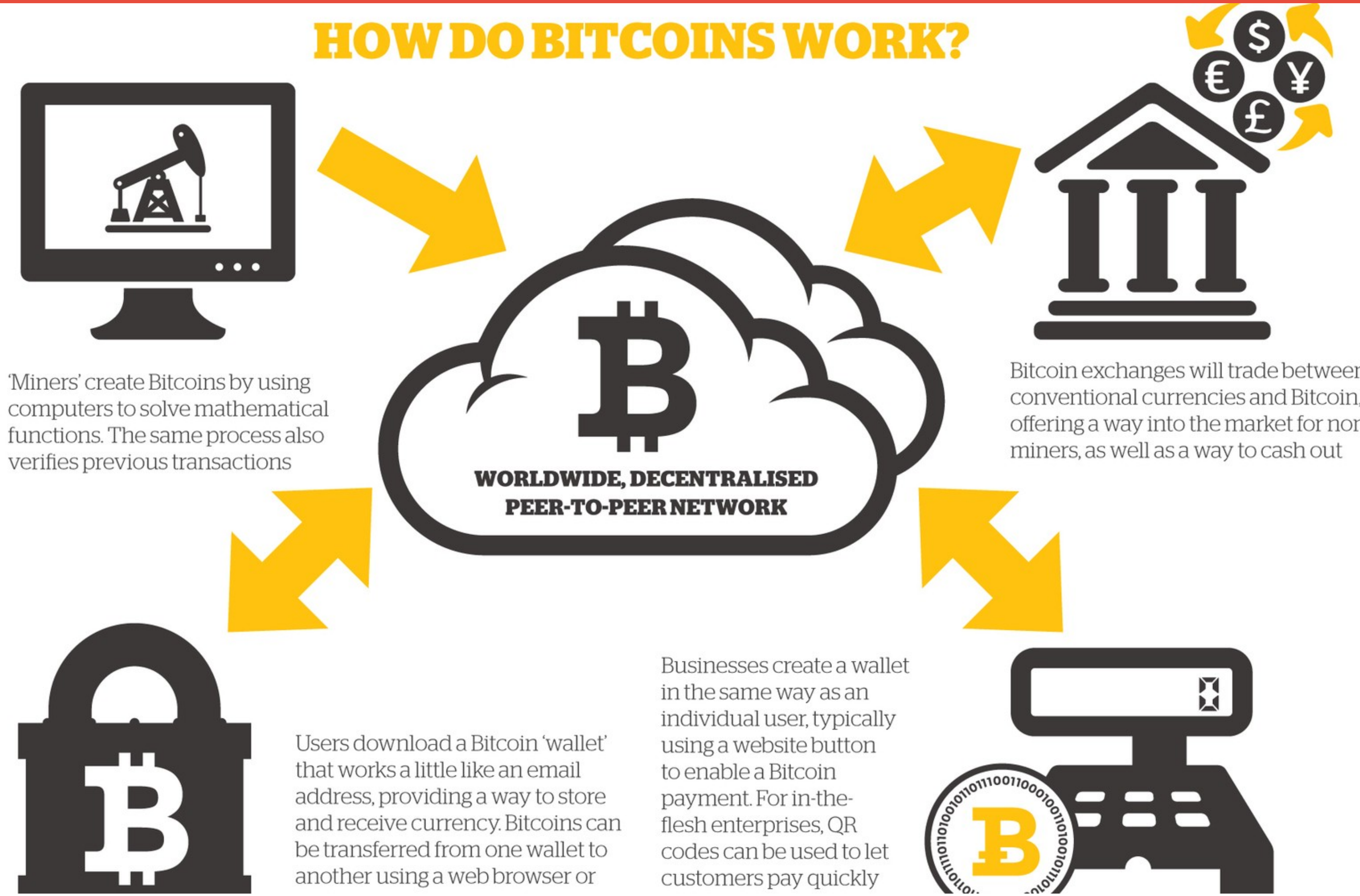
Deep Learning



Deep Learning



Bitcoin Mining



Embedded Systems

Apple A17 Bionic

Mobile chip inside iPhone 15

64-bit ARM-based system on a chip (SoC) designed by Apple

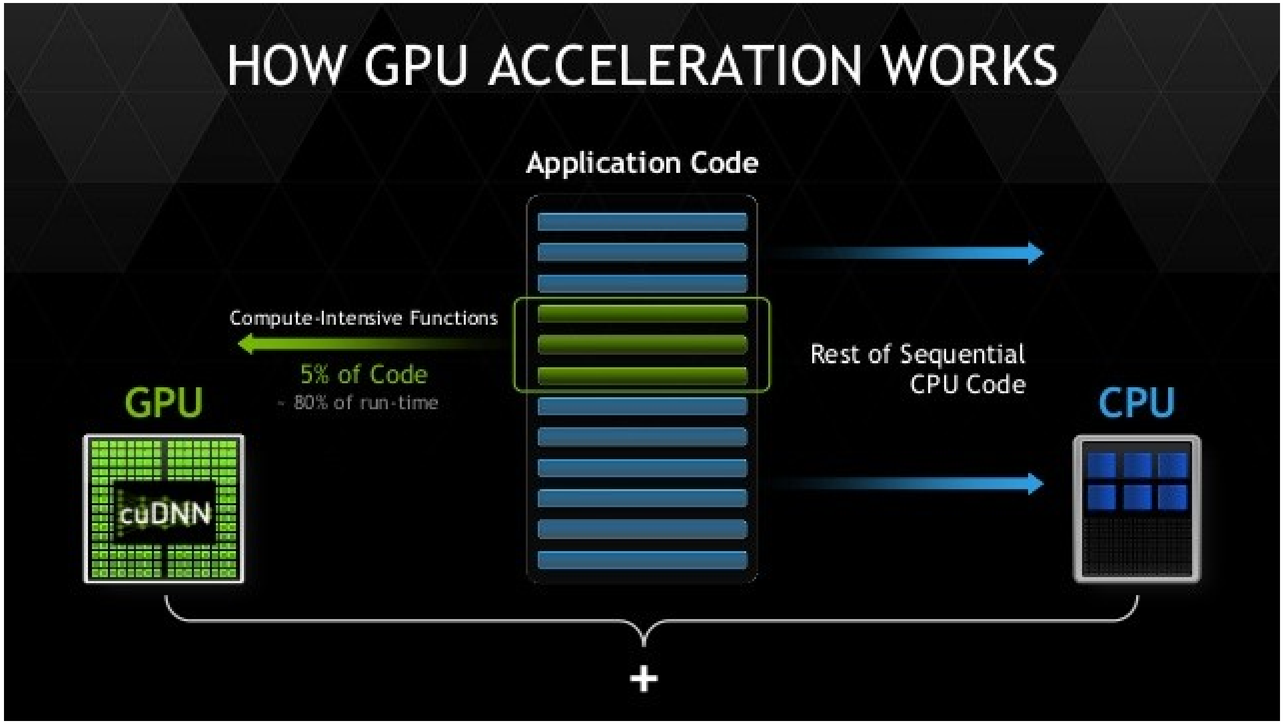
6-core multicore

- Two high-performance cores (3.78 GHz)
- Four low-power energy-efficient cores (2.11 GHz)

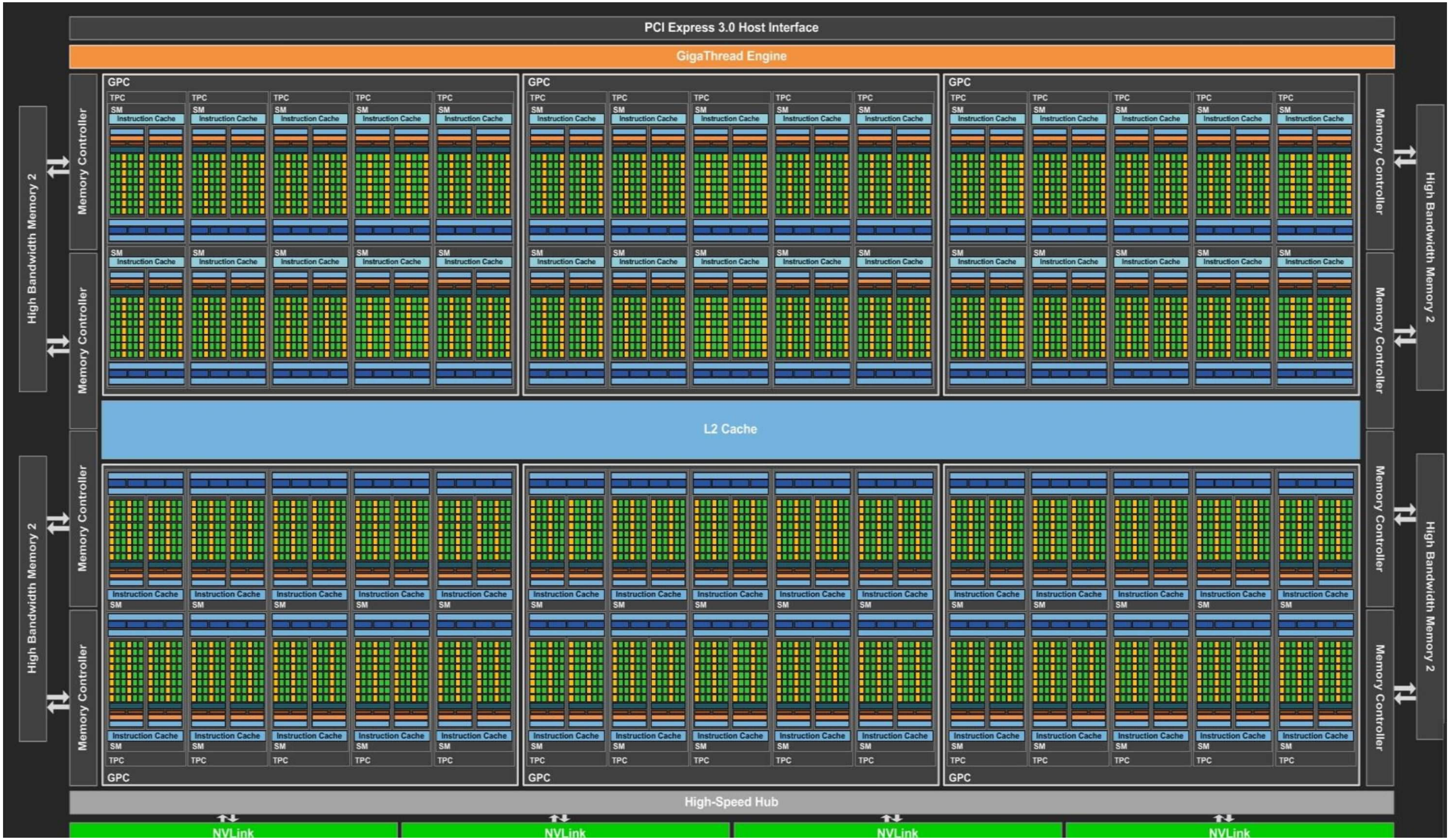
6-core GPU

16-core neural engine

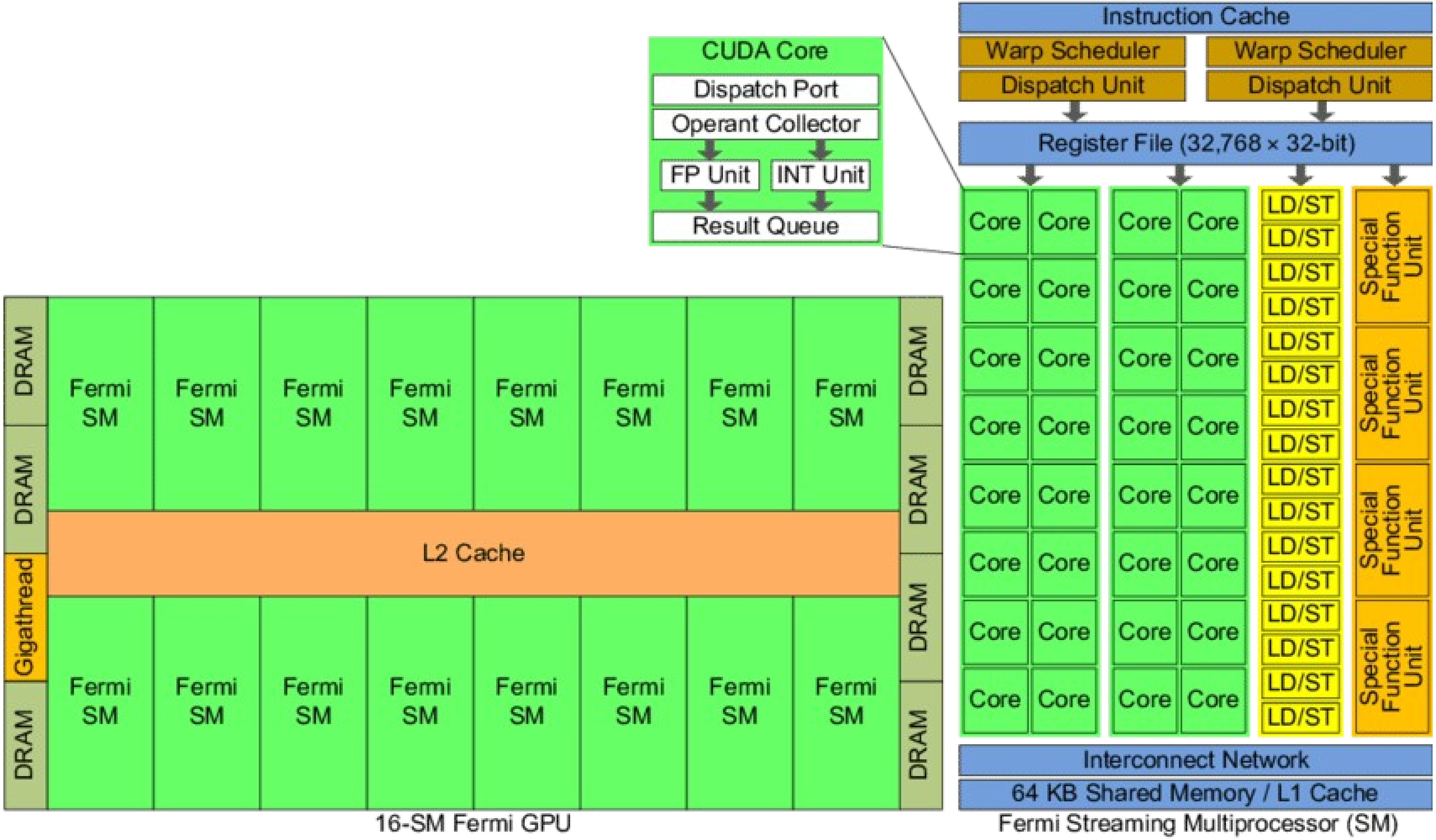
GPU Acceleration



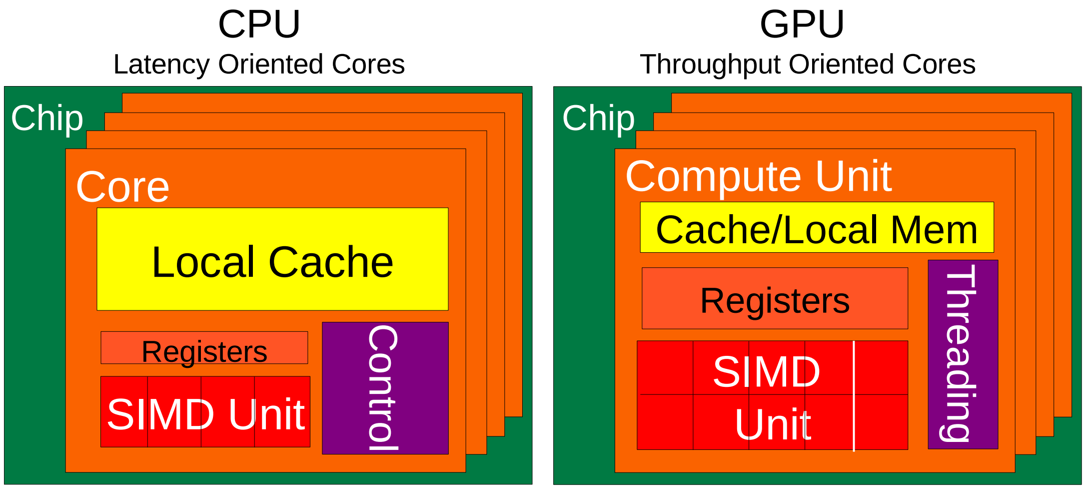
GPU Architecture



GPU Architecture Example



CPU vs GPU


CPU
Latency Oriented Cores
Chip
Core
Local Cache
Registers
SIMD Unit
Control
GPU
Throughput Oriented Cores
Chip
Compute Unit
Cache/Local Mem
Registers
SIMD
Unit
Threading

Latency vs Throughput

Latency

Elapsed time

Throughput

Events per unit time

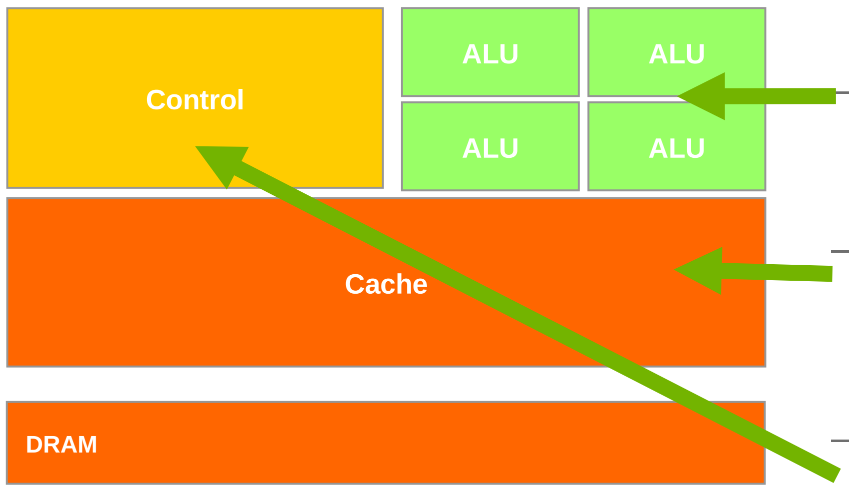
Example: perform 100 multiplications

Single-core CPU: 1 mul = 1 sec, 100 mul = 100 sec

Quad-core CPU: 1 mul = 2 sec, 100 mul = 50 sec

GPU (50 PUs): 1 mul = 5 sec, 100 mul = 10 sec

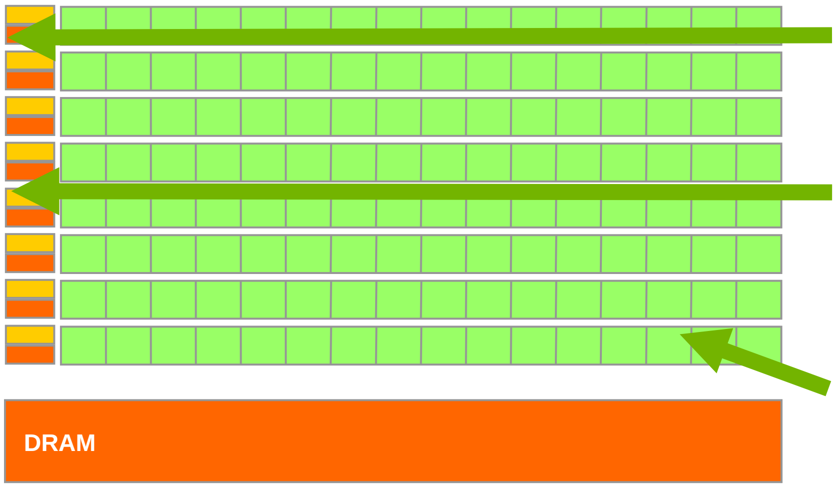
CPUs: Latency Oriented Design



- 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

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

Problems with GPUs

Need enough parallelism

Data-centric applications

Under-utilization

Memory bottleneck

Bandwidth to CPU

Separate memories

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

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 language extensions to enable heterogeneous programming, and also 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