GPU Programming with CUDA

Ph.D. Pedro Velho
and
Eng. Mônica Liliana Hernandez Ariza

Disclaimer

I am not an expert on the subject

Working with GPU for less than one year

I wish I have had this course at the time!

Download the slides and examples

http://190.168.117.201/~velho/cuda.html

Meeting the audience!

How many of you used concurrent programming before?

How many threads?

How many already used CUDA?

Introduction
from **games** to **science**

1

2 **Architecture**

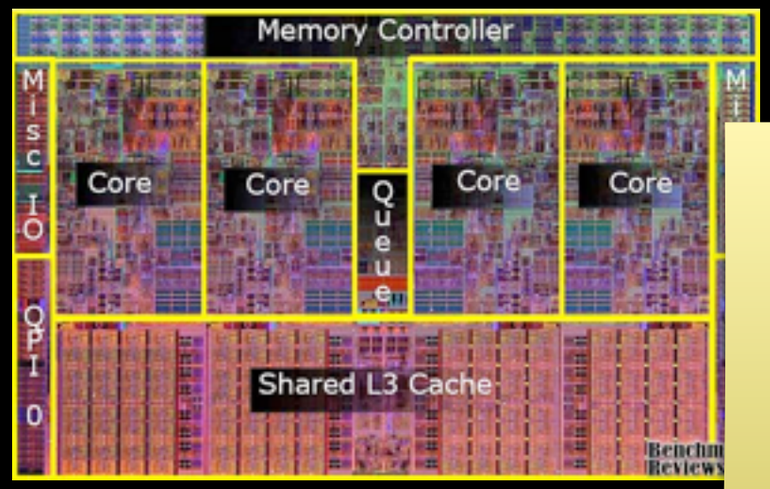
3 **GPU** Programming

CUDA 4

Final Remarks 5

CPU vs GPU

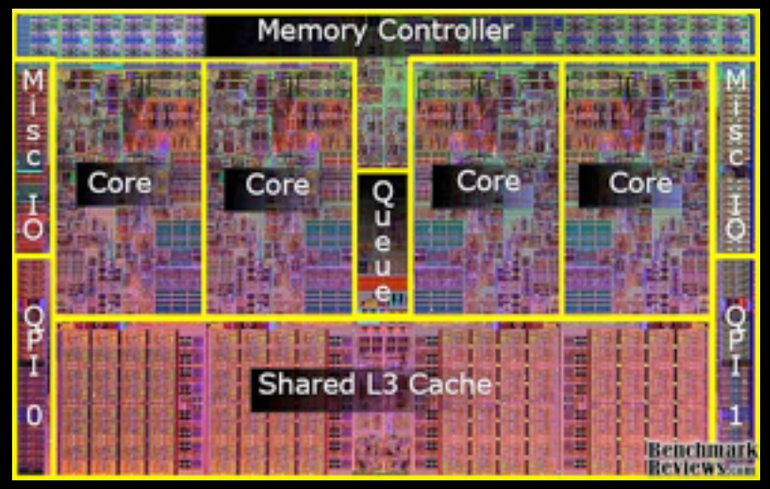
- A few general purpose cores
- Big cache memory
- Eg.: Nehalem i7 quad-core
  - 4 cores (8 threads)
  - Cache is about 50% of die area



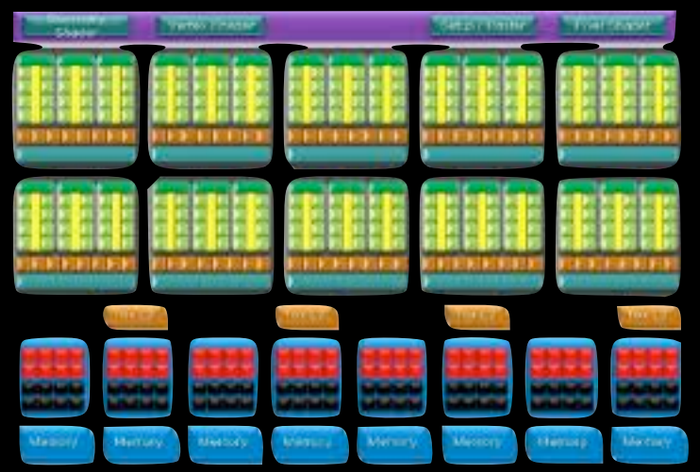
Litografia

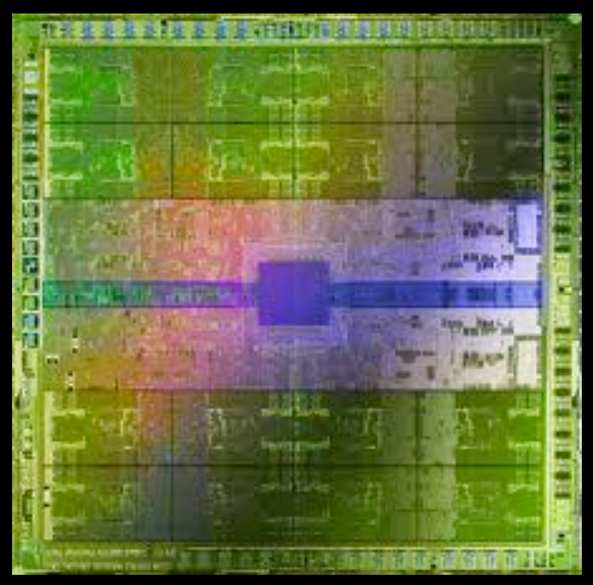
CPU vs GPU

- A few general purpose cores
- Big cache memory
- Eg.: Nehalem i7 quad-core
  - 4 cores (8 threads)
  - Cache is about 50% of die area



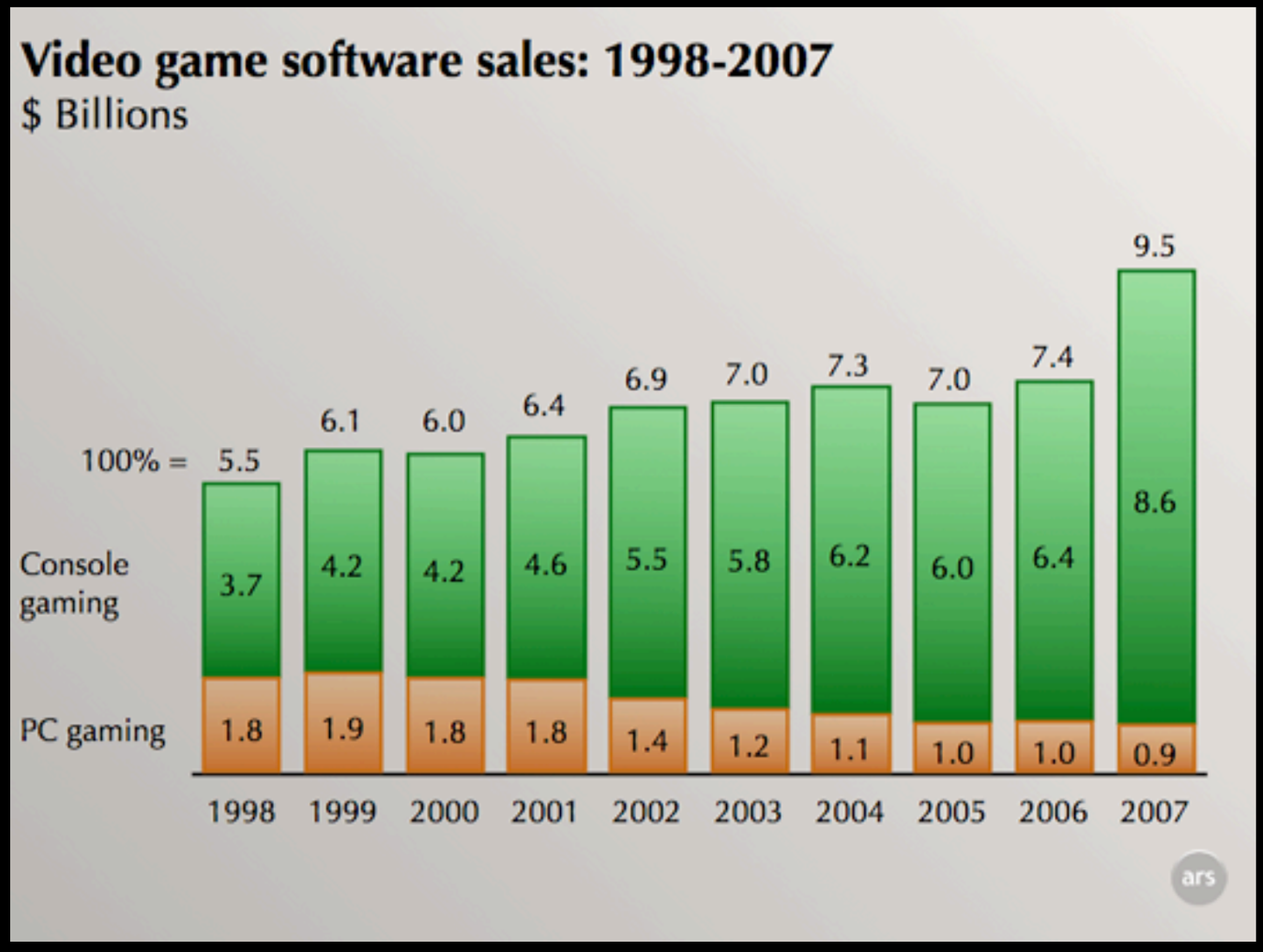
CPU vs GPU





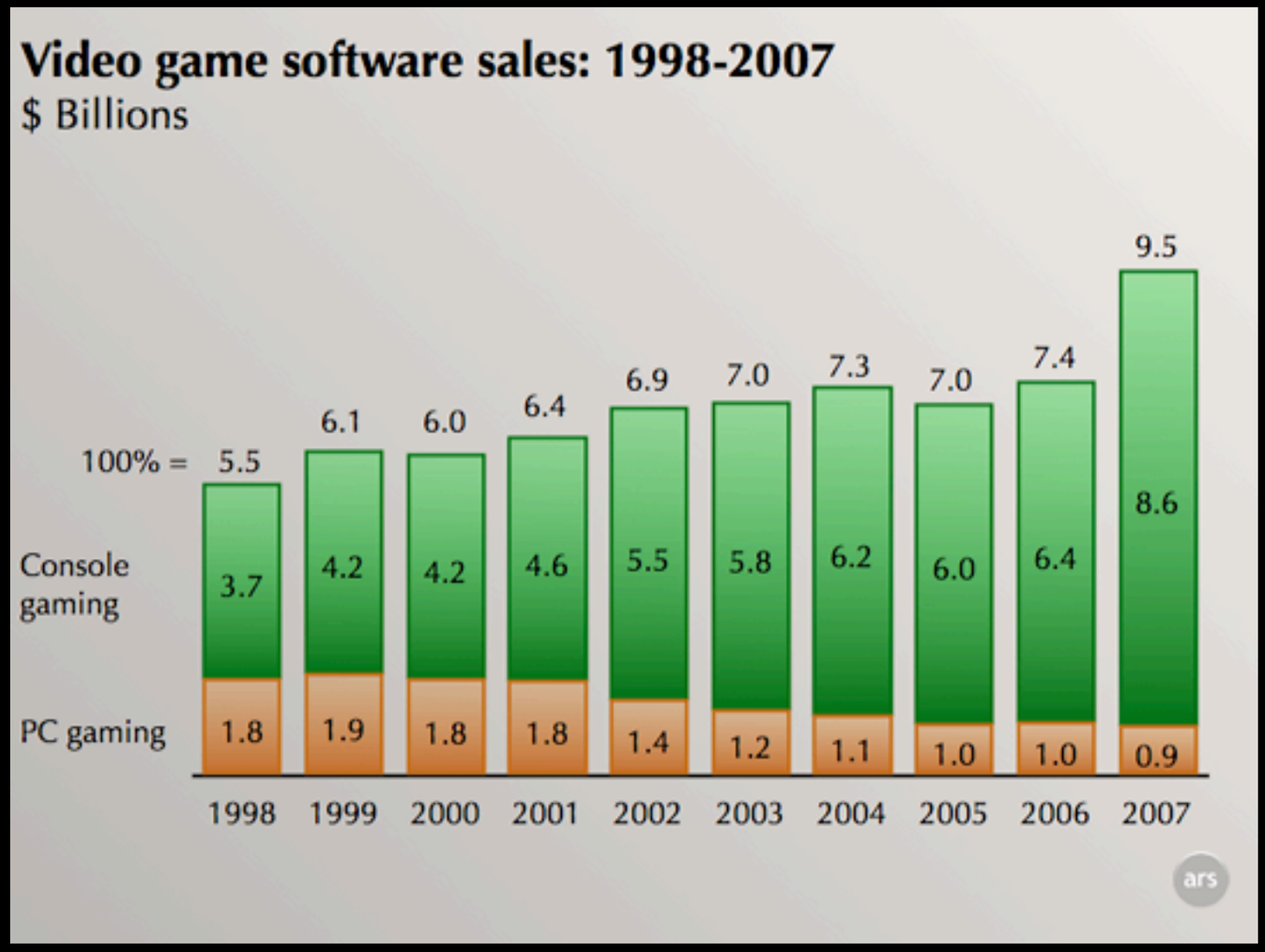
- Design goal massively parallel graphics
- A lot of replicated functional units
- Small cache size
- Eg.: NVIDIA GTX280
  - 240 SP (streaming processors)
  - support for 30720 simultaneous threads

Computer Graphics is a
Computational intensive application



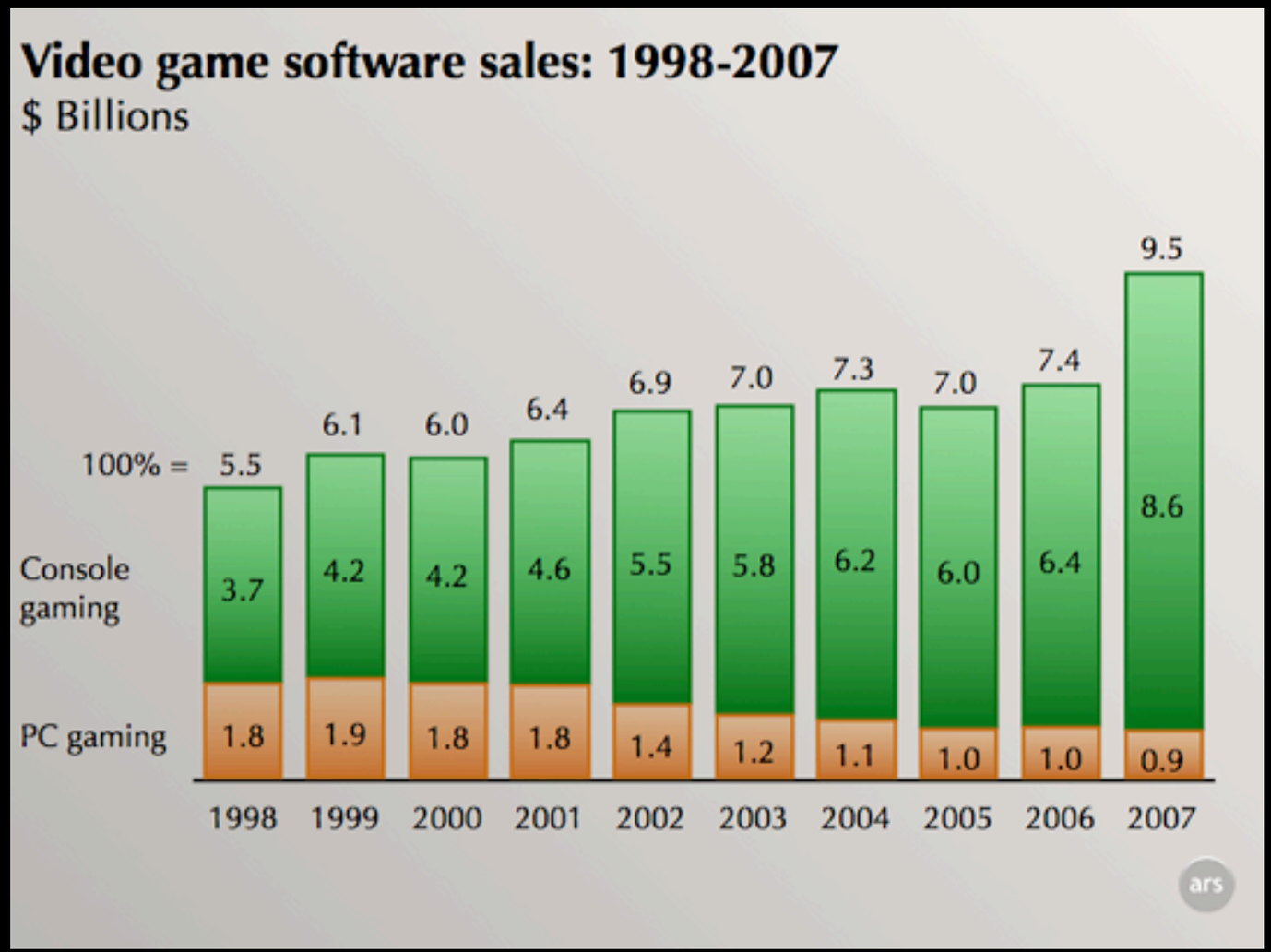
Computer Graphics is a Computational intensive application

A lot of $$$ from game industry



Computer Graphics is a Computational intensive application

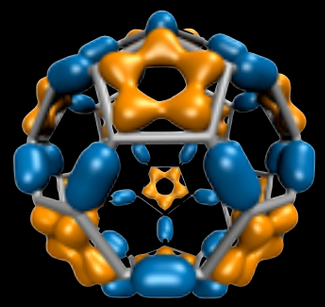
A lot of $$$ from game industry



Expressive gain in performance for parallel graphics rendering

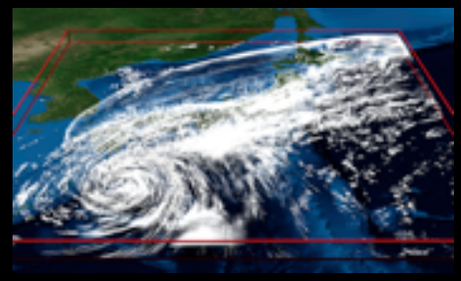
Caught attention from the scientific community

GPU is also adapted to several scientific applications



Molecular Biology

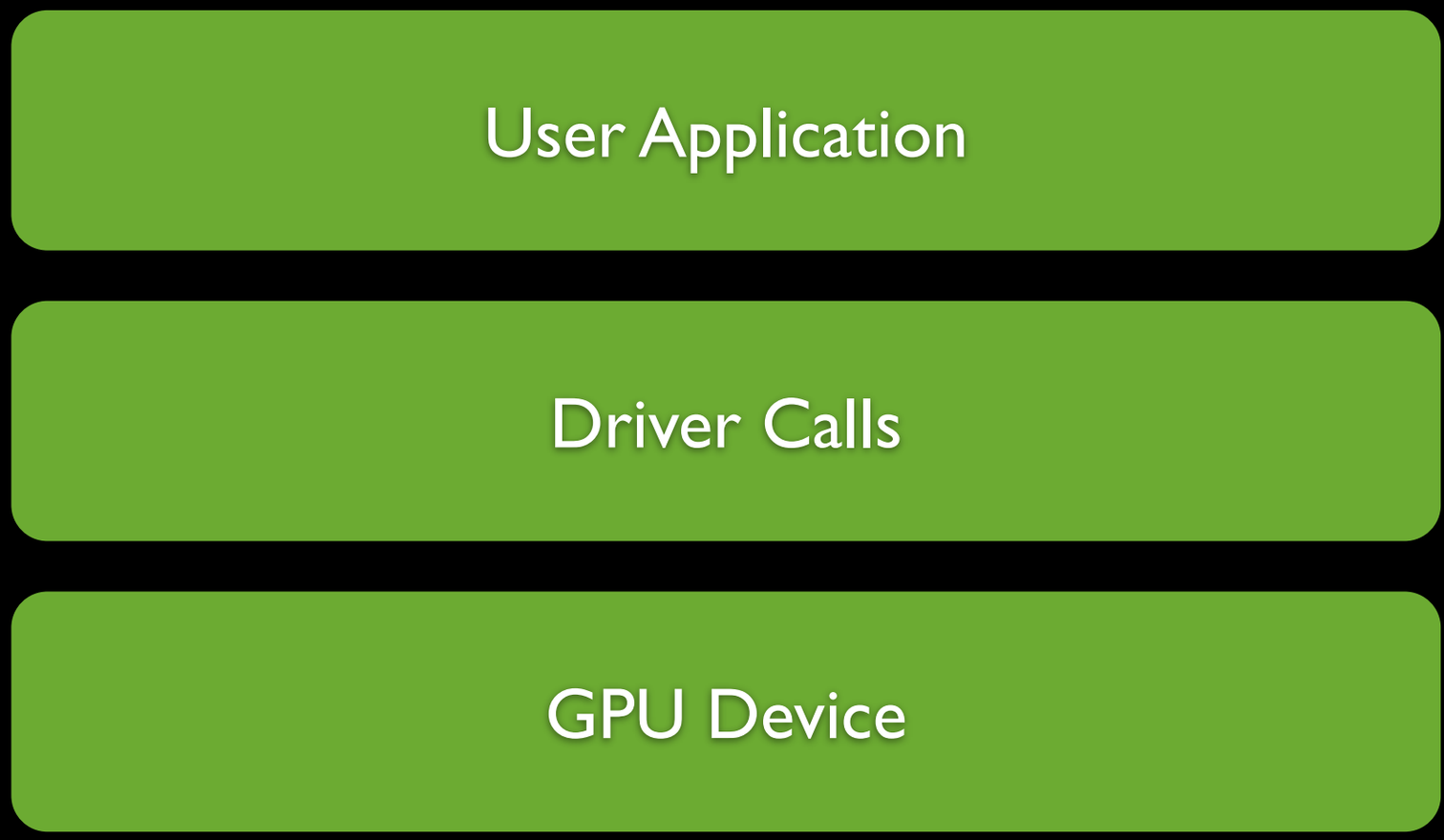
Fluid Simulation



Weather Forecast

GPGPU

Model the application directly using Computing Graphics driver calls



Need to port the problem to a complete different domain

DirectX or OpenGL are not easy to figure out

Potential Gain in Performance

Potential Gain in Performance

100 times faster!

Several guys from Intel

This is a myth!

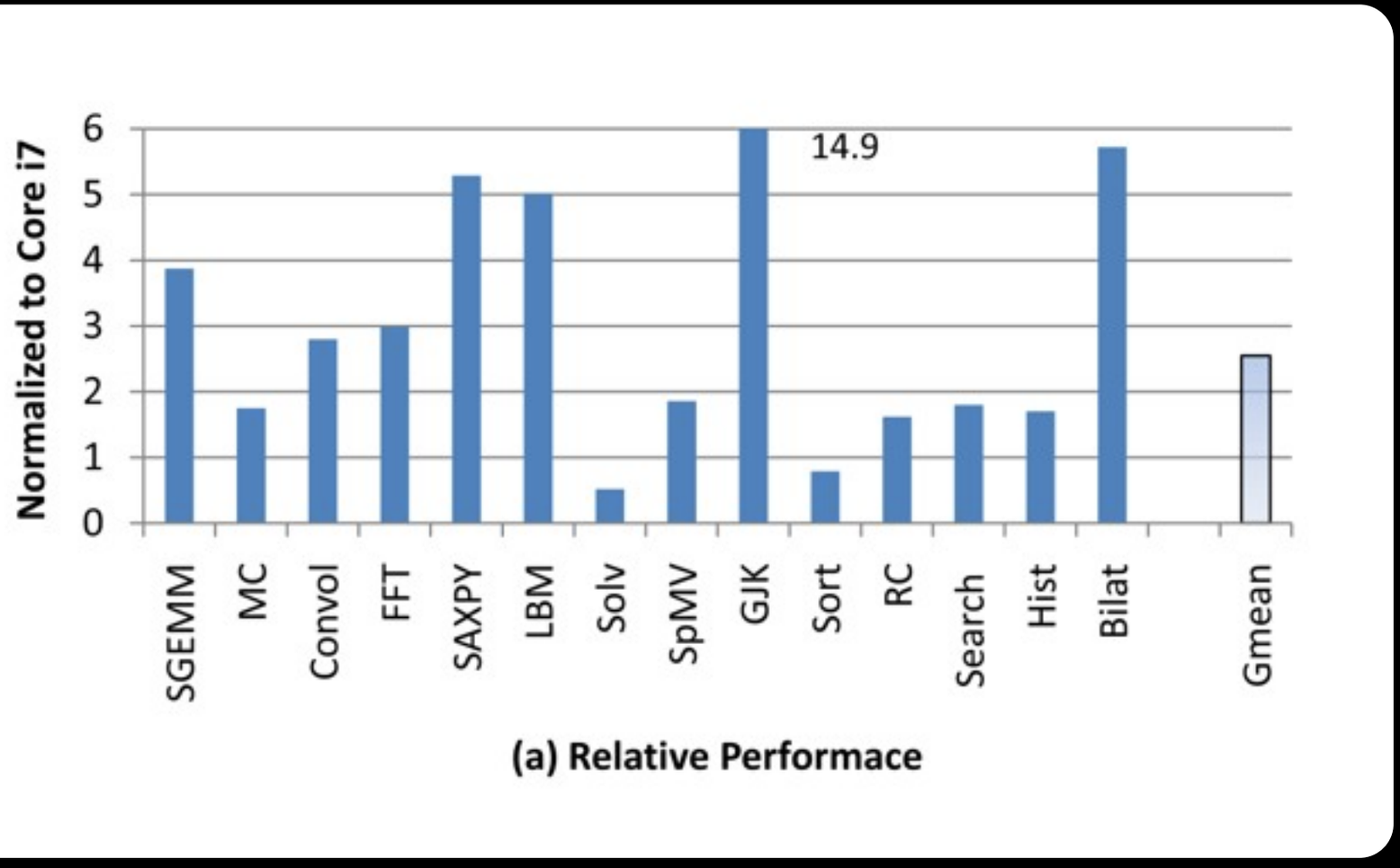
Victor W Lee *et. al.*, Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU

Core i7 - quadcore
vs.
GTX280

14 kernels

relative performance!

Reason:
Rethink your problem is challenging



Potential Gain in Performance

100 times faster!

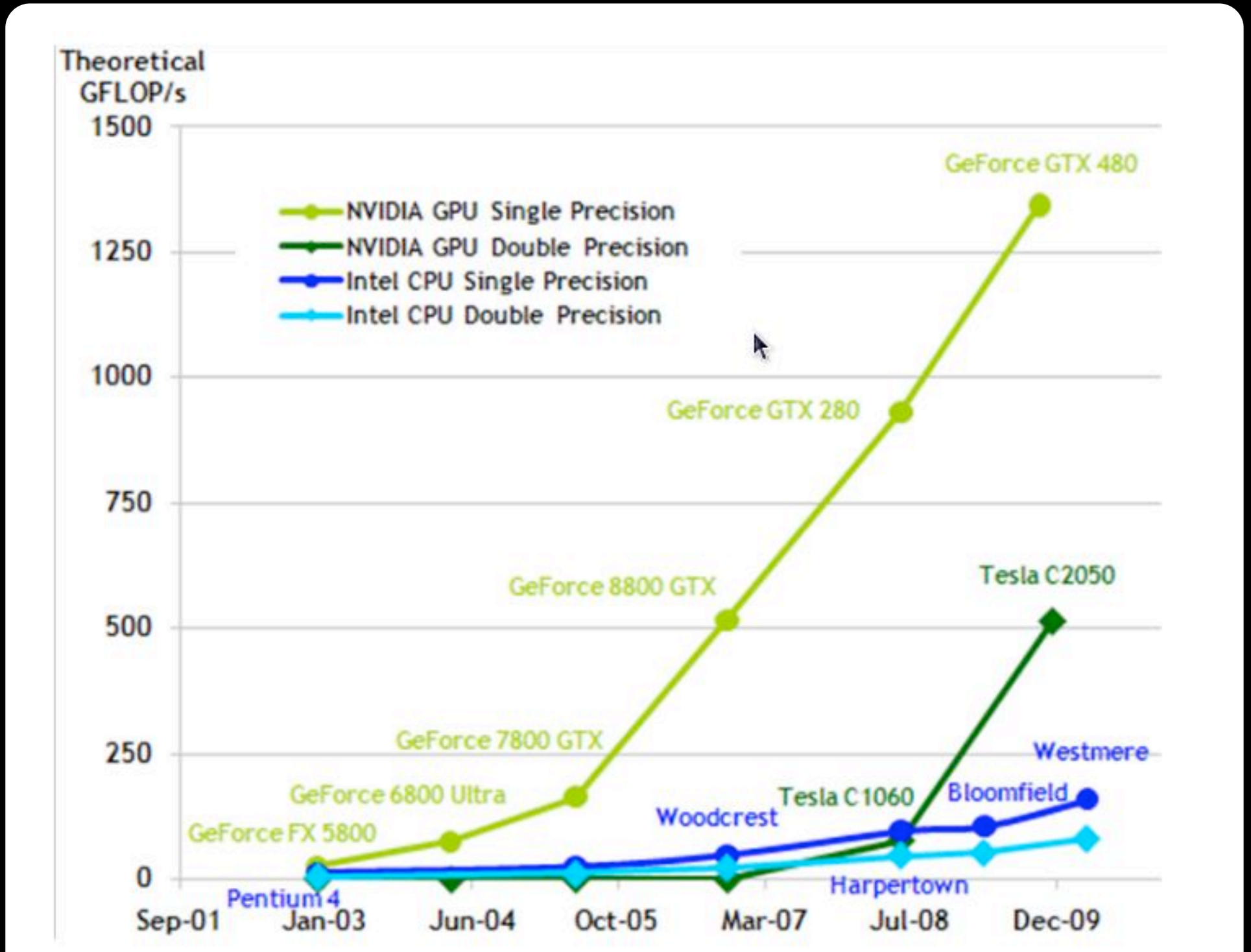
Several guys from Intel

This is a myth!

Victor W Lee *et. al.*, Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU

$$\frac{T_{\text{GTX280}}}{T_{\text{Core i7}}}$$

CPU vs GPU

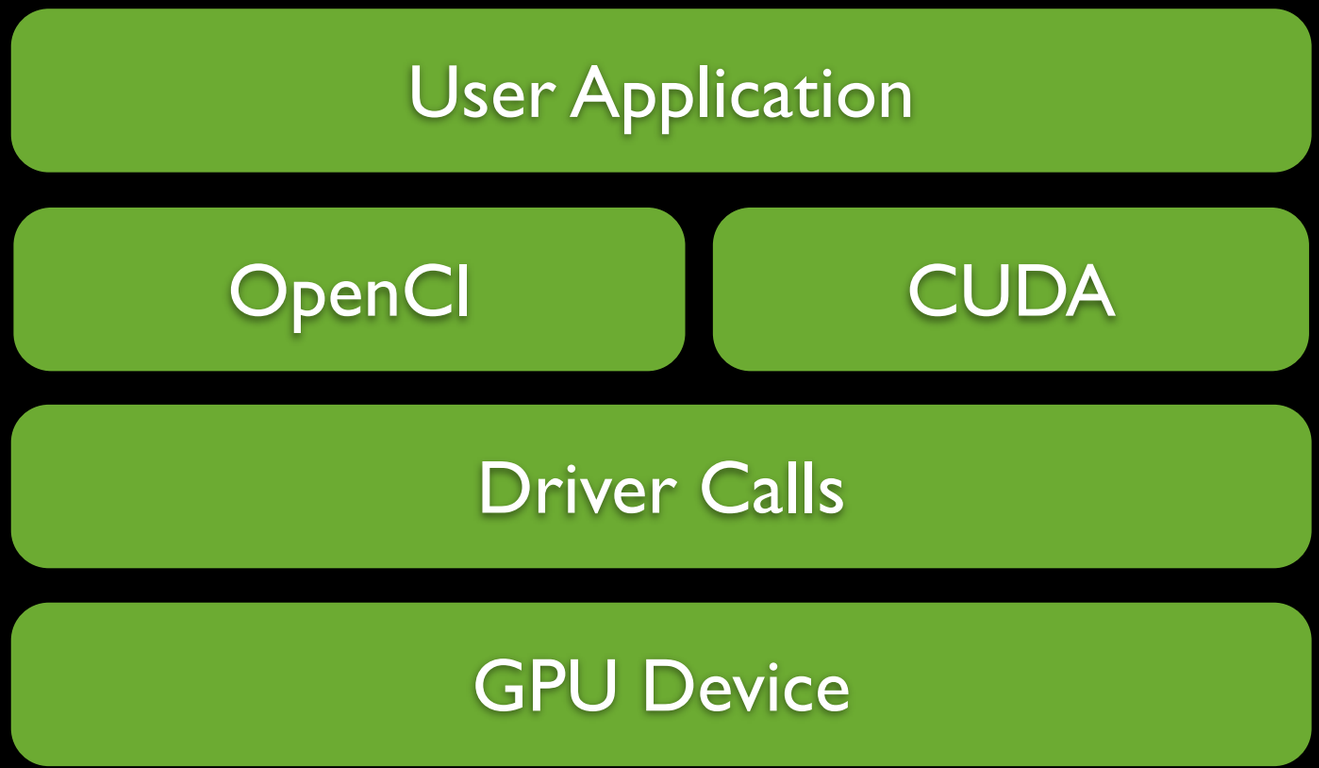

Theoretical
GFLOP/s
1500
1250
1000
750
500
250
0
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla C2050
Tesla C1060
Westmere
Bloomfield
Woodcrest
Harpertown
Pentium 4
Sep-01
Jan-03
Jun-04
Oct-05
Mar-07
Jul-08
Dec-09

Substantial gain in execution time (10x)!

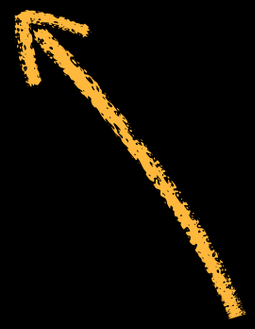
before GPU	**with** GPU
one year	one month plus a week
one day	two hours and twinty four minutes
one hour	six minutes

GPU Programming today

Don't need to port the application to DirectX or OpenGL



- Propretary (only work on NVIDIA)
- Enhanced software support
- Several software libraries and examples



CUDA vs OpenCl

- Open specification
- Work on NVIDIA and ATI video cards
- Aim at any computing device

Introduction
from **games** to **science**

1

2 **Architecture**

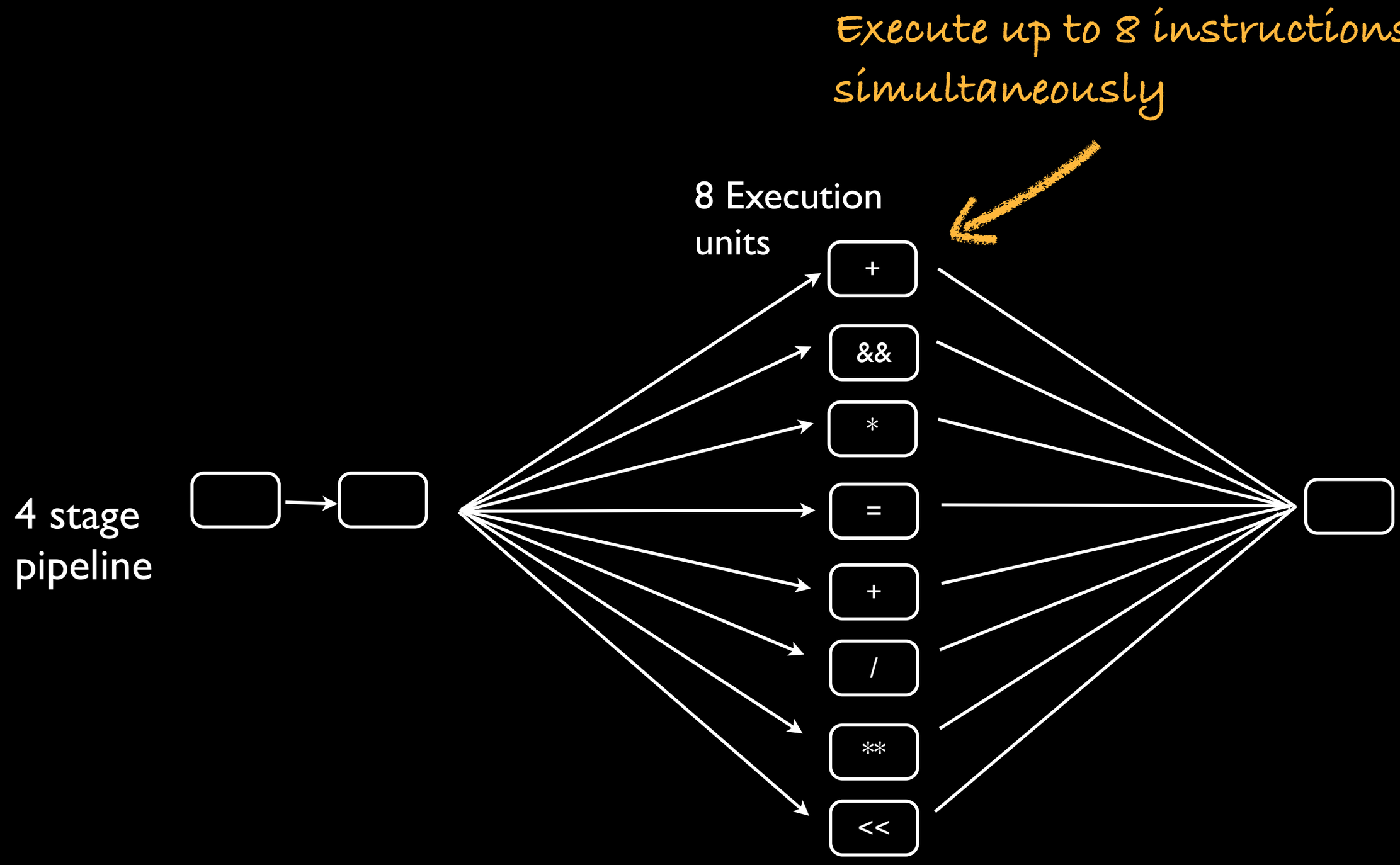
3 **GPU** Programming

CUDA 4

Final Remarks 5

Previously on parallel architectures

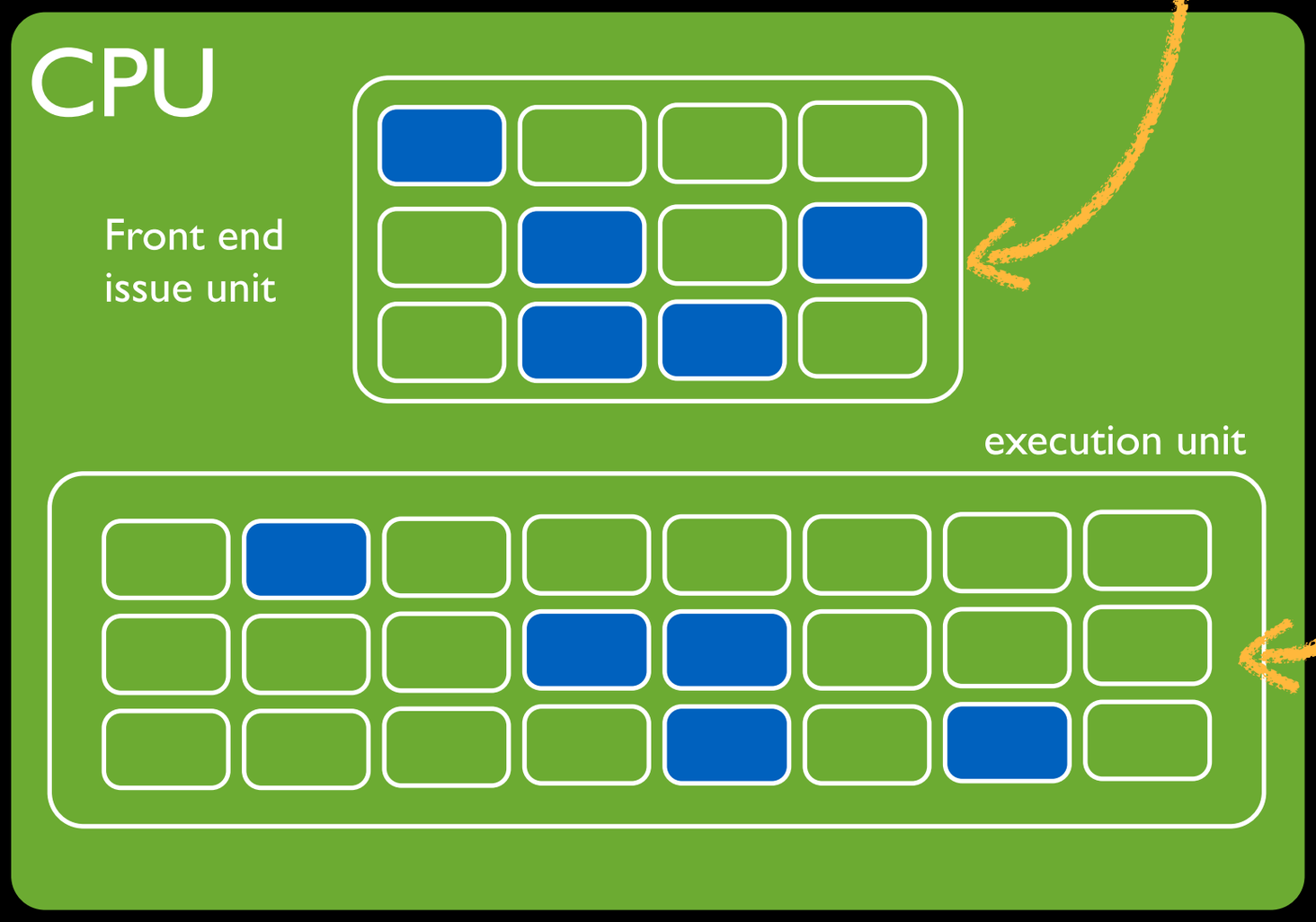
Superscalar processors



Superscalar processors make the illusion of concurrent execution

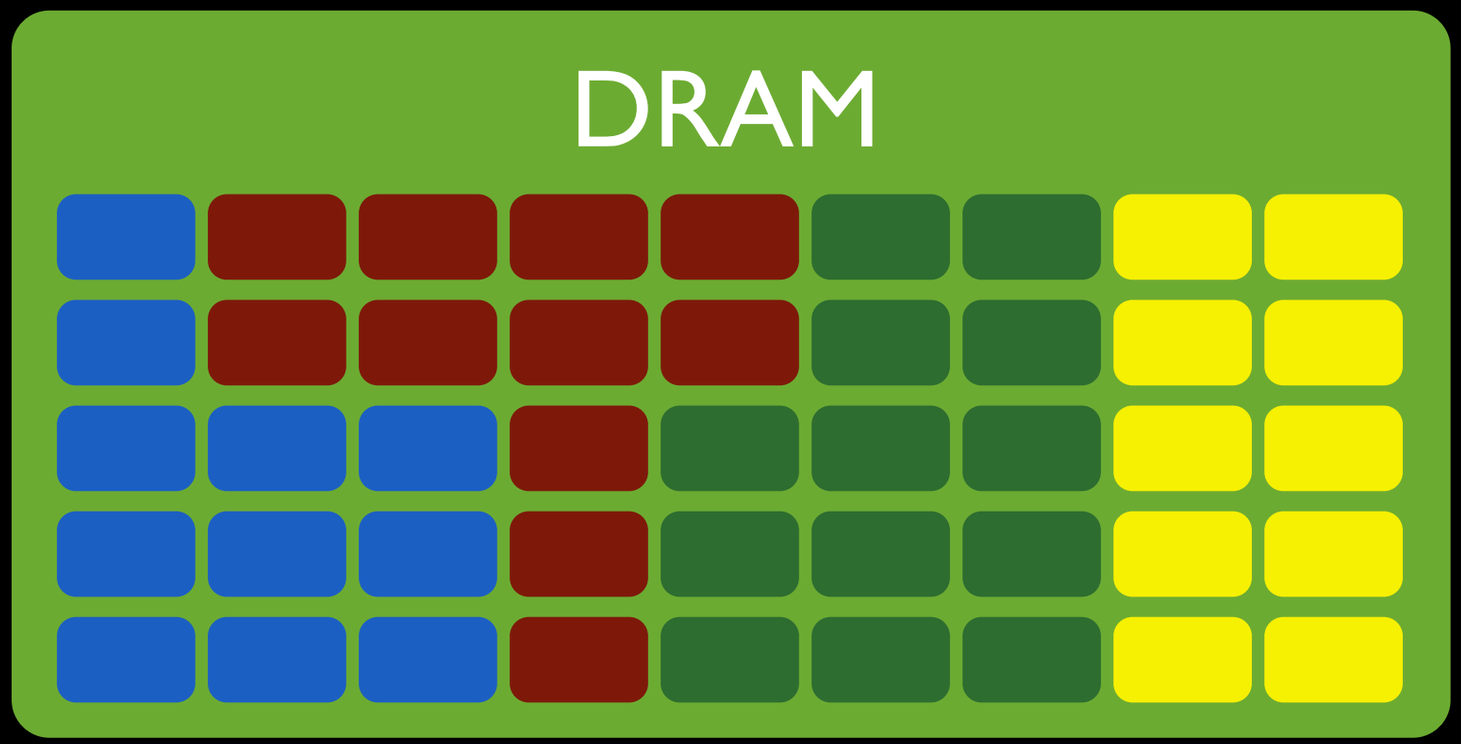
Instruction from one thread arrive

A hardware issue unit decides which instructions can execute simultaneously



waste due to instruction dependecy (bubbles)

A program has instructions for several threads in memory



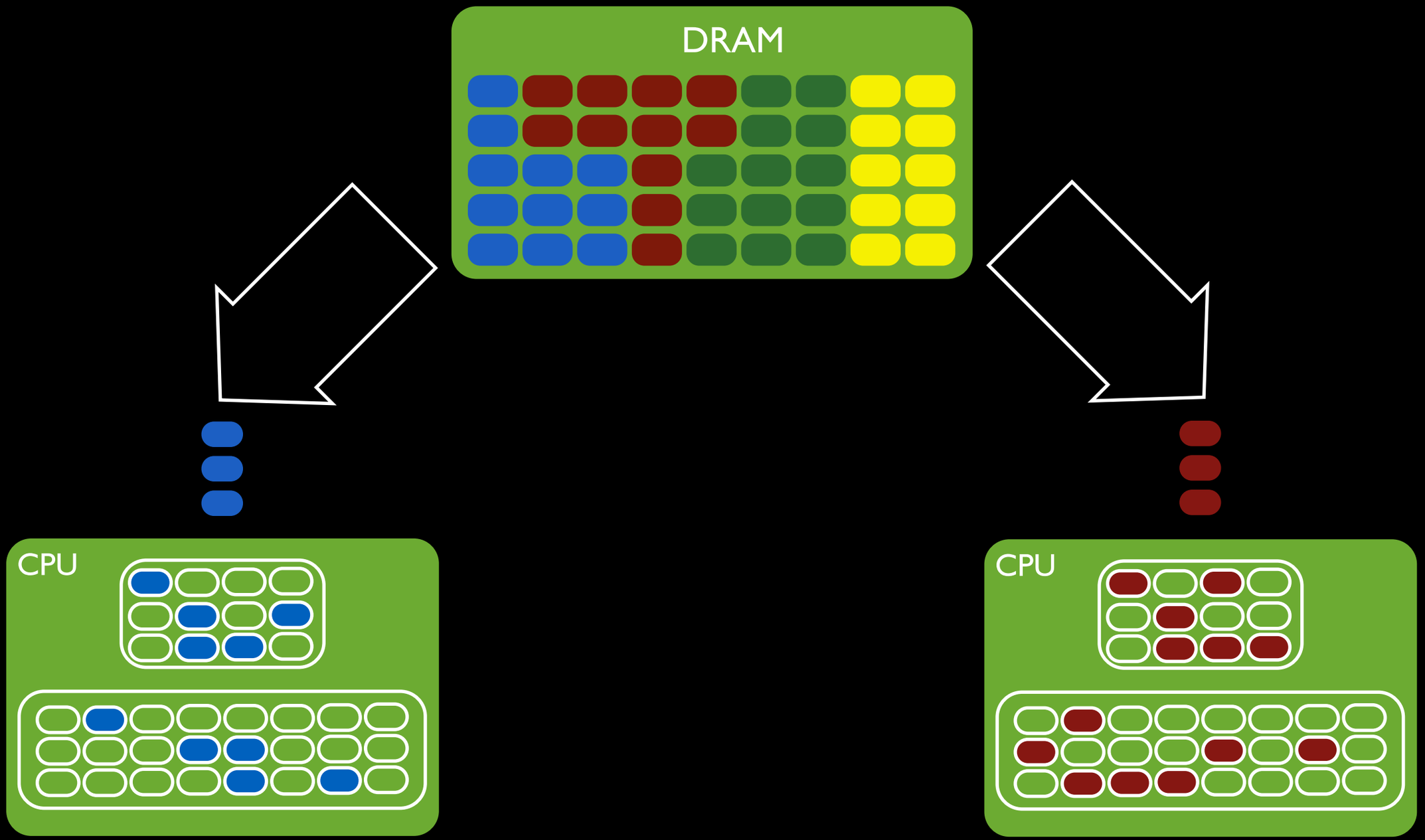
blue thread

red thread

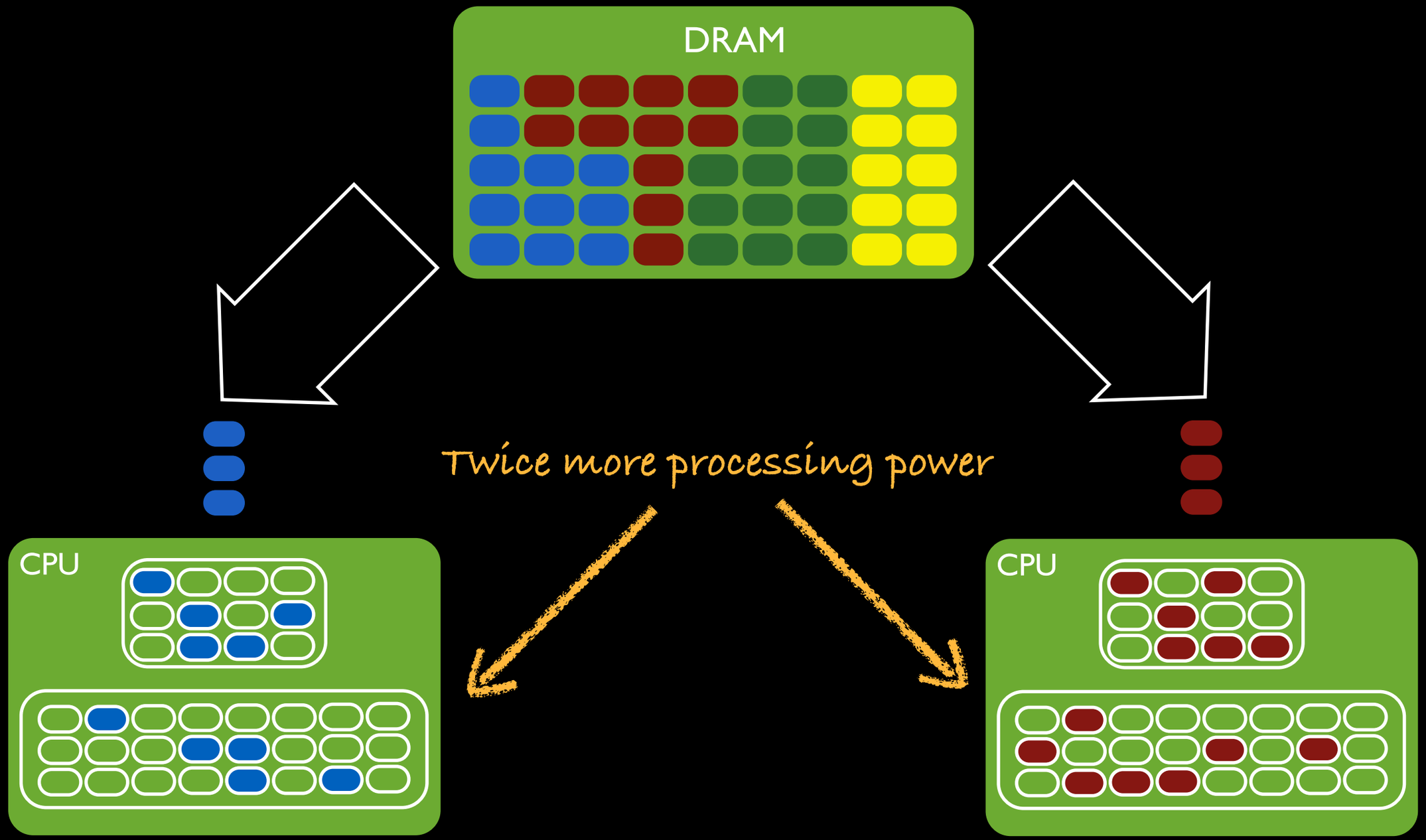
green thread

yellow thread

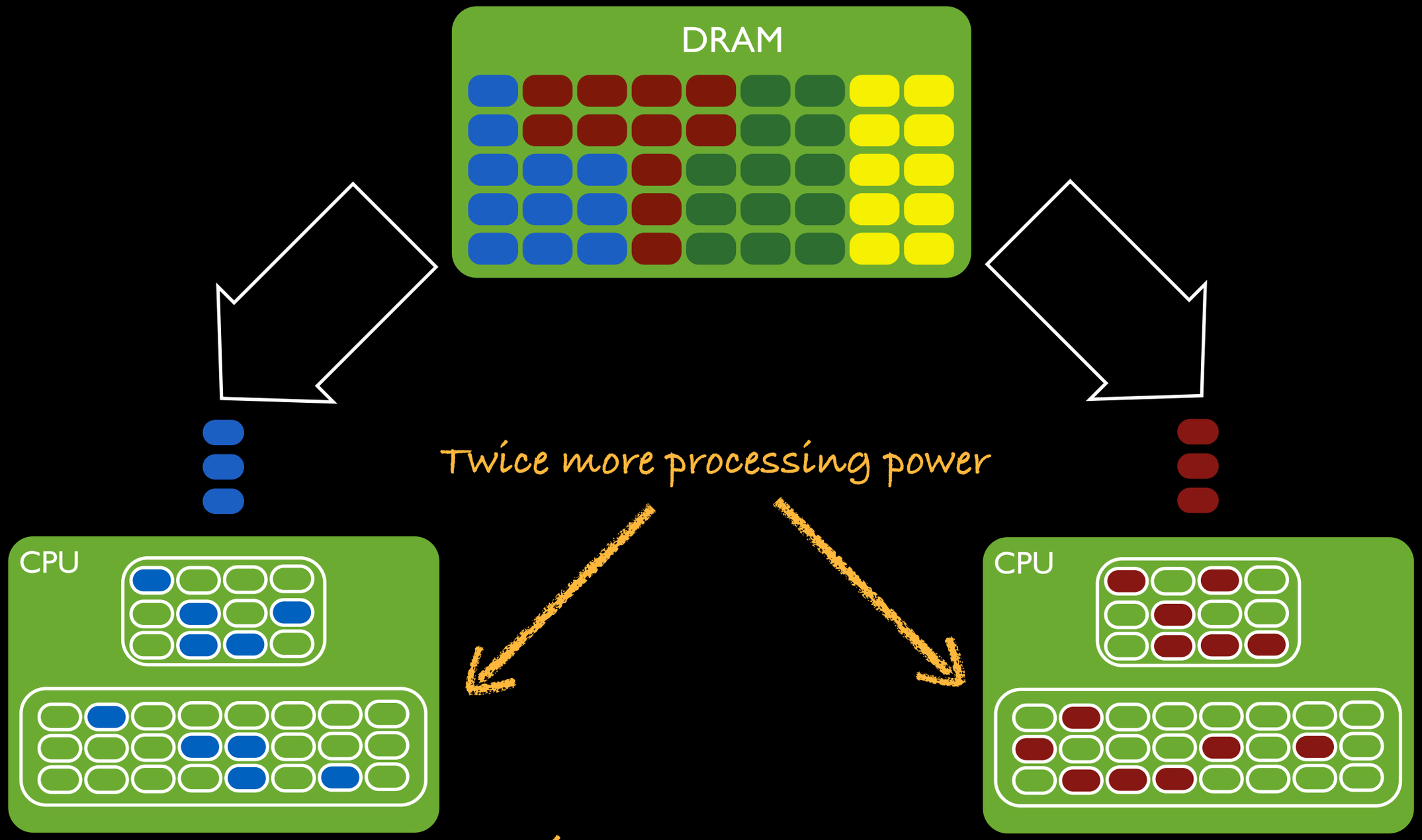
Single threaded multicore



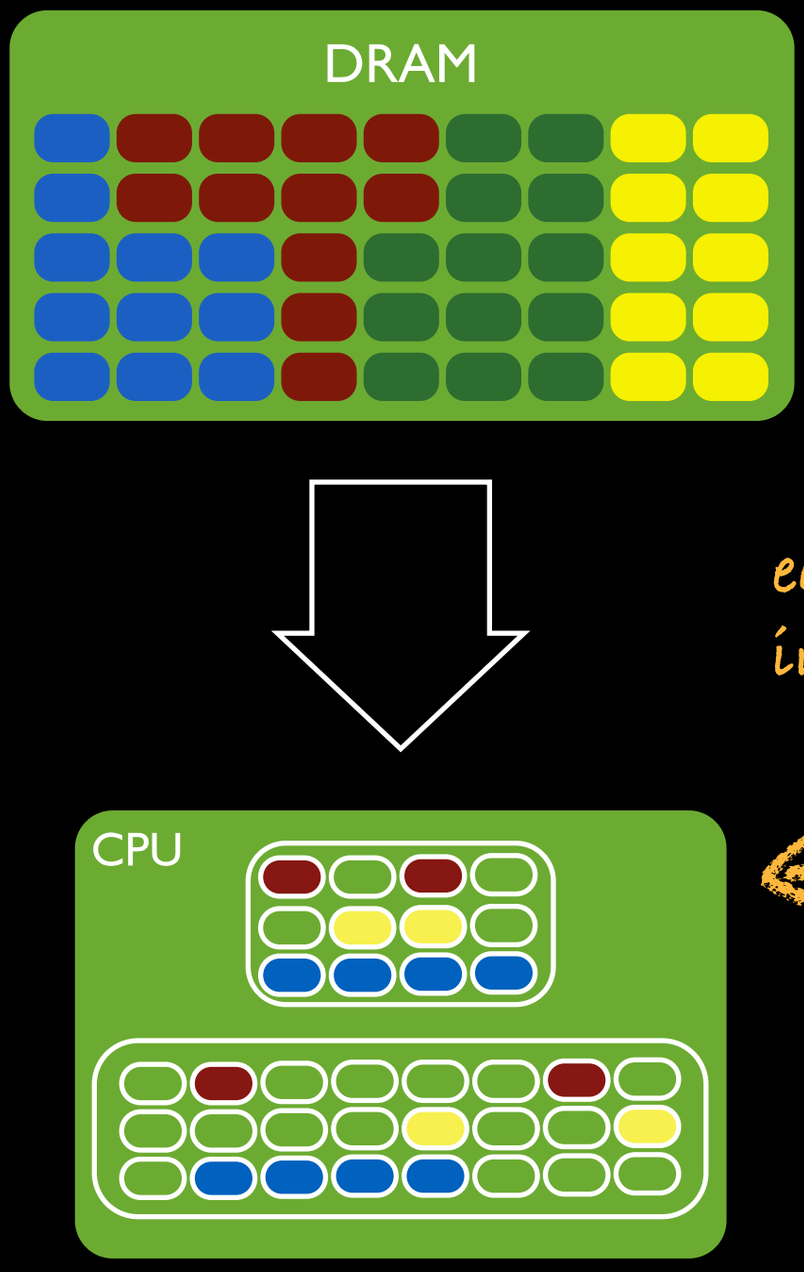
Single threaded multicore



Single threaded multicore

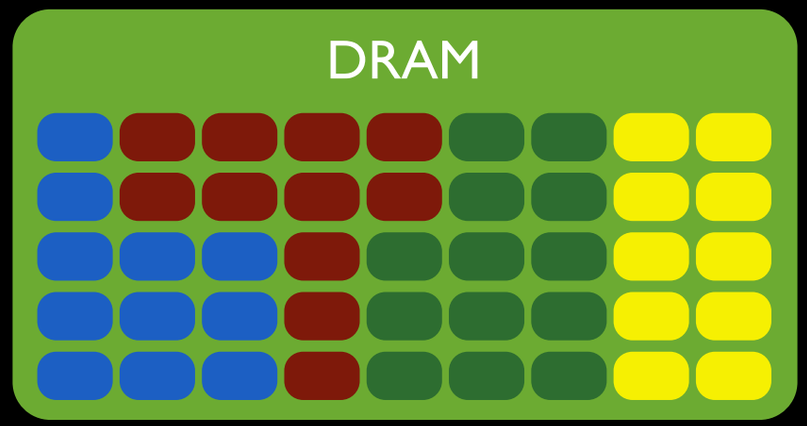


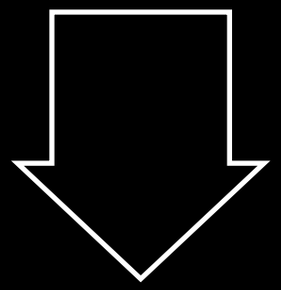
Super-threadeding



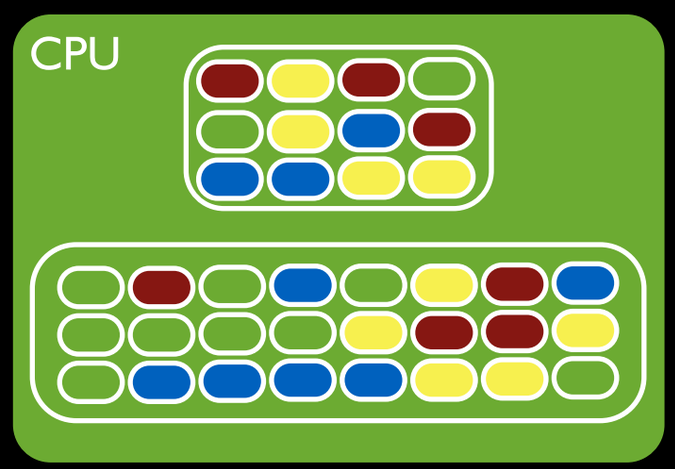
each stage only run one instruction from one thread

Multi-threadeding



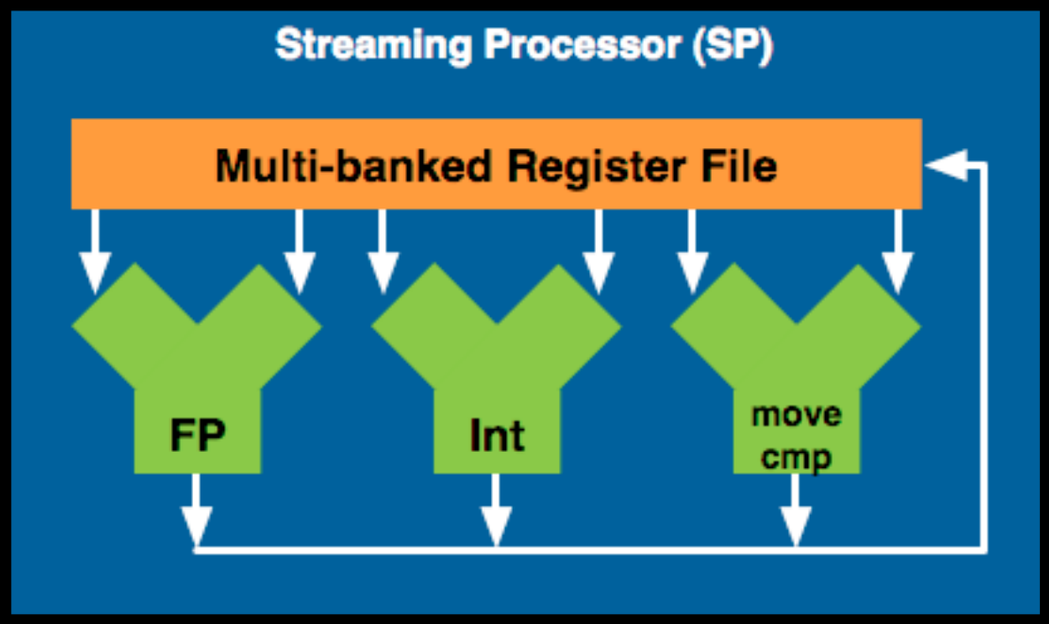


Can execute instruction from more than 1 thread at a time

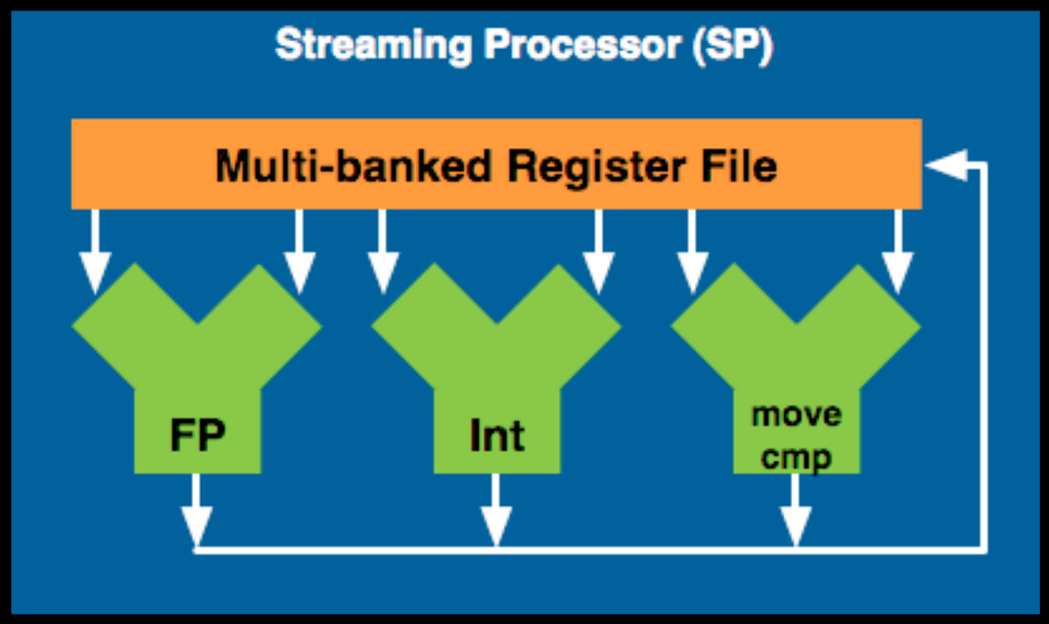


Back to GPU architecture

Streaming Processor (SP)

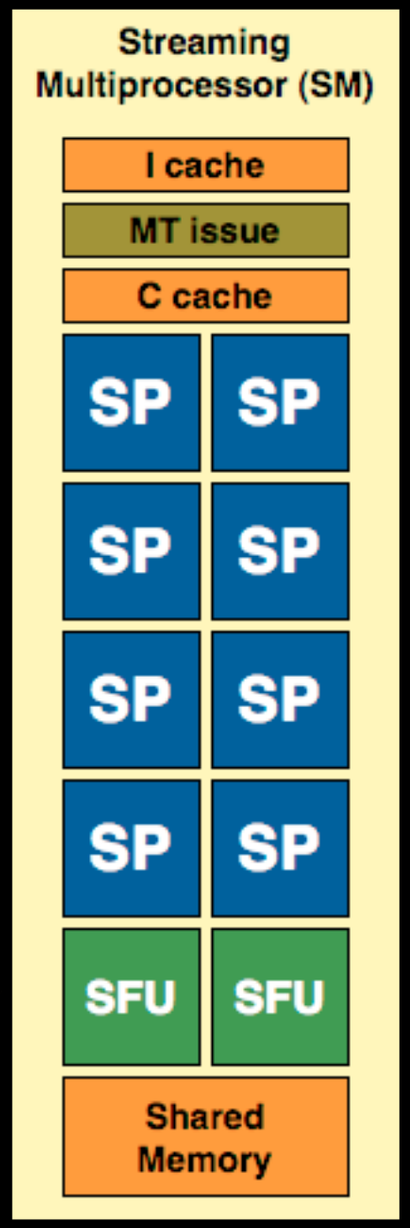


Streaming Processor (SP)



Cacheless
Pipilened
Single issue

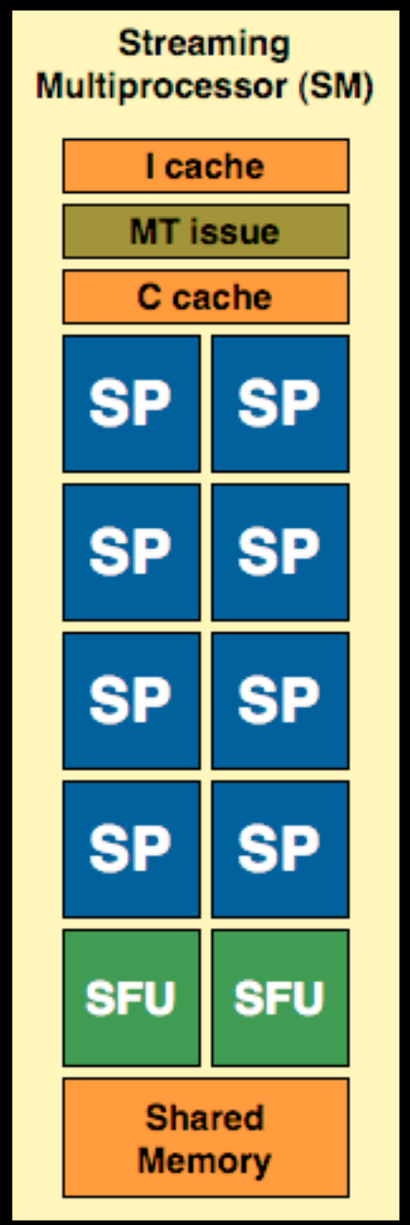
Streaming Multiprocessor (SM)



Each SFU
4 FP multiply
for sin, cosin

Array of 8 (eight) SPs

Streaming Multiprocessor (SM)



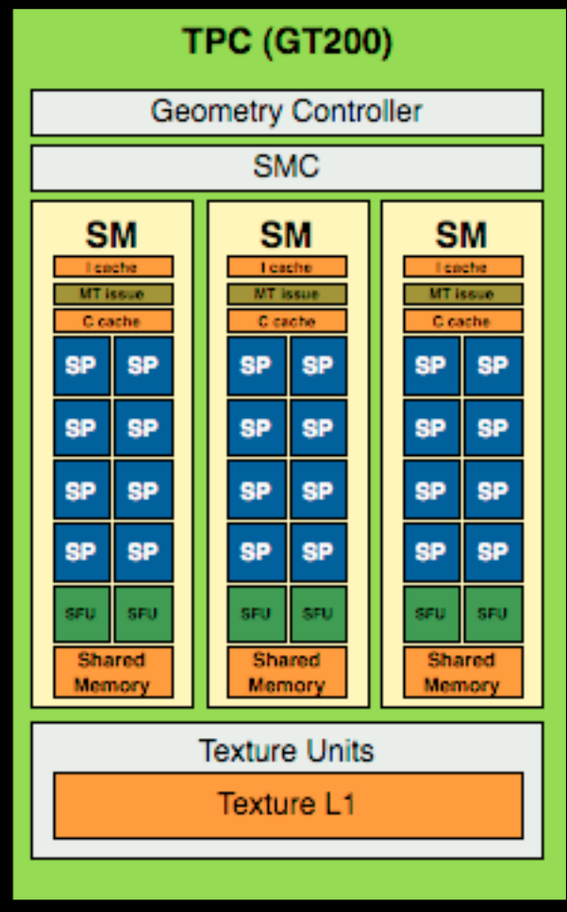
Each SFU
4 FP multiply
for sin, cosin

Multi-threaded
can issue several instructions

Array of 8 (eight) SPs

GPU Architecture (GT200)

Texture Processor Cluster
3 SM's



GPU Architecture (GT200)

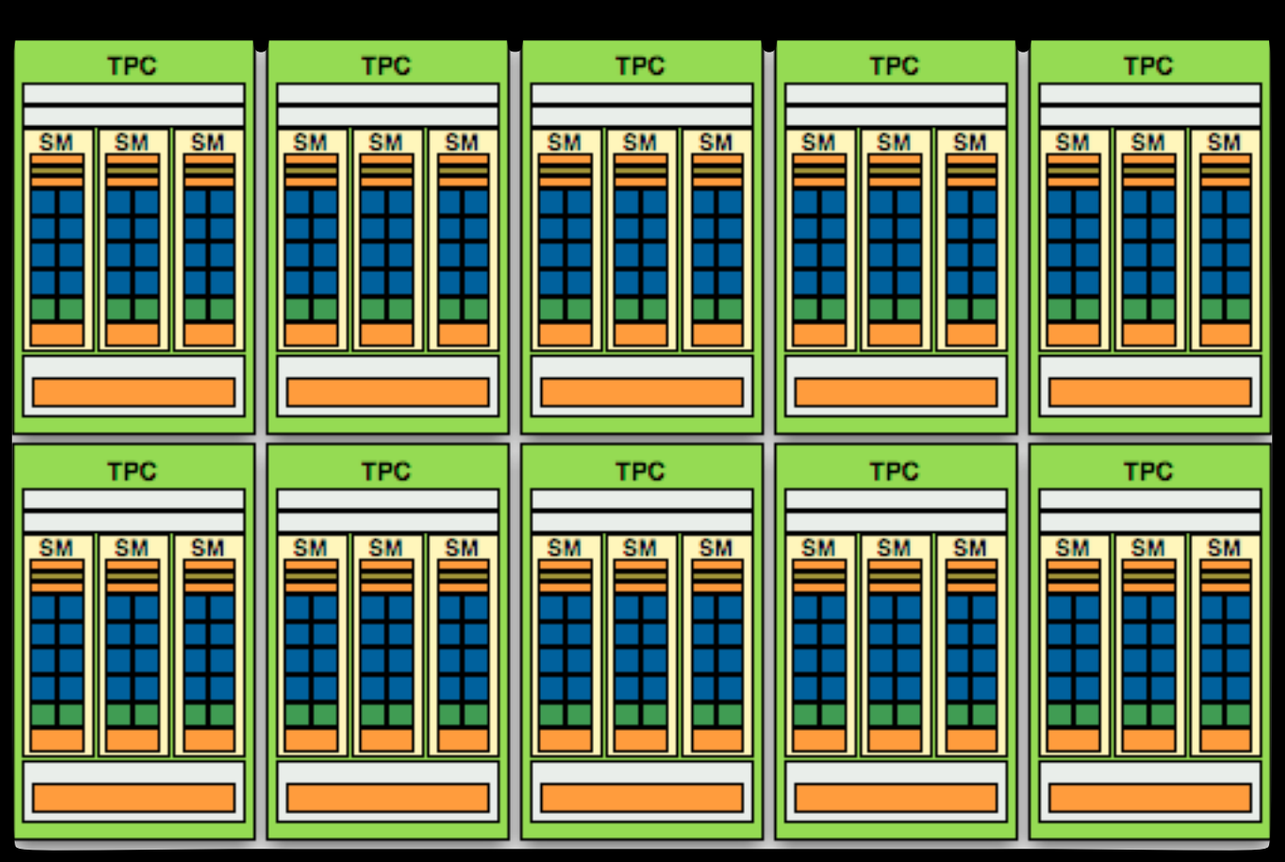
The beast

10 TPC's

3 SM's per TPC

8 SP's per SM

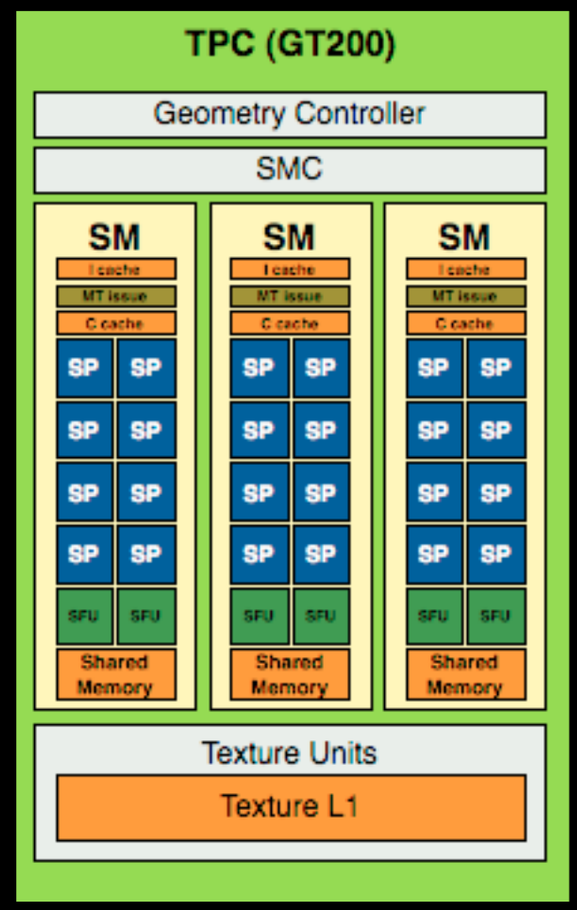
Total of 240 SP's



GPU Architecture (GT200)

very small cache

To hide memory latency need several threads available per SM

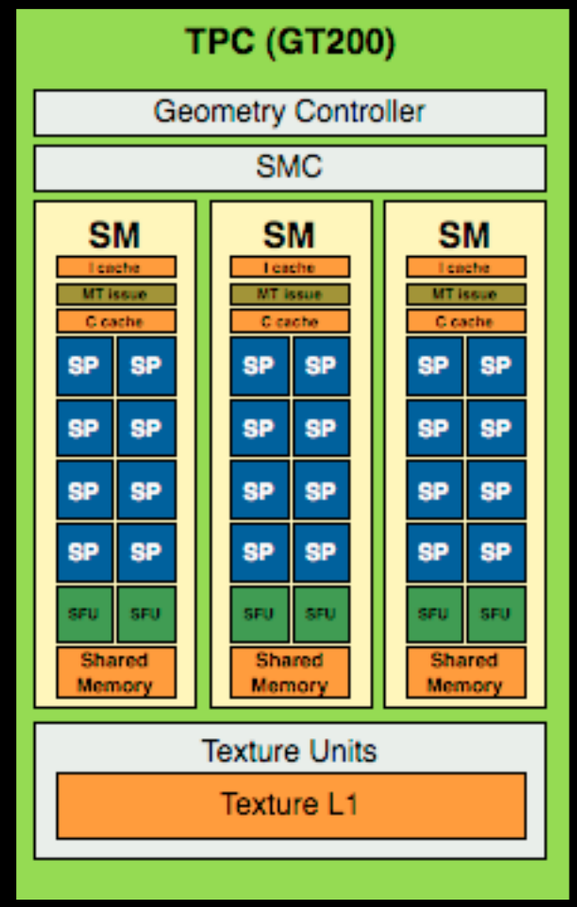


Schedule per group of 32 threads, called a **warp**

GPU Architecture (GT200)

Each SM handles 32 **warps** simultaneously

32 x 32 = 1024 threads per SM



1024 x 30 = 30720 simultaneous threads

Introduction
from **games** to **science**

1

2 **Architecture**

3 **GPU** Programming

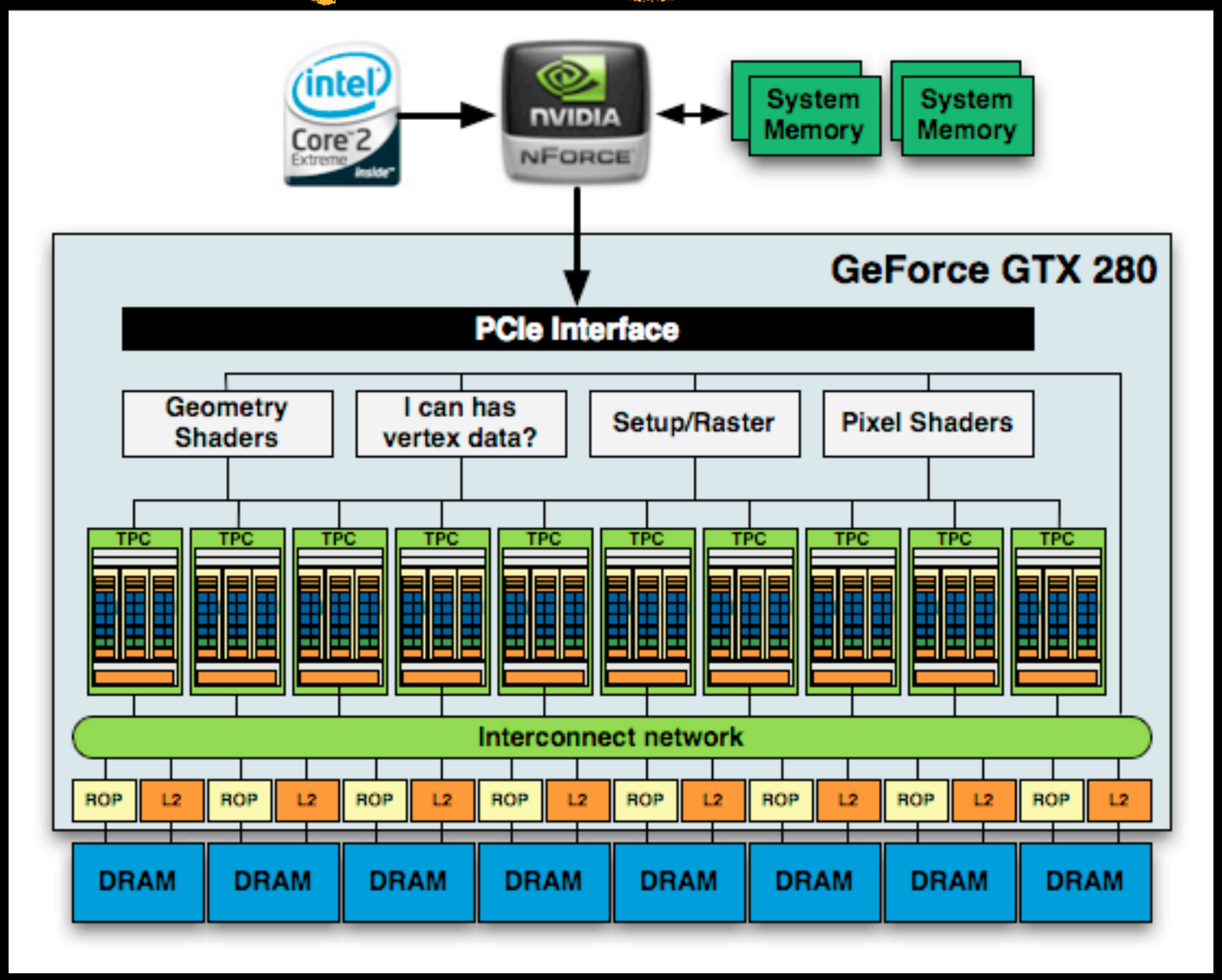
CUDA 4

Final Remarks 5

GPU Programming

CPU is the HOST Processor

GPU is a co-processor



GPU has its own DRAM

GPU Programming

Massively parallel processor (GT200 - 30720 Threads)

- CPU send burst of threads to execute on the GPU

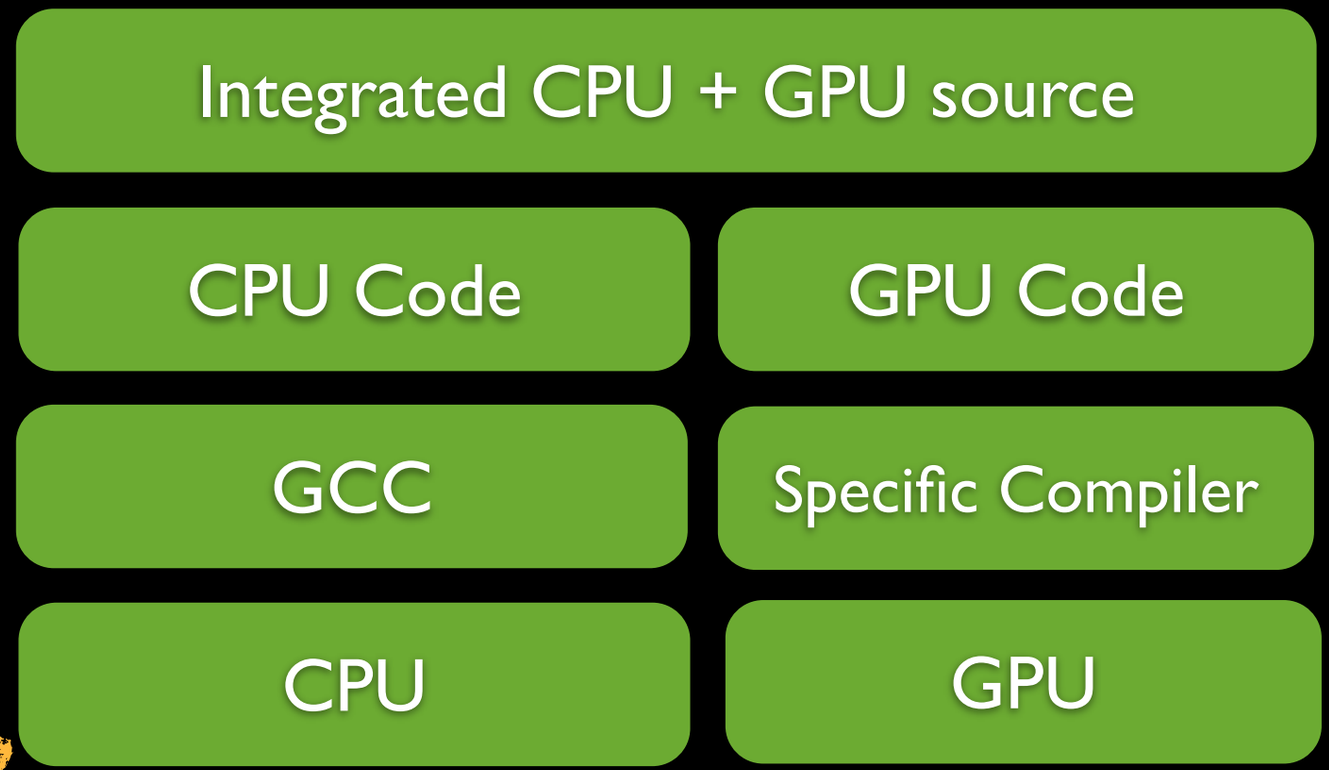
Use DMA to transfer from CPU DRAM to GPU DRAM

CPU becomes free after dispatching the threads

- Can do something useful meanwhile

Applications must be rewritten to cope with GPU

GPU Programming



Same code can run on both devices CPU and GPU

GPU Programming

Single threaded application

```
int a[1024];
int b[1024];
int c[1024];

int main()
{
    for(int i=0; i<1024; i++){
        c[i] = a[i] + b[i];
    }
}
```

Multi threaded application

```
int main()
{
    int i = thread.id;
    c[i] = a[i] + b[i];
}
```

Need to instantiate 1024 threads

GT200 supports up to 30720 threads simultaneously!!!

GPU Programming

Definition of a single thread computing function (or **kernel**)

```
int kernel()
{
    int i = thread.id;
    a[i] = a[i] + b[i];
}
```

1- How to Compute the thread ID?

2- How do we copy data from CPU to GPU?

3- How to dispatch kernel on the device?

4- How to get results back when done?

GPU Programming

Have support for operations on the Host (CPU) and Device (GPU)

1- Copy data from Host to Device
2- Execute kernel on the device
3- Wait for kernel to finish
4- Copy data from Device to Host

Depends on the programming interface

mallocDeviceMemory
copyFromHostToDevice
computeKernel
copyFromDeviceToHost

Introduction
from **games** to **science**

1

2 **Architecture**

3 **GPU** Programming

CUDA 4

Final Remarks 5

CUDA Programming

- C extension

- Support for several platforms:
  - Linux
  - Windows
  - MacOS

- Need to install NVIDIA Driver, Toolbox and SDK

CUDA Programming

Provide several libraries

STL C++ Port to CUDA

Linear Algebra
cuBLAS

CUDA Programming

Requirements for Linux

- 1 NVIDIA CUDA aware card
- GCC installed
- Downloaded Toolkit, Driver, and SDK

Step-by-step installation:

- Exit GUI (Ctrl+Alt+Backspace)

- Install the CUDA Toolkit
  $ ./cudatoolkit_4.2.9_linux_64_ubuntu11.04.run

MOSTRAR SITE

- Install the driver
  $ sudo ./devdriver_4.2_linux_64_295.41.run

- Restart GUI
  $ sudo /etc/init.d/gdm start

- Install SDK
  $ ./gpucomputingsdk_4.2.9_linux.run

Only the driver requires superuser priviledges

Exploring SDK demos

CUDA Programming

SDK has many toy applications:

```
$ cd $HOME/NVIDIA_CUDA_SDK

$ make

$ make check

$ C/bin/linux/release/programName
```

CUDA Programming

Example

```
...
float *aHost, *bHost, *cHost;
...
__global__ void kernel(float *a, float *b, float *c){
    int i = threadidx.x;
    c[i] = a[i] + b[i];
}

int main(){
    float *aDev, *bDev, *cDev;

    cudaMalloc(void *aDev, 1024 * sizeof(float));
    cudaMemcpy(aDev, aHost, 1024 * sizeof(float));

    cudaMalloc(void *bDev, 1024 * sizeof(float));
    cudaMemcpy(bDev, bHost, 1024 * sizeof(float));

    kernel<<<1,1024>>> (aDev, bDev, cDev);

    cudaFree(aDev); cudaFree(bDev); cudaFree(cDev);
}
```

Exercise Zero

Device query.

CUDA Programming API

Function directives

Kernel function has strict properties
- must return void
- no static variables
- no recurrency
- no variable arguments

	Execute on	Called from
__device__ float DeviceFunc(...)	device	device
__global__ void kernelFunc(...)	device	host
__host__ float HostFunc(...)	host	host

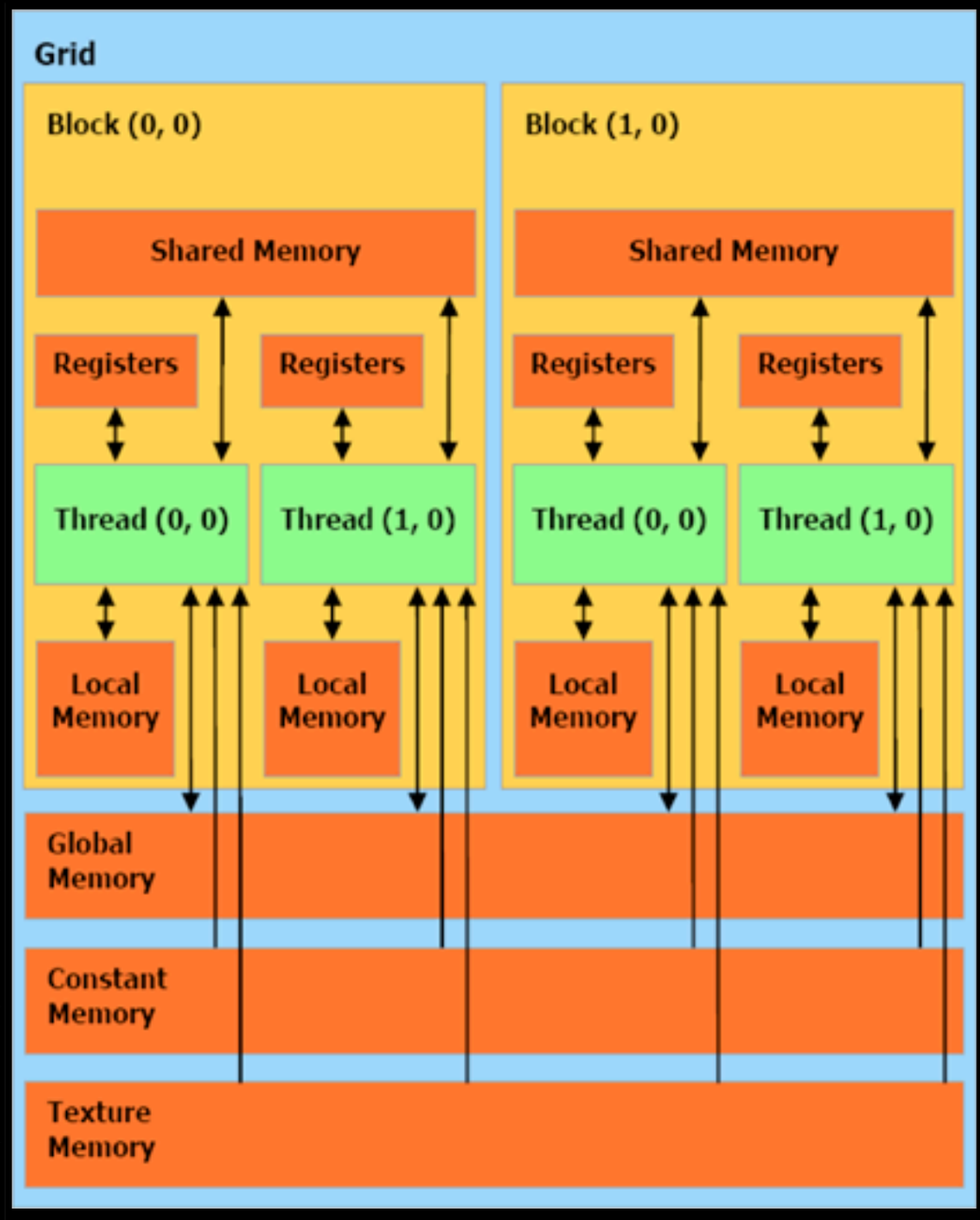
Can be used combined with __device__

CUDA Programming API

Memory allocation

cudaMalloc(...)

Allocate global memory
2 parameters:
Pointer
Number of bytes



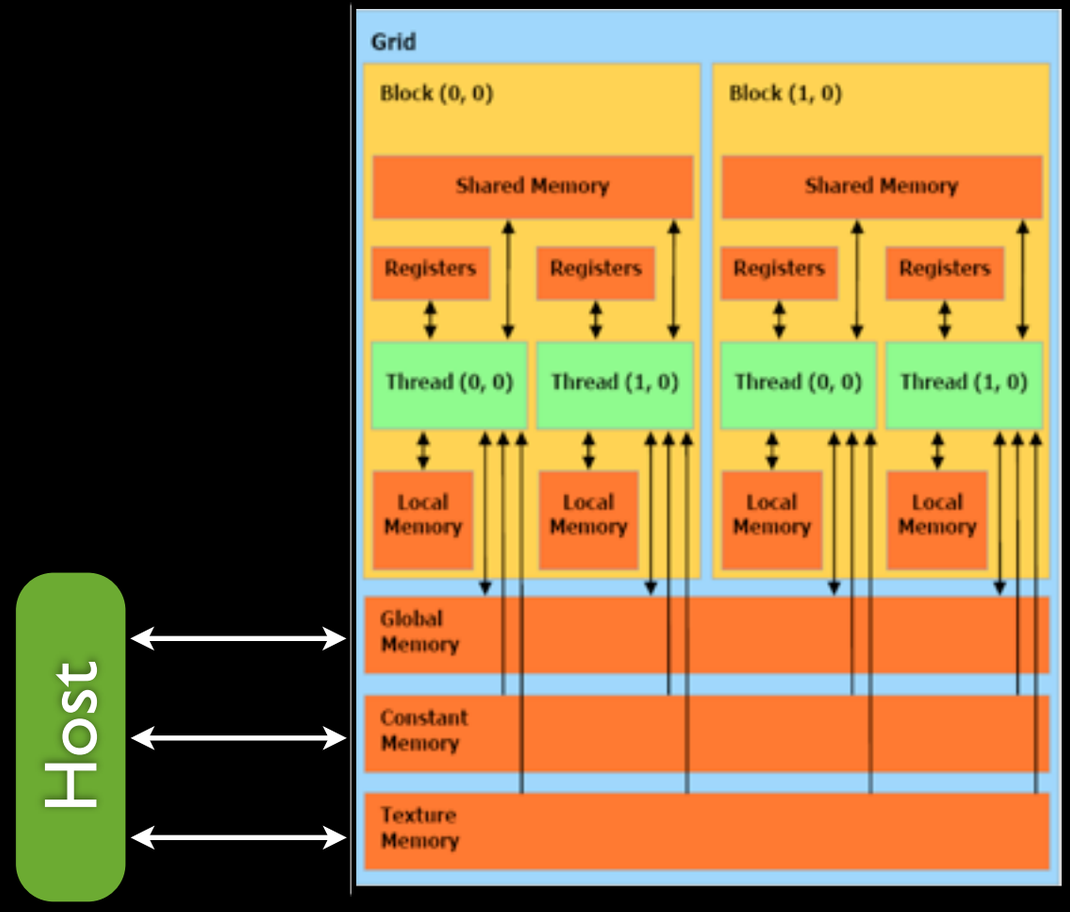
CUDA Programming API

Transfer data

cudaMemcpy(...)

4 parameters:
Source pointer
Destination pointer
Bytes to copy
Transfer type

HostToHost
HostToDevice
DeviceToHost
DeviceToDevice



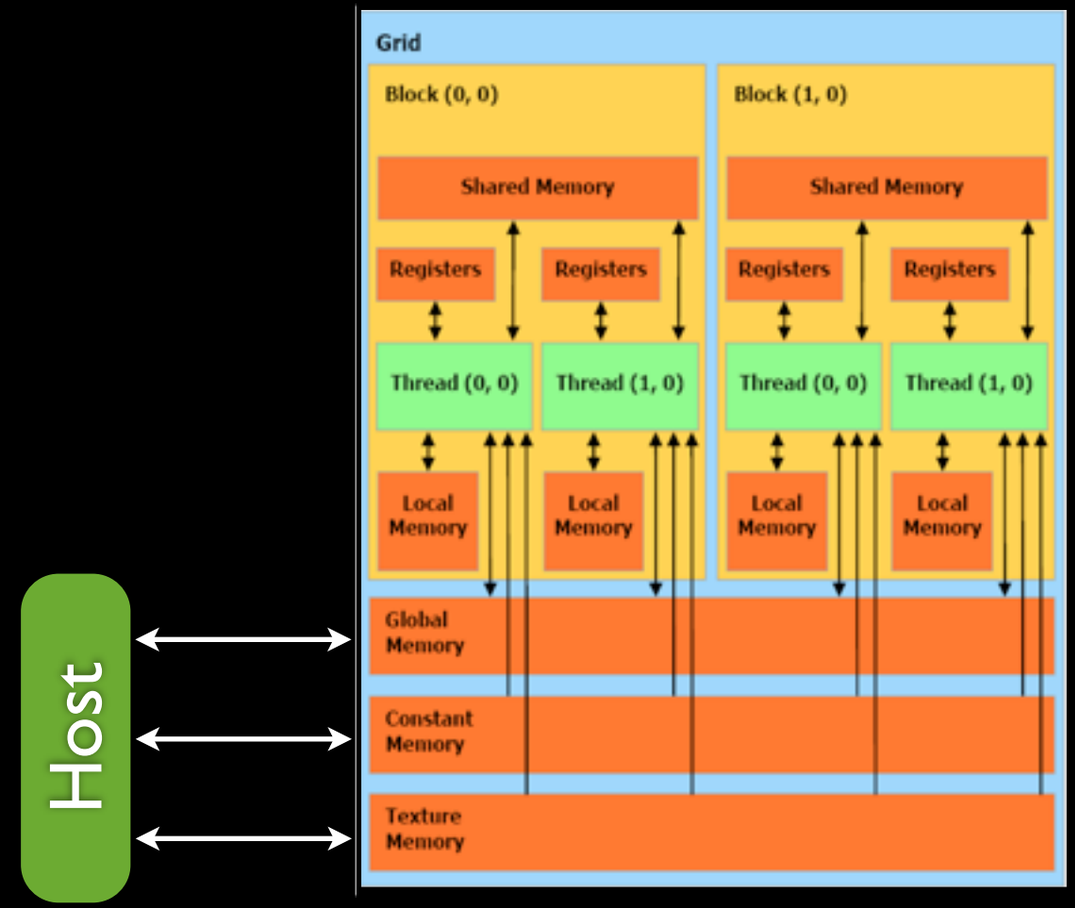
CUDA Programming API

Memory allocation

cudaFree(...)

Frees global memory

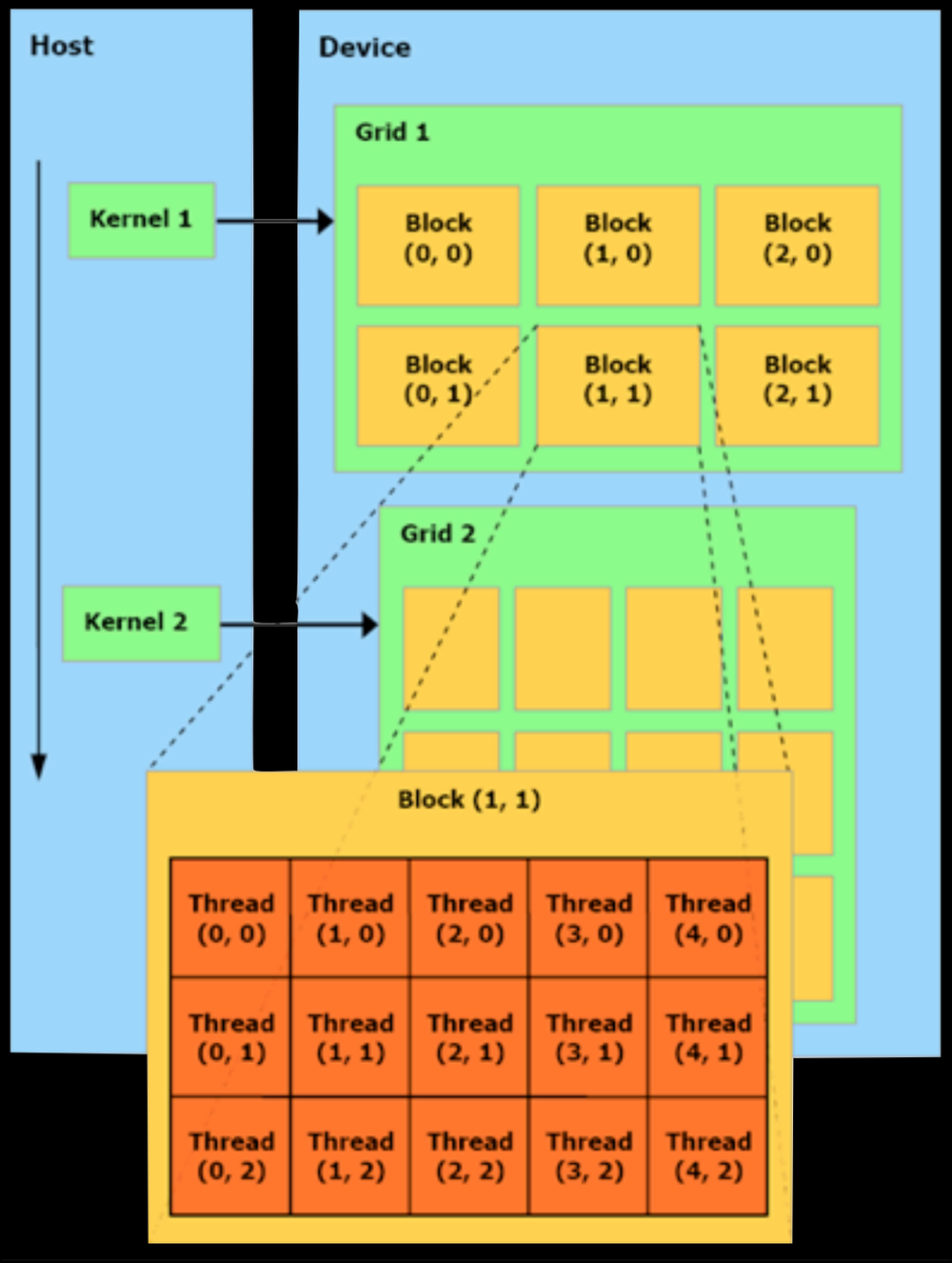
1 parameter:
Pointer



Exercise 1

Implementing the sum of two vectors using CUDA. You can assume that the array has a maximum of 1024 elements.

CUDA Programming API



Thread indexing

Threads are organized in blocks

Blocks are organized in grids

Legacy from CG applications

Indexing may be in **3D**!!!

CUDA Threads

Grid

Block (0,0)	Block (1,0)	Block (2,0)
Block (0,1)	Block (1,1)	Block (2,1)
Block (0,2)	Block (1,2)	Block (2,2)

CUDA Threads

Grid

Block (0,0)	Block (1,0)	Block (2,0)
Block (0,1)	Block (1,1)	Block (2,1)
Block (0,2)	Block (1,2)	Block (2,2)

Block

Thread (0,0)	Thread (1,0)	Thread (2,0)	Thread (3,0)
Thread (0,1)	Thread (1,1)	Thread (2,1)	Thread (3,1)
Thread (0,2)	Thread (1,2)	Thread (2,2)	Thread (3,2)

CUDA Threads

mapping threads

Block (0,0)

Thread (0,0)	Thread (1,0)	Thread (0,0)	Thread (1,0)
Thread (0,1)	Thread (1,1)	Thread (0,1)	Thread (1,1)
Thread (0,0)	Thread (1,0)	Thread (0,0)	Thread (1,0)
Thread (0,1)	Thread (1,1)	Thread (0,1)	Thread (1,1)

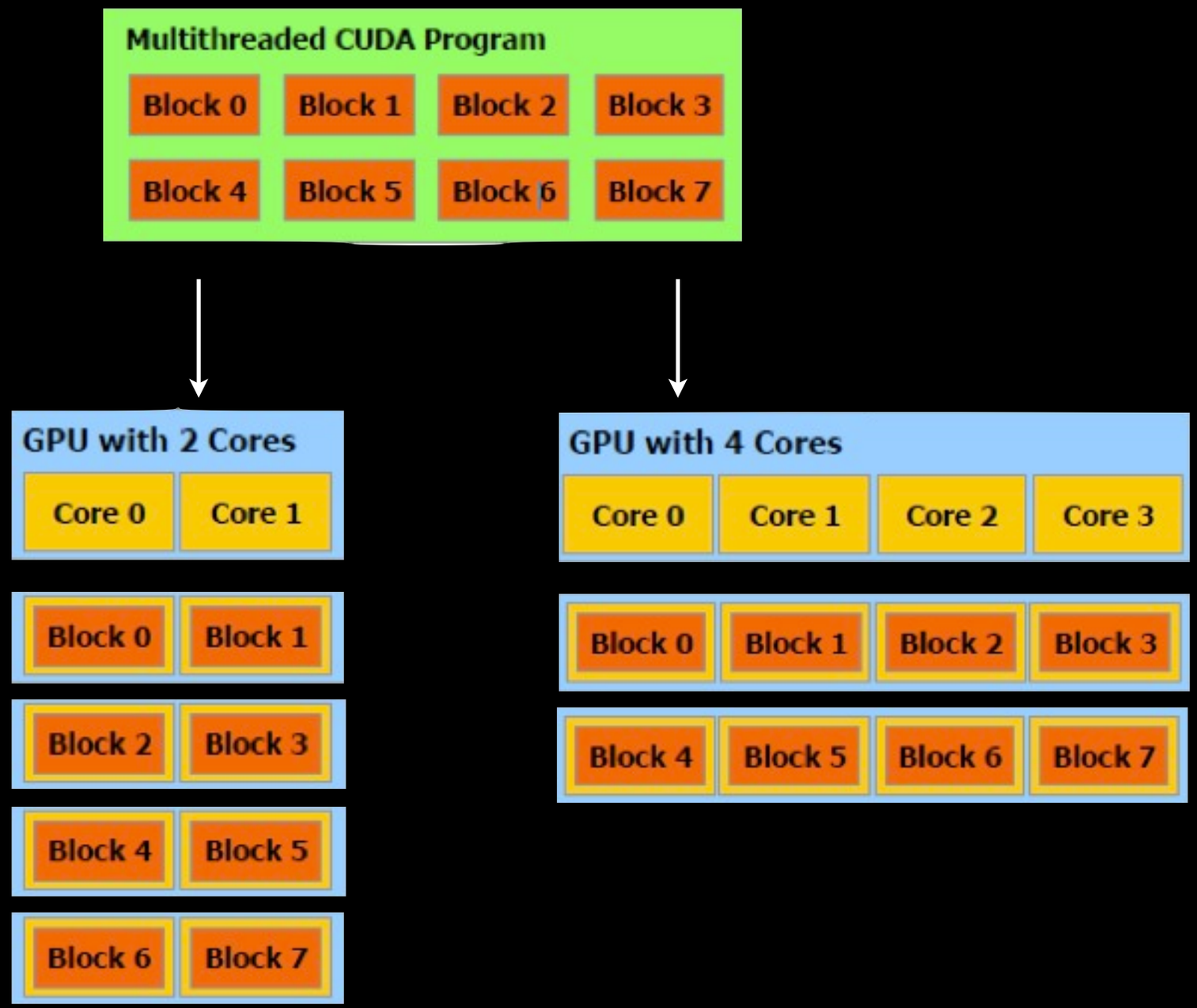
Block (1,0)

Block (0,1)

Block (1,1)

```
dim3 Grid(2,2);
dim3 Block(2,2);
kernel<<<Grid,Block>>>(parameters);
```

CUDA Threads



CUDA Threads

How can we arrange 6 threads?

Block (0,0)

Thread (0,0)	Thread (1,0)	Thread (2,0)	Thread (3,0)	Thread (4,0)	Thread (5,0)

MAX THREADS PER BLOCK DEPEND ON THE ARCHITECTURE

DEVICE QUERY

CUDA Threads

How can we arrange 6 threads?

Block (0,0)

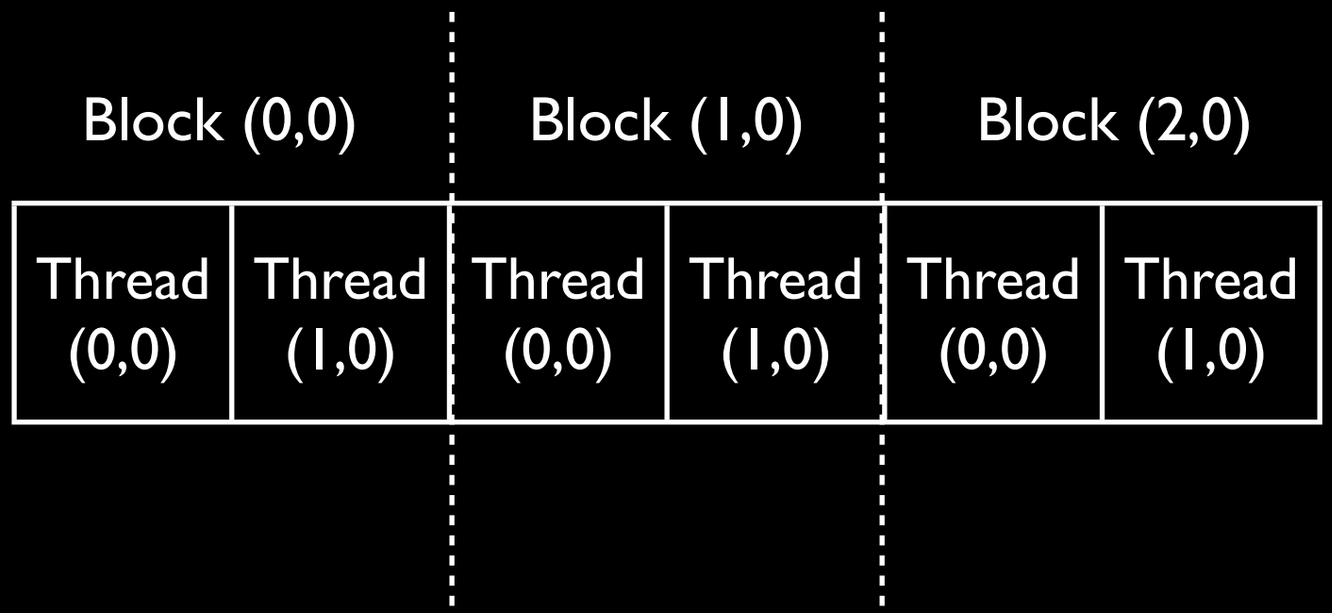
Thread (0,0)	Thread (1,0)	Thread (2,0)	Thread (3,0)	Thread (4,0)	Thread (5,0)

Block (0,0) | Block (1,0)

Thread (0,0)	Thread (1,0)	Thread (2,0)	Thread (0,0)	Thread (1,0)	Thread (2,0)

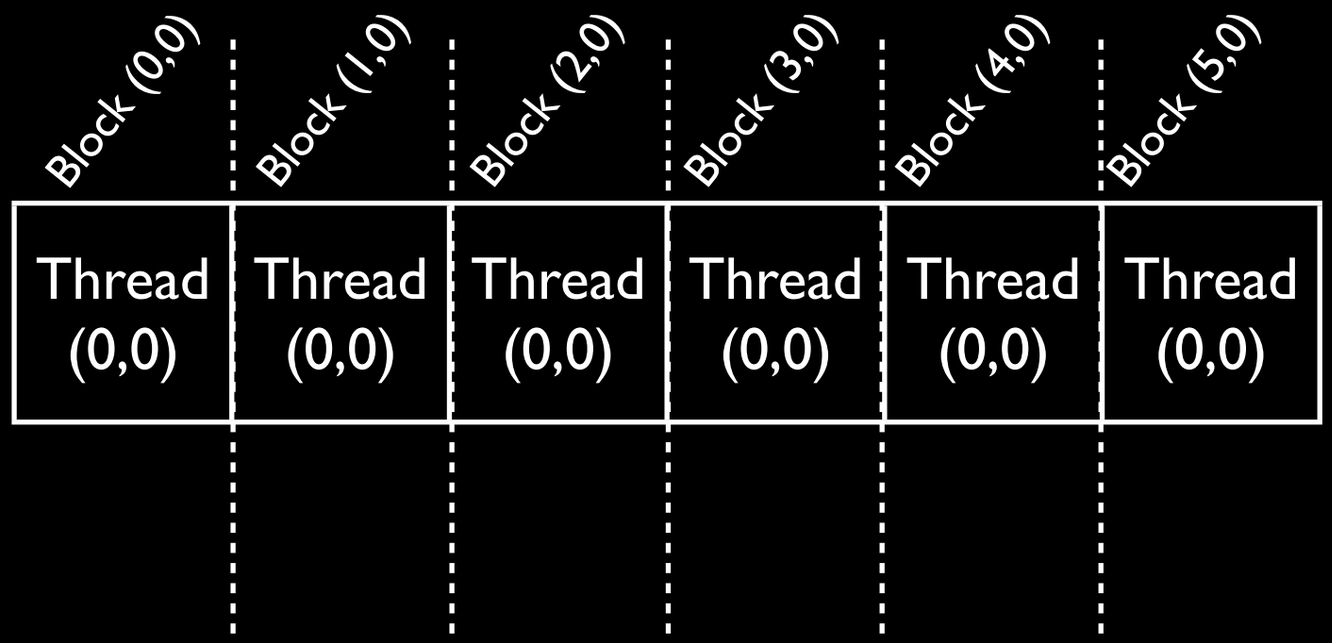
CUDA Threads

How can we arrange 6 threads?



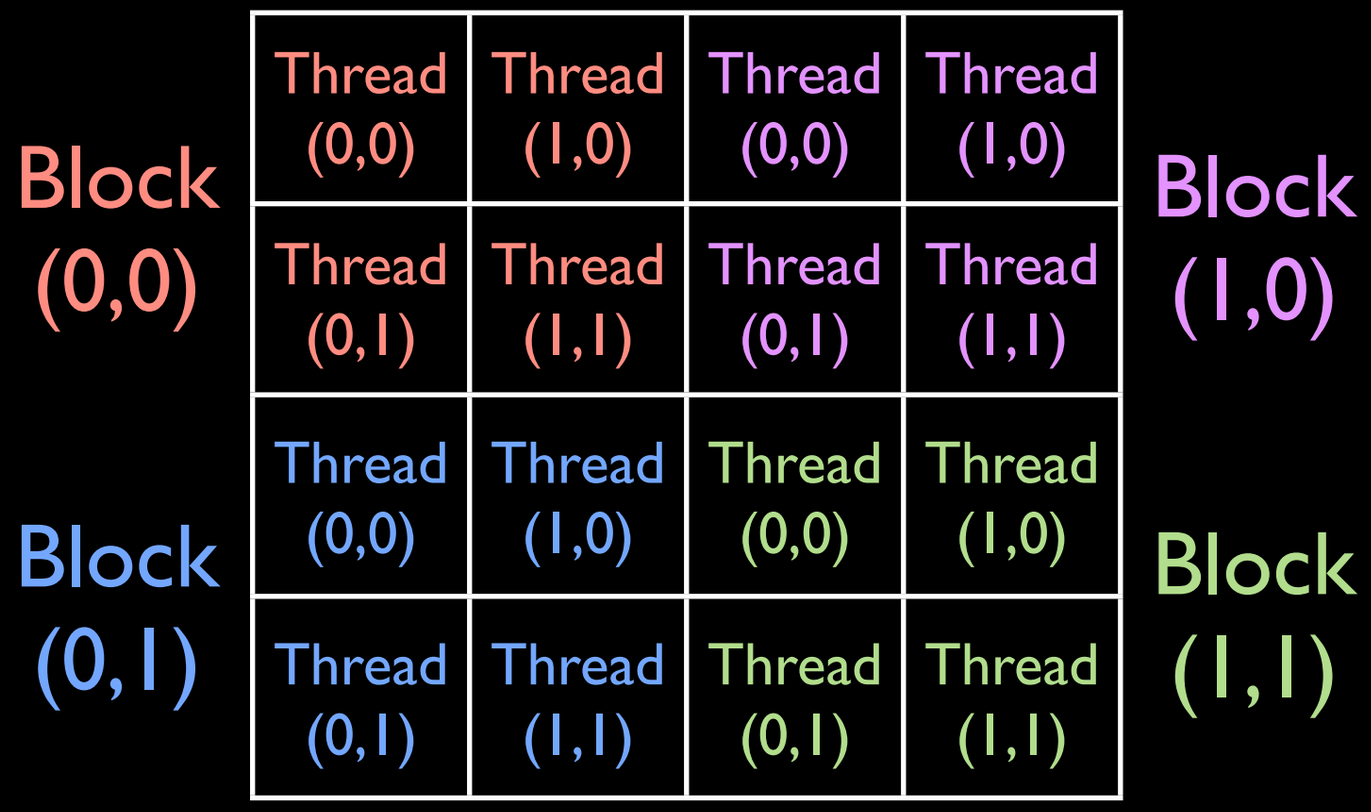
CUDA Threads

How can we arrange 6 threads?



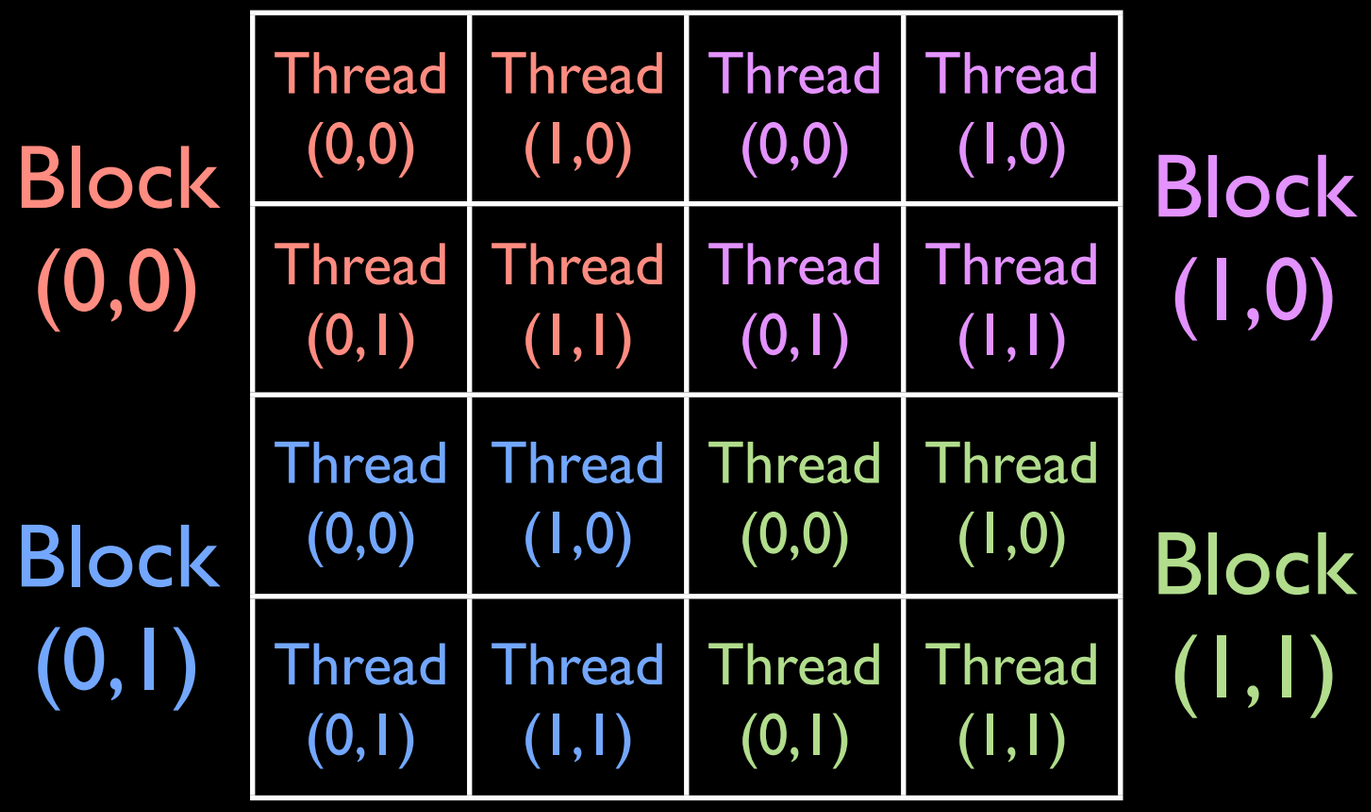
CUDA Threads

Mapping on an unique grid



CUDA Threads

Mapping on an unique grid



idx = blockIdx.x*blockDim.x + threadIdx.x;

idy = blockIdx.y*blockDim.y + threadIdx.y;

CUDA Threads

Mapping on an unique grid

Block (0,0)

Thread (0,0)	Thread (1,0)	Thread (0,0)	Thread (1,0)
Thread (0,1)	Thread (1,1)	Thread (0,1)	Thread (1,1)

Block (0,1)

Thread (0,0)	Thread (1,0)	Thread (0,0)	Thread (1,0)
Thread (0,1)	Thread (1,1)	Thread (0,1)	Thread (1,1)

Block (1,0)

Block (1,1)

Thread (0,0)	Thread (1,0)	Thread (2,0)	Thread (3,0)
Thread (0,1)	Thread (1,1)	Thread (2,1)	Thread (3,1)
Thread (0,2)	Thread (1,2)	Thread (2,2)	Thread (3,2)
Thread (0,3)	Thread (1,3)	Thread (2,3)	Thread (3,3)

idx = blockIdx.x*blockDim.x + threadIdx.x;

idy = blockIdx.y*blockDim.y + threadIdx.y;

CUDA Threads

Get an unique thread index

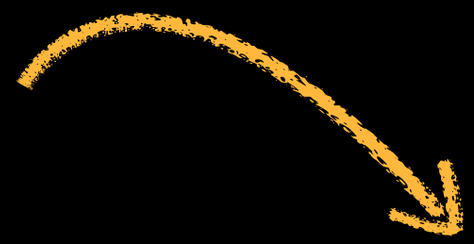
Thread (0,0)	Thread (1,0)	Thread (2,0)	Thread (3,0)
Thread (0,1)	Thread (1,1)	Thread (2,1)	Thread (3,1)
Thread (0,2)	Thread (1,2)	Thread (2,2)	Thread (3,2)
Thread (0,3)	Thread (1,3)	Thread (2,3)	Thread (3,3)

```
k = idx + idy*blockDim.x*gridDim.x;
```

CUDA Threads

Get an unique thread index

Thread (0,0)	Thread (1,0)	Thread (2,0)	Thread (3,0)
Thread (0,1)	Thread (1,1)	Thread (2,1)	Thread (3,1)
Thread (0,2)	Thread (1,2)	Thread (2,2)	Thread (3,2)
Thread (0,3)	Thread (1,3)	Thread (2,3)	Thread (3,3)



Thread (0)	Thread (1)	Thread (2)	Thread (3)
Thread (4)	Thread (5)	Thread (6)	Thread (7)
Thread (8)	Thread (9)	Thread (10)	Thread (11)
Thread (12)	Thread (13)	Thread (14)	Thread (15)

```
k = idx + idy*blockDim.x*gridDim.x;
```

Exercise II

Implementing the sum of two vectors using CUDA of a unlimited number of elements.

Exercise III

Filling the gaps of the third example.

Exercise IV

Implement a MPI/CUDA hybrid application.

Introduction
from **games** to **science** 1

2 **Architecture**

3 **GPU** Programming

CUDA 4

Final Remarks 5

GPU is good for...

loosely coupled threads (avoid synchronization)

computing bound applications

these architectures can not replace general purpose CPU

great insight for low power consumption and future architectures

CUDA Pros

Support for several OS

A lot of documentation

Many libraries available

Great performance

CUDA Cons

NVIDIA propretary

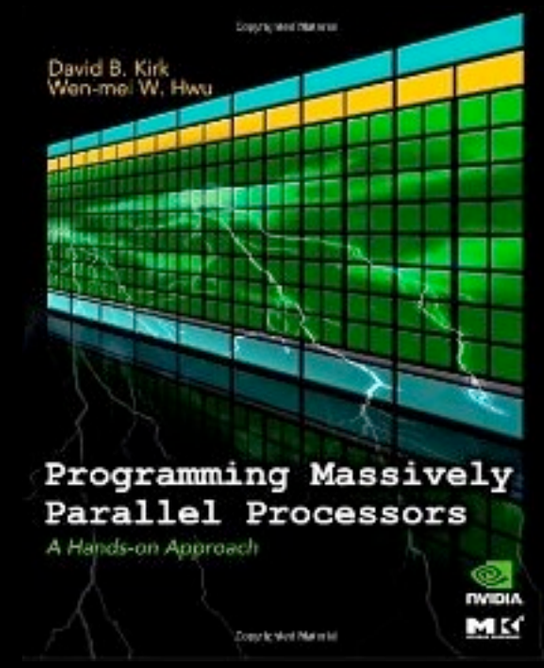
Architectures of the Future

Highly heterogeneous

Intel SandBridge
AMD Fusion
NVIDIA Tegra

Bibliography

Programming Massively Parallel Processors: A Handson Approach, David B. Kirk and Wen-Mei Hwu, Second Edition, Morgan Kaufmann, 2009



NVIDIA developer zone, http://developer.nvidia.com/