

# WHAT CAN GPU COMPUTING DO FOR NUMERICAL SIMULATIONS?

**Rastko Sknepnek**

Division of Physics

School of Engineering, Physics and Mathematics

University of Dundee



Discreet | NVIDIA



HD:  $1920 \times 1080 = 2,073,600$  pixels

60 frames per second

$10^4$  operations per pixel

$$10^6 \times 10^2 \times 10^4 = 10^{12}$$

A **TRILLION** operations per second!

**We need a TFLOPS device!**

# Outline

- CPU vs. GPU architecture
- Data parallelization
- NVIDIA™ CUDA™ overview
- CUDA™ in real life
- Limitations



DISCOVER THE NVIDIA





# CPU

- General purpose
- Sufficiently fast, yet versatile (from spreadsheets to games and from Facebook to fluid dynamic)
- Easily programmable



Some task are too demanding (e.g., graphics) and need to be offloaded to coprocessors.



Specialized hardware components (such as GPU)



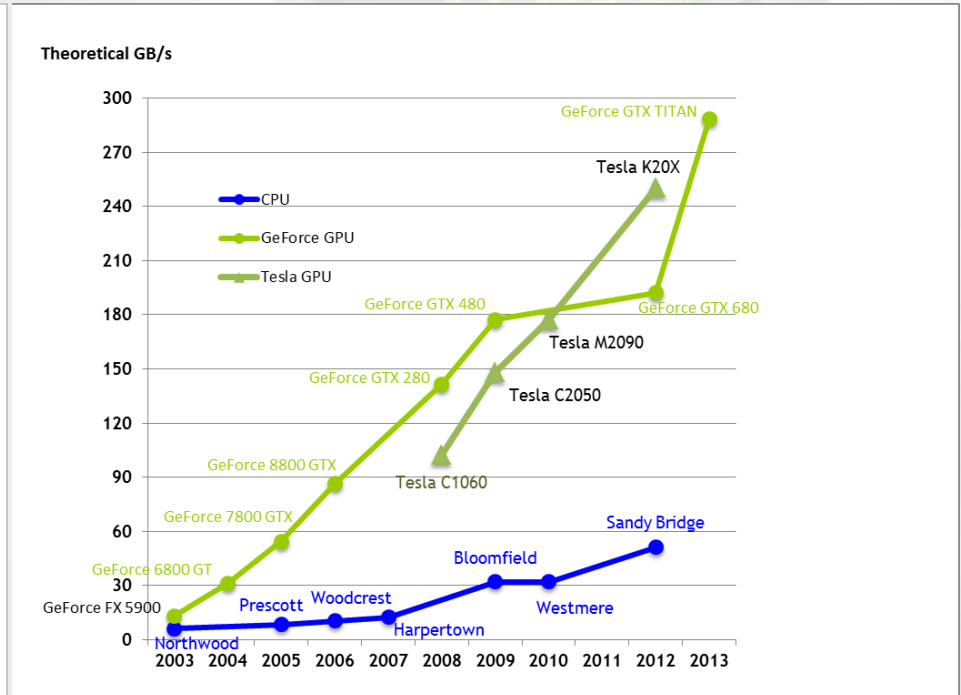
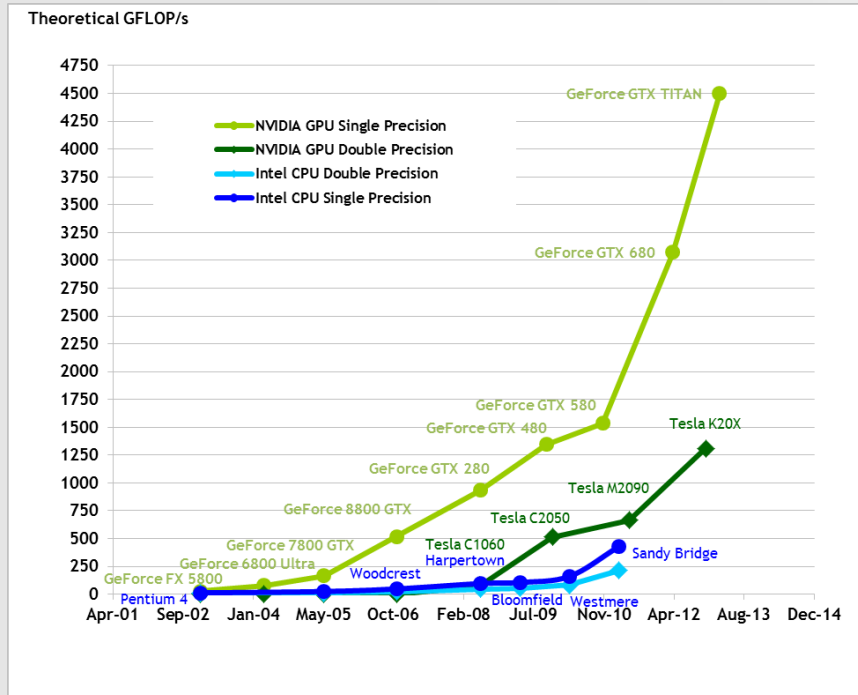
## In 2006 NVIDIA released **Compute Unified Device Architecture (CUDA)**

G80 GPU – general purpose parallel computing platform opened access to GPU's tremendous computational capabilities (over 500 GFLOPS and 90GB/s bandwidth) to non graphical applications.

**GPGPU WAS BORN!**



(GeForce 8800 GTX, 2006)



# Side by side comparison

## CPU



Intel Core i7-4960X (\$1,000)

- 22nm manufacturing
- **6** cores/**15MB** shared L3 cache
- 3.6 GHz
- ~1.9 billion transistors
- ~**60 GB/s** memory bandwidth (theoretical)
- ~**160 GFLOPS**
- ~200W of power

## GPU

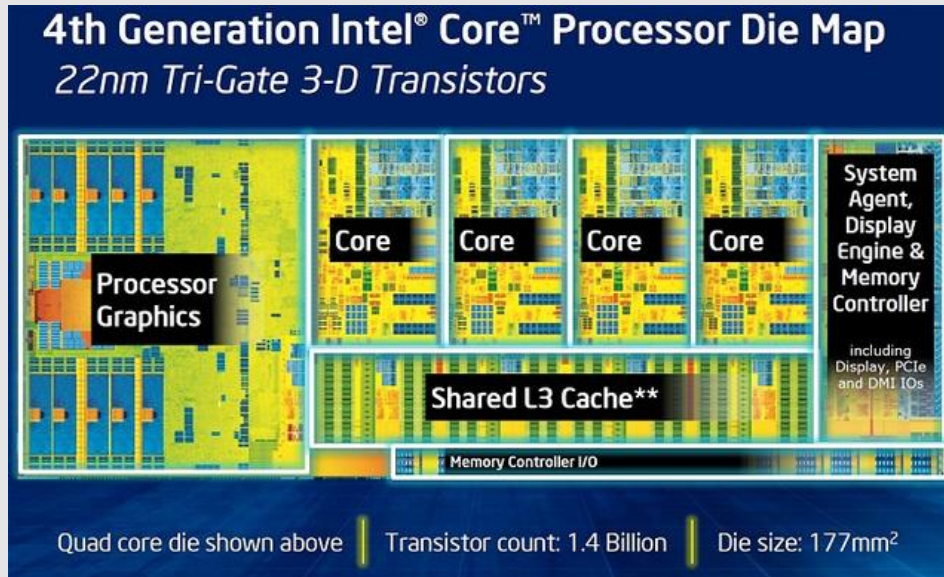


NVIDIA TITAN GK110 (Kepler)  
(~\$1,000)

- 28 nm manufacturing
- **2688** CUDA cores/**1,536KB** of L2 cache
- 837 MHz
- ~7.1 billion transistors
- ~**290 GB/s** memory bandwidth (theoretical)
- ~**4 TFLOPS** (single) and **1.3 TFLOPS** (double)
- 250W of power



# Radically different architectures



Runs handful of threads at the same time

Small number of registers (16 per core) (expensive context switching)

Complex logic control

CPU hides memory latency by large multilevel caches (L1, L2, and L3)

Runs thousands of threads simultaneously (Single Instruction Multiple Threads – SIMT)

Huge number of registers (~65k) (cheap context switching)

Simple control logic and very little cache - most transistors devoted to number crunching

Memory latency hidden by computations.

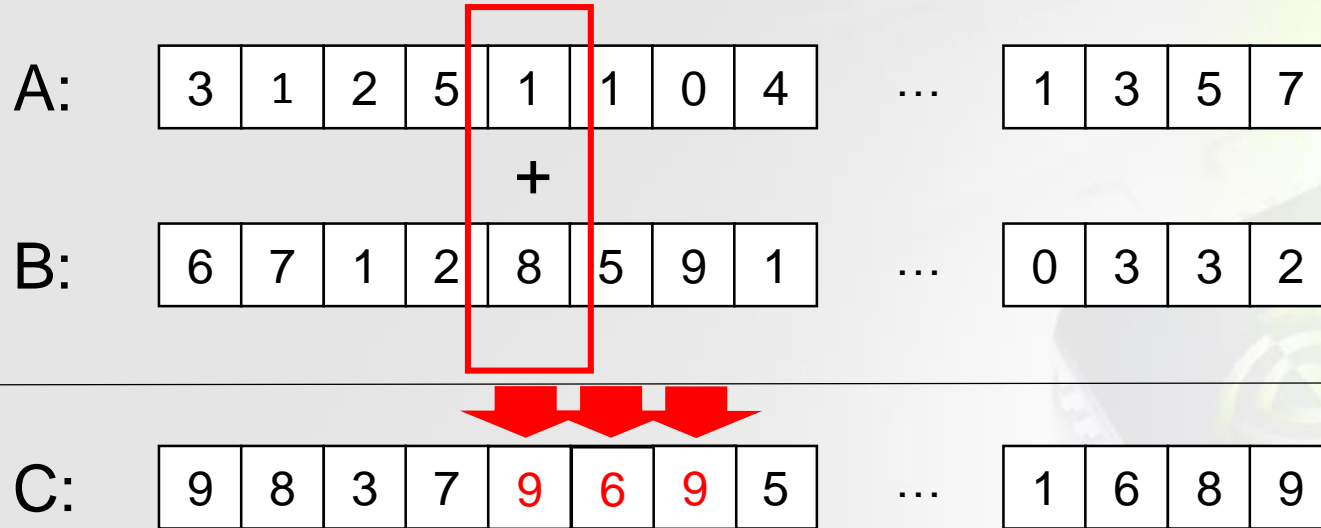


GTX TITAN GK110 GPU (Kepler)

# Introducing CUDA

Let's start with an example...

Add to vectors ( $\mathbf{C} = \mathbf{A} + \mathbf{B}$ ):



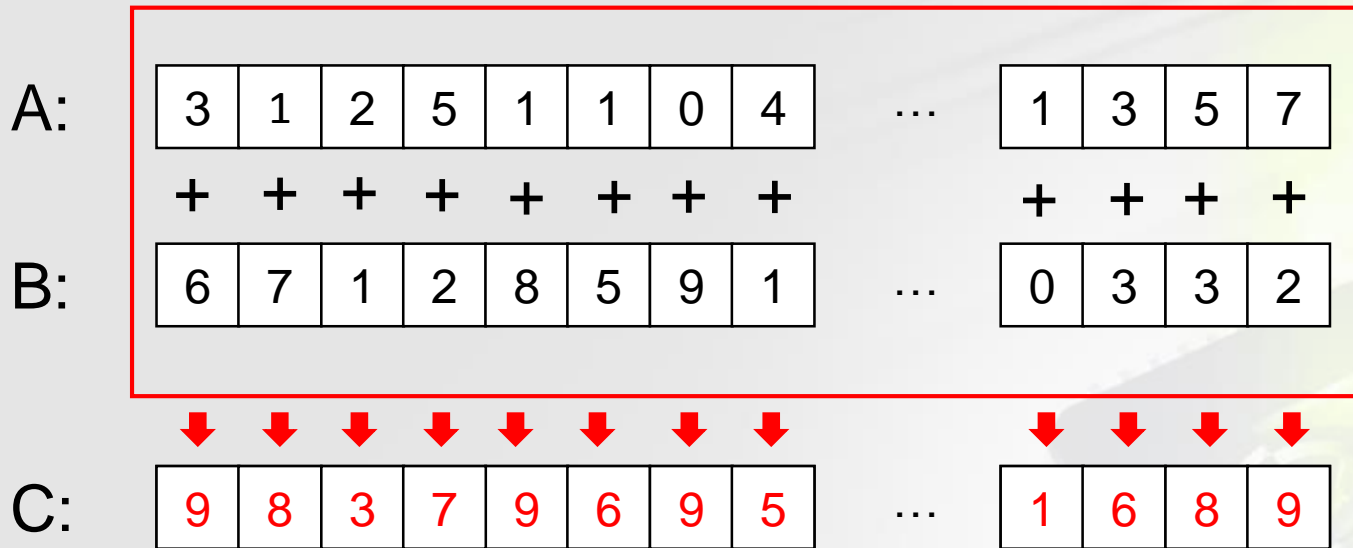
On CPU in ANSI-C

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

} loop



It's much faster to do it in parallel (with CUDA)



```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    VecAdd<<<1, N>>>(A, B, C);
}
```

Looks like C, but what actually happened here?

```
// Kernel definition
```

```
__global__ void VecAdd(float* A, float* B, float* C)
```

```
{  
    int i = threadIdx.x;  
    C[i] = A[i] + B[i];  
}
```

```
int main()
```

```
{  
    VecAdd<<<1, N>>>(A, B, C);  
}
```

**GPU (device) code**

**CPU (host) code**

**\_\_global\_\_** : new keyword indicating that this code will run on the device (GPU)

**threadIdx.x** : internal variable that specifies id of the current thread

**<<<1,N>>>** : host will invoke N threads stored in one block in the device

Code is split into host and device parts (more below).

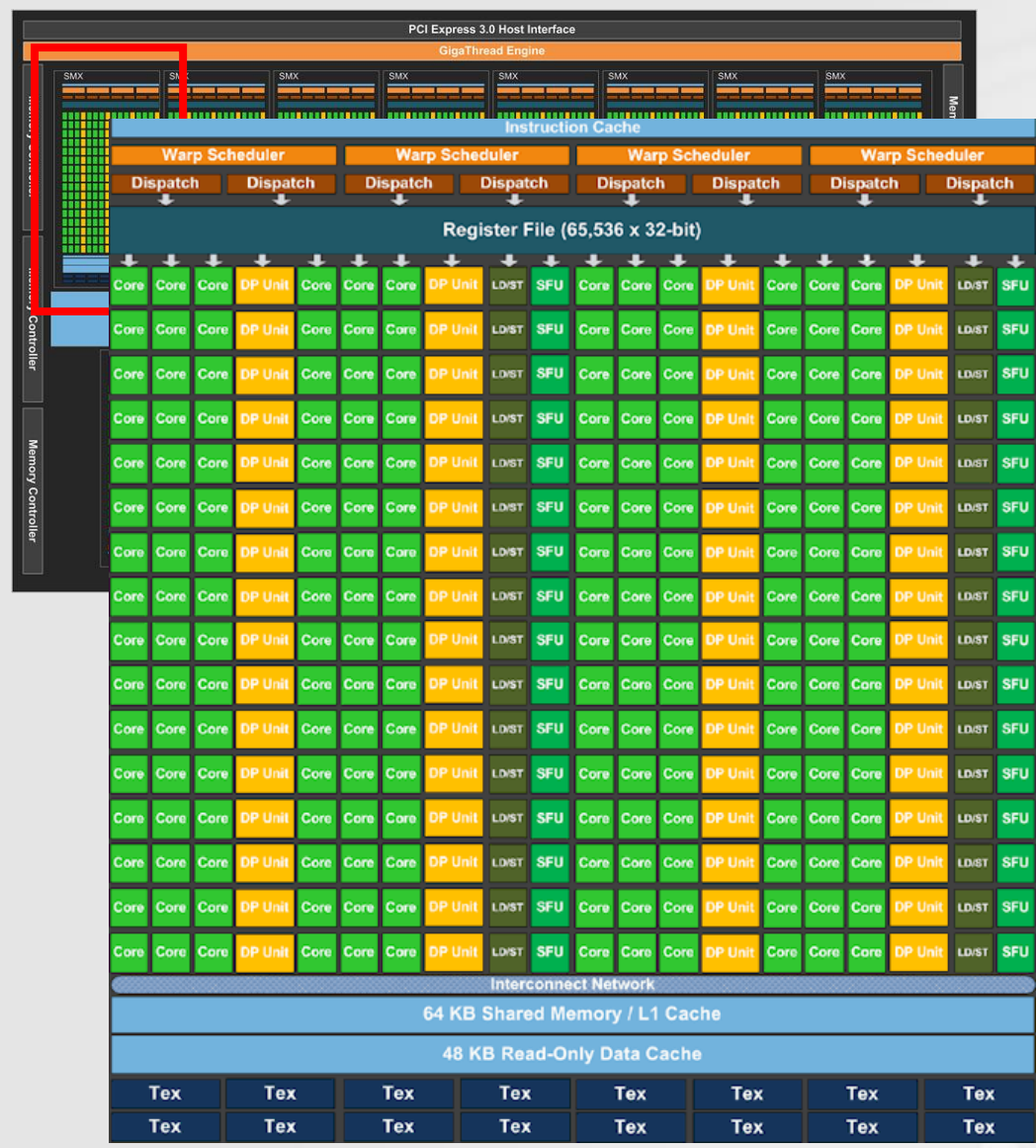
Device code runs N threads in parallel.

\* Actual code would need memory allocation.

Glossary:

- Host = CPU
- Device = GPU
- Code on device = “kernel”

GPU contains up to 15 SMX (streaming multiprocessors)

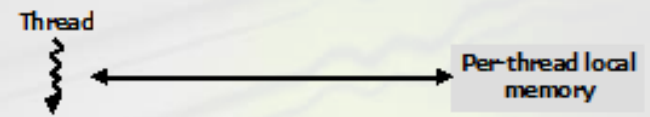


Each SMX has:

- 192 single-precision CUDA cores
  - 64 double-precision units
  - 32 special function units (SFU)
  - 32 load/store units (LD/ST).
  - 64 KB Configurable Shared Memory and L1 Cache
  - 48KB Read-Only Data Cache
  - 255 registers per thread
- The SMX schedules threads in groups of **32** parallel threads called **warps**.
  - 4 warp schedulers/SMX
  - 8 instruction dispatch units
  - 4 warps to be issued and executed concurrently.

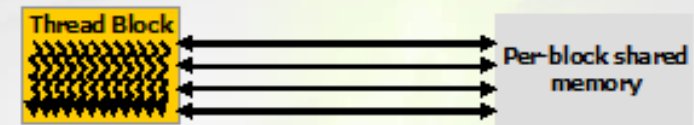


Thread – basic unit of execution (runs on a core)



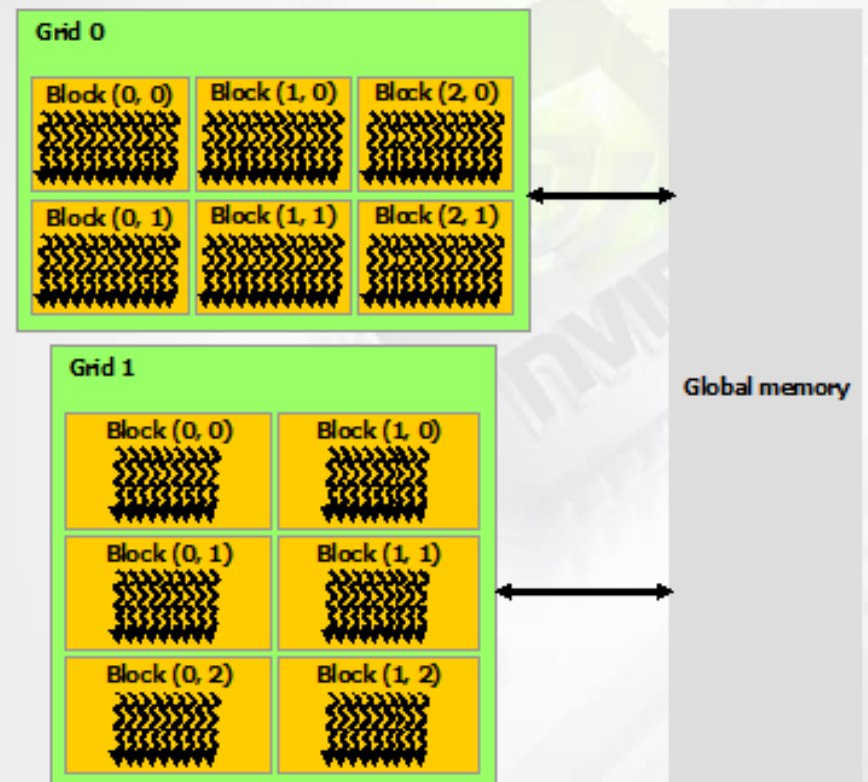
Threads are grouped into blocks

- Currently each block can contain up to 1024 threads
- Typically one uses 256 (16x16) threads per block
- Each block runs on one SMX
- Threads within block can access common shared memory



Thread block are organized into a grid.

- Thread blocks are required to execute independently: It must be possible to execute them in any order.
- Thread blocks can be scheduled in any order across any number of SMXs.

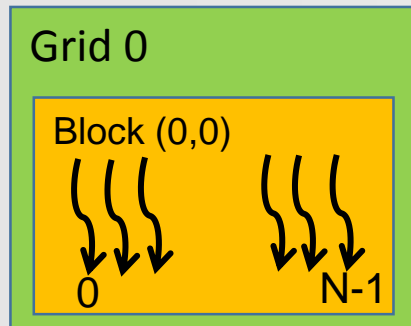


## Back to the example

```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    VecAdd<<<1, N>>>>(A, B, C);
}
```

One thread  
(single core)

Grid with one thread block  
Thread block with N threads



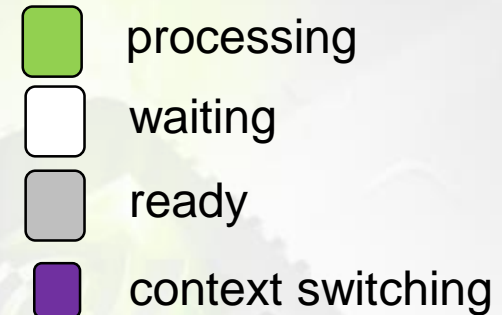
- SMX executes groups of 32 threads (called **warp**) at the same time.
- Each thread executes same instruction (SIMT)
- Divergence is possible!
- Latency hidden by fast context changes.

**Note:** For convenience threads within a block can be indexed with one-, two-, or three-dimensional indices. Same is true for block within a grid.

## Latency hiding

Although running at 200 GB/s bandwidth<sup>\*</sup>, memory access is still much slower than any computation (up to 800 cycles). For optimal performance one needs to hide this latency by keeping cores busy.

CPU hides memory latency through multilevel caches



GPU hides latency with high throughput.



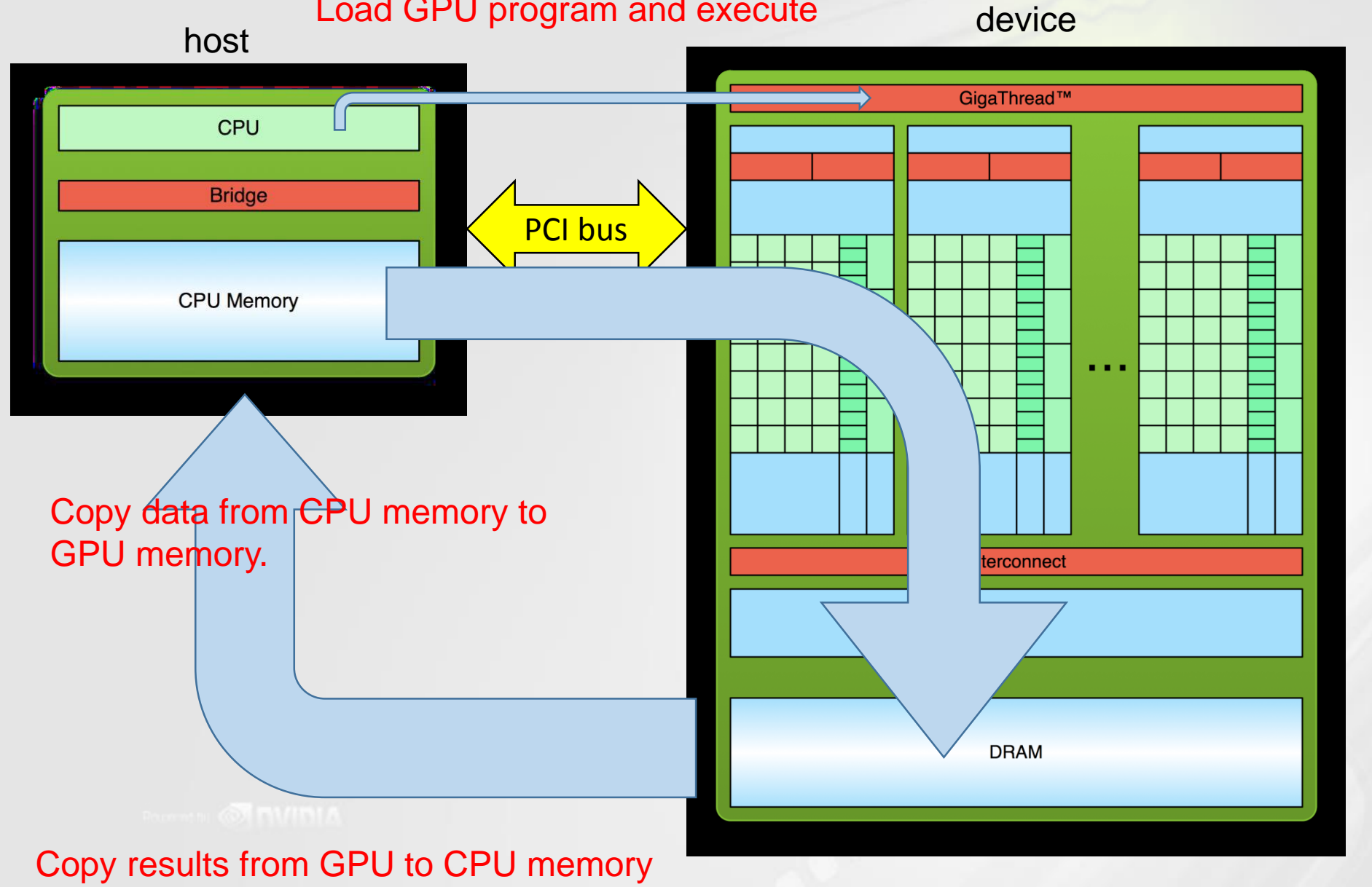
For maximum performance one needs to spawn many (thousands!) of threads.

**\* Word of caution:** For maximal bandwidth memory access needs to be coalesced.



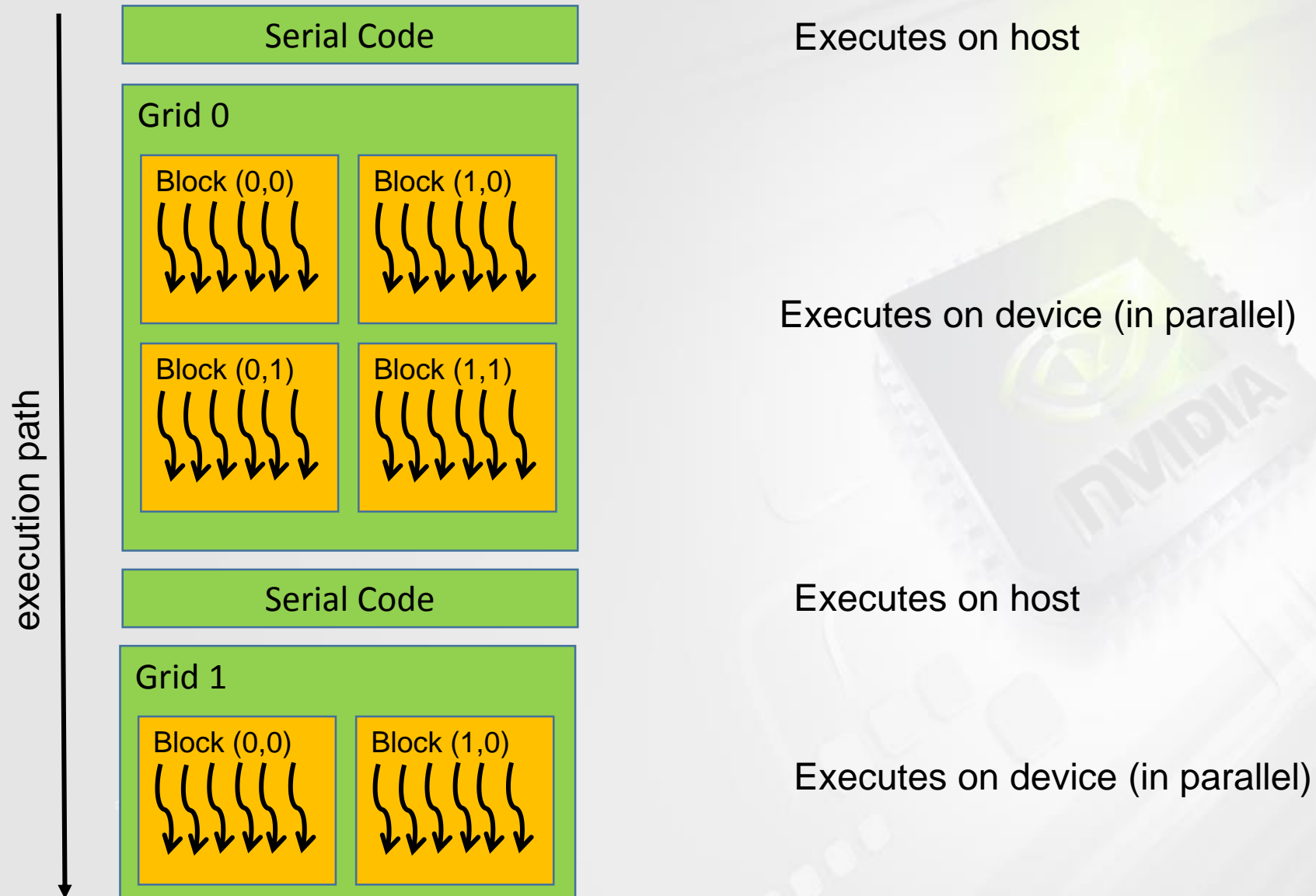
Programming model: Device and host are separate entities with their own memory

Load GPU program and execute



# CUDA introduces heterogeneous computing

Parts of the code are executed on CPU and parts on GPU



# Full example...

```
#include <stdio.h>
```

```
__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];
}
```

device kernel

```
int main(void)
```

```
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));
    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));
    for (int i = 0; i < N; i++)
    {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }
```

host and device arrays

system size  $N = 2^{20}$  elements

allocating memory on host.

allocating memory on device.

populating host arrays

```
    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

copy data to device

```
    saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
```

execute tread block grid

```
    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

copy data back to host

```
    float maxError = 0.0f;
    for (int i = 0; i < N; i++)
        maxError = max(maxError, abs(y[i]-4.0f));
    printf("Max error: %fn", maxError);
```

do something with the result



Where one can see great speedups...





- Problems that require huge amount of computations and/or bandwidth
- Problems that have regular (predicable) memory access patterns
- Problems that do not require diverging code

Most of dense linear algebra (PDE solvers), large scale N-body simulations (gravity), molecular dynamics (MD) fall into this category.\*

**Word of caution:** Unfortunately, it is not sufficient to simply recompile your code on GPU and see a 100x speed up (most likely you'd see a slow down). Significant rewriting and even algorithm redesign is often necessary.

\*Different level of complexity to implement them such that the GPU's power is fully utilized.

Many libraries and software packages are already available!

GPU Computing Applications						
Libraries and Middleware						
CUFFT CUBLAS CURAND CUSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX	iray	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
 CUDA-Enabled NVIDIA GPUs						
<u>Kepler Architecture</u> (compute capabilities 3.x)	GeForce 600 Series		<u>Quadro Kepler Series</u>		Tesla K20 Tesla K10	
<u>Fermi Architecture</u> (compute capabilities 2.x)	GeForce 500 Series GeForce 400 Series		<u>Quadro Fermi Series</u>		Tesla 20 Series	
<u>Tesla Architecture</u> (compute capabilities 1.x)	<u>GeForce 200 Series</u> <u>GeForce 9 Series</u> <u>GeForce 8 Series</u>		<u>Quadro FX Series</u> <u>Quadro Plex Series</u> <u>Quadro NVS Series</u>		Tesla 10 Series	
 Entertainment		 Professional Graphics		 High Performance Computing		

## What GPUs are not good for...

GPUs are **NOT** the silver bullet!  
(They are not here to replace CPU.)

Types of problems that don't run well on GPUs:

- Most graph algorithms (too unpredictable, especially in memory access)
- Sparse linear algebra (but this is bad on the CPU too)
- Small signal processing problems (FFTs smaller than 1000 points, for example)
- Search
- Sort
- Complex data structures



# Summary

- Modern GPUs are much more than just devices for producing fancy graphics.
- They deliver tremendous computational power (TFLOPS) and bandwidth at very low cost and power consumption.
- GPU power comes from high level of parallelization accompanied by was majority of the chip being devoted to computations at the expense of drastically simplified control flows.

## Pros

- Huge computational power at low cost.
- Programmable for non-graphical applications.
- Programmable in C/C++ with only a few language extensions.



## Cons

- Often requires fundamental redesign of the code.
- Programmer needs to be fairly familiar with the internal workings of the hardware.
- Bad programming is severely penalized.