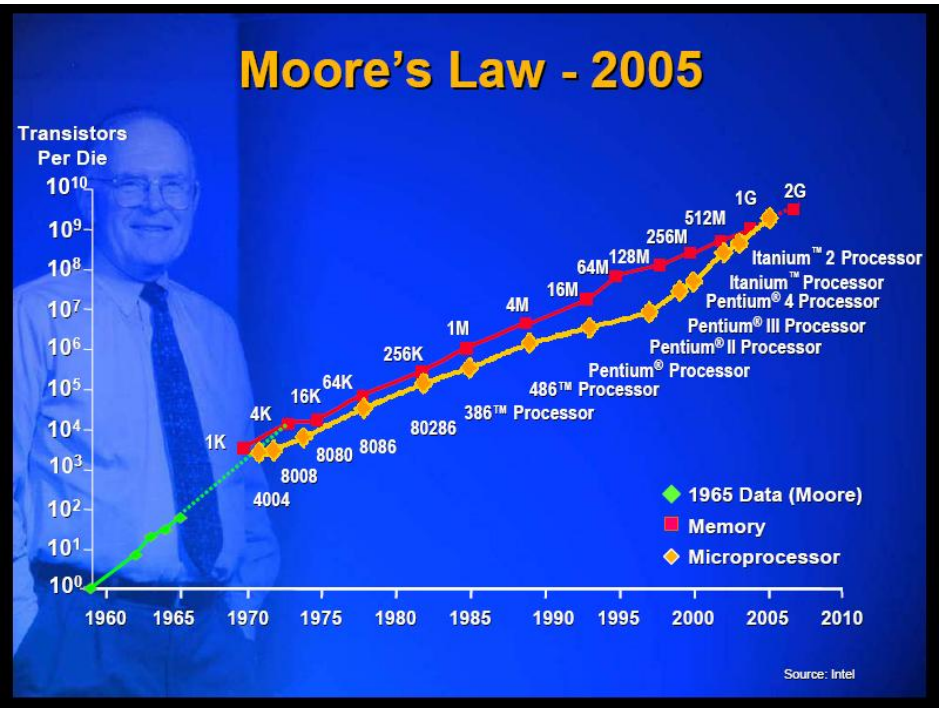
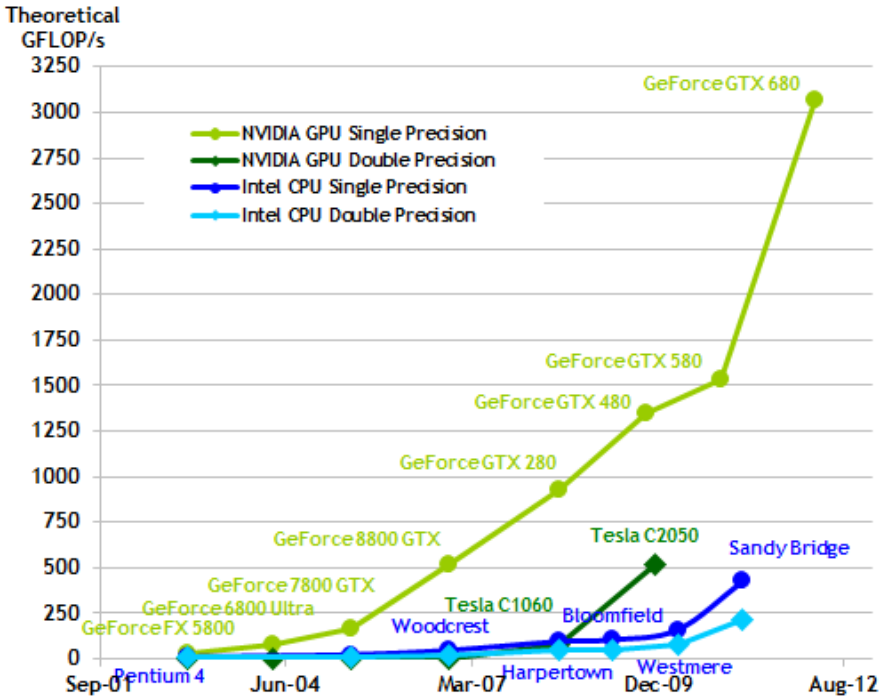
High Performance Computing with Graphics Processing Units (GPUs)

Matthew Beckler
CMU / HackPittsburgh
January 11, 2012

Overview

- Traditional CPUs have run into power limit
- Solution? MOAR CORES!
- Graphics Processing Units: Many, many cores



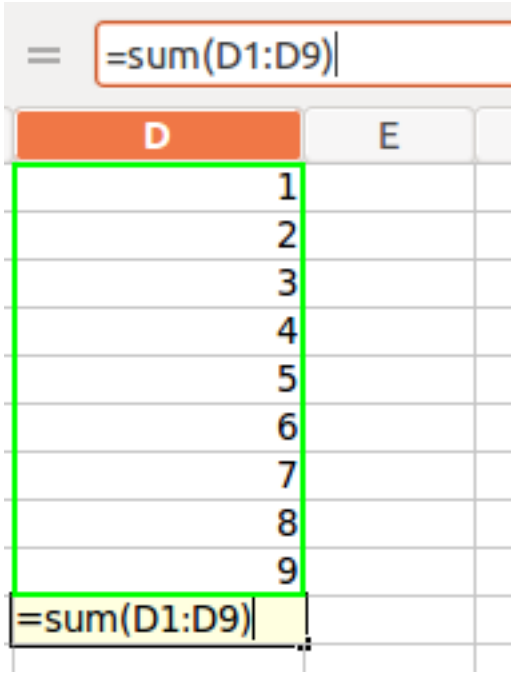


Outline

- ~~Motivation~~
- How computers actually work
- Moore’s Law
- Graphics Processing Units
- Your First CUDA Application!
- Optimization of CUDA applications
- Questions?

How computers work

- Naturally, this is a bit complex
- Execute a sequence of simple “instructions”



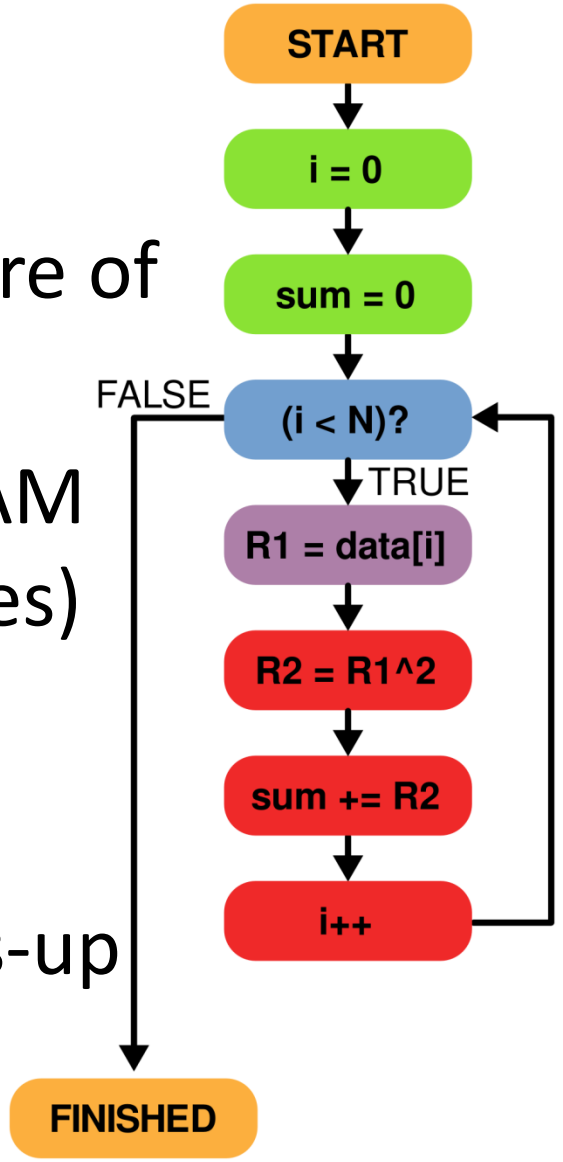
X <- 0
X = X + 1
X = X + 2
X = X + 3
…
X = X + 9

What are “instructions”?

- Data / Memory instructions:
  - Move data between the RAM and CPU (registers)
  - Read and write data from hardware devices
- Arithmetic and logic
  - Add, subtract, multiply, divide, etc
  - Bitwise operations like AND, OR, XOR, etc
  - Compare two values (R1 <= R3?)
- Control flow
  - Branch instructions “IF R1 < R3 THEN GOTO LINE4”

Different types of processors

- This is a “Control Flow Graph”
- Shows how the code adds up square of numbers in the `data[]` array
- The `R1 = data[i]` step accesses RAM which is very, very slow (100+ cycles)
- **Simple CPU**: Just wait it out
- **Smarter CPU**: Pre-load the data to reduce the waiting time, speeds-up the execution of a single “thread”

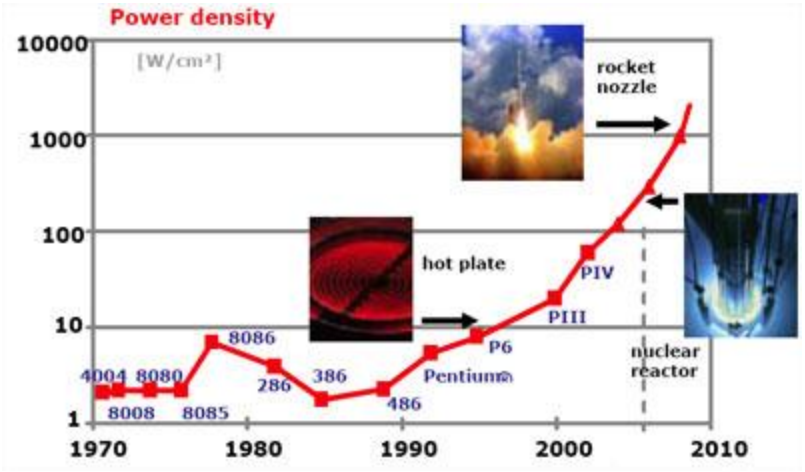


Moore's Law

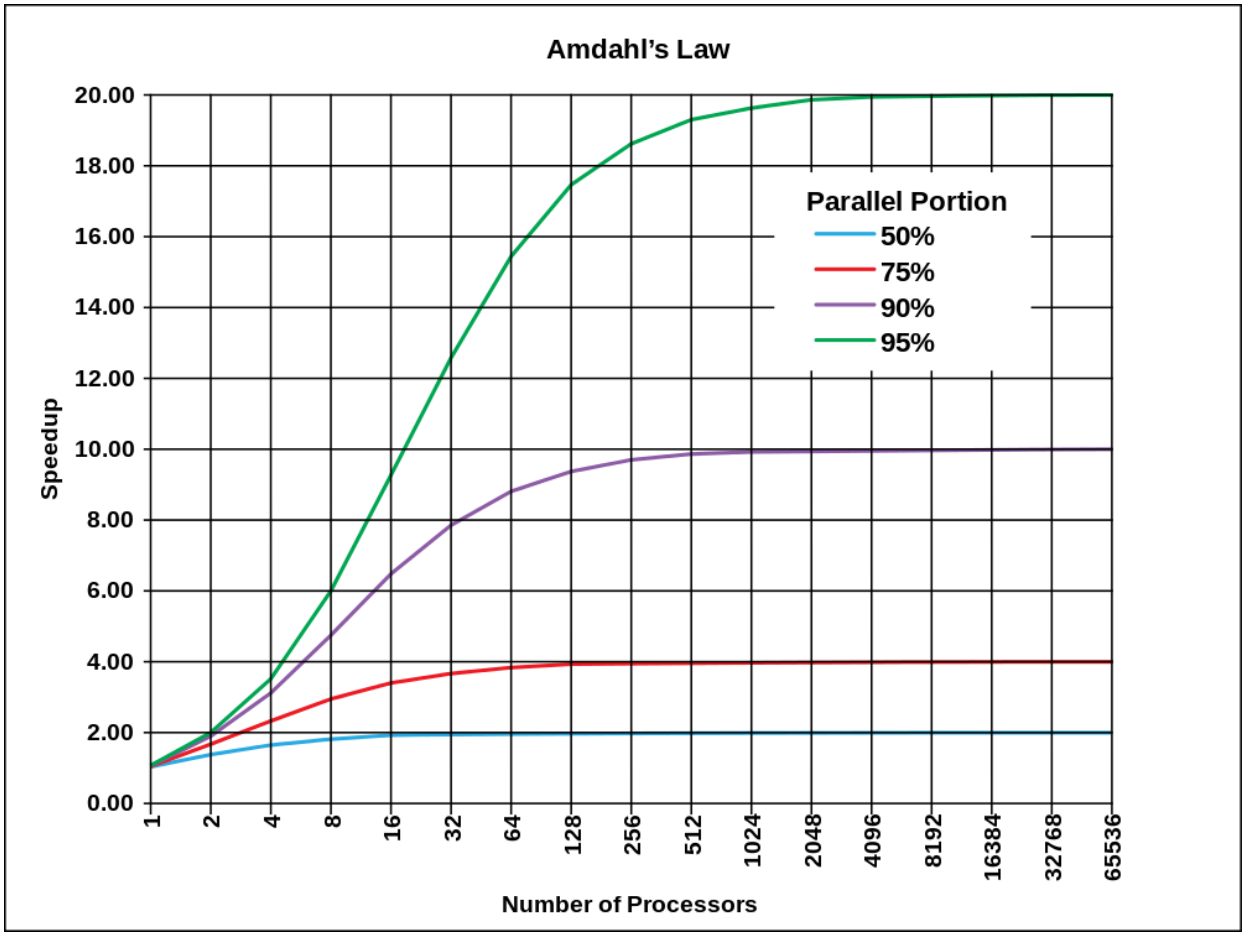
- Since 1965, transistors/chip doubled every 18 mo.

- Slowing down now, due to number of factors
  - Difficulties shrinking down chip designs
  - Power density [Watts/cm^2] approaching nuclear reactor!

- **Solution: Use extra transistors to make more computing cores**



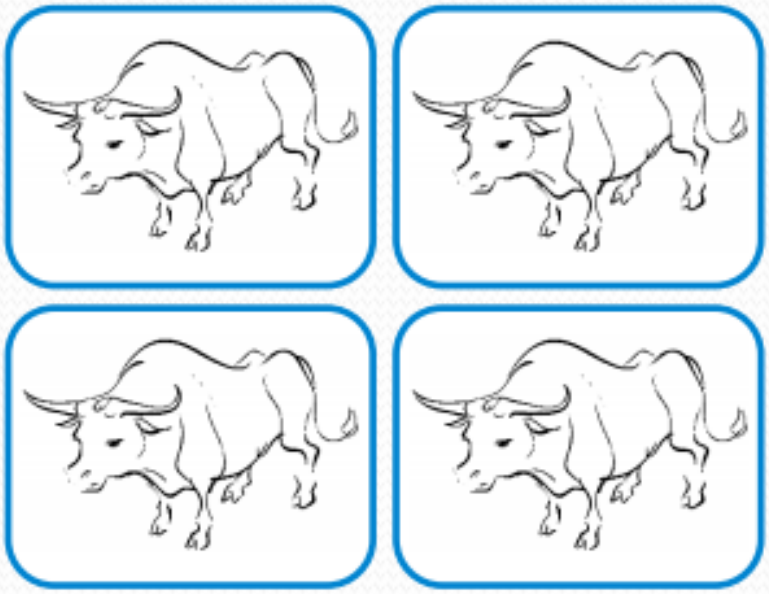
- **Amdahl's Law (1967):** Maximum theoretical speedup using multiple processors



- Ex: 95% parallelizable -> 20x speedup, max

Multicore vs **Many**core

- Multicore: Yoke of oxen (each very powerful)
- Manycore: Flock of chickens (tiny and plentiful)



Multicore

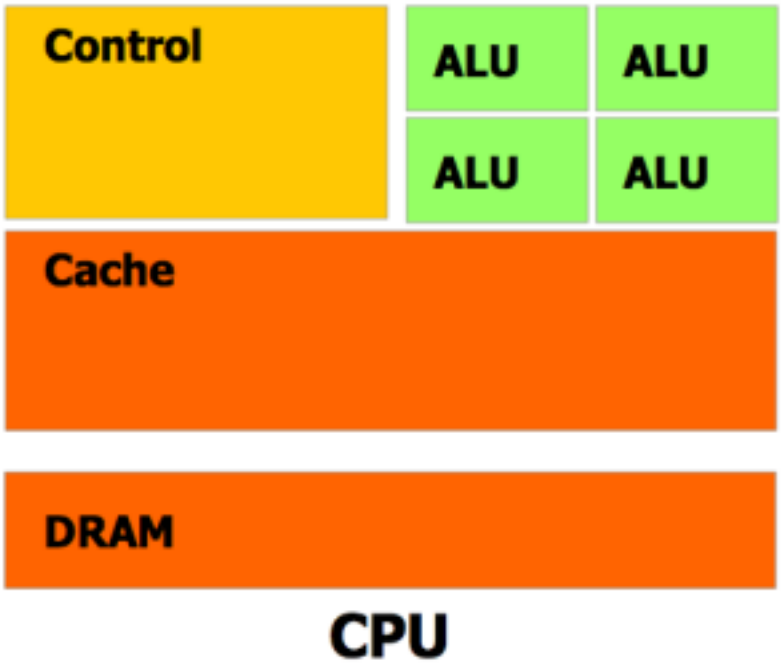
Manycore

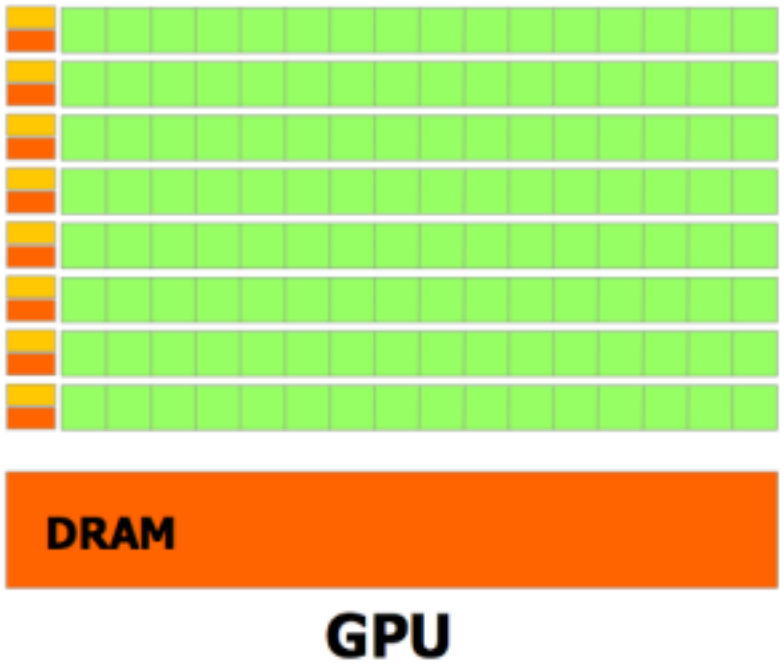
NVIDIA GPUs

- Quadro FX 5800
  - 4 GB memory (DRAM)
  - 1.30 GHz
  - 240 CUDA “cores”
  - Over 30,000 resident threads
  - $3500 in 2008

Differences between CPU and GPU

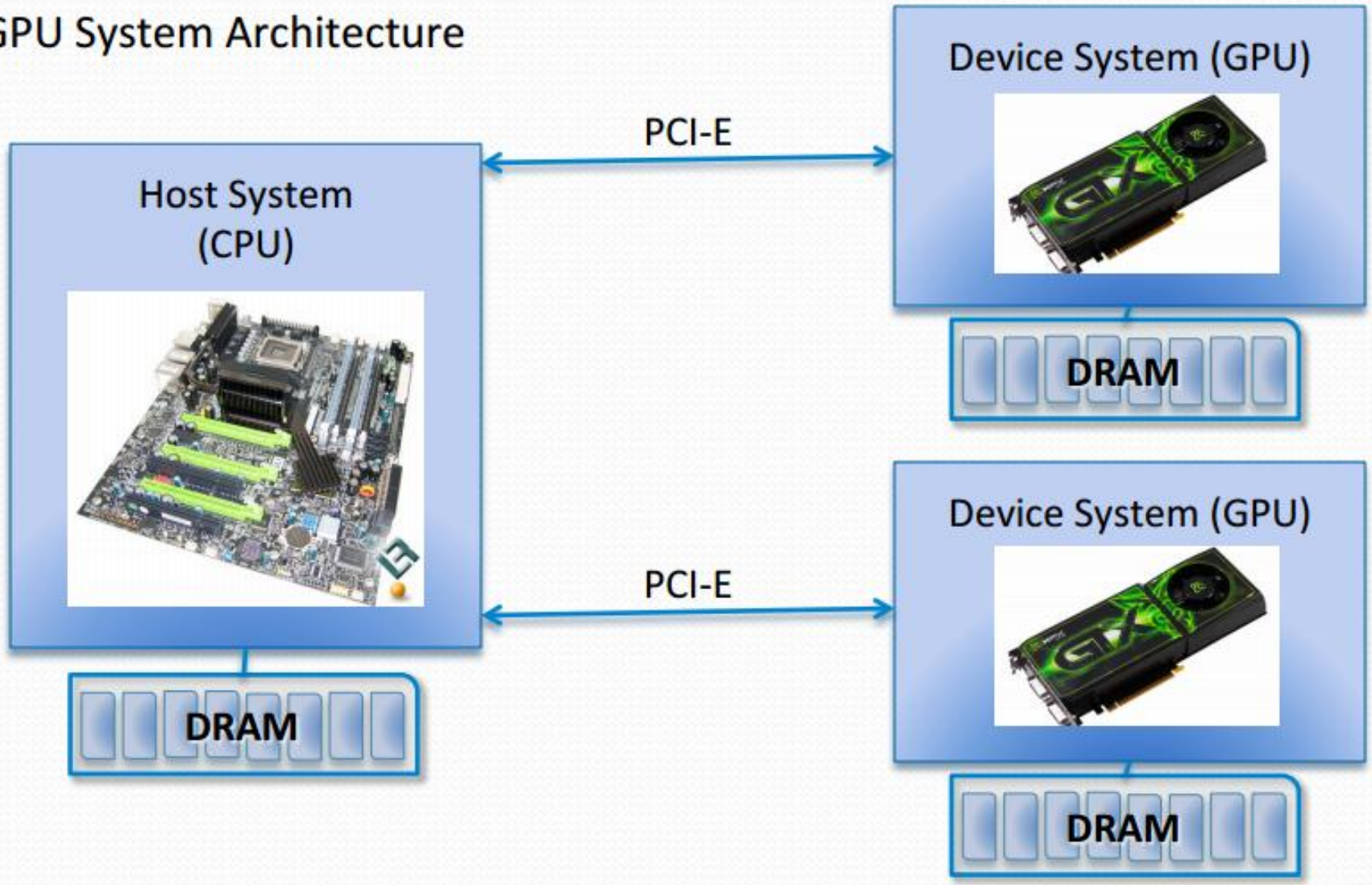
- **Multicore CPUs:** Fast execution of a few threads
  - Very sophisticated cores, with large memory caches
- **Manycore GPUs:** Exploit parallelism in problem
  - Hundreds of simple cores, aggregate throughput





GPU is an Accelerator

- GPU System Architecture



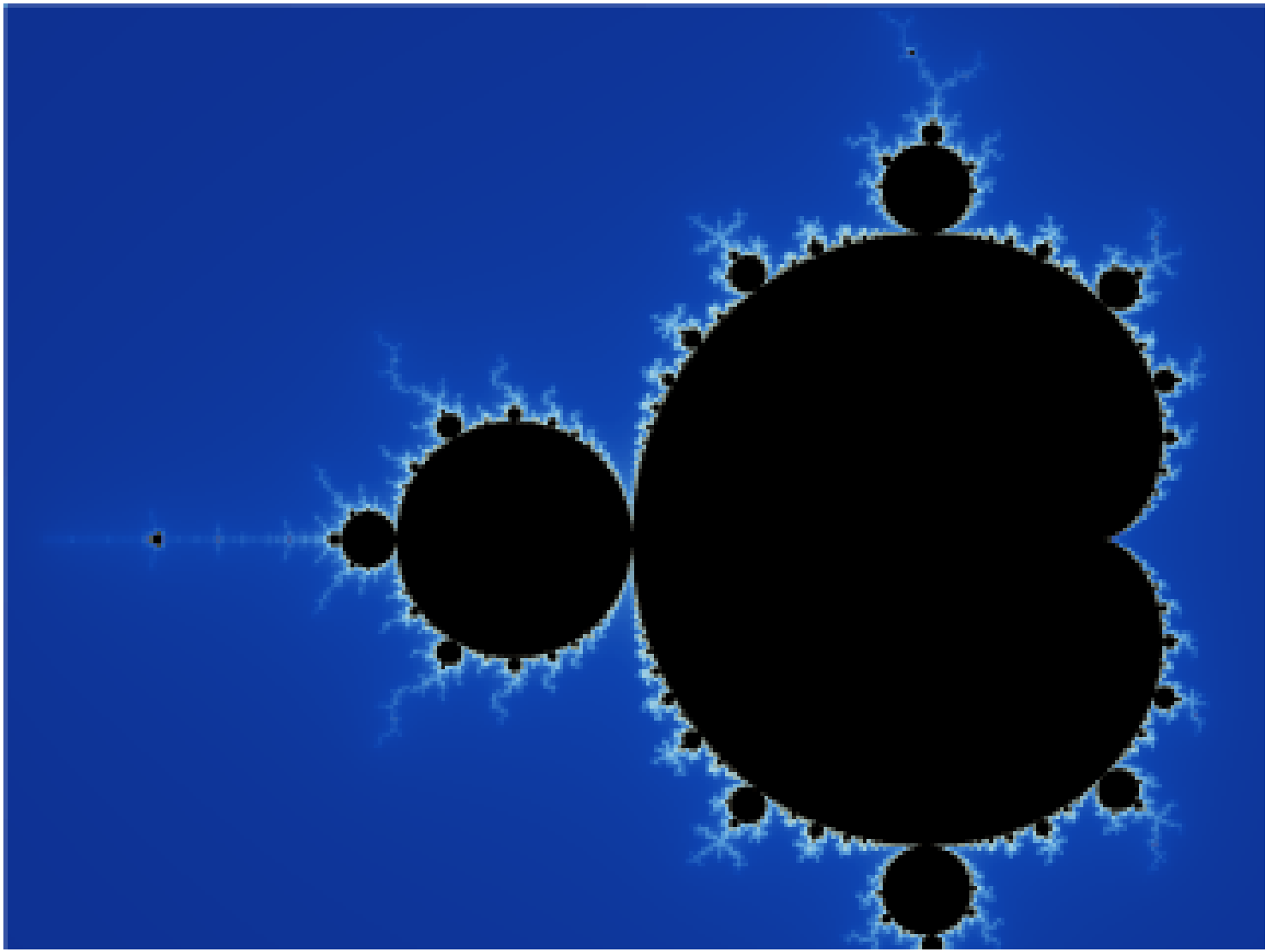
GPU is not always best choice

- Need an application with lots of ***concurrency***
  - Many small, independent computations
  - For example, each pixel in Mandelbrot Set fractal
  - "Embarrassingly parallel" problems are great!

- Need "data parallelism"
  - Simple threads operate on different parts of data
  - Regular memory accesses (not random or "jumpy")

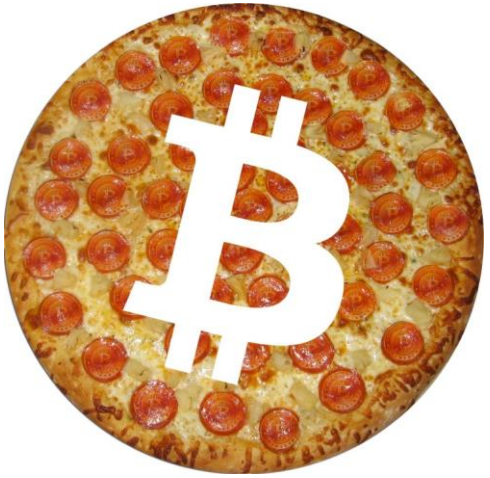
Parallel problems: Mandelbrot Set

$Z_{n+1} = Z_n^2 + c$ ($Z_0 = 0$, for all pixels)

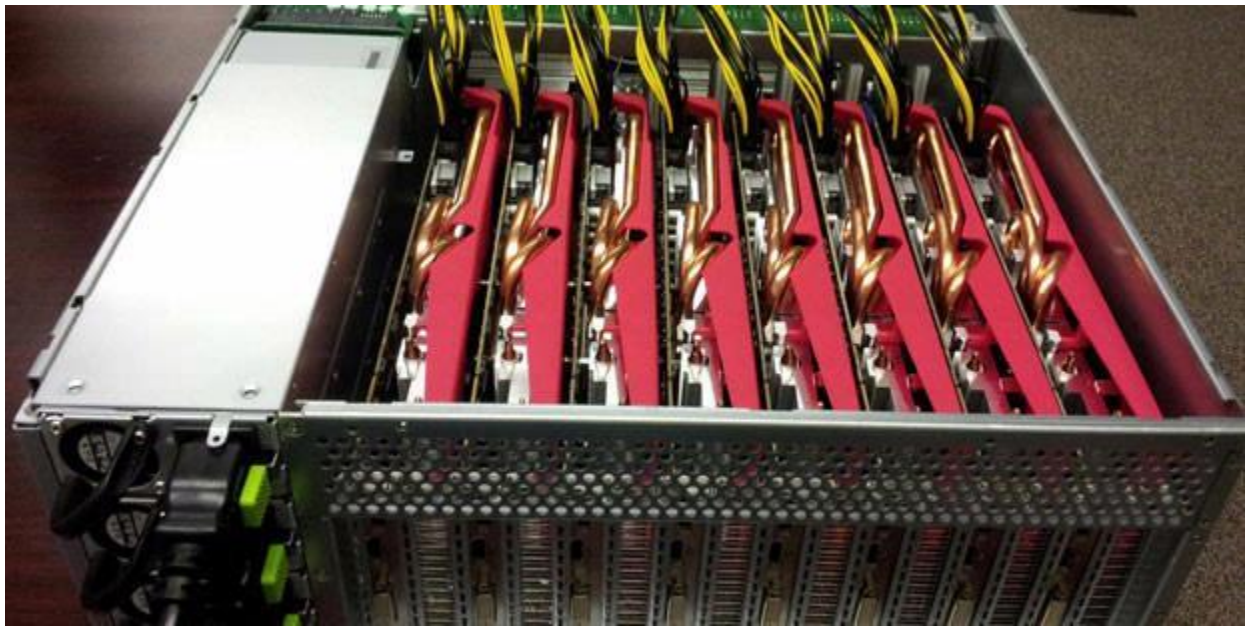


Parallel problems: Raytracing

Parallel problems: Hashing

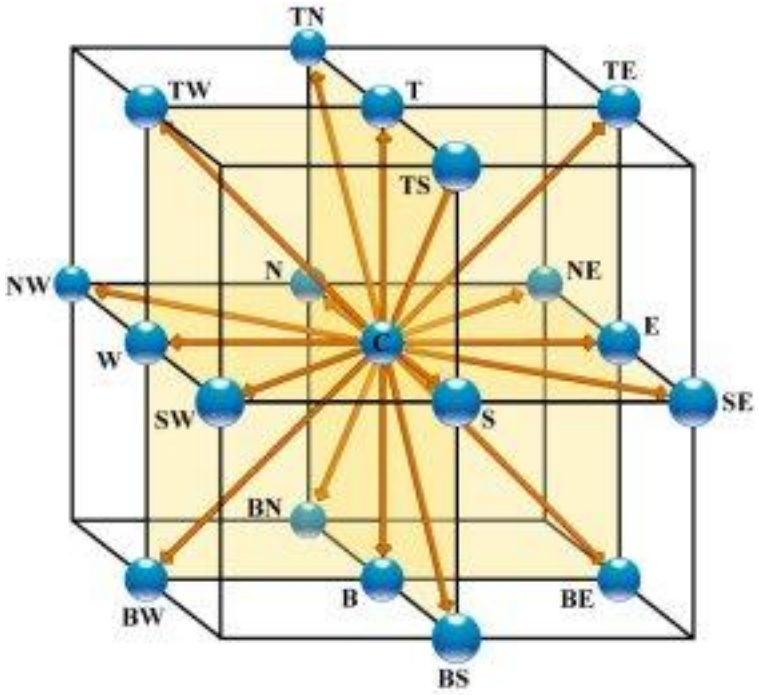


- Bitcoin mining!
- Password file breaking
  - 25 GPU machine!
  - 5.5 hr to break 8-char NTLM password



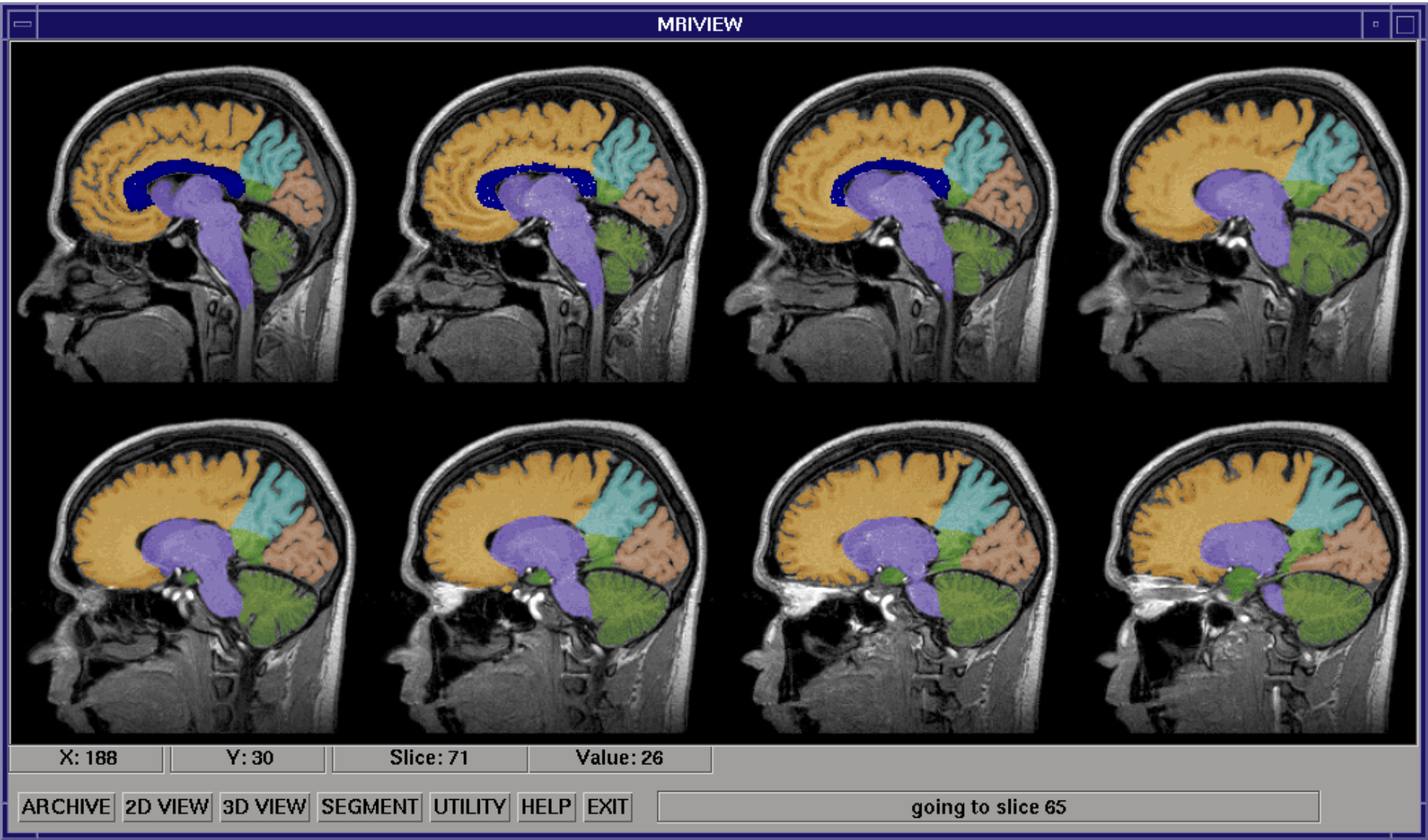
Parallel problems: Physics Simulations

- i.e.: Navier-Stokes, Lattice Boltzmann, FEA, etc
- Simulate each particle independently

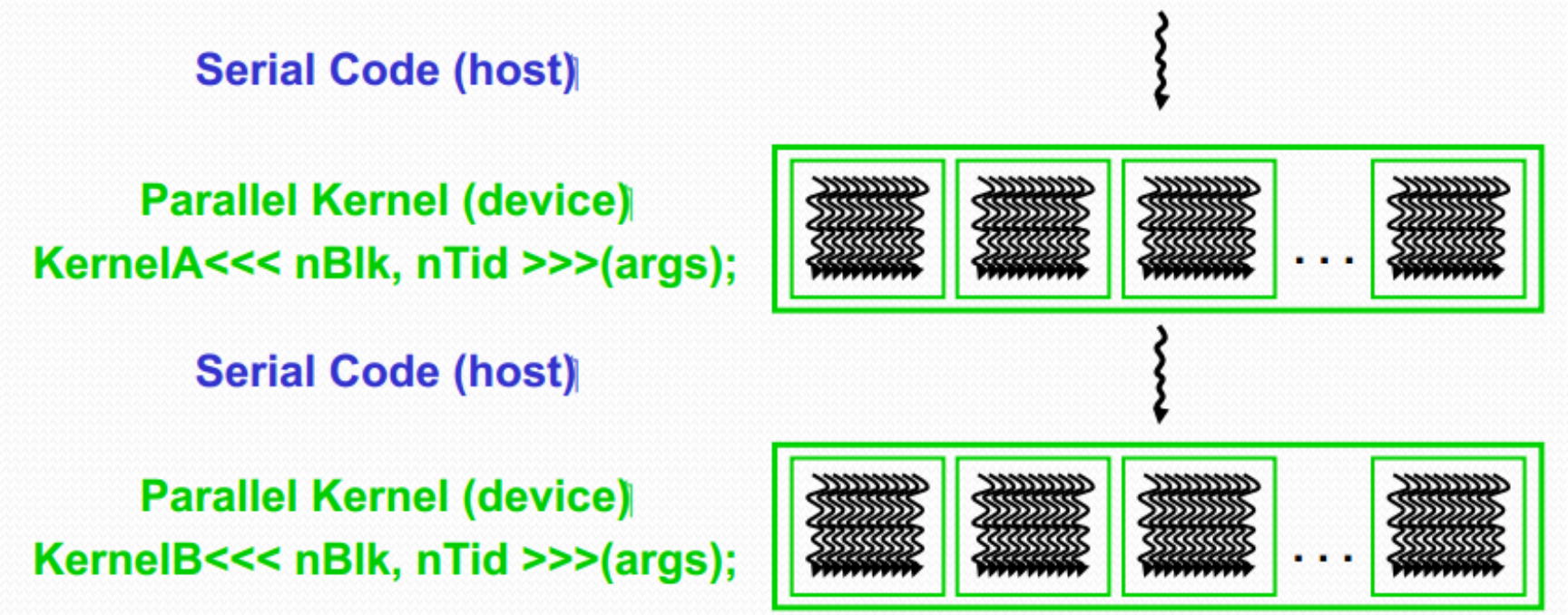


Parallel problems: MRI Data

- Gigabytes of 4-dimensional data, full of noise

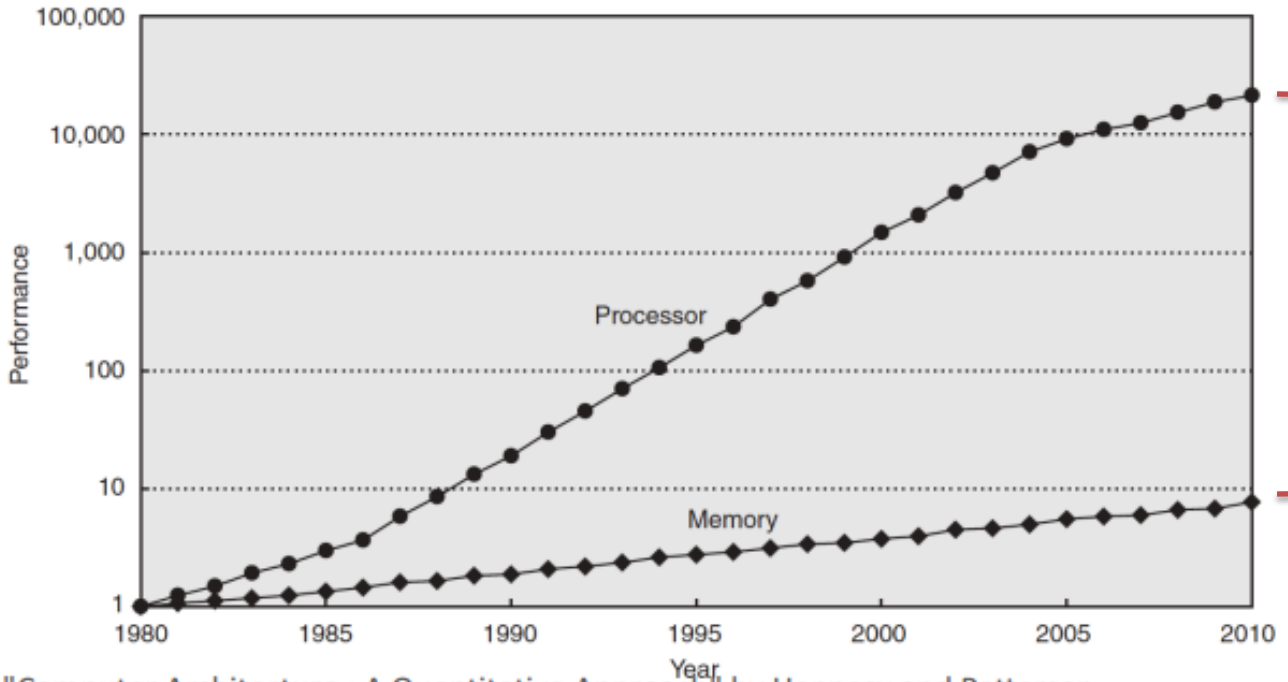


CUDA: Compute Unified Device Architecture



Kirk, Hwu, Programming Massively Parallel Processors, Figure 3.2

Why organize threads in blocks?

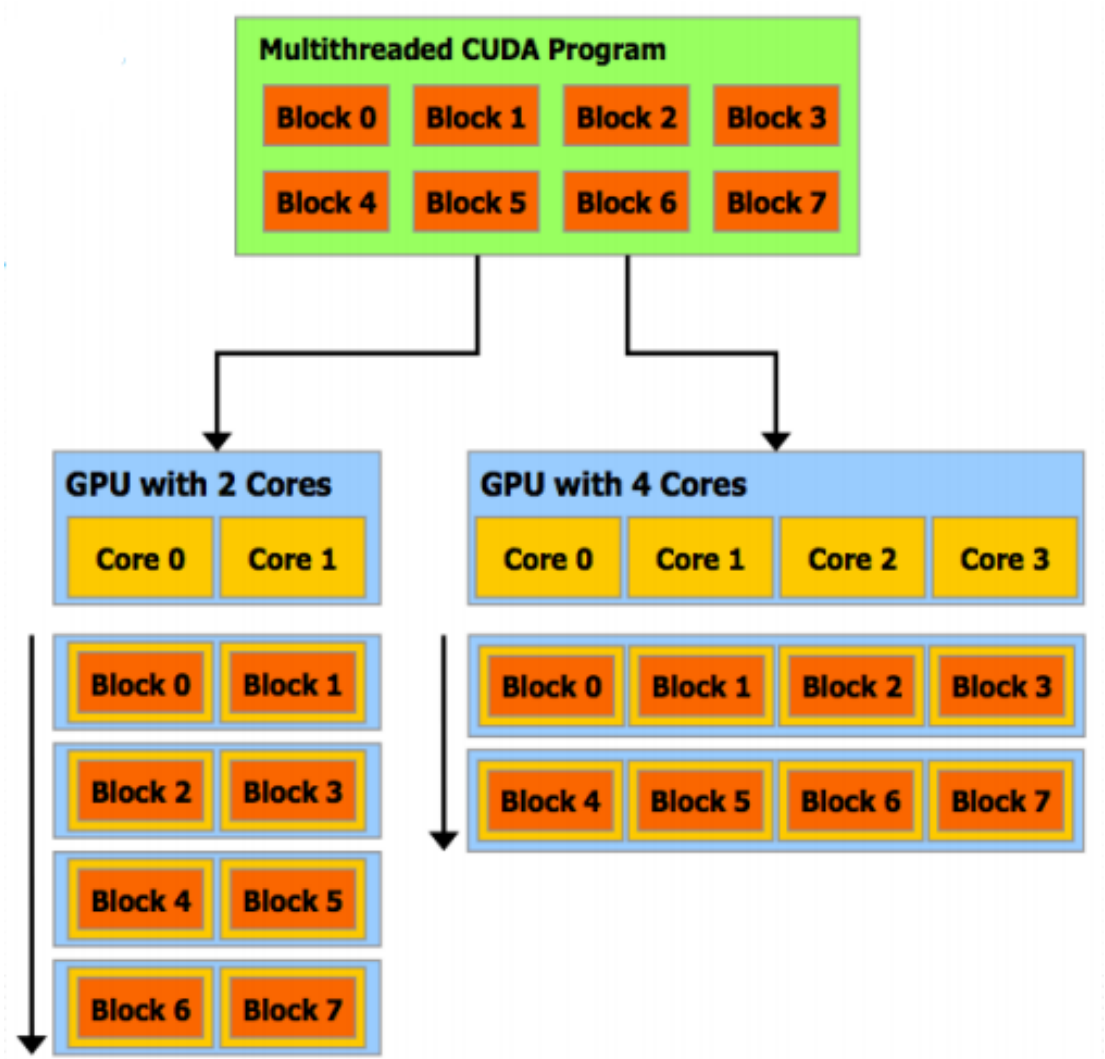


"Computer Architecture : A Quantitative Approach" by Hennesy and Patterson.

Figure 5.2 Starting with 1980 performance as a baseline, the gap in performance between memory and processors is plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 5.13 on page 313). The processor line assumes a 1.25 improvement per year until 1986, and a 1.52 improvement until 2004, and a 1.20 improvement thereafter; see Figure 1.1 in Chapter 1.

Huge gap between processor and memory performance!

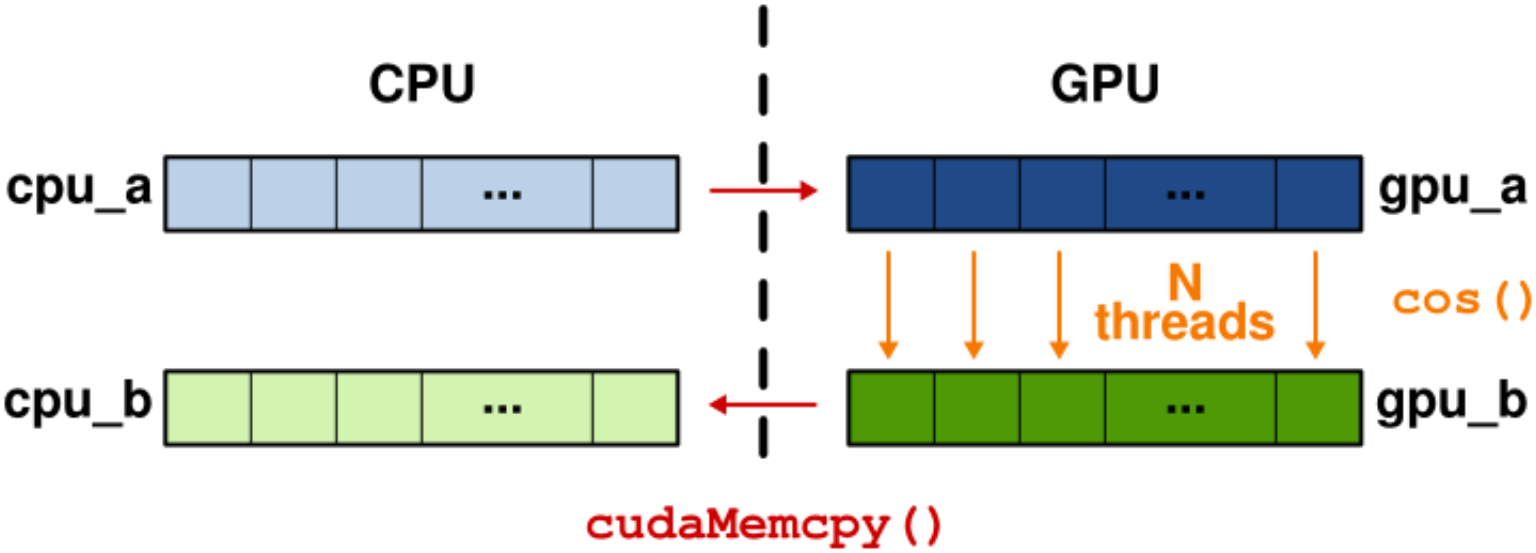
GPU-independent Design



NVIDIA CUDA C Programming Guide Version 4.0, Chapter 1.3

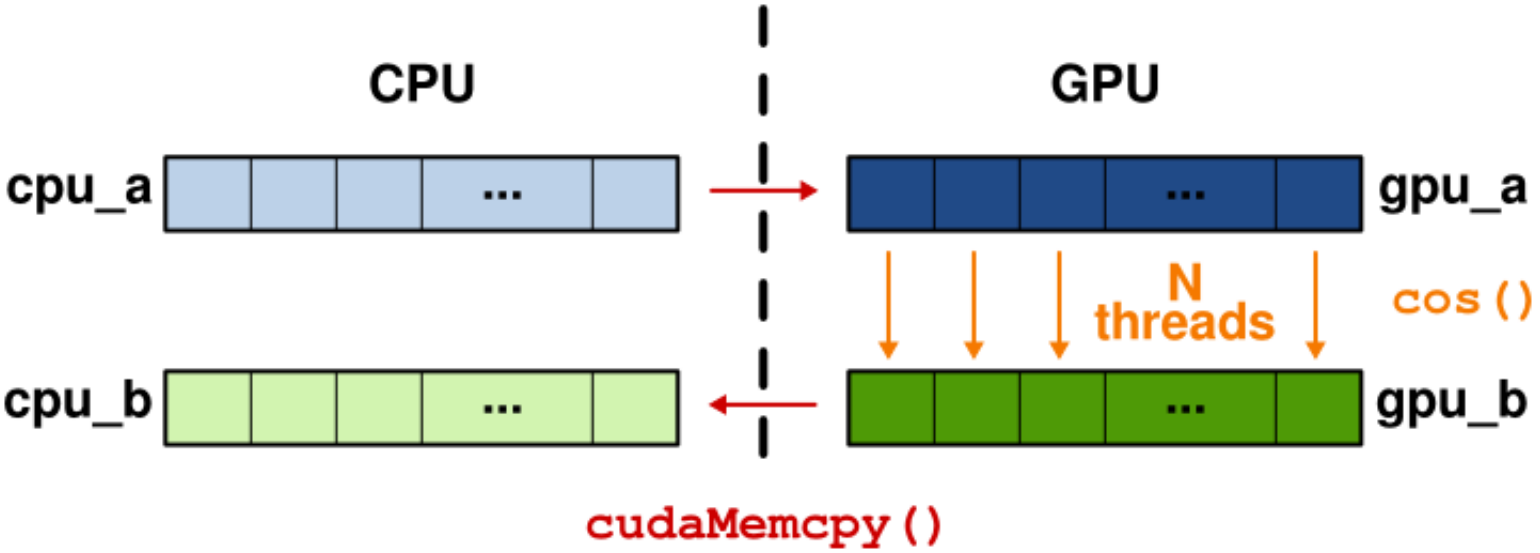
Your First CUDA Program!

- Compute *cos(x)* for all values in the array A

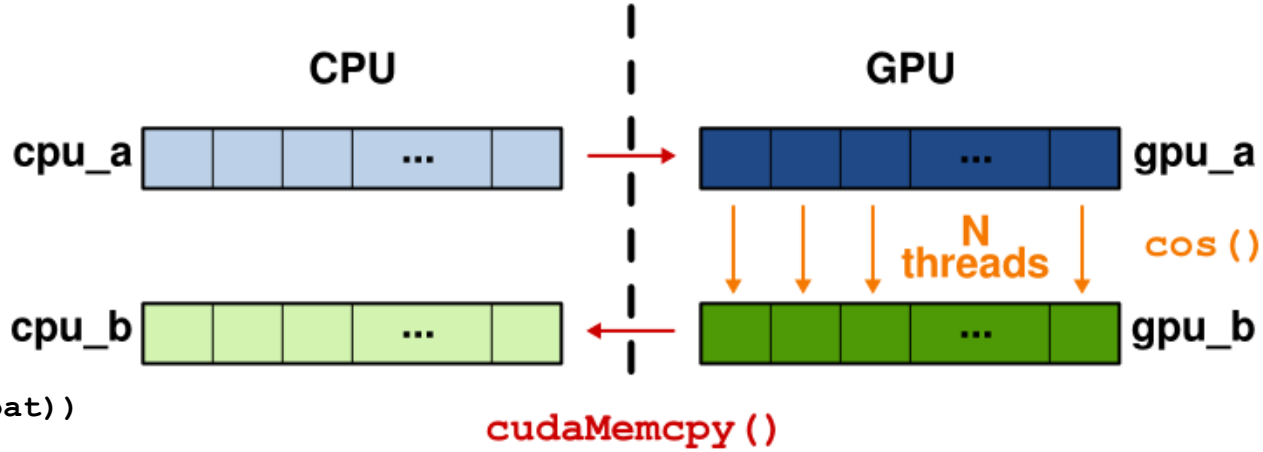


Your First CUDA Program!

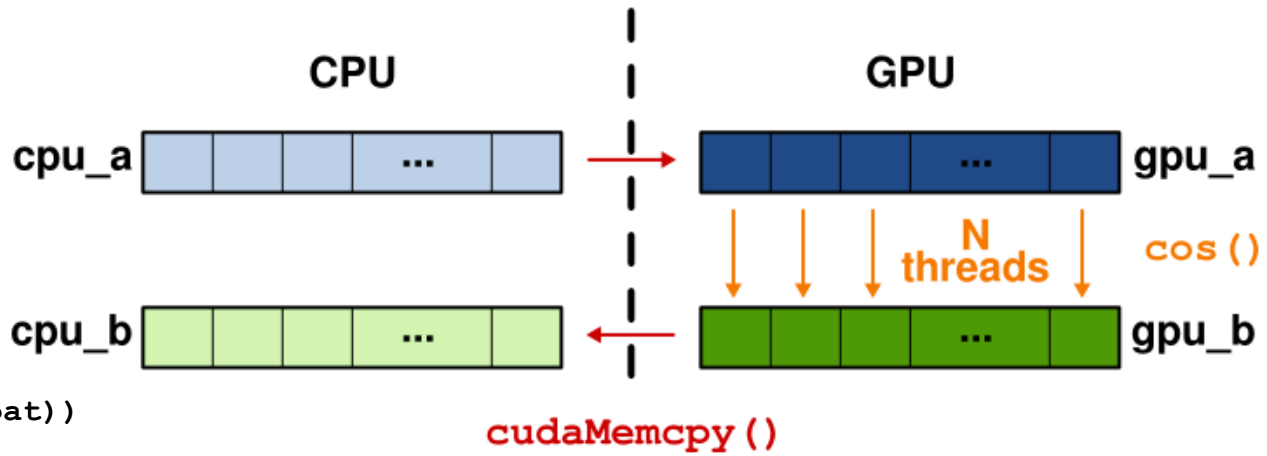
- Compute *cos(x)* for all values in the array A



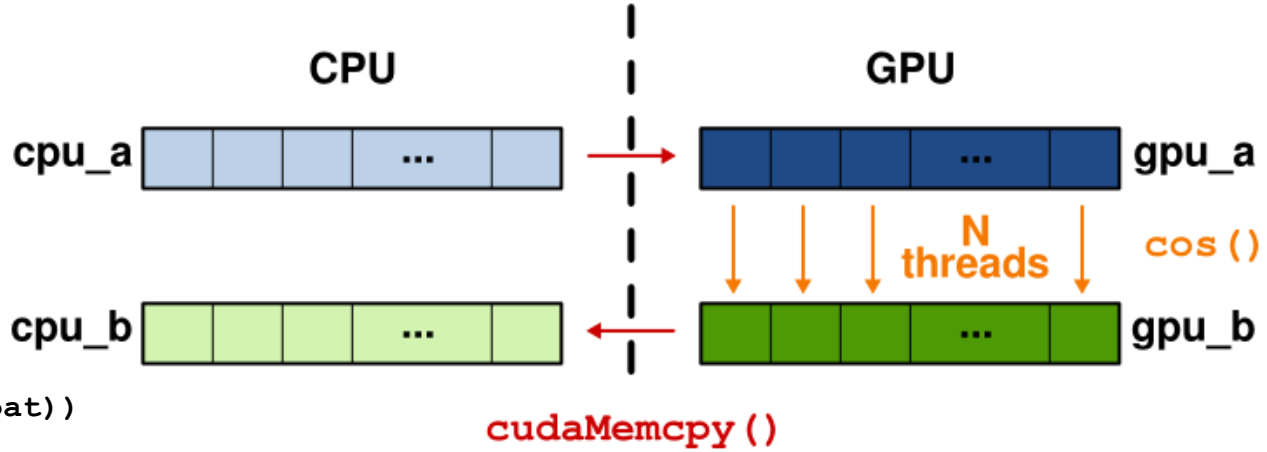
1. Reserve memory on CPU and GPU for original A and result B
2. Load data into A from file
3. Transfer A from CPU to GPU
4. Execute GPU "kernel" code to do the actual work (cos(A) -> B)
5. Transfer result B from GPU to CPU
6. Save result to file



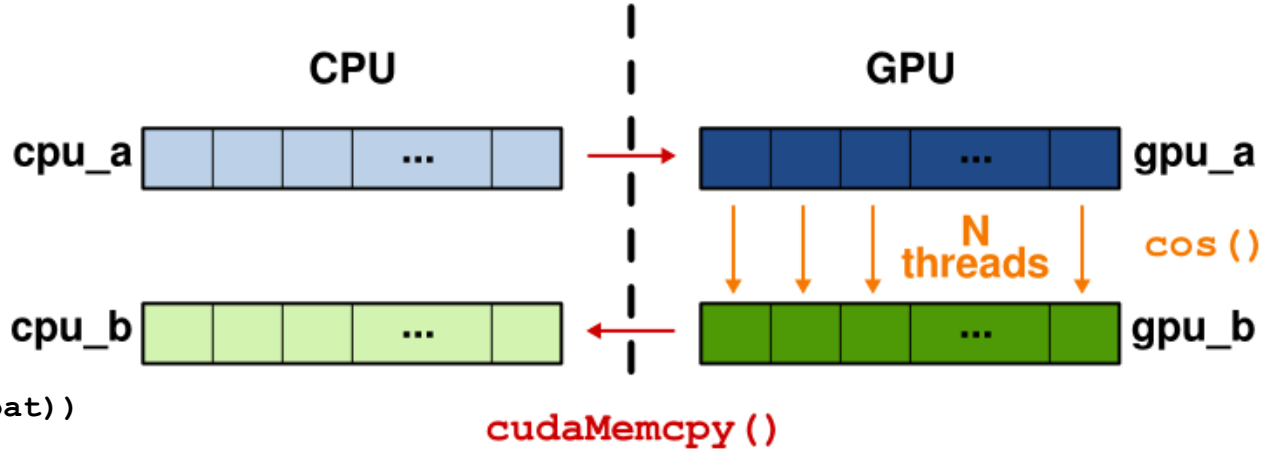

```
#define N 1024
#define SIZE (N * sizeof(float))
int main() {
    float *cpu_a, *cpu_b; float *gpu_a, *gpu_b;

}
```



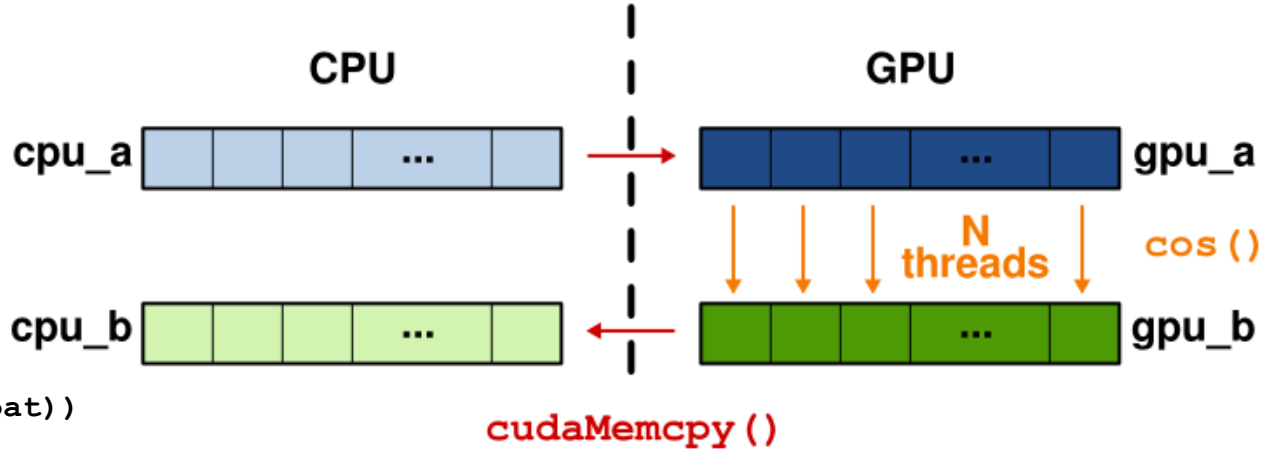

```
#define N 1024
#define SIZE (N * sizeof(float))
int main() {
    float *cpu_a, *cpu_b; float *gpu_a, *gpu_b;

    cpu_a = (float*) malloc(SIZE);
    cpu_b = (float*) malloc(SIZE);
    cudaMalloc(&gpu_a, SIZE);
    cudaMalloc(&gpu_b, SIZE);
    // TODO: assign values to cpu_a (from data file on disk)









}
```



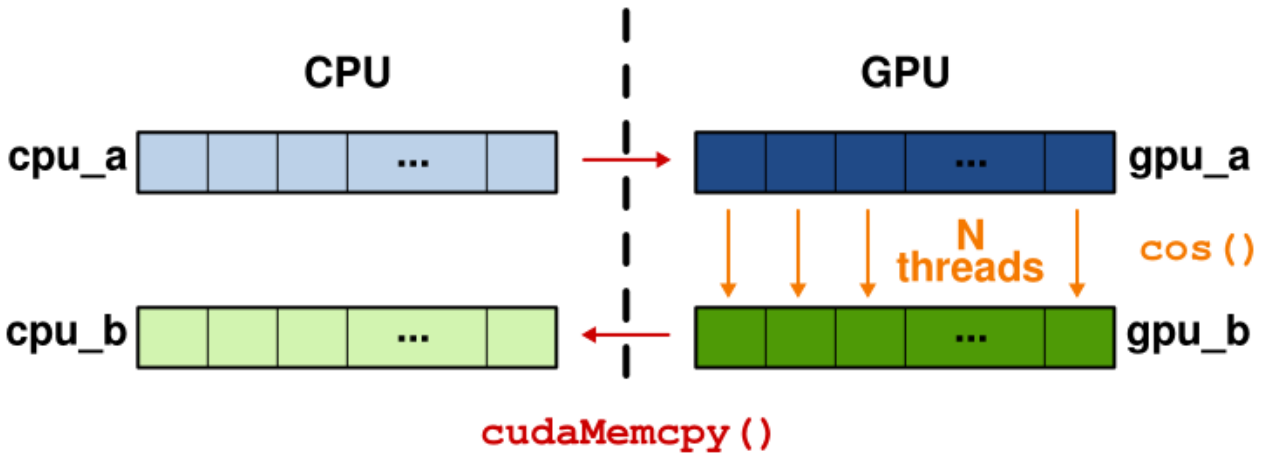

```
#define N 1024
#define SIZE (N * sizeof(float))
int main() {
    float *cpu_a, *cpu_b; float *gpu_a, *gpu_b;

    cpu_a = (float*) malloc(SIZE);
    cpu_b = (float*) malloc(SIZE);
    cudaMalloc(&gpu_a, SIZE);
    cudaMalloc(&gpu_b, SIZE);
    // TODO: assign values to cpu_a (from data file on disk)

    // transfer source data from CPU to GPU:
    cudaMemcpy(gpu_a, cpu_a, SIZE, cudaMemcpyHostToDevice);









}
```




```
#define N 1024
#define SIZE (N * sizeof(float))
int main() {
    float *cpu_a, *cpu_b; float *gpu_a, *gpu_b;

    cpu_a = (float*) malloc(SIZE);
    cpu_b = (float*) malloc(SIZE);
    cudaMalloc(&gpu_a, SIZE);
    cudaMalloc(&gpu_b, SIZE);
    // TODO: assign values to cpu_a (from data file on disk)

    // transfer source data from CPU to GPU:
    cudaMemcpy(gpu_a, cpu_a, SIZE, cudaMemcpyHostToDevice);

    // launch 1 thread per element on the GPU (256 threads per block):
    int num_blocks = N / 256;
    gpu_cosine_kernel<<<num_blocks, 256>>>( gpu_a, gpu_b );




}
```



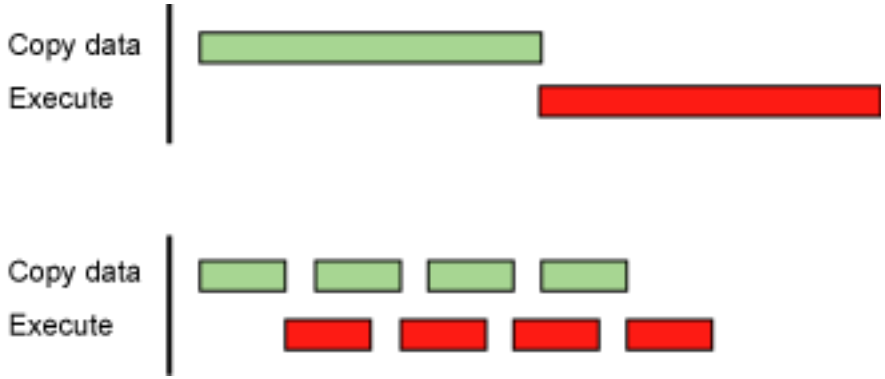

```
#define N 1024
#define SIZE (N * sizeof(float))
int main() {
    float *cpu_a, *cpu_b; float *gpu_a, *gpu_b;

    cpu_a = (float*) malloc(SIZE);
    cpu_b = (float*) malloc(SIZE);
    cudaMalloc(&gpu_a, SIZE);
    cudaMalloc(&gpu_b, SIZE);
    // TODO: assign values to cpu_a (from data file on disk)

    // transfer source data from CPU to GPU:
    cudaMemcpy(gpu_a, cpu_a, SIZE, cudaMemcpyHostToDevice);

    // launch 1 thread per element on the GPU (256 threads per block):
    int num_blocks = N / 256;
    gpu_cosine_kernel<<<num_blocks, 256>>>( gpu_a, gpu_b );

    // transfer result data from GPU to CPU:
    cudaMemcpy(cpu_b, gpu_b, SIZE, cudaMemcpyHostToDevice);
    // TODO: save values from cpu_b (to data file on disk)

}
```

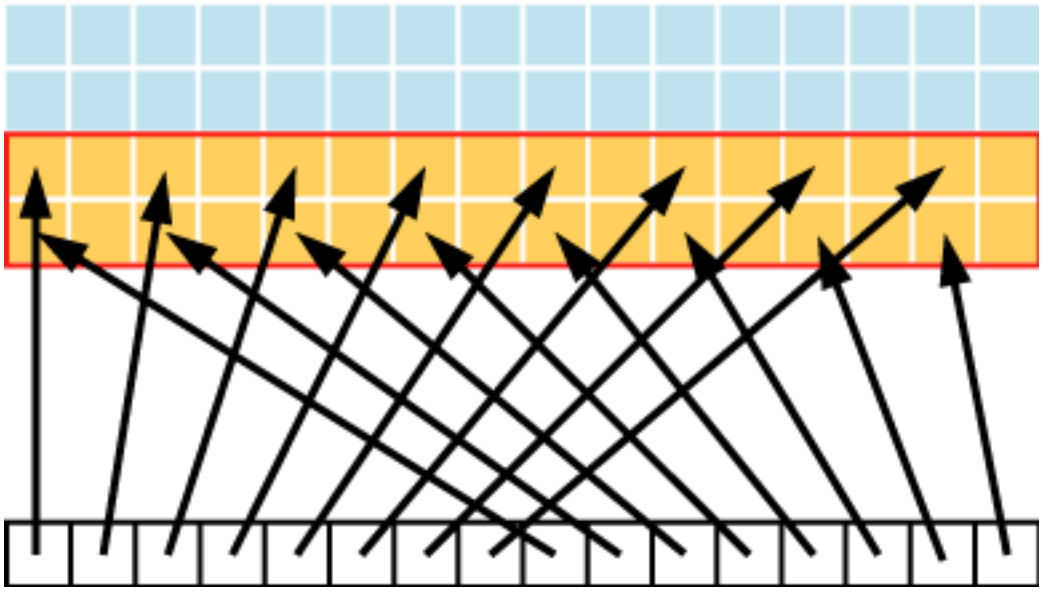


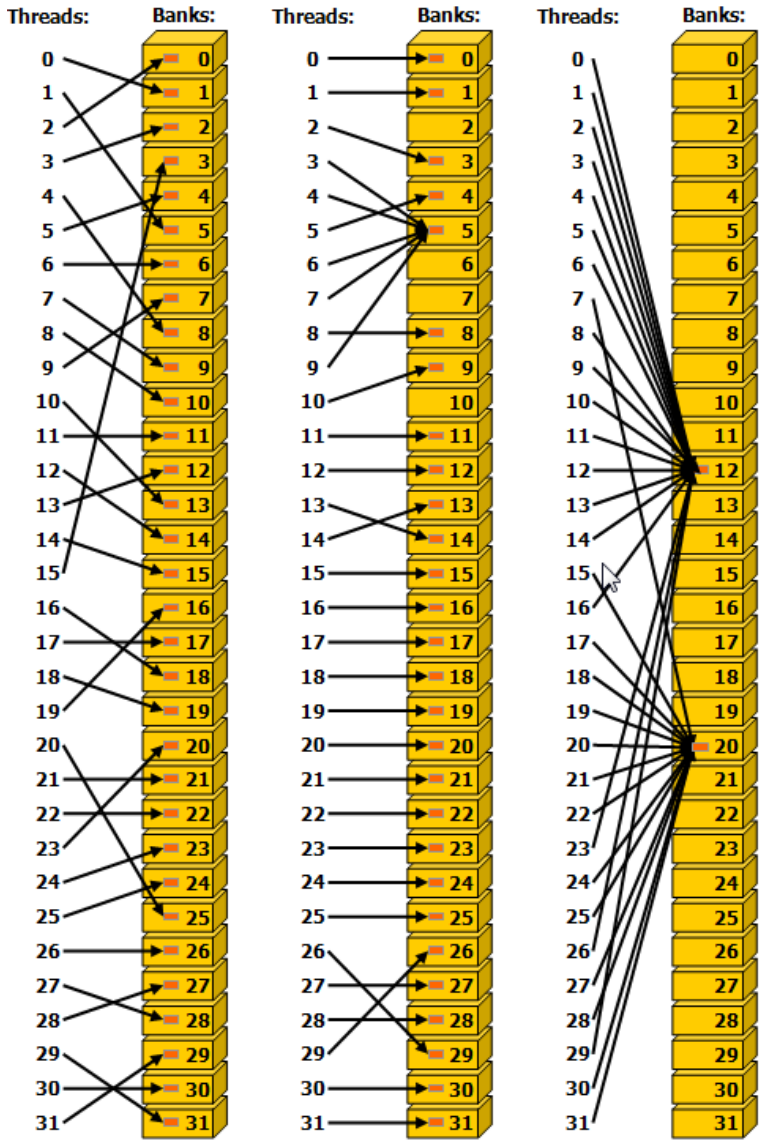
GPU code

```
#define N 1024
#define SIZE (N * sizeof(float))

__global__ void gpu_cosine_kernel(float *a, float *b)
{
    int index = blockIdx.x * 256 + threadIdx.x;
    b[index] = cos( a[index] );
}
```

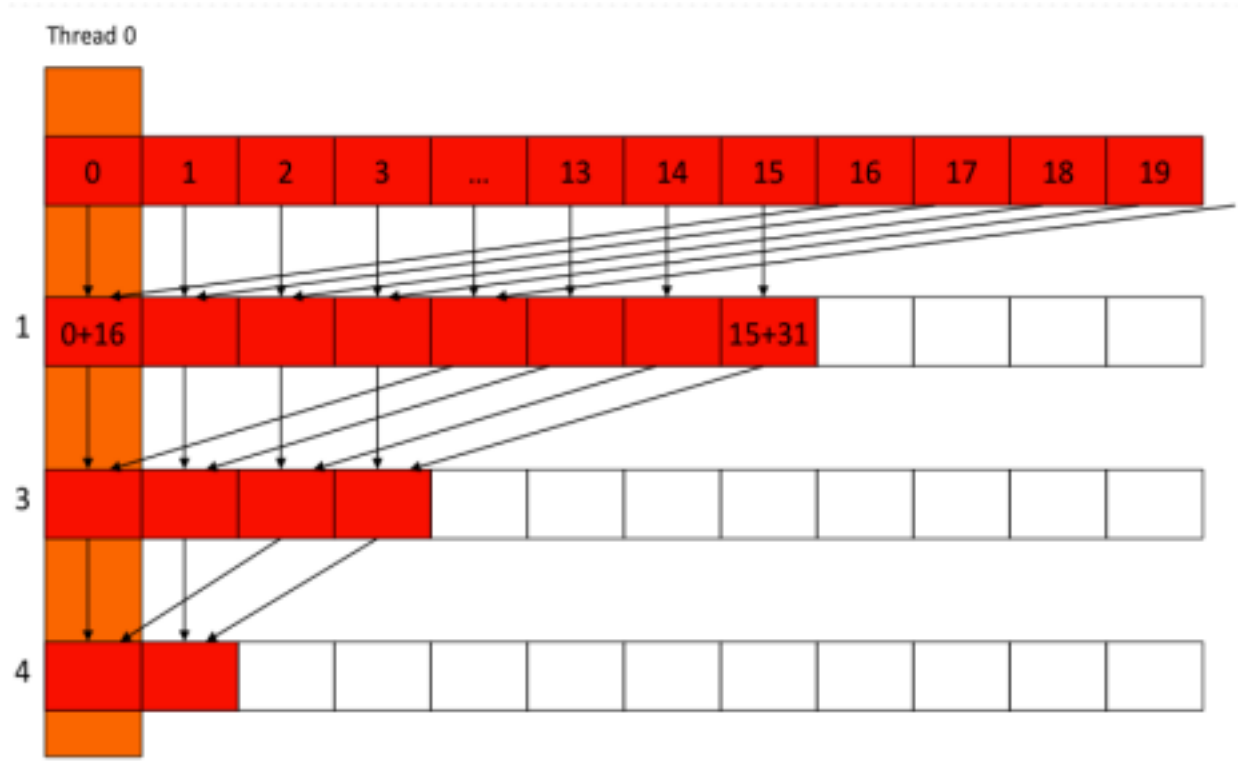
Optimization of CUDA code



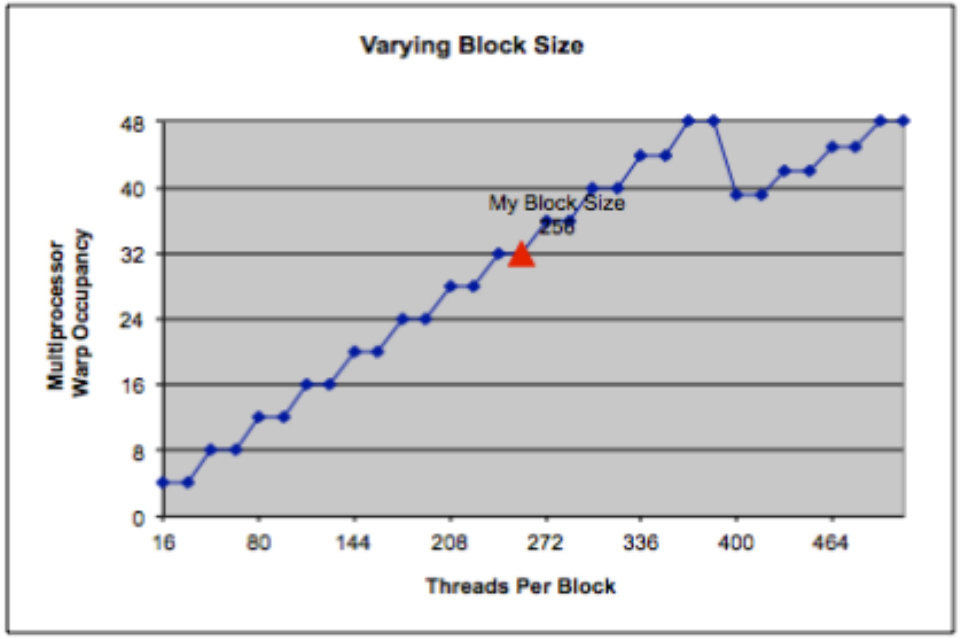


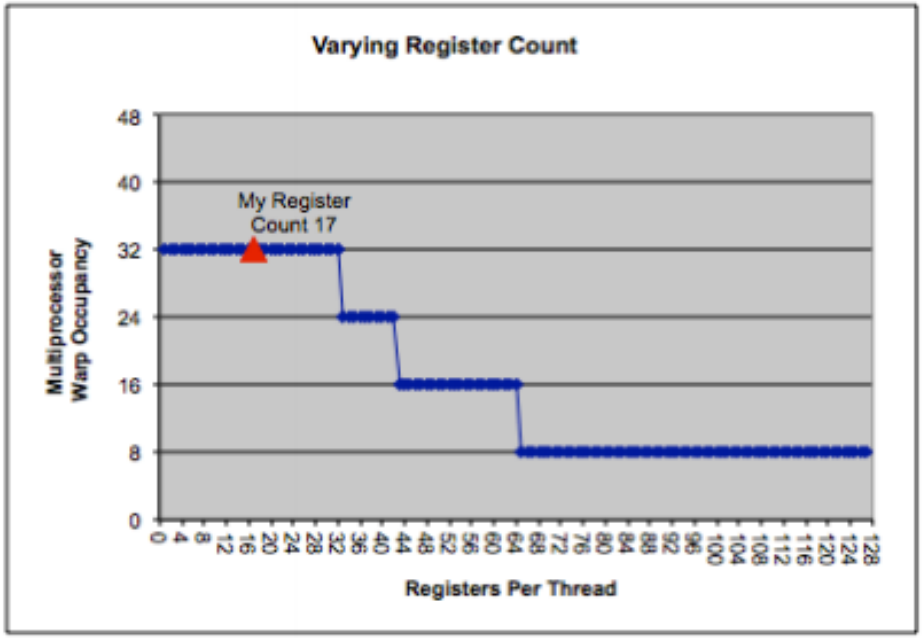


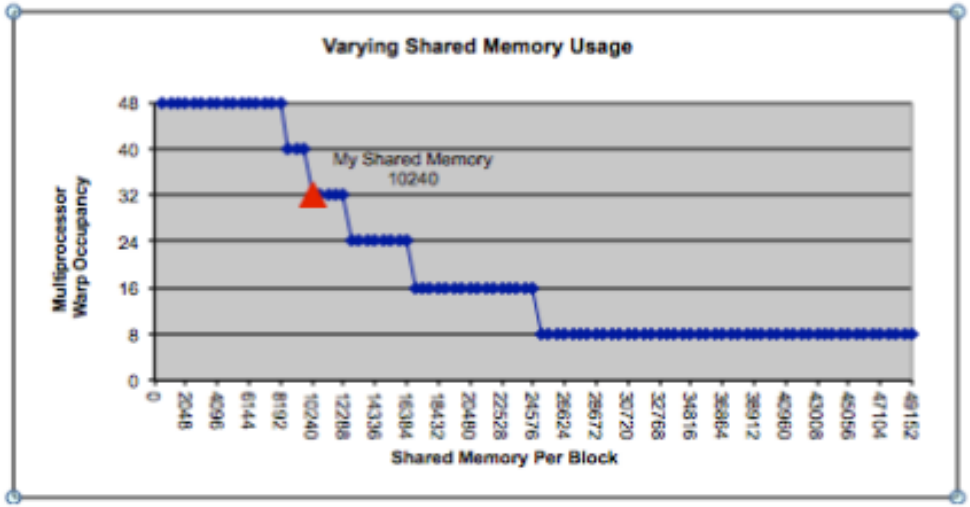
Optimization of CUDA code



Optimization of CUDA code

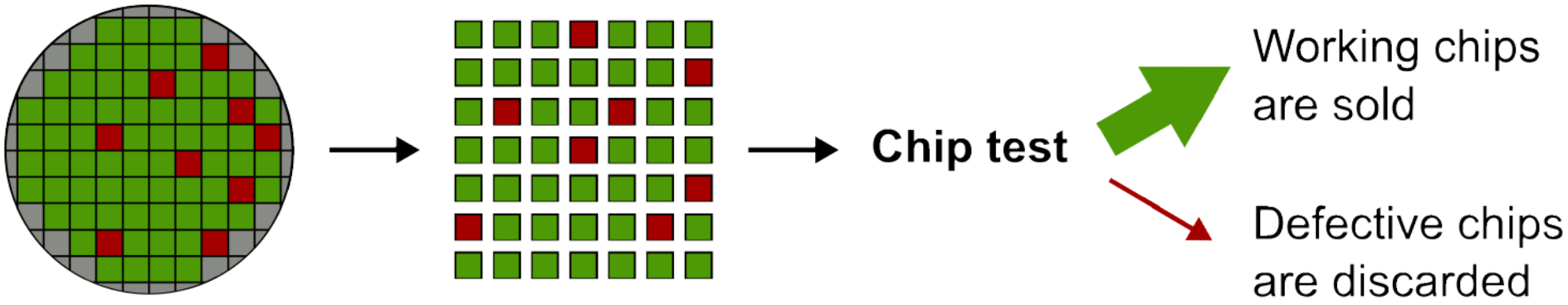






- Kernel 1:
  - 256 threads/block
  - 17 registers/thread
  - 10KB Shared mem/block
- Occupancy: 67%

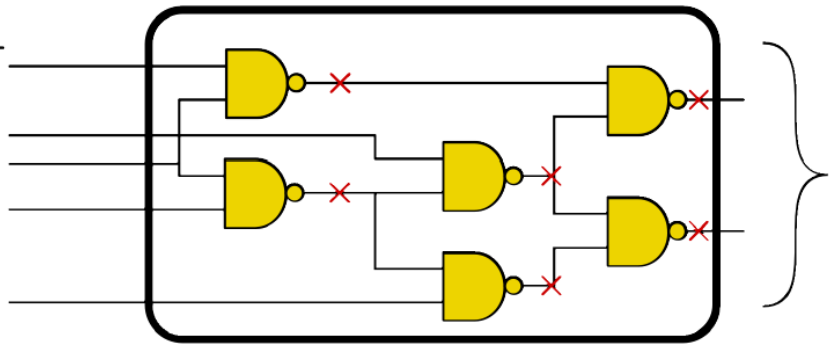
My own research using GPUs



Test set

t0	t1	t2	t3	t4
1	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	1	0	1	1
0	1	0	1	0

Circuit under test



Detected faults

- t0: f0
- t1: f1, f5, f6
- t3: f2, f3
- t4: f0, f1, f2
- t5: f5, f7

- t0 does not uniquely detect any faults, and can be removed

Any questions?