University of BRISTOL

Tutorial on GPU computing

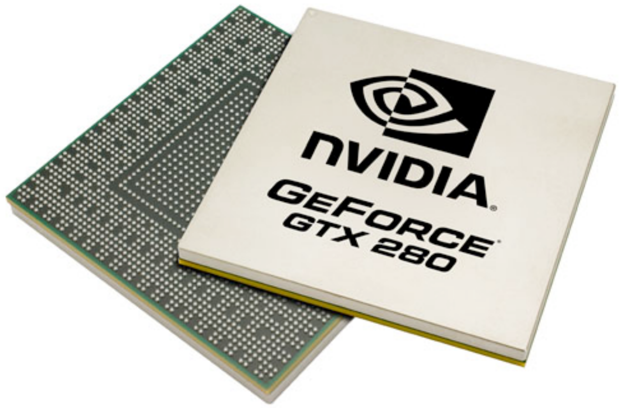
With an introduction to CUDA

Felipe A. Cruz

University of Bristol, Bristol, United Kingdom.

Marco Ferretti acknoledges authorship and applies some minor changes

The GPU evolution

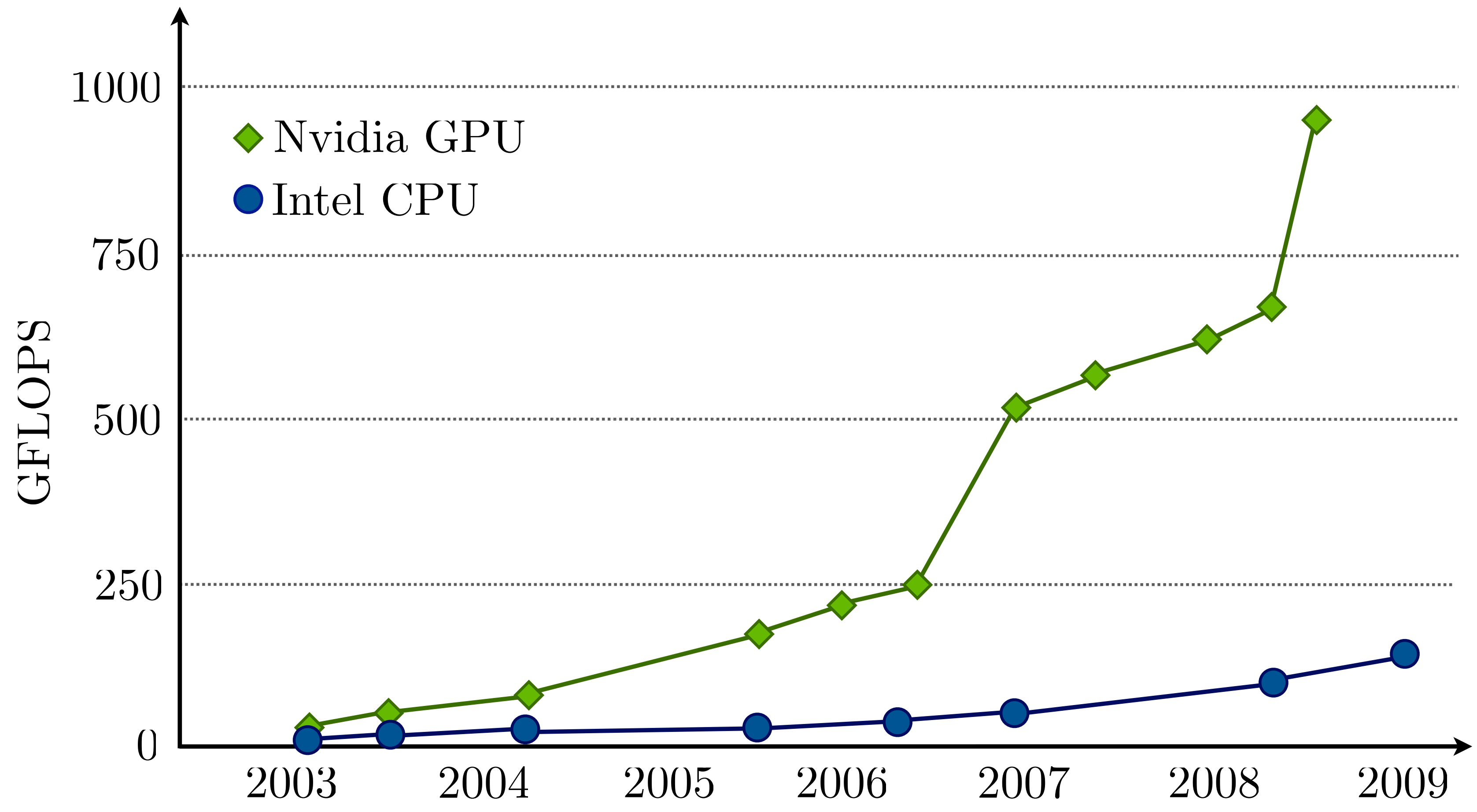


- The **Graphic Processing Unit** (GPU) is a processor that was **specialized** for processing graphics.
- The GPU has recently **evolved** towards a **more flexible** architecture.
- **Opportunity**: We can implement ***any algorithm***, not only graphics.
- **Challenge**: obtain **efficiency** and **high performance**.

The motivation

GPU computing - key ideas:

- Massively parallel.
- Hundreds of cores.
- Thousands of threads.
- Cheap.
- Highly available.
- Programable: CUDA



CUDA: Compute Unified Device Architecture

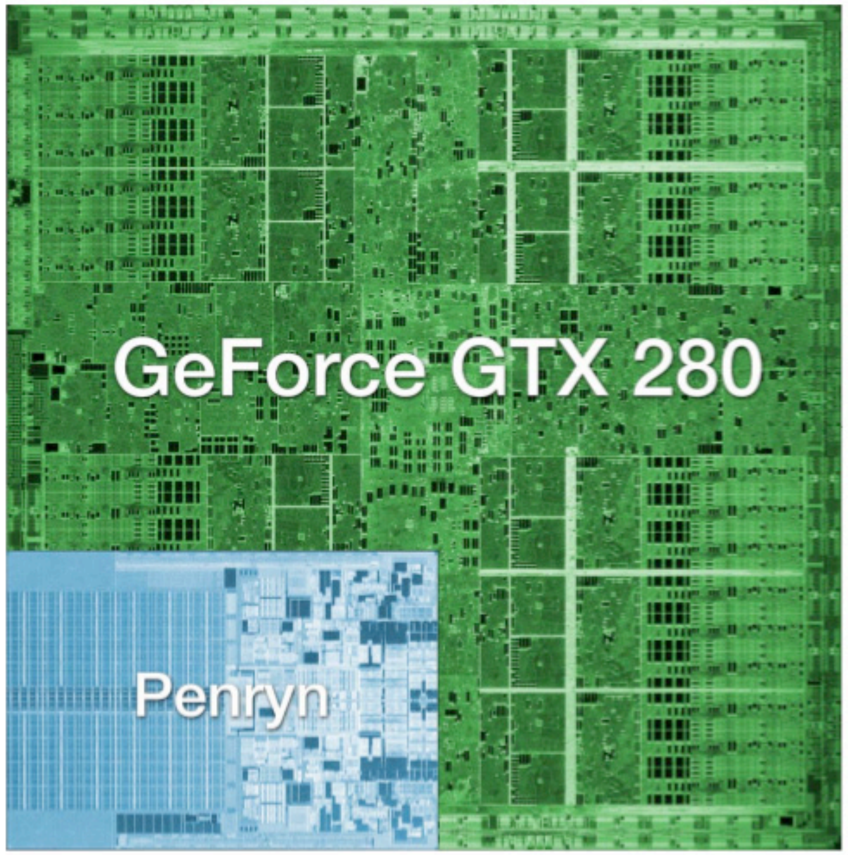
- Introduced by Nvidia in late 2006.
- CUDA is a **compiler and toolkit** for programming NVIDIA GPUs.
- CUDA API **extends the C** programming language.
- Runs on **thousands of threads**.
- It is an **scalable model**.
- Objectives:
  - **Express parallelism**.
  - Give a high level **abstraction from hardware**.

NVIDIA: GPU vendor

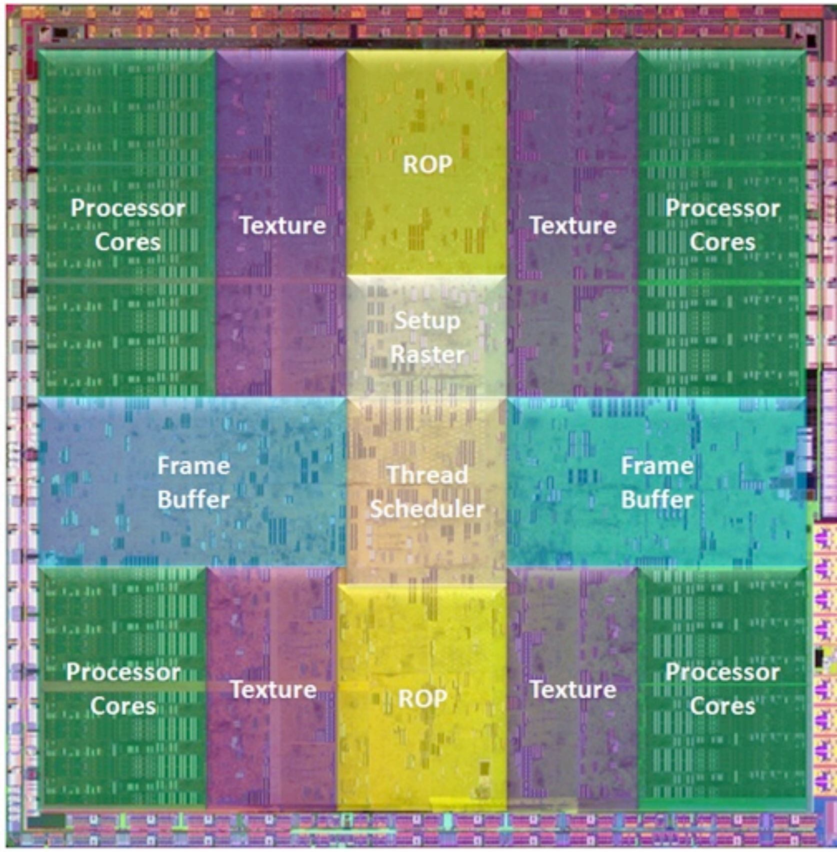
- GPU **market: multi-billion** dollars! (Nvidia **+30% market**)
- **Sold hundreds of millions** of CUDA-capable GPUs.
  - HPC market is tiny in comparison.
- New GPU generation every ~**18 months**.
- **Strong support** to GPU computing:
  - Hardware side: **developing flexible GPUs**.
  - Software side: releasing and improving **development tools**.
  - Community side: **support to academics**.
- Links: www.nvidia.com, http://www.nvidia.com/object/cuda_home.html

How a GPU looks like?

- **Most computers have one.**
- **Billions of transistors.**
- Computing:
    - **1 Teraflop** (Single precision)
    - **100 Gflops** (Double precision
- Also:
    - A **heater for winter** time!
- Supercomputer for the masses?



Die comparison



Chip areas


Tesla card

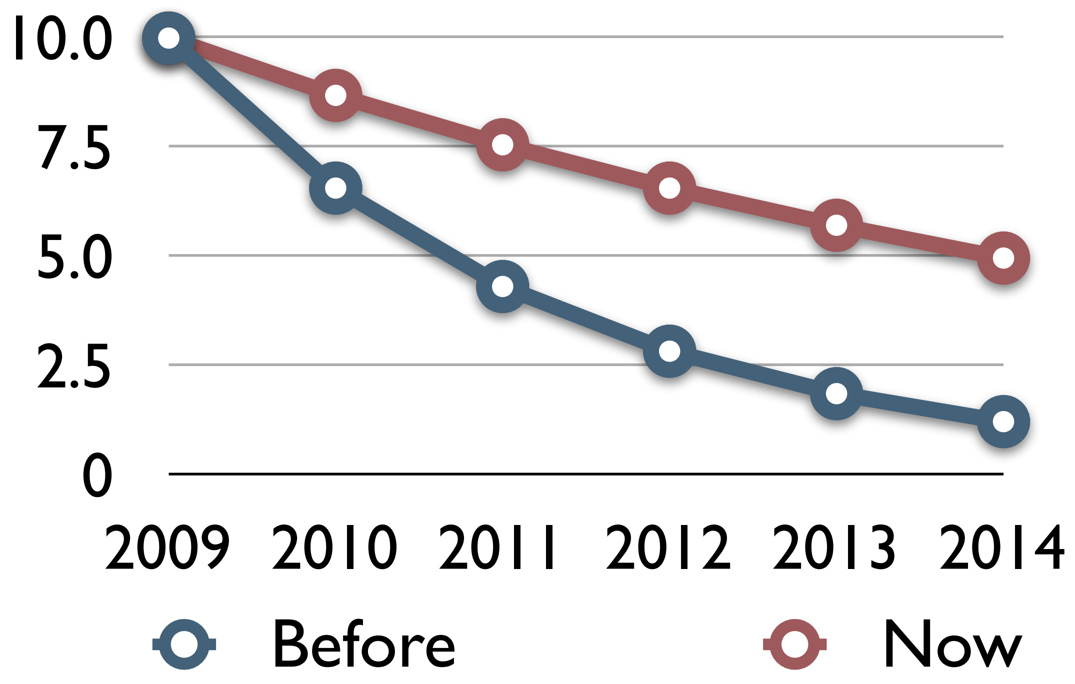

Tesla S1070: 4 cards

Ok... after the buzz

- Question 1: **Why accelerator technology today?** If it has been around since the 70's!
- Question 2: **Can I really get 100x in my application?**
- Question 3: **CUDA? vendor dependent?**
- Question 4: **GPU computing = General-purpose on GPU?**

Why accelerator technology today?

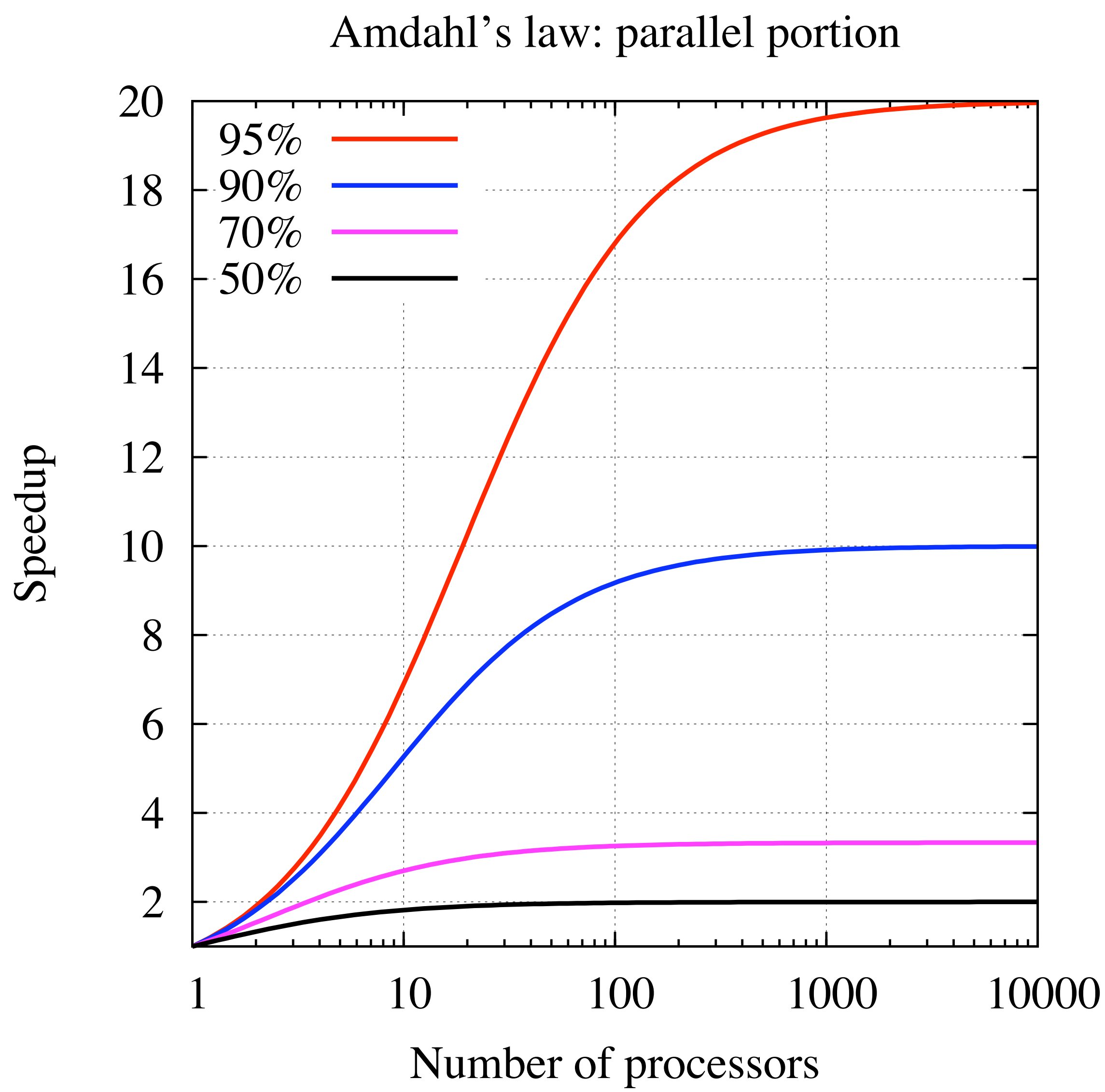
- Investment on GPU technology makes **more sense today** than in 2004.
- CPU **uni-processor speed is not doubling** every 2 years anymore!
- Case: **investing in an accelerator** that gives a ~10x speedup:



  - **2004** speedup **1.52x** per year: 10x today would be **1.3x** acceleration in 5 years.
  - **TODAY** speedup **1.15x** per year: 10x today would be **4.9x** acceleration in 5 years.
  - Consider the point that **GPU parallel performance is doubling** every 18 months!

Can I get 100x speedups?

- **You can** get hundred-fold speedup **for some** algorithms.
- It depends on the non-parallel part: **Amdahl's law**.
- Complex application normally make use of many algorithms.
- Look for **alternative ways** to perform the computations that are more parallel.
- **Significance**: An accelerated program is going to be as fast as its serial part!



Amdahl's Law

Maximum speedup

CUDA language is vendor dependent?

- Yes, and nobody wants to **locked to a single vendor**.
- OpenCL is **going to become an industry standard**. (Some time in the future.)
- OpenCL is a **low level specification**, **more complex to program with** than CUDA C.
- **CUDA C is more mature** and currently makes more sense (to me).
- However, OpenCL is not **“*that*”** different from CUDA. **Porting CUDA to OpenCL** should be easy in the future.
- **Personally, I’ll wait** until OpenCL standard & tools are more mature.

GPU computing = General-purpose GPU?

- With CUDA **you can program in C** but with some restrictions.
- **Next CUDA** generation will have **full support C/C++** (and much more.)
- However, **GPU** are still **highly specialized** hardware.
- Performance in the GPU does not come from the flexibility...

GPU computing features

- **Fast GPU cycle**: New hardware every ~18 months.
- Requires **special programming** but similar to **C**.
- CUDA code is **forward compatible** with future hardware.
- **Cheap** and available hardware (£200 to £1000).
- **Number crunching**: 1 card ~= 1 teraflop ~= small cluster.
- **Small factor** of the GPU.
- Important factors to consider: **power** and **cooling**!

CUDA introduction

with images from CUDA programming guide

Felipe A. Cruz

What's better?

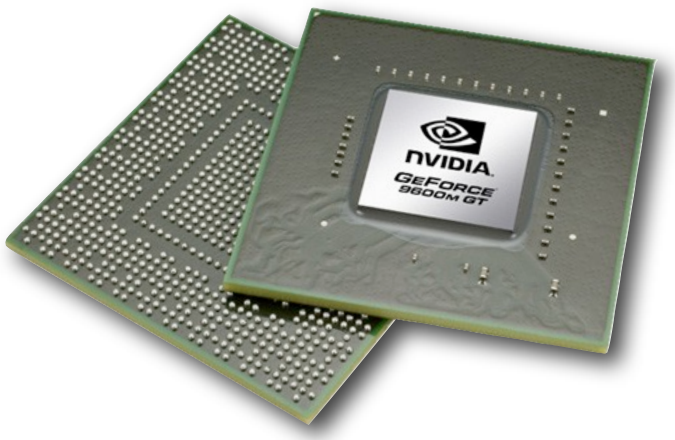
Many scooters

Sport car

Deliver many packages within a reasonable timescale.

Deliver a package as soon as possible

What do you need?



High throughput
and
reasonable latency

Low latency
and
reasonable throughput

Compute many jobs
within a reasonable timeframe.

Compute a job as
fast as possible.

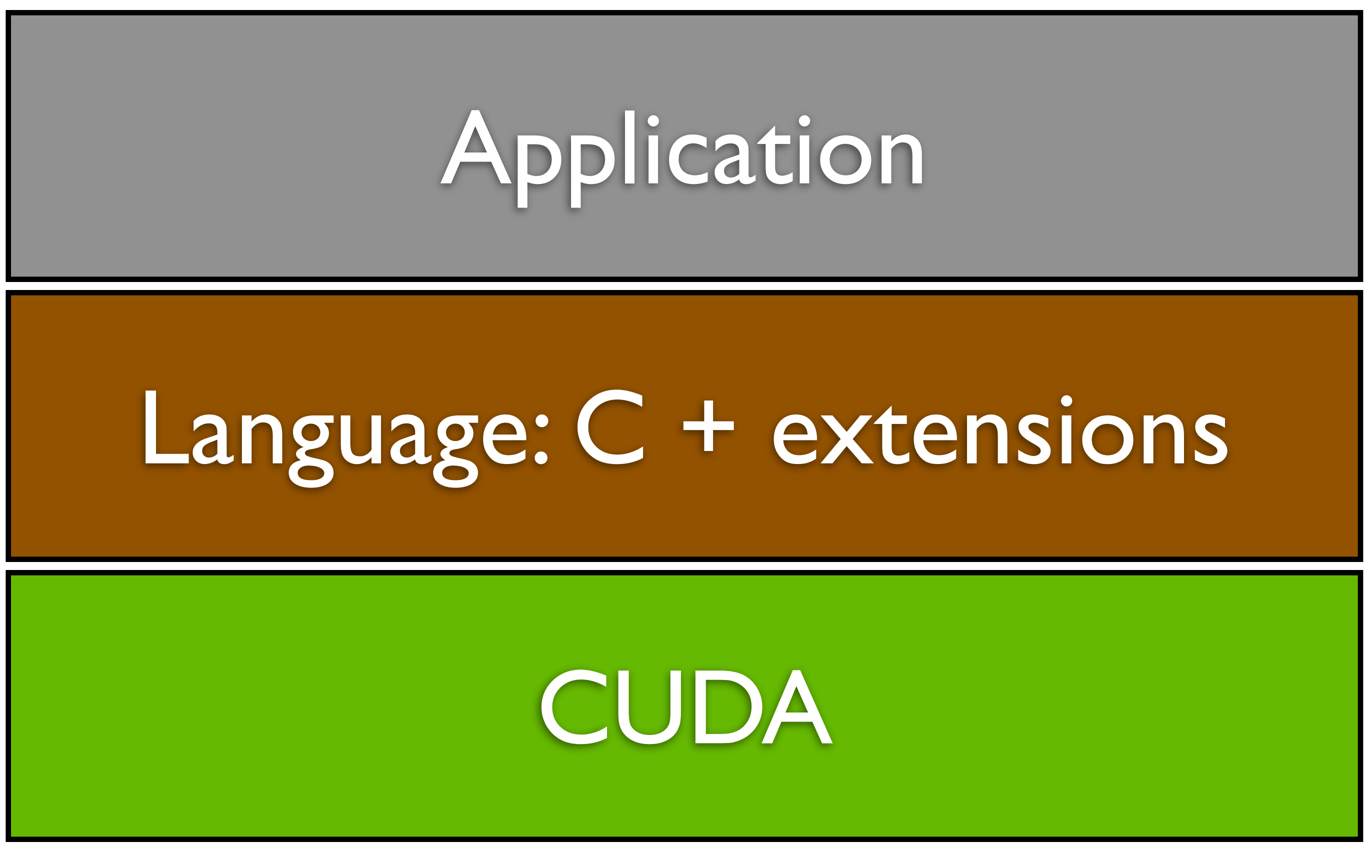
NVIDIA GPU Architecture

GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision Floating Point Capability	None	30 FMA ops / clock	256 FMA ops /clock
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops / clock	512 FMA ops /clock
Special Function Units (SFUs) / SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Configurable 48 KB or 16 KB
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

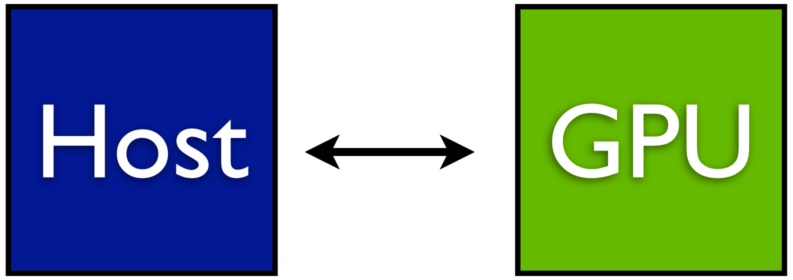
Comparison of NVIDIA GPU generations. Current generation: GT200. Table from NVIDIA Fermi whitepaper.

CUDA architecture

- Support of languages: C, C++, OpenCL.
- Windows, linux, OS X compatible.



Architecture



CPU and GPU model

University of BRISTOL

Strong points of CUDA

- **Abstracting from the hardware**
  - Abstraction by the **CUDA API**. You don't see every little aspect of the machine.
  - Gives **flexibility to the vendor**. Change hardware but keep legacy code.
    - **Forward compatible**.
- **Automatic Thread management** (can handle **+100k threads**)
  - **Multithreading**: hides latency and helps maximize the GPU utilization.
  - Transparent for the programmer (you don't worry about this.)
  - Limited **synchronization between threads** is provided.
  - **Difficult to dead-lock**. (No message passing!)

Programmer effort

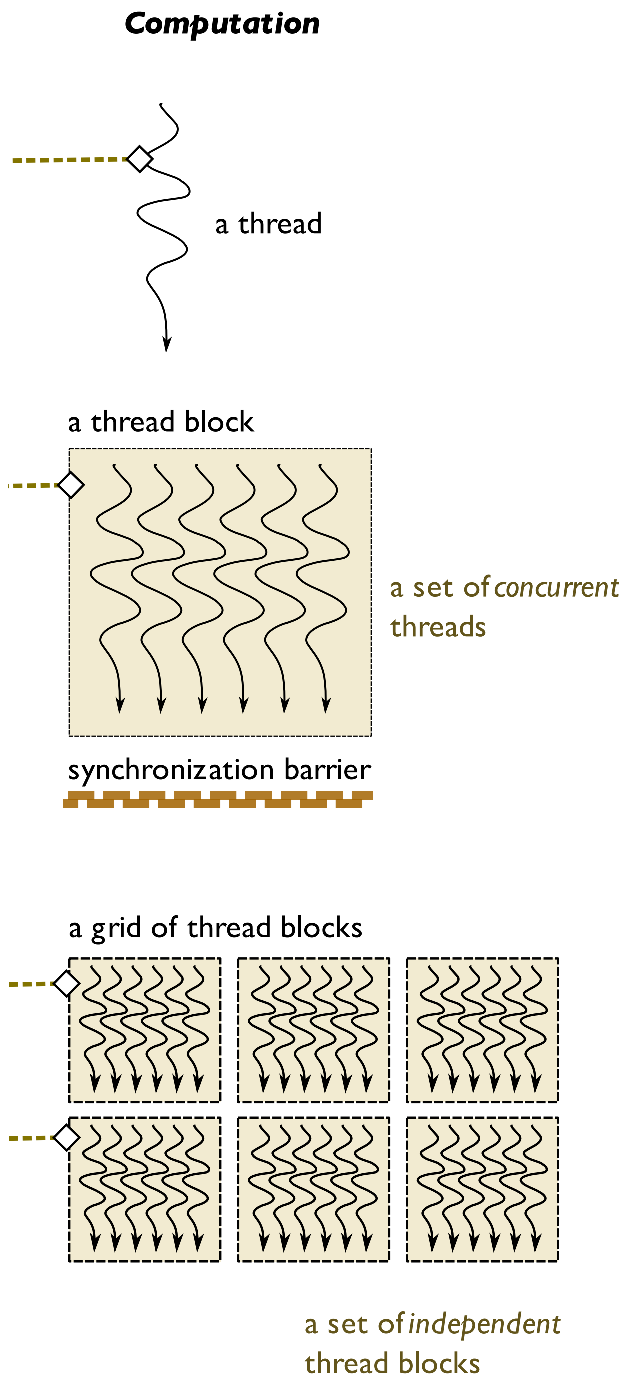
- Analyze algorithm for **exposing parallelism**:
    - Block size
    - Number of threads
    - **Tool: pen and paper**
- Challenge: **Keep machine busy** (with limited resources)
    - Global data set (Have efficient data transfers)
    - Local data set (Limited on-chip memory)
    - Register space (Limited on-chip memory)
    - **Tool: Occupancy calculator**

Outline

- Memory hierarchy.
- Thread hierarchy.
- Basic C extensions.
- GPU execution.
- Resources.

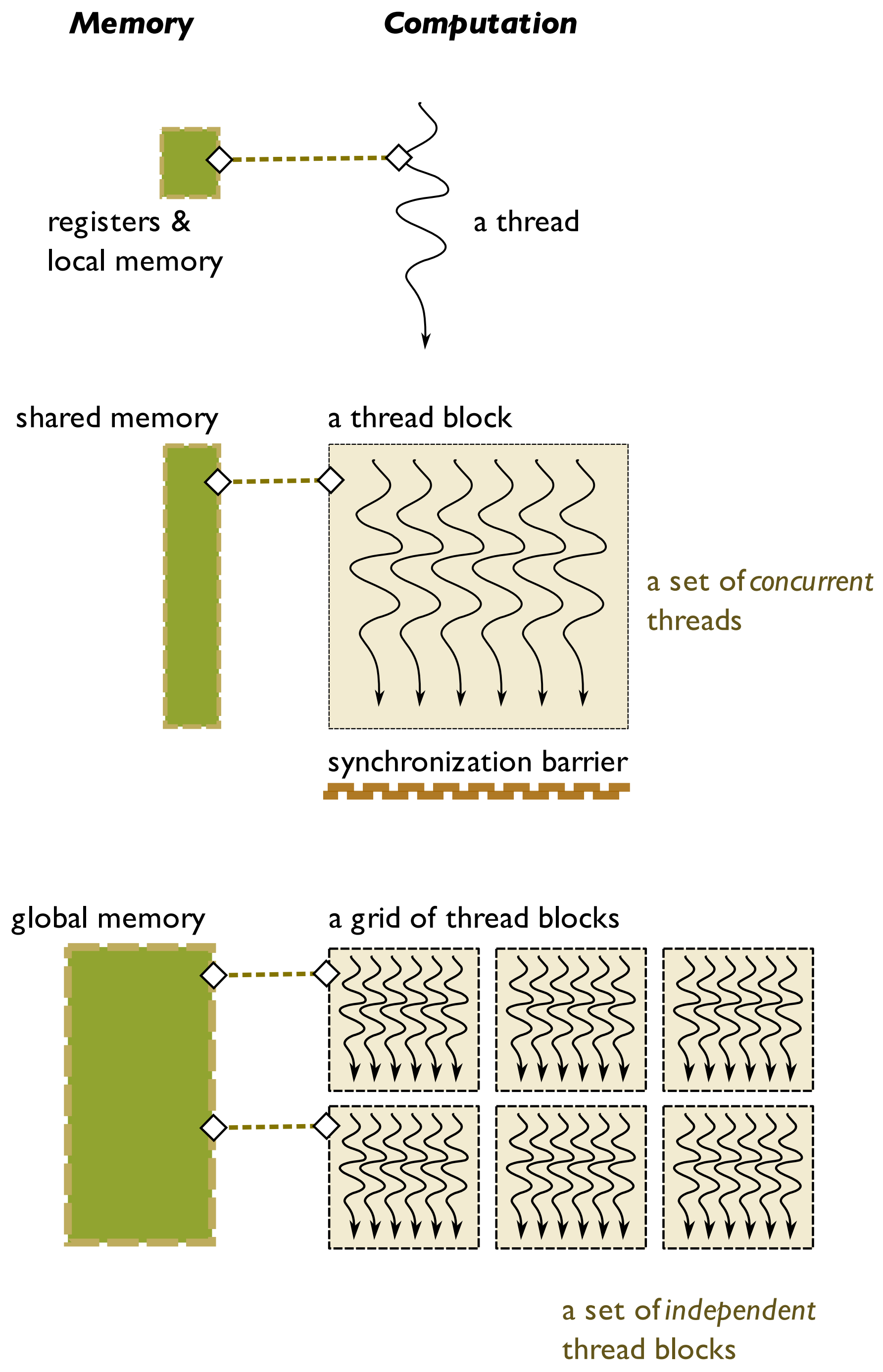
Thread hierarchy

- Kernels are executed by **thread**.
  - A kernel is a **simple C** program.
  - Each thread has it **own ID**.
  - **Thousands** of threads execute same kernel.
- Threads are grouped into **blocks**.
  - Threads in a block can **synchronize** execution.
- Blocks are grouped in a **grid**.
  - Blocks are **independent** (Must be able to be executed in any order.)



Memory hierarchy

- Three **types** of memory in the graphic card:
  - Global memory: 4GB
  - Shared memory: 16 KB
  - Registers: 16 KB
- **Latency**:
  - Global memory: 400-600 cycles
  - Shared memory: Fast
  - Register: Fast
- **Purpose**:
  - Global memory: IO for grid
  - Shared memory: thread collaboration
  - Registers: thread space



Basic C extensions

Function modifiers

- __global__ : to be called by the host but executed by the GPU.
- __host__ : to be called and executed by the host.

Kernel launch parameters

- Block size: (x, y, z). x*y*z = Maximum of 768 threads total. (Hw dependent)
- Grid size: (x, y). Maximum of thousands of threads. (Hw dependent)

Variable modifiers

- __shared__ : variable in shared memory.
- __syncthreads() : sync of threads within a block.

Check CUDA programming guide for all the features!

Example:device

- Simple example: add two arrays
- Not strange code: It is C with extensions.

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

- Example from CUDA programming guide

Example:device

- Simple example: add two arrays
- Not strange code: It is C with extensions.

Thread id

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

- Example from CUDA programming guide

Example: host

```
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Example: host

Memory allocation

```
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Example: host

Memory
copy: Host -> GPU

```
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Example: host

```
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Kernel call

Example: host

```
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Memory copy: GPU -> Host

Example: host

```
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Free GPU memory

Example: host

```
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Work flow

Time

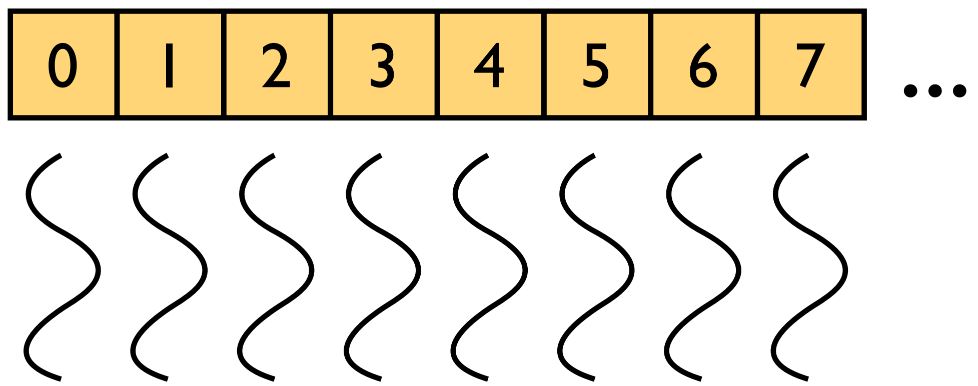
Memory allocation

Memory copy: Host -> GPU

Kernel call

Memory copy: GPU -> Host

Free GPU memory




```
// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```