

GPU Computing with CUDA

Lecture 1 - Introduction

Christopher Cooper
Boston University

August, 2011
UTFSM, Valparaíso, Chile

General Ideas

- ▶ Objectives
 - Learn CUDA
 - Recognize CUDA friendly algorithms and practices
- ▶ Requirements
 - C/C++ (hopefully)
- ▶ Lab oriented course!

Resources

- ▶ Material available in <http://www.bu.edu/pasi/materials/post-pasi-training/>
- ▶ Machines to be used
 - Cluster USM
 - BUNGEE/BUDGE at Boston University
- ▶ Main references
 - Kirk, D. and Hwu, W. Programming Massively Parallel Processors
 - Sanders, J. and Kandrot, E. CUDA by Example
 - CUDA C Programming Guide
 - CUDA C Best Practices Guide

Outline of course

- ▶ Week 1
 - Basic CUDA
- ▶ Week 2
 - Optimization
- ▶ Week 3
 - CUDA Libraries
- ▶ Week 4
 - Applications

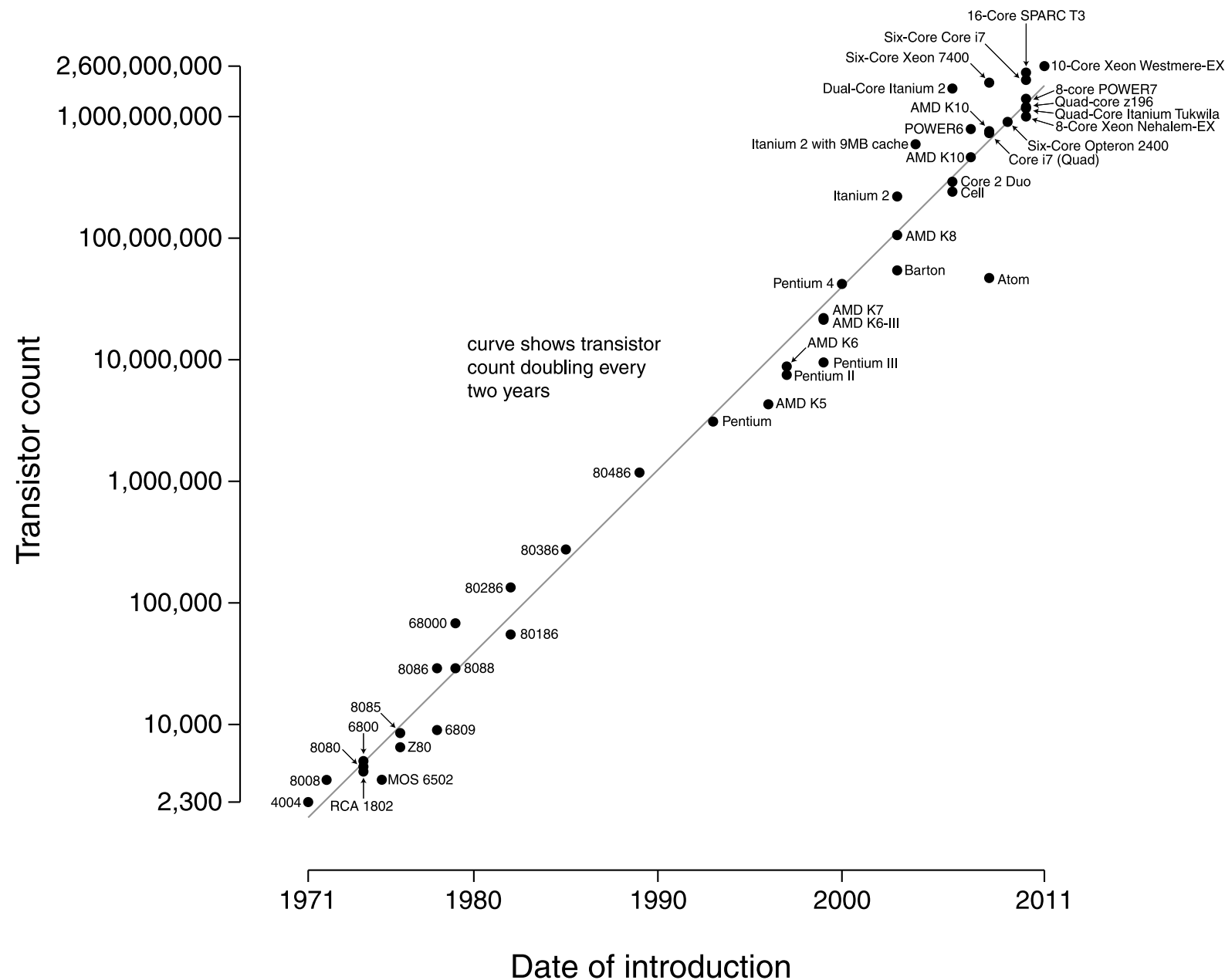
Outline

- ▶ Understanding the need of multicore architectures
- ▶ Overview of the GPU hardware
- ▶ Introduction to CUDA
 - Main features
 - Thread hierarchy
 - Simple example
 - Concepts behind a CUDA friendly code

Moore's Law

- ▶ Transistor count of integrated circuits doubles every two years

Microprocessor Transistor Counts 1971-2011 & Moore's Law



Getting performance

- ▶ We have more and more transistors, then?
- ▶ How can we get more **performance**?
 - Increase processor speed
 - ▶ Have gone from MHz to GHz in the last 30 years
 - ▶ Power scales as frequency³!
 - ▶ Memory needs to catch up
 - Parallel executing
 - ▶ Concurrency, multithreading

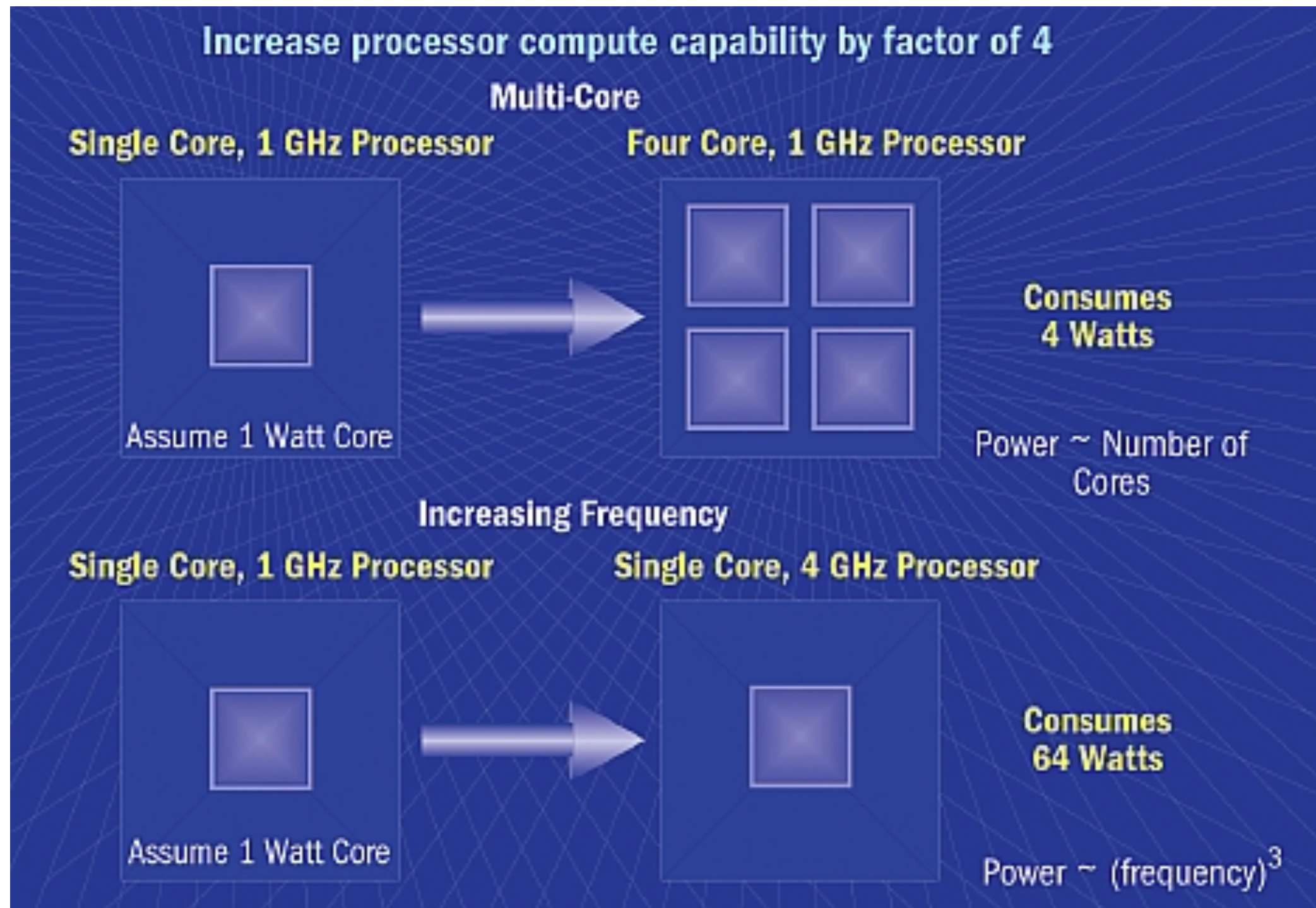
Getting performance

- ▶ We have more and more transistors, then?
- ▶ How can we get more **performance**?
 - Increase processor speed
 - ▶ Have gone from MHz to GHz in the last 30 years 
 - ▶ Power scales as frequency³!
 - ▶ Memory needs to catch up
 - Parallel executing
 - ▶ Concurrency, multithreading

Getting performance

- ▶ We have more and more transistors, then?
- ▶ How can we get more **performance**?
 - Increase processor speed
 - ▶ Have gone from MHz to GHz in the last 30 years 
 - ▶ Power scales as frequency³! 
 - ▶ Memory needs to catch up 
 - Parallel executing
 - ▶ Concurrency, multithreading

Getting performance



Moore's Law

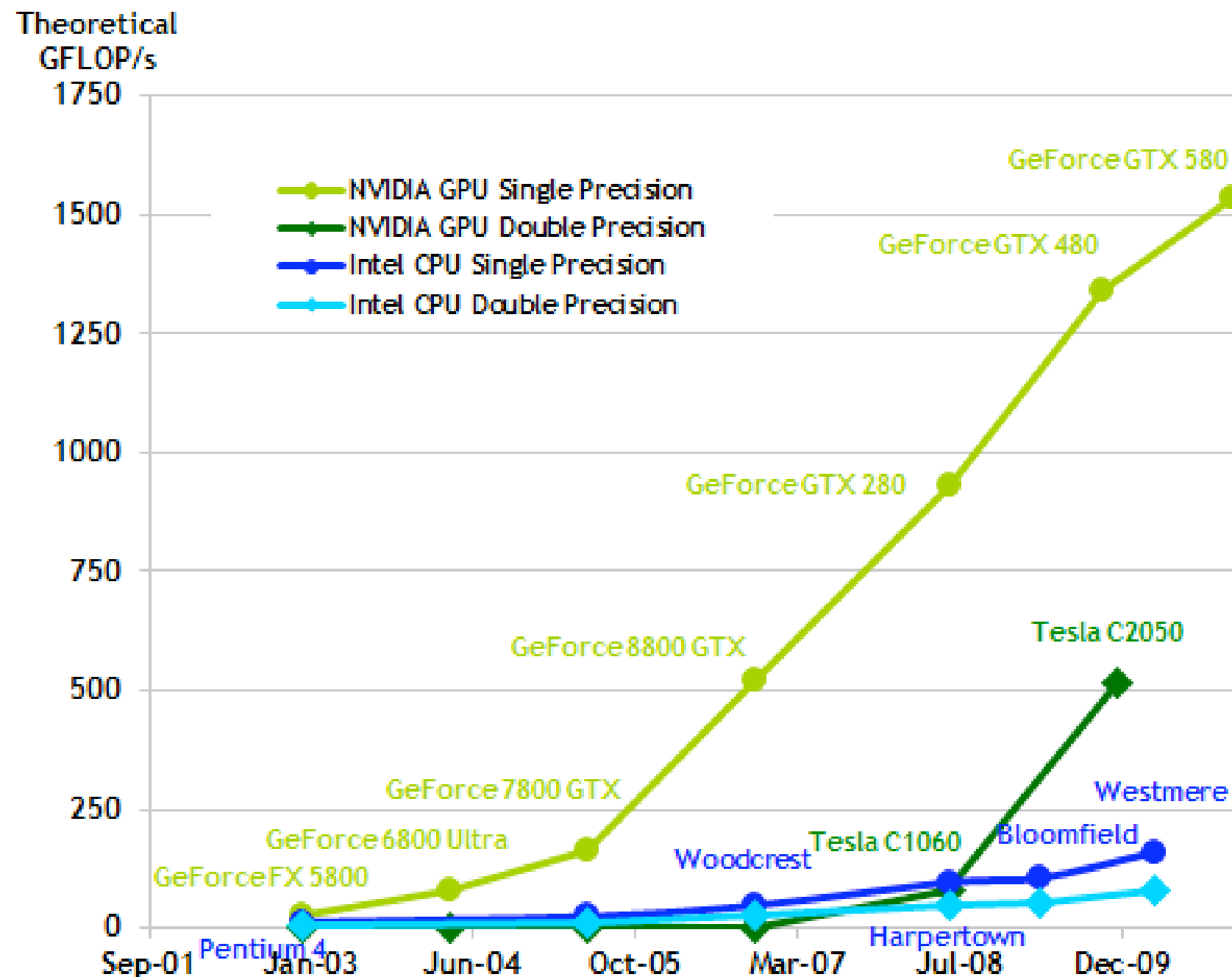
- ▶ Serial performance scaling reached its peak
- ▶ Processors are not getting faster, but wider
- ▶ Challenge: **parallel thinking**

Graphic Processing Units (GPU)



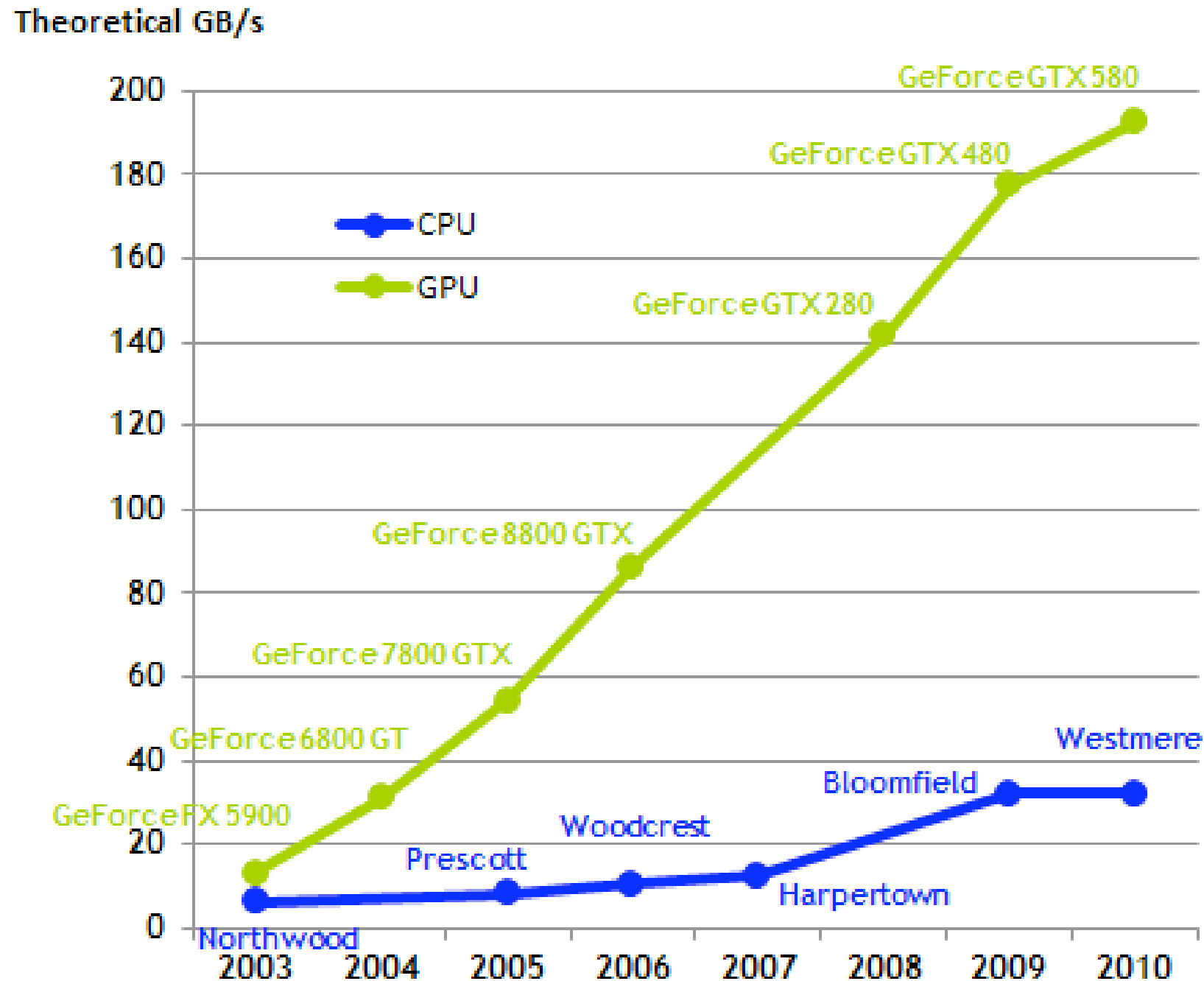
Graphic Processing Units (GPU)

- ▶ GPU is a hardware specially designed for highly parallel applications (graphics)



Graphic Processing Units (GPU)

- Fast processing must come with high bandwidth!



Graphic Processing Units (GPU)

- ▶ Graphics market is hungry for better and faster rendering
- ▶ Development is pushed by this HUGE industry
 - High quality product
 - Cheap!
- ▶ Proven to be a real alternative for scientific applications

Rank	Site	Computer/Year Vendor	Cores	R _{max}	R _{peak}	Power
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	548352	8162.00	8773.63	9898.56
2	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
3	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
4	National Supercomputing Centre in Shenzhen (NSCS) China	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU / 2010 Dawning	120640	1271.00	2984.30	2580.00
5	GSIC Center, Tokyo Institute of Technology Japan	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows / 2010 NEC/HP	73278	1192.00	2287.63	1398.61
6	DOE/NNSA/LANL/SNL United States	Cielo - Cray XE6 8-core 2.4 GHz / 2011 Cray Inc.	142272	1110.00	1365.81	3980.00

Top 500 list. June 2011

GPU chip design

► CPU vs GPU

Control

ALU

ALU

ALU

ALU

ALU

ALU

ALU

ALU

Cache

DRAM

CPU

CPU

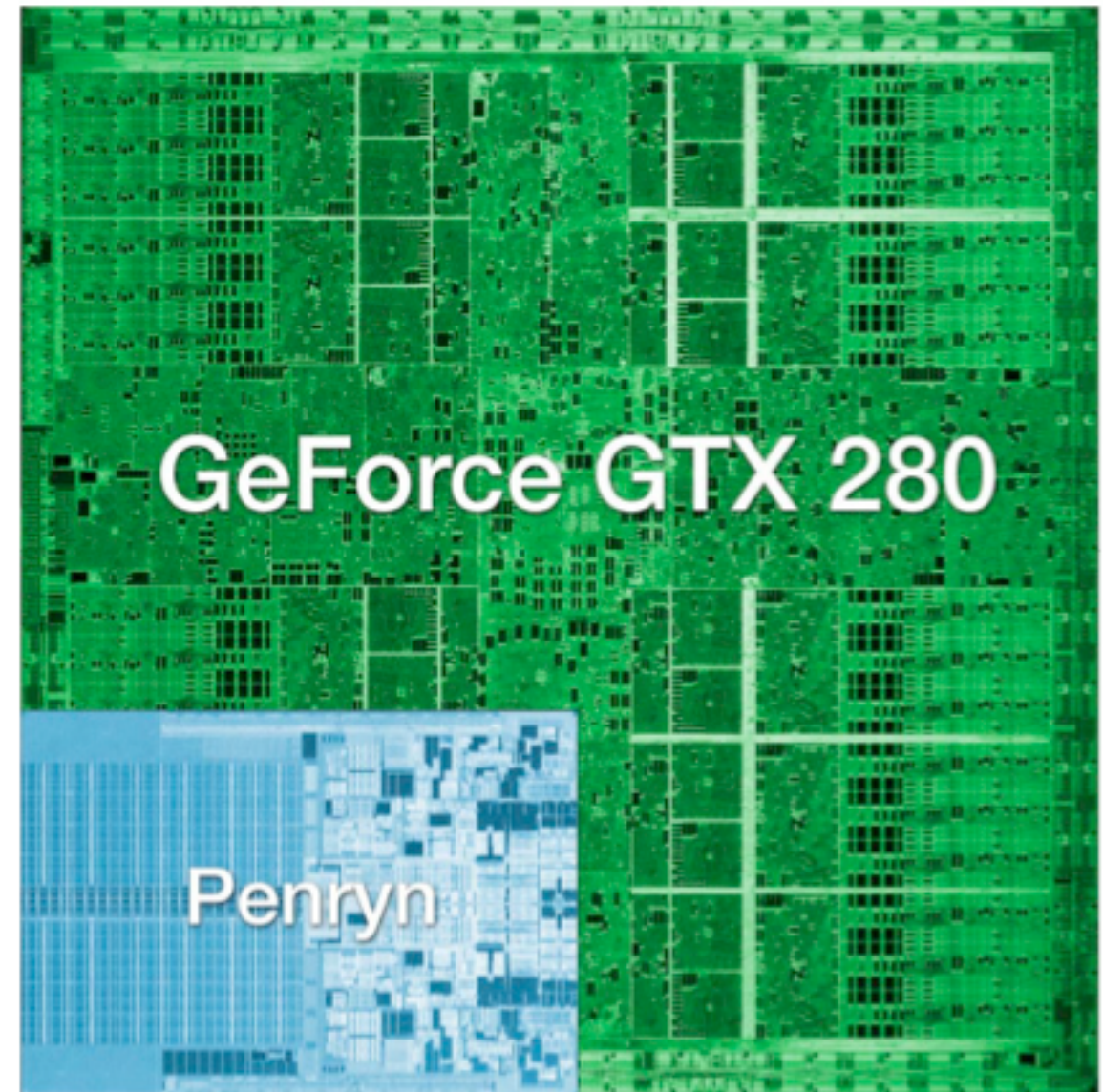
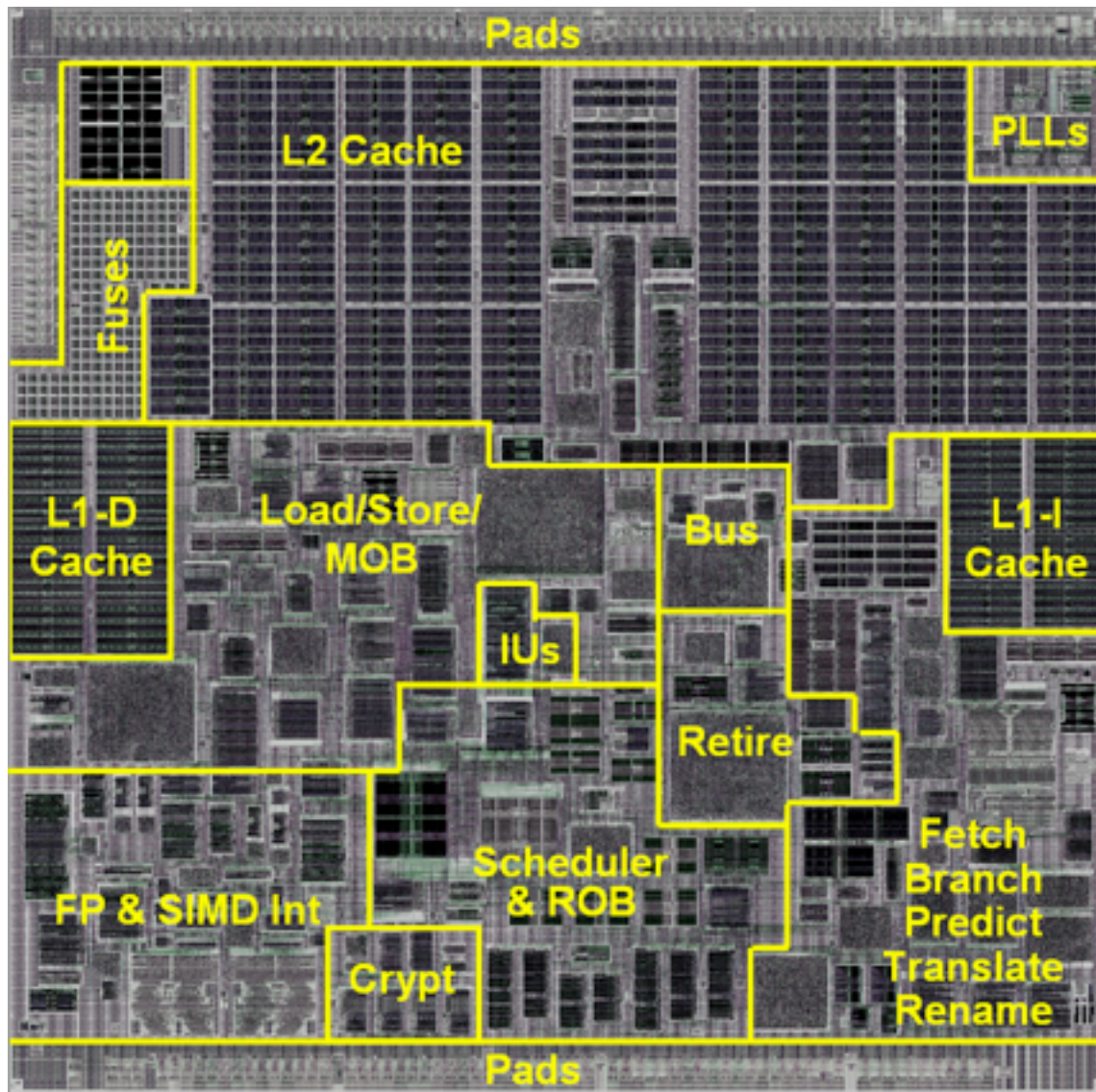
DRAM

GPU

GPU

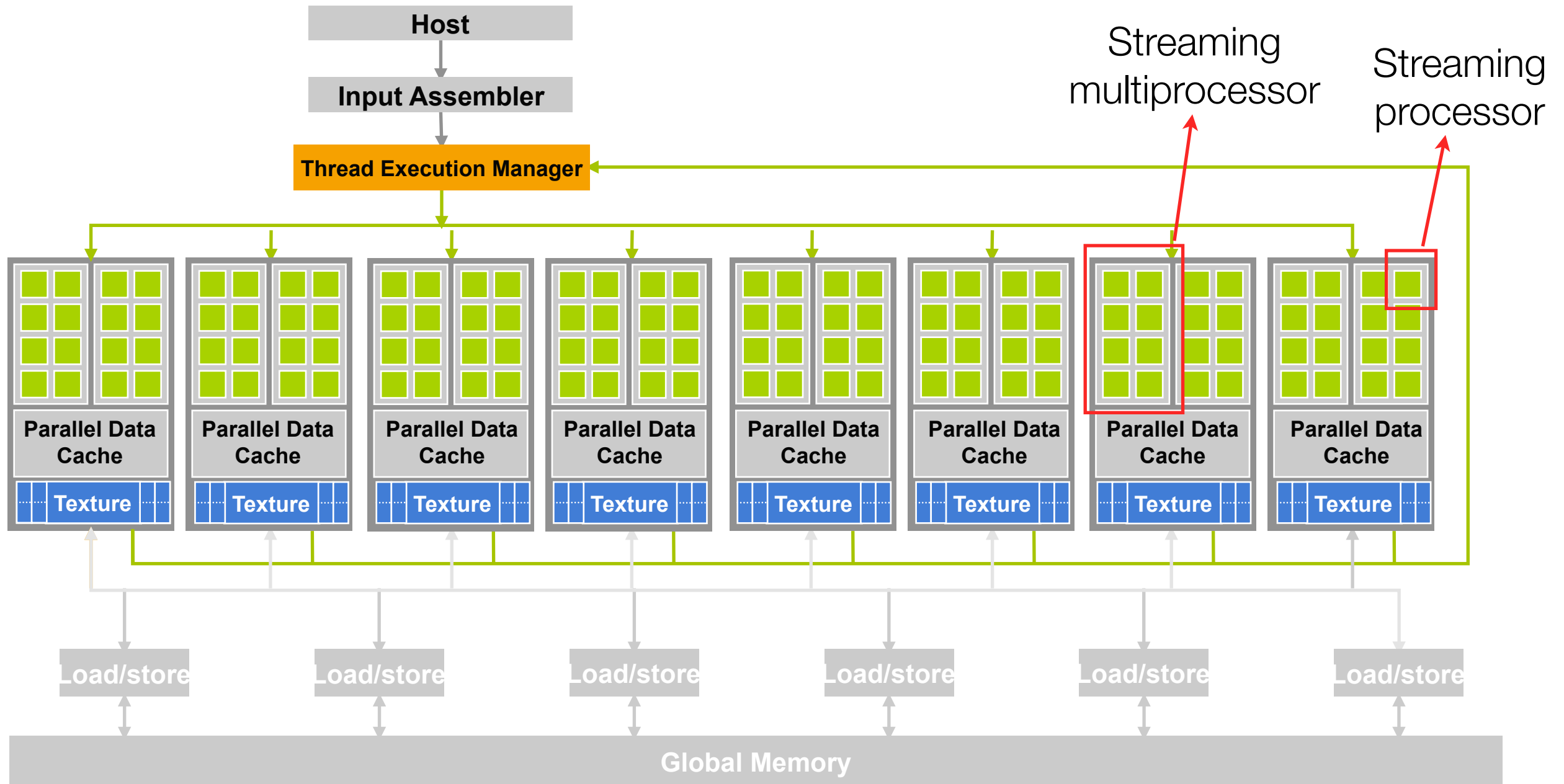
GPU devotes more transistors to data processing

GPU chip design



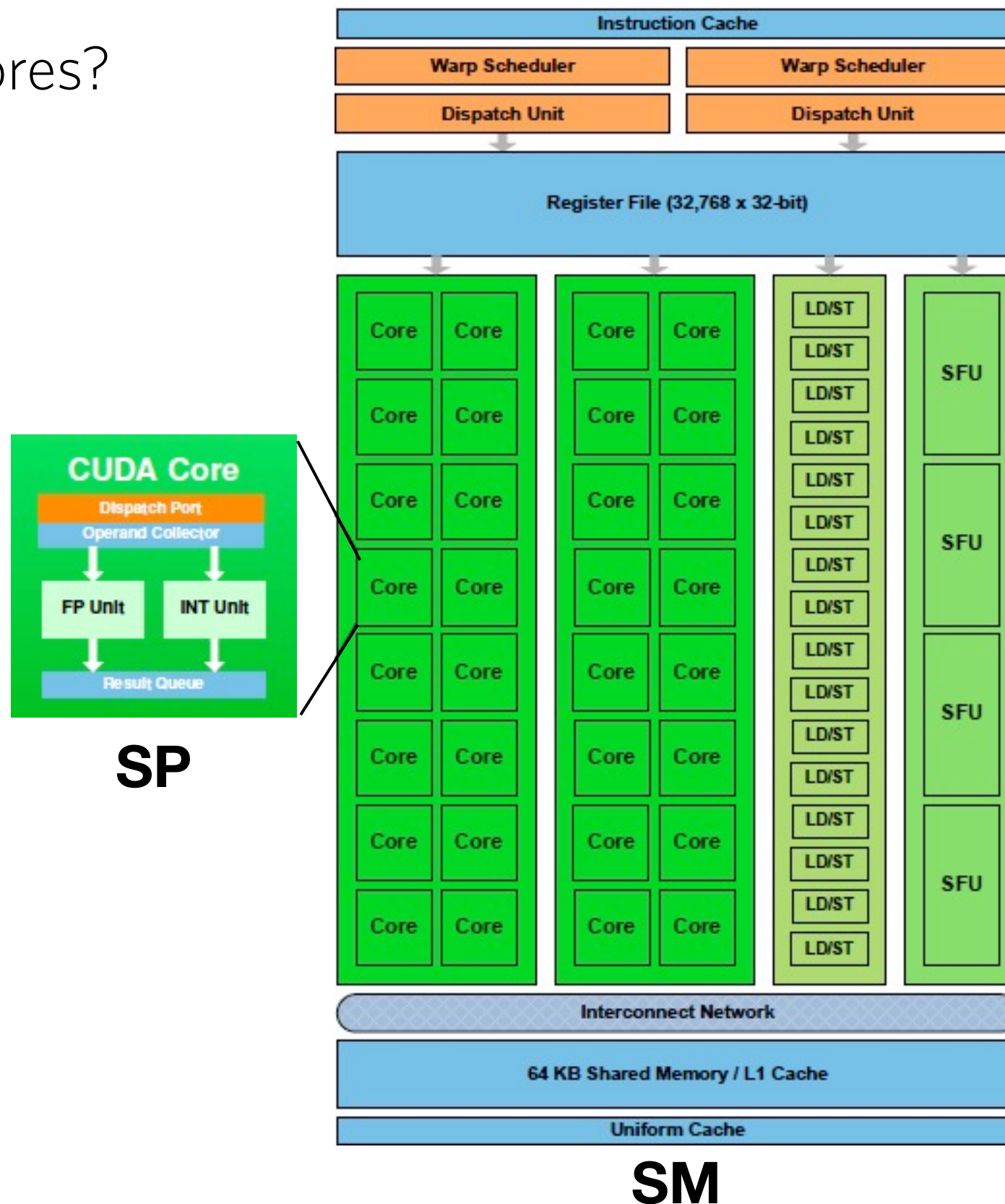
GPU chip design

► A glance at the GeForce 8800

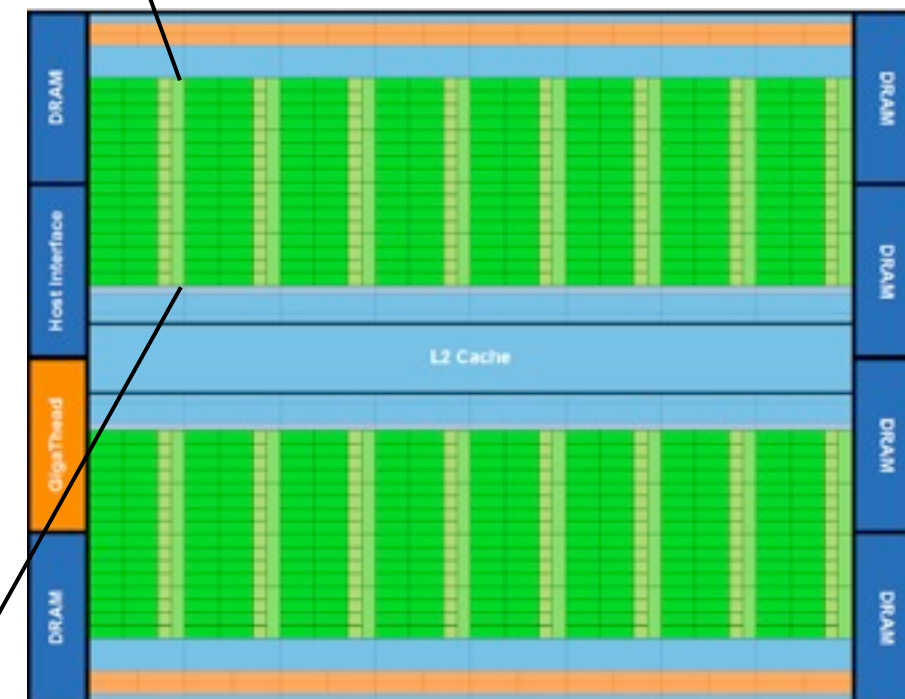


GPU chip design

► Cores?

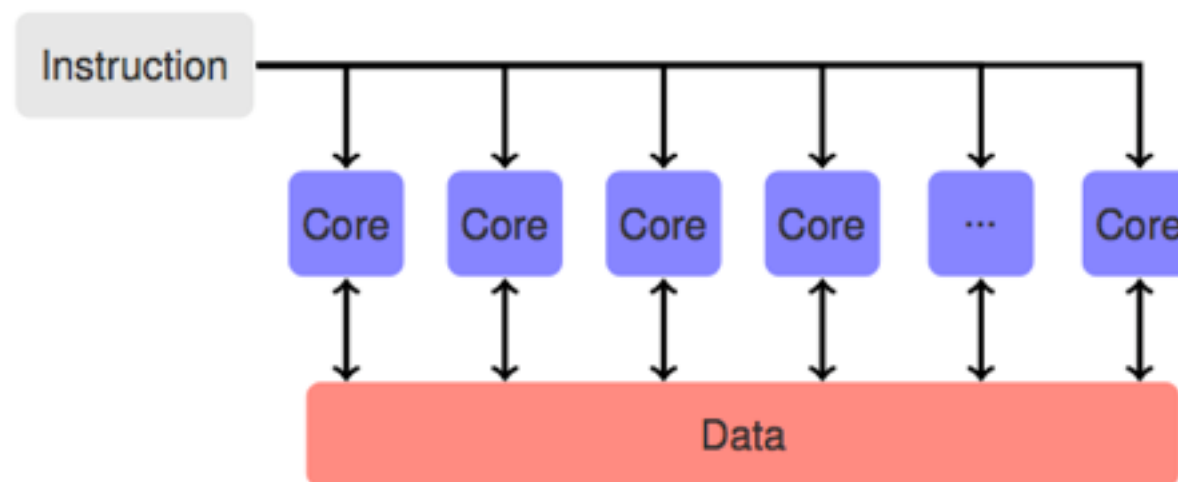


3 billion transistors
512 CUDA cores (32 in 16 SMs)
64kB of on chip RAM
High bandwidth

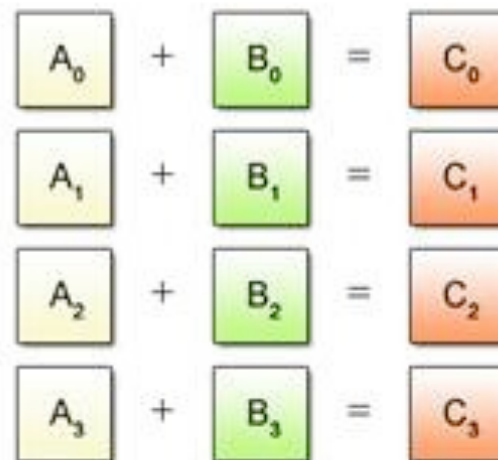


GPU chip design

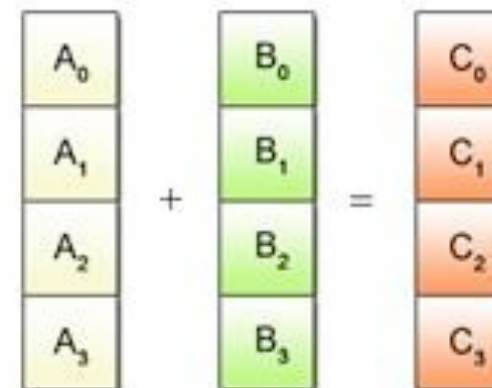
- ▶ The GPU core is the **stream processor**
- ▶ Stream processors are grouped in **Stream Multiprocessors**
 - SM is basically a **SIMD** processor (Single Instruction Multiple Data)



(a) Scalar Operation



(b) SIMD Operation



GPU chip design

► Core ideas

- GPUs consist of many “simple” cores
 - Designed for many simpler tasks: **high throughput**
 - Fewer control units: **latency**



Maximize throughput
of all threads



Minimize latency
of a thread

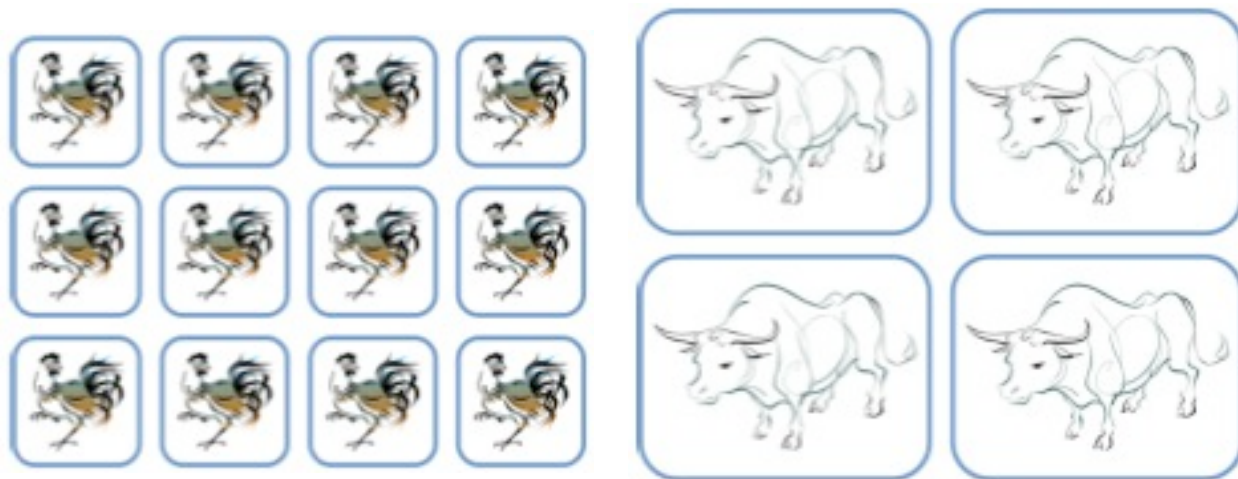
Parallel thinking

► Latency? Throughput?

- Latency: time to complete a task —————→ **Execution time**
- Throughput: number of tasks in a fixed time —————→ **Bandwidth**

► Decisions!?!

- Depends on the problem



“If you were plowing a field,
which would you rather use: two
strong oxen or 1024 chicken?”

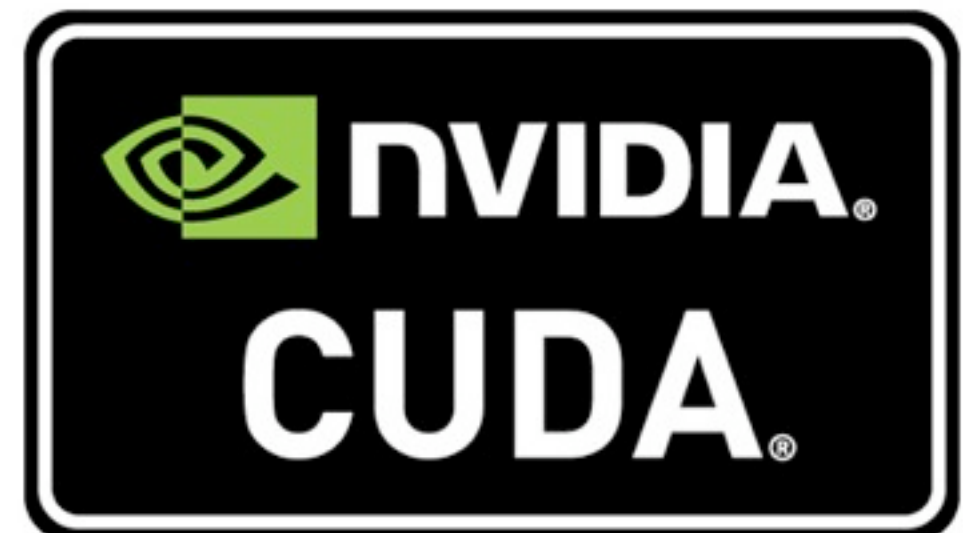
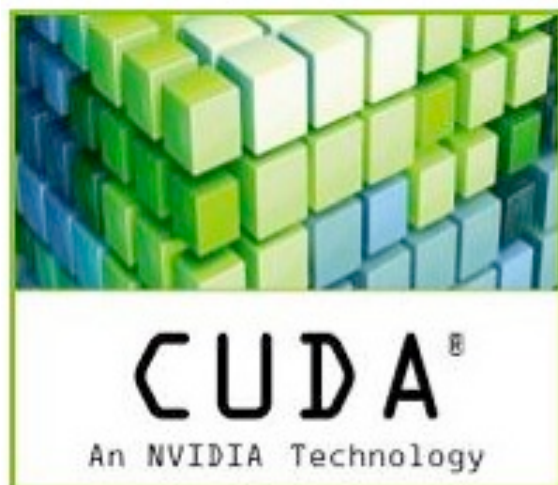
Seymour Cray

Parallel thinking

- ▶ 1024 chicken? We better have a strategy!
 - Rethink our algorithms to be more parallel friendly
 - Massively parallel:
 - ▶ Data parallelism
 - ▶ Load balancing
 - ▶ Regular computations
 - ▶ Data access
 - ▶ Avoid conflicts
 - ▶ and so on...

Computer Unified Device Architecture (CUDA)

- ▶ Parallel computer architecture developed by NVIDIA
- ▶ General purpose programming model
- ▶ Specially designed for General Purpose GPU computing:
 - Offers a compute designed API
 - Explicit GPU memory managing



CUDA enabled GPUs

- ▶ What is the difference between GPUs
 - CUDA enabled GPUs
 - GPUs classified according to compute capability

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
GeForce GTX 560 Ti	2.1	8	384
GeForce GTX 460	2.1	7	336
GeForce GTX 470M	2.1	6	288
GeForce GTS 450, GTX 460M	2.1	4	192
GeForce GT 445M	2.1	3	144
GeForce GT 435M, GT 425M, GT 420M	2.1	2	96
GeForce GT 415M	2.1	1	48
GeForce GTX 580	2.0	16	512
GeForce GTX 570, GTX 480	2.0	15	480
GeForce GTX 470	2.0	14	448
GeForce GTX 465, GTX 480M	2.0	11	352
GeForce GTX 295	1.3	2x30	2x240
GeForce GTX 285, GTX 280, GTX 275	1.3	30	240

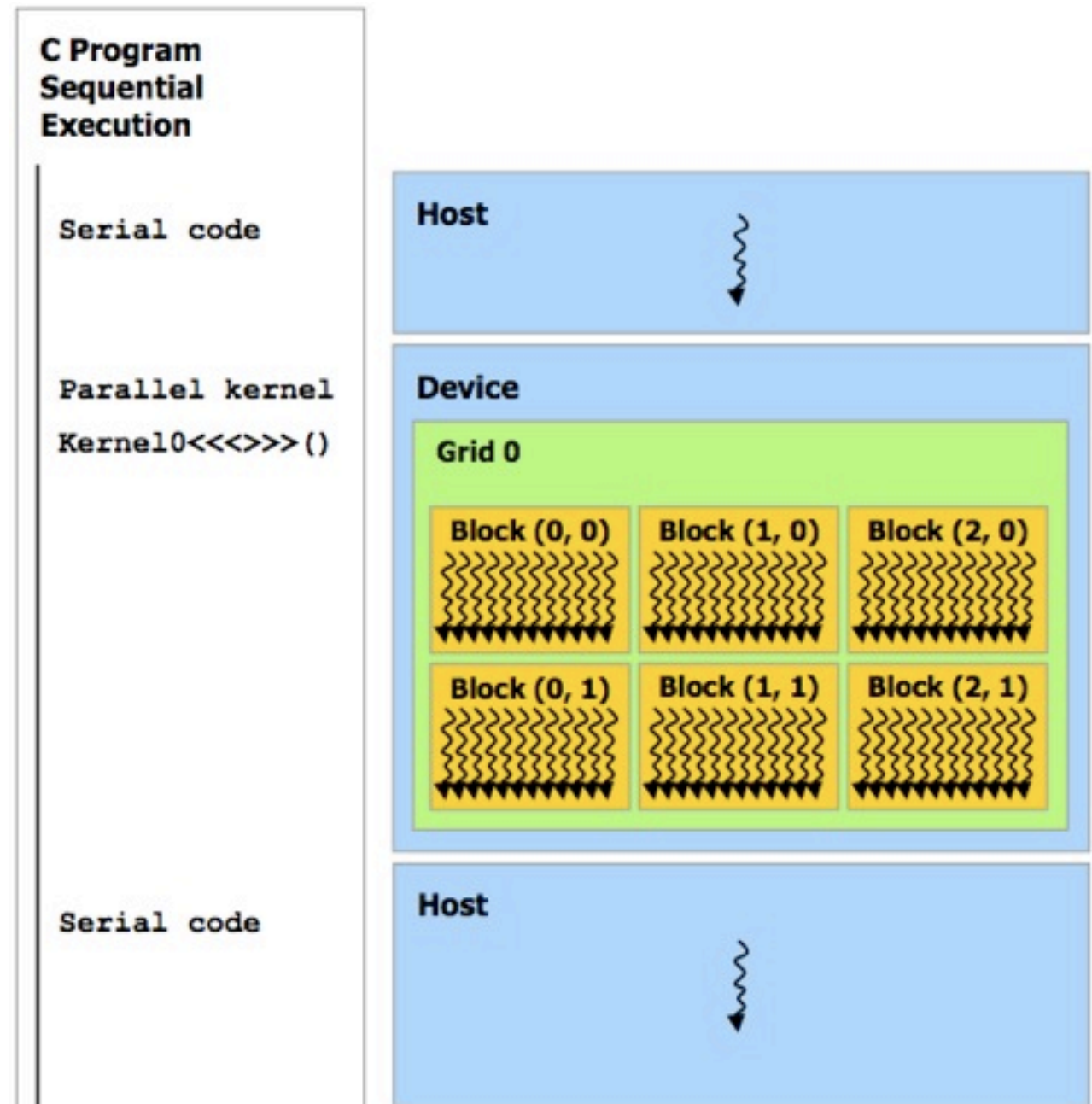
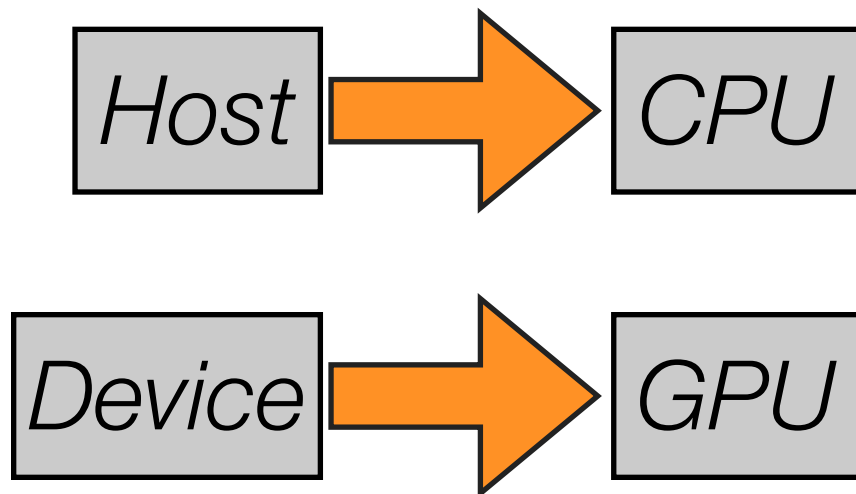
CUDA Programming Guide Appendix A

Technical Specifications	Compute Capability				
	1.0	1.1	1.2	1.3	2.x
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Maximum number of threads per block	512				1024
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24		32		48
Maximum number of resident threads per multiprocessor	768		1024		1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K

CUDA Programming Guide Appendix F

CUDA - Main features

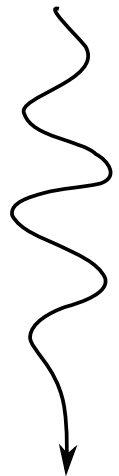
- ▶ C/C++ with extensions
- ▶ Heterogeneous programming model:
 - Operates in CPU (host) and GPU (device)



CUDA - Threads

► In CUDA, a **kernel** is executed by many **threads**

- A thread is a sequence of executions
- Multi-thread: many threads will be running at the same time

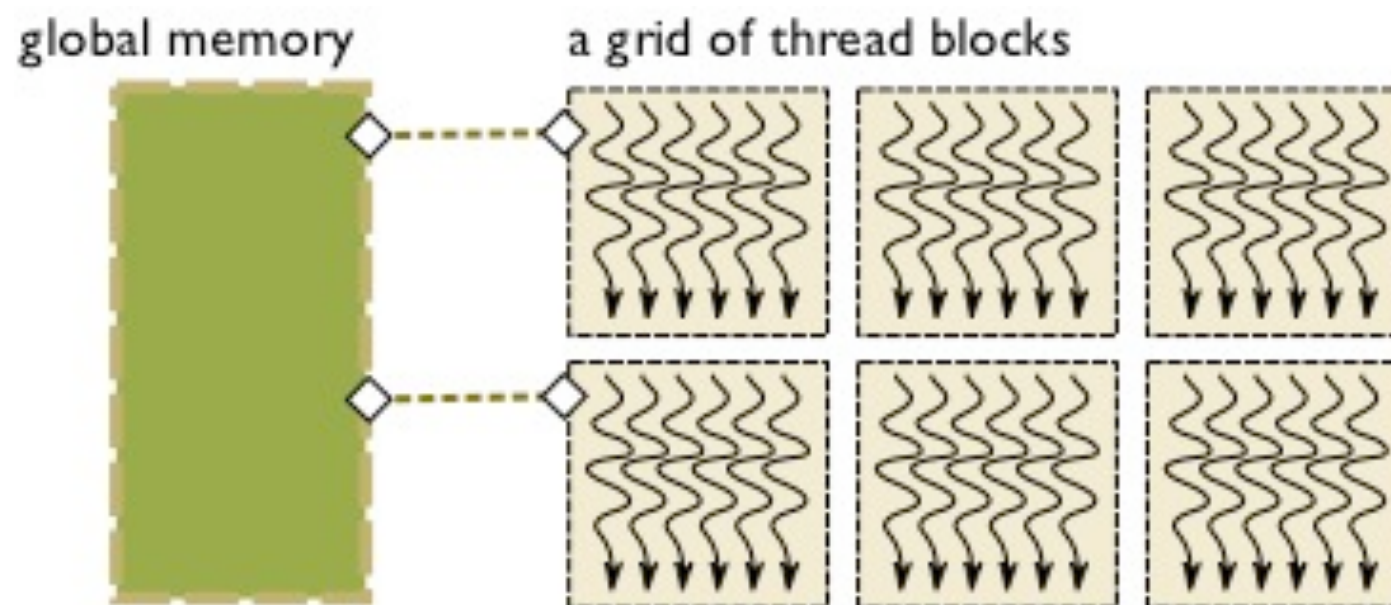
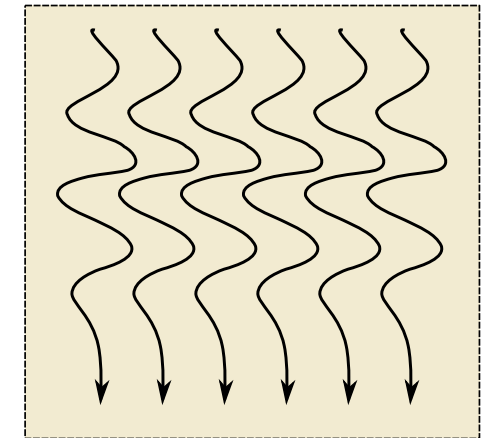


```
__global__ void vec_add (float *A, float *B, float *C, int N){  
    int i = threadIdx.x + blockDim.x*blockIdx.x;  
  
    if (i>=N) {return;}  
    C[i] = A[i] + B[i];  
}
```

```
void vec_add (float *A, float *B, float *C, int N){  
    for (int i=0; i<N; i++)  
        C[i] = A[i] + B[i];  
}
```

CUDA - Threads

- ▶ Threads are grouped into **thread blocks**
 - Programming abstraction
 - All threads within a thread block run in the same SM
 - Threads of the same block can communicate
- ▶ Thread blocks conform a **grid**

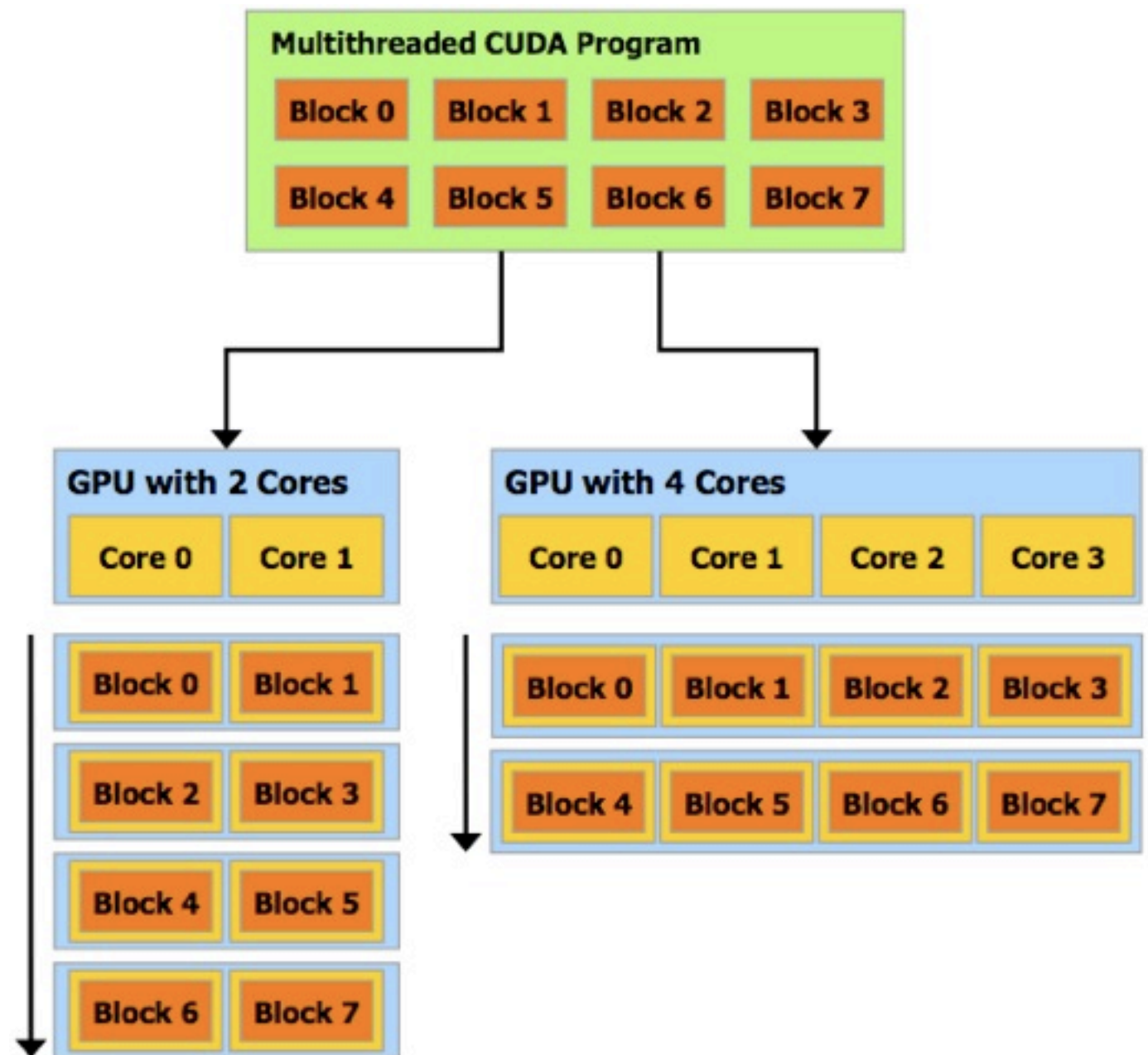


CUDA - Threads

- ▶ CUDA virtualizes the physical hardware
 - **Thread** is a virtualized scalar processor
 - **Thread blocks** is a virtualized multiprocessors
- ▶ Thread blocks need to be **independent**
 - They run to completion
 - No idea in which order they run

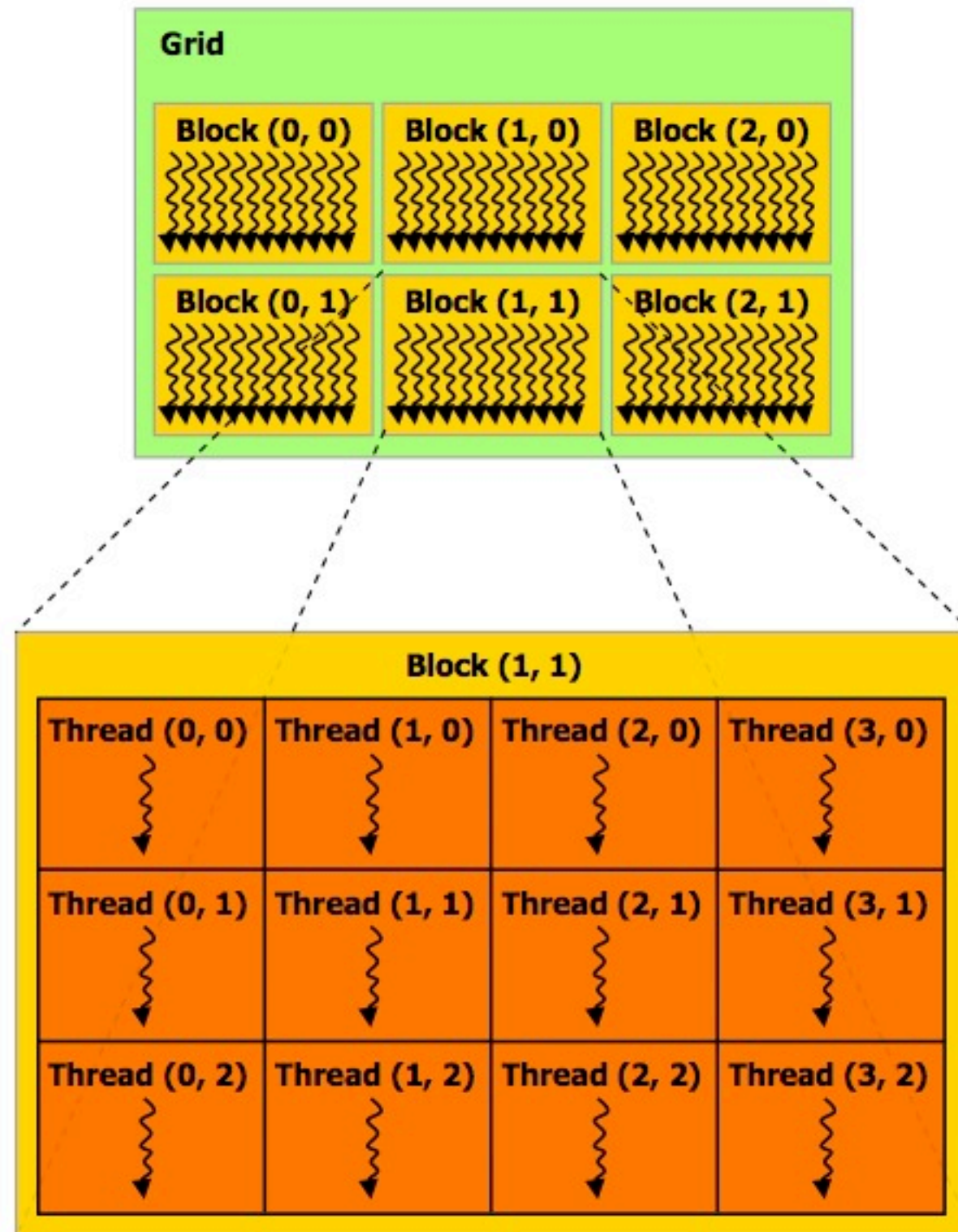
CUDA - Threads

- Provides automatic scalability across GPUs
 - Thread hierarchy
 - Shared memories
 - Barrier synchronization



CUDA - Threads

- ▶ Threads have a unique combination of block ID and thread ID
 - We can operate in different parts of the data
 - SIMT: Single Instruction Multiple Threads
 - Threads: 1D, 2D or 3D
 - Blocks: 1D or 2D



CUDA - Extensions to C

- ▶ Minimal effort from C to CUDA

- ▶ Function declarations

```
__global__ void kernel()  
__device__ float function()
```

- ▶ Variable qualifiers

```
__shared__ float array[5]  
__constant__ float array[5]
```

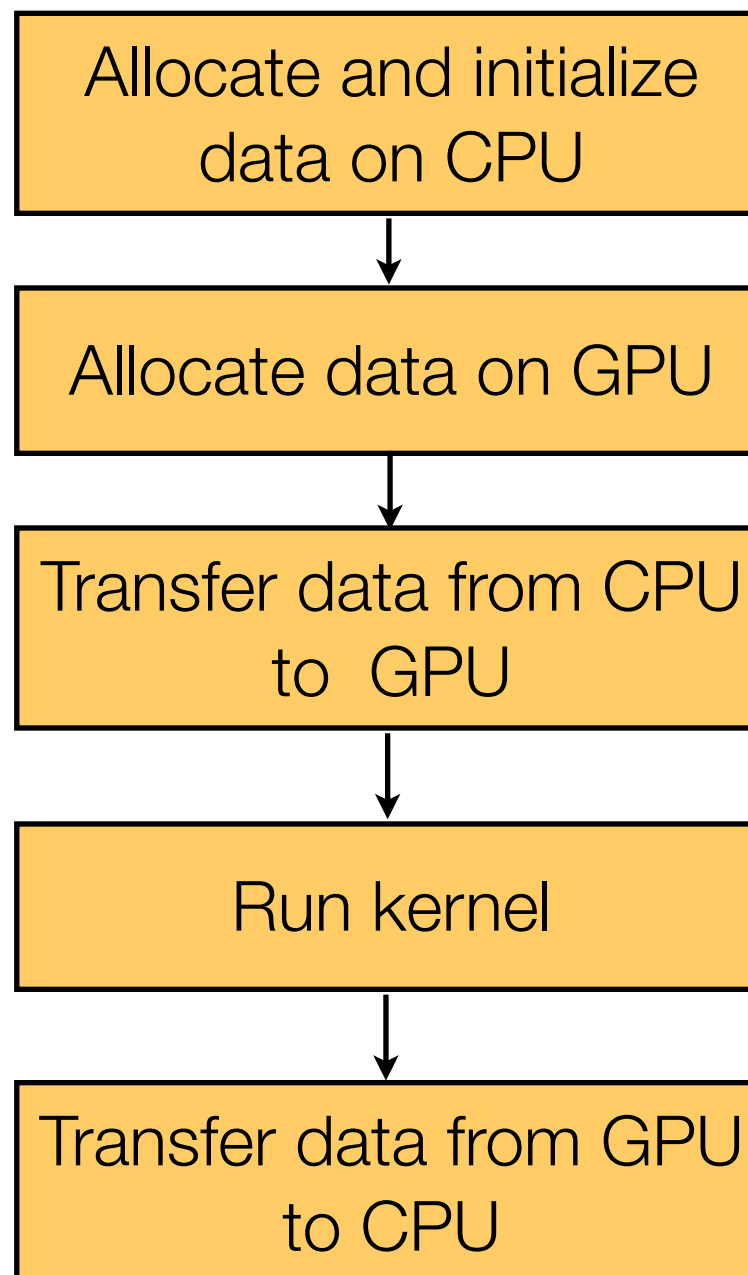
- ▶ Execution configuration

```
dim3 dim_grid(100,100); // 10000 blocks  
dim3 dim_block(16, 16); // 256 threads per block in 2D  
kernel <<<dim_grid, dim_block>>> (); // Launch kernel
```

- ▶ Others

```
threadIdx.x  
blockIdx.x
```

CUDA - Program execution



CUDA - Vector add example

```
__global__ void  vec_add (float *A, float *B, float *C, int N)
{
    // Using 1D threadID and block ID
    int i = threadIdx.x + blockDim.x*blockIdx.x;

    if (i>=N) {return;}

    C[i] = A[i] + B[i];
}

int main()
{
    ....
    // Launch kernel with N/256 blocks of 256 threads
    int blocks = int(N-0.5)/256 + 1;
    vec_add<<<blocks, 256>>> (A_d, B_d, C_d, N);
}
```

CUDA - Vector add example

```
__global__ void vec_add (float *A, float *B, float *C, int N)
{
    // Using 1D threadIdx and block ID
    int i = threadIdx.x + blockDim.x*blockIdx.x;

    if (i>=N) {return;}

    C[i] = A[i] + B[i];
}
```

Thread indexing

```
int main()
{
    ....
    // Launch kernel with N/256 blocks of 256 threads
    int blocks = int(N-0.5)/256 + 1;
    vec_add<<<blocks, 256>>> (A_d, B_d, C_d, N);
}
```

CUDA - Vector add example

```
__global__ void vec_add (float *A, float *B, float *C, int N)
{
    // Using 1D threadID and block ID
    int i = threadIdx.x + blockDim.x*blockIdx.x;

    if (i>=N) {return;}

    C[i] = A[i] + B[i];
}

int main()
{
    ....
    // Launch kernel with N/256 blocks of 256 threads
    int blocks = int(N-0.5)/256 + 1;
    vec_add<<<blocks, 256>>> (A_d, B_d, C_d, N);
}
```

Perform operation

CUDA - Vector add example

```
int N = 200;

// Allocate and initialize host CPU
float *A_h = new float[N];
float *B_h = new float[N];
float *C_h = new float[N];

for(int i=0; i<N; i++)
{
    A_h[i] = 1.3f;
    B_h[i] = 2.0f;
}

// Allocate memory on the GPU
float *A_d, *B_d, *C_d;

cudaMalloc( (void**) &A_d, N*sizeof(float));
cudaMalloc( (void**) &B_d, N*sizeof(float));
cudaMalloc( (void**) &C_d, N*sizeof(float));

// Copy host memory to device
cudaMemcpy(A_d, A_h, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(float), cudaMemcpyHostToDevice);
```


CUDA - Vector add example

```
int N = 200;
```

```
// Allocate and initialize host CPU  
float *A_h = new float[N];  
float *B_h = new float[N];  
float *C_h = new float[N];  
  
for(int i=0; i<N; i++)  
{  
    A_h[i] = 1.3f;  
    B_h[i] = 2.0f;  
}
```

CPU allocation

```
// Allocate memory on the GPU  
float *A_d, *B_d, *C_d;
```

```
cudaMalloc( (void**) &A_d, N*sizeof(float));  
cudaMalloc( (void**) &B_d, N*sizeof(float));  
cudaMalloc( (void**) &C_d, N*sizeof(float));
```

```
// Copy host memory to device  
cudaMemcpy(A_d, A_h, N*sizeof(float), cudaMemcpyHostToDevice);  
cudaMemcpy(B_d, B_h, N*sizeof(float), cudaMemcpyHostToDevice);
```

CUDA - Vector add example

```
int N = 200;

// Allocate and initialize host CPU
float *A_h = new float[N];
float *B_h = new float[N];
float *C_h = new float[N];

for(int i=0; i<N; i++)
{
    A_h[i] = 1.3f;
    B_h[i] = 2.0f;
}
```

```
// Allocate memory on the GPU
float *A_d, *B_d, *C_d;

cudaMalloc( (void**) &A_d, N*sizeof(float));
cudaMalloc( (void**) &B_d, N*sizeof(float));
cudaMalloc( (void**) &C_d, N*sizeof(float));
```

GPU allocation

```
// Copy host memory to device
cudaMemcpy(A_d, A_h, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(float), cudaMemcpyHostToDevice);
```

CUDA - Vector add example

```
int N = 200;

// Allocate and initialize host CPU
float *A_h = new float[N];
float *B_h = new float[N];
float *C_h = new float[N];

for(int i=0; i<N; i++)
{
    A_h[i] = 1.3f;
    B_h[i] = 2.0f;
}

// Allocate memory on the GPU
float *A_d, *B_d, *C_d;

cudaMalloc( (void**) &A_d, N*sizeof(float));
cudaMalloc( (void**) &B_d, N*sizeof(float));
cudaMalloc( (void**) &C_d, N*sizeof(float)); Copy to device

// Copy host memory to device
cudaMemcpy(A_d, A_h, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(float), cudaMemcpyHostToDevice);
```

CUDA - Vector add example

```
// Copy host memory to device
cudaMemcpy(A_d, A_h, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(float), cudaMemcpyHostToDevice);

// Run grid of N/256 blocks of 256 threads each
int blocks = int(N-0.5)/256 + 1;
vec_add<<<blocks, 256>>> (A_d, B_d, C_d, N);

// Copy back from device to host
cudaMemcpy(C_h, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);

// Free device
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
```

CUDA - Vector add example

Copy to device

```
// Copy host memory to device
cudaMemcpy(A_d, A_h, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(float), cudaMemcpyHostToDevice);
```

```
// Run grid of N/256 blocks of 256 threads each
int blocks = int(N-0.5)/256 + 1;
vec_add<<<blocks, 256>>> (A_d, B_d, C_d, N);
```

```
// Copy back from device to host
cudaMemcpy(C_h, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);
```

```
// Free device
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
```

CUDA - Vector add example

```
// Copy host memory to device
cudaMemcpy(A_d, A_h, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(float), cudaMemcpyHostToDevice);
```

```
// Run grid of N/256 blocks of 256 threads each
int blocks = int(N-0.5)/256 + 1;
vec_add<<<blocks, 256>>> (A_d, B_d, C_d, N);
```

Launch kernel

```
// Copy back from device to host
cudaMemcpy(C_h, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);
```

```
// Free device
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
```

CUDA - Vector add example

```
// Copy host memory to device
cudaMemcpy(A_d, A_h, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(float), cudaMemcpyHostToDevice);
```

```
// Run grid of N/256 blocks of 256 threads each
int blocks = int(N-0.5)/256 + 1;
vec_add<<<blocks, 256>>> (A_d, B_d, C_d, N);
```

```
// Copy back from device to host
cudaMemcpy(C_h, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);
```

```
// Free device
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
```

Copy back to host

CUDA - Vector add example

```
// Copy host memory to device
cudaMemcpy(A_d, A_h, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(float), cudaMemcpyHostToDevice);
```

```
// Run grid of N/256 blocks of 256 threads each
int blocks = int(N-0.5)/256 + 1;
vec_add<<<blocks, 256>>> (A_d, B_d, C_d, N);
```

```
// Copy back from device to host
cudaMemcpy(C_h, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);
```

```
// Free device
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
```

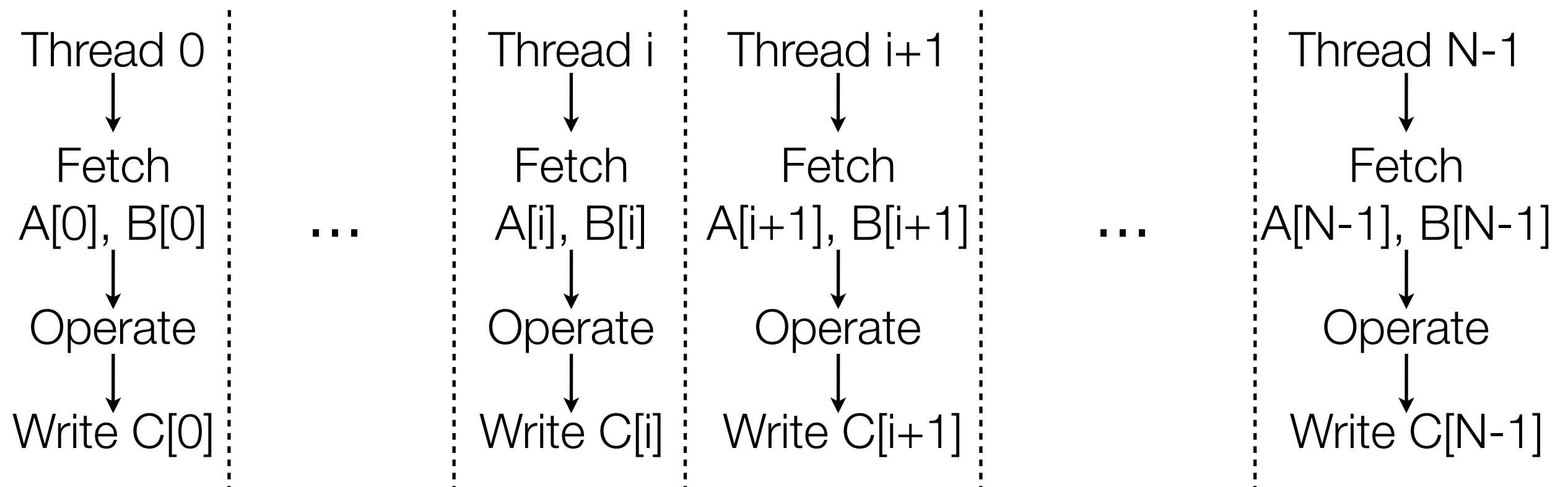
Free device

CUDA - Vector add example

```
__global__ void vec_add (float *A, float *B, float *C, int N)
{
    // Using 1D threadIdx and block ID
    int i = threadIdx.x + blockDim.x*blockIdx.x;

    if (i>=N) {return;}

    C[i] = A[i] + B[i];
}
```



CUDA - To keep in mind

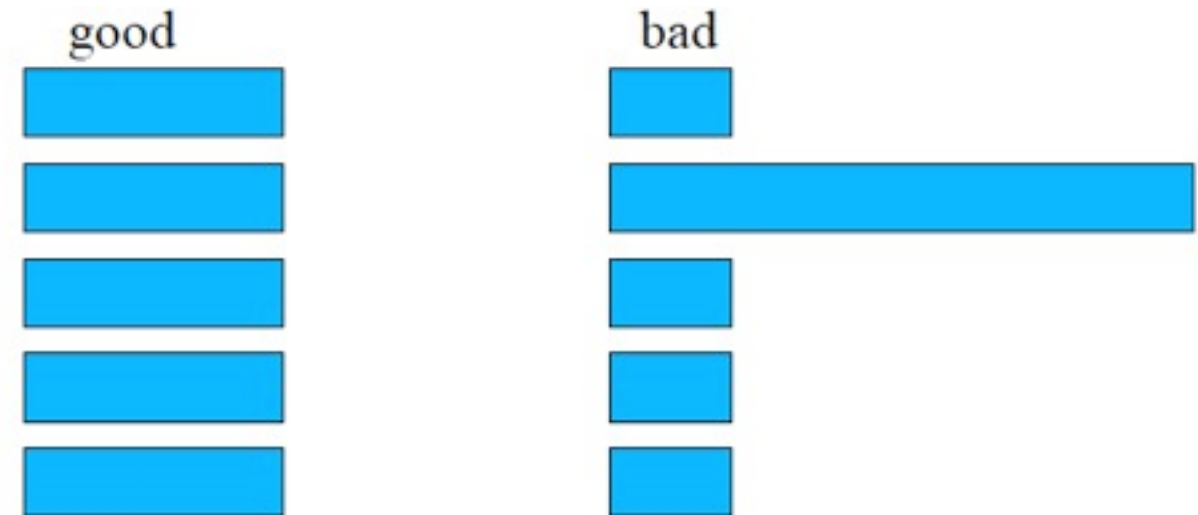
- ▶ Keep executions simple
 - Leave tough things to the CPU
- ▶ Performance:
 - Load balance
 - Data parallelism
 - Avoid conflicts
 - Keep the GPU busy!
- ▶ How fast can we go?



CUDA - To keep in mind

- ▶ Load balance

- All threads should do the same amount of work



- ▶ Data Parallelism (Massive)

- Arrange your algorithm so is data parallel friendly
- Look for regularity



- ▶ Avoid conflicts

- Data access conflicts will serialize your applications or give you wrong answers!

CUDA - To keep in mind

- ▶ Keep the GPU busy!
 - High peak compute throughput compared to bandwidth
 - Fermi:
 - ▶ 1TFLOP peak throughput, 144 GB/s peak off chip memory access (36 Gfloats per second)
 - ▶ $4 \times 1\text{TFLOP} = 4000\text{GB/s}$ for peak throughput!
 - ▶ $1000/36 \approx 28$ operations per fetched value
 - Need to hide latency!

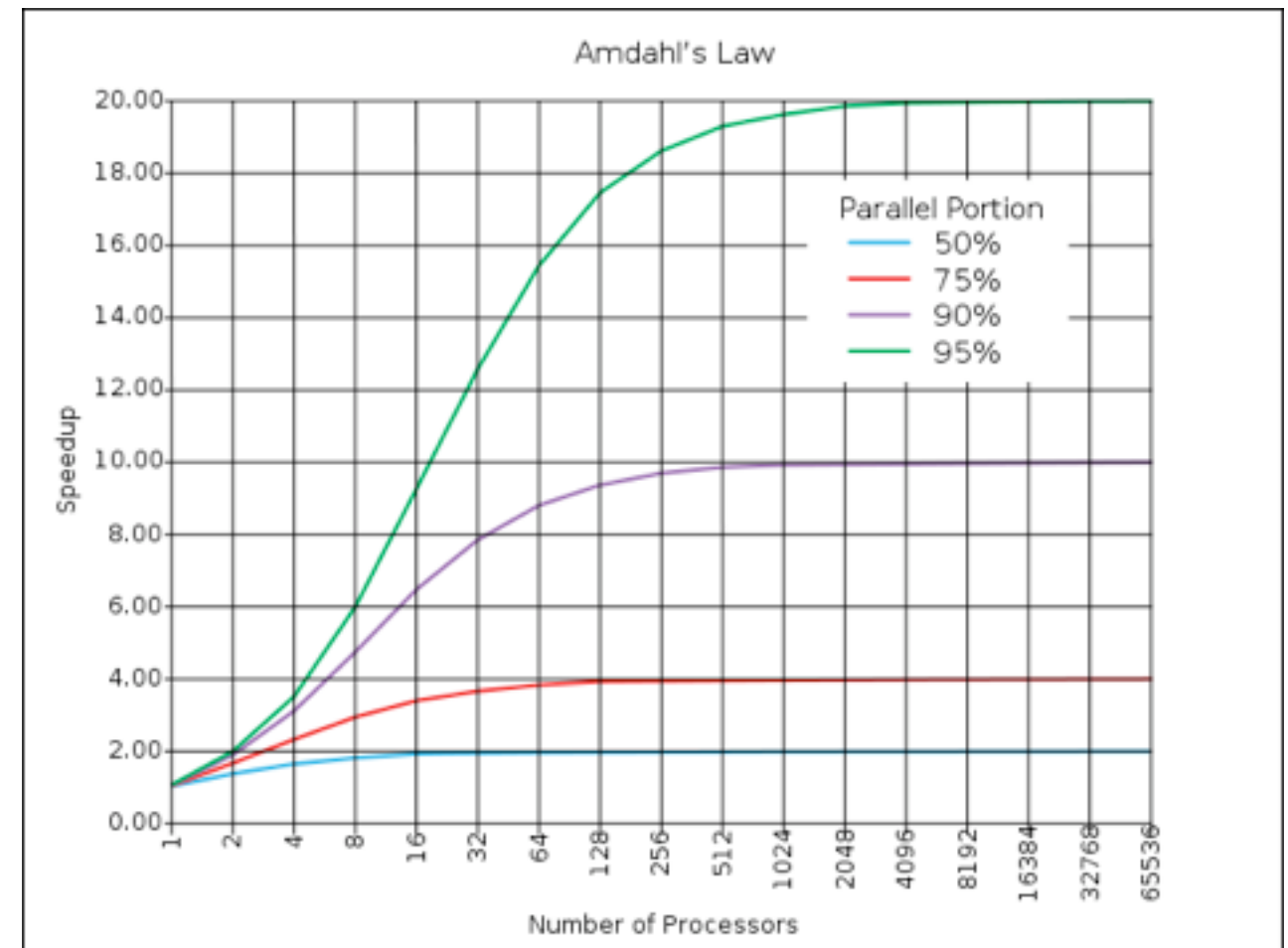


CUDA - To keep in mind

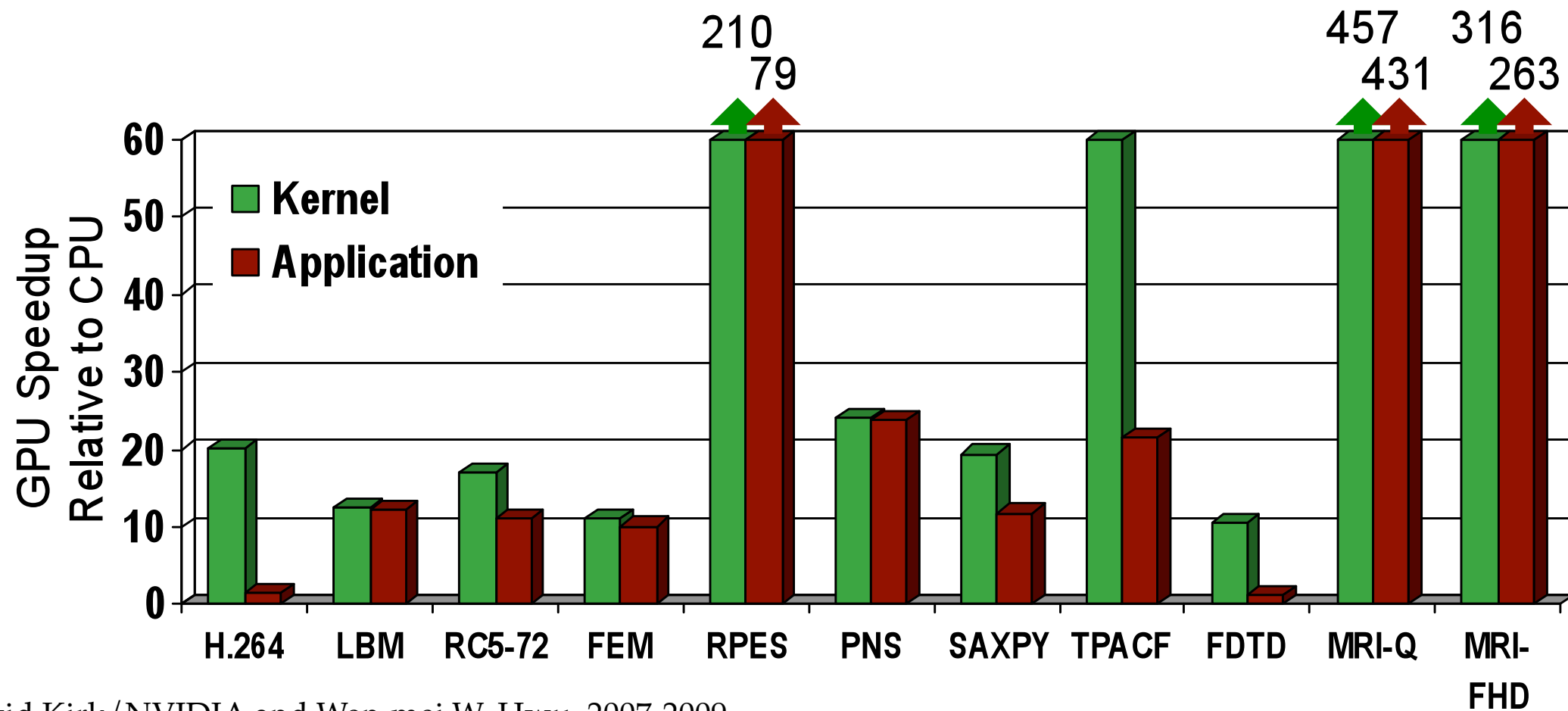
- ▶ How fast can we go?
 - Keep Amdahl's law in mind
 - Know how much parallelization can be done in your application
 - Measure independent parts of your algorithm before going to the GPU

$$\text{speedup} = \frac{1}{(1 - P) + \frac{P}{S}}$$

P: parallel portion
S: sequential portion



CUDA - Applications



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE 498AL Spring 2010, University of Illinois, Urbana-Champaign

- ▶ Ultrasound imaging
- ▶ Molecular dynamics
- ▶ Seismic migration
- ▶ Astrophysics simulations
- ▶ Graphics
- ▶