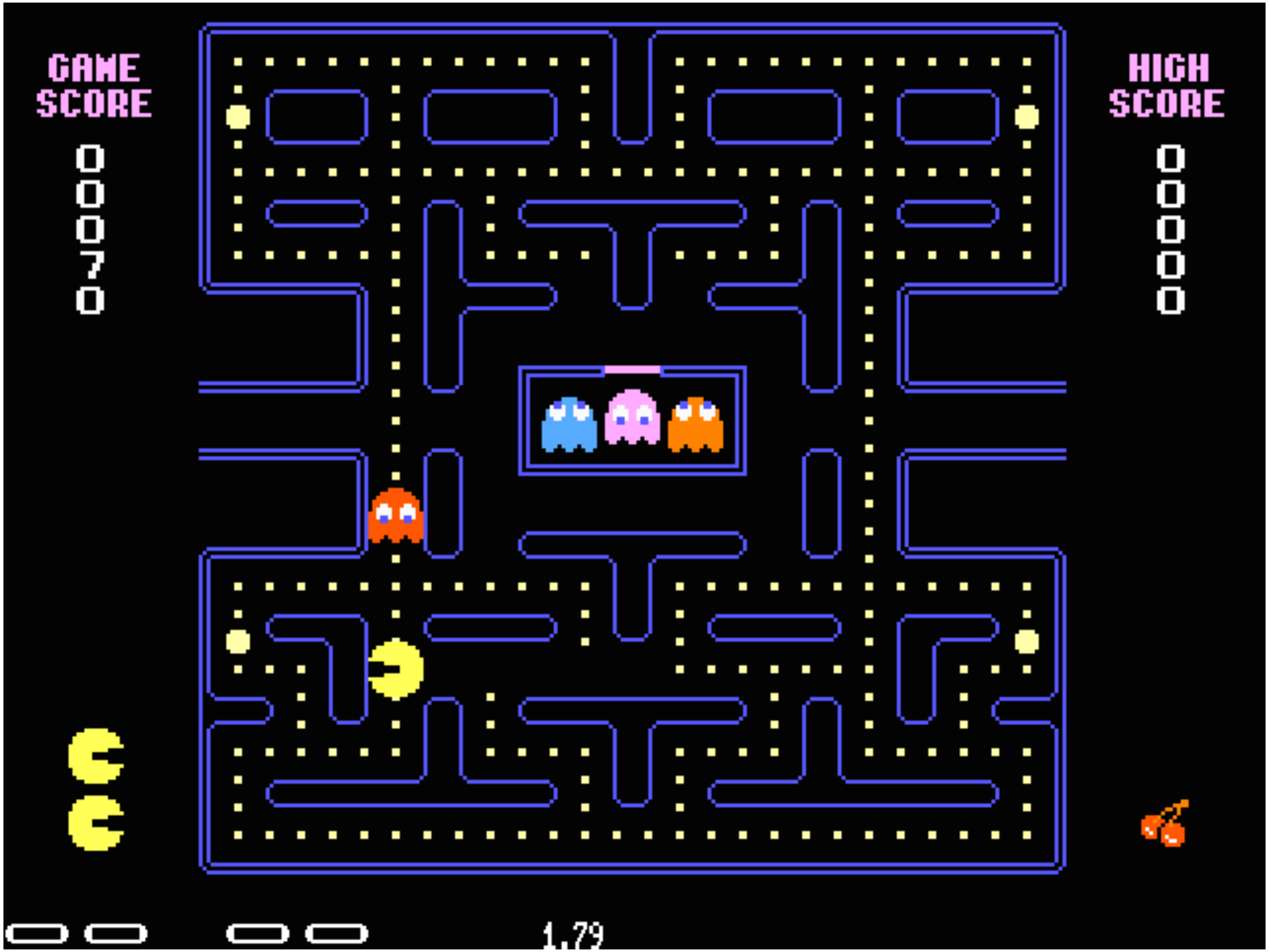
GPU COMPUTING

Part 1

Ana Lucia Varbanescu (UvA)

Graphics in the 80s



Graphics in the 90s

Graphics in 2015

GPUs in movies

- From Ariel in Little Mermaid to Brave

So …

- GPUs are a steady market
  - Gaming
  - CAD-like activities
    - Traditional or not …
  - Visualisation
    - Scientific or not …
- GPUs are increasingly used for other types of applications
  - Number crunching in science, finance, image processing
  - (fast) Memory operations in big data processing

TODO List

1. The hardware
2. The software
3. Programming examples

INTRODUCTION TO GPUS

GPU = the processor

GPGPU = general purpose computing on GPUs
(typically refers to non-graphics stuff)

GPGPU History

Use graphics primitives for HPC

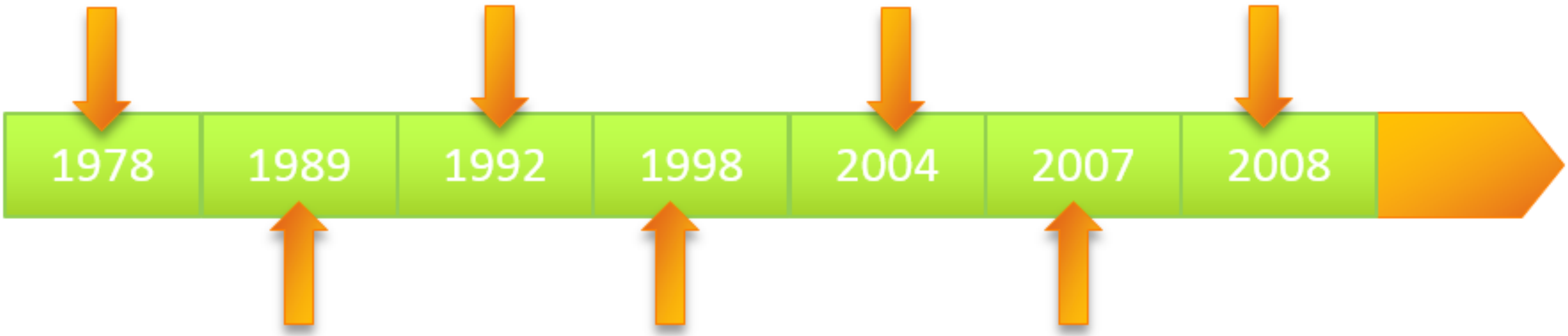
Graphics Programming on GPU (GPGPU)

Ikonas [England 1978]

Pixel-planes 5 [Rhoades, et al. 1992]

Brook (2004)

OpenCL (December 2008)



Pixel machine [Potmesil & Hoffert 1989]

DirectX/OpenGL

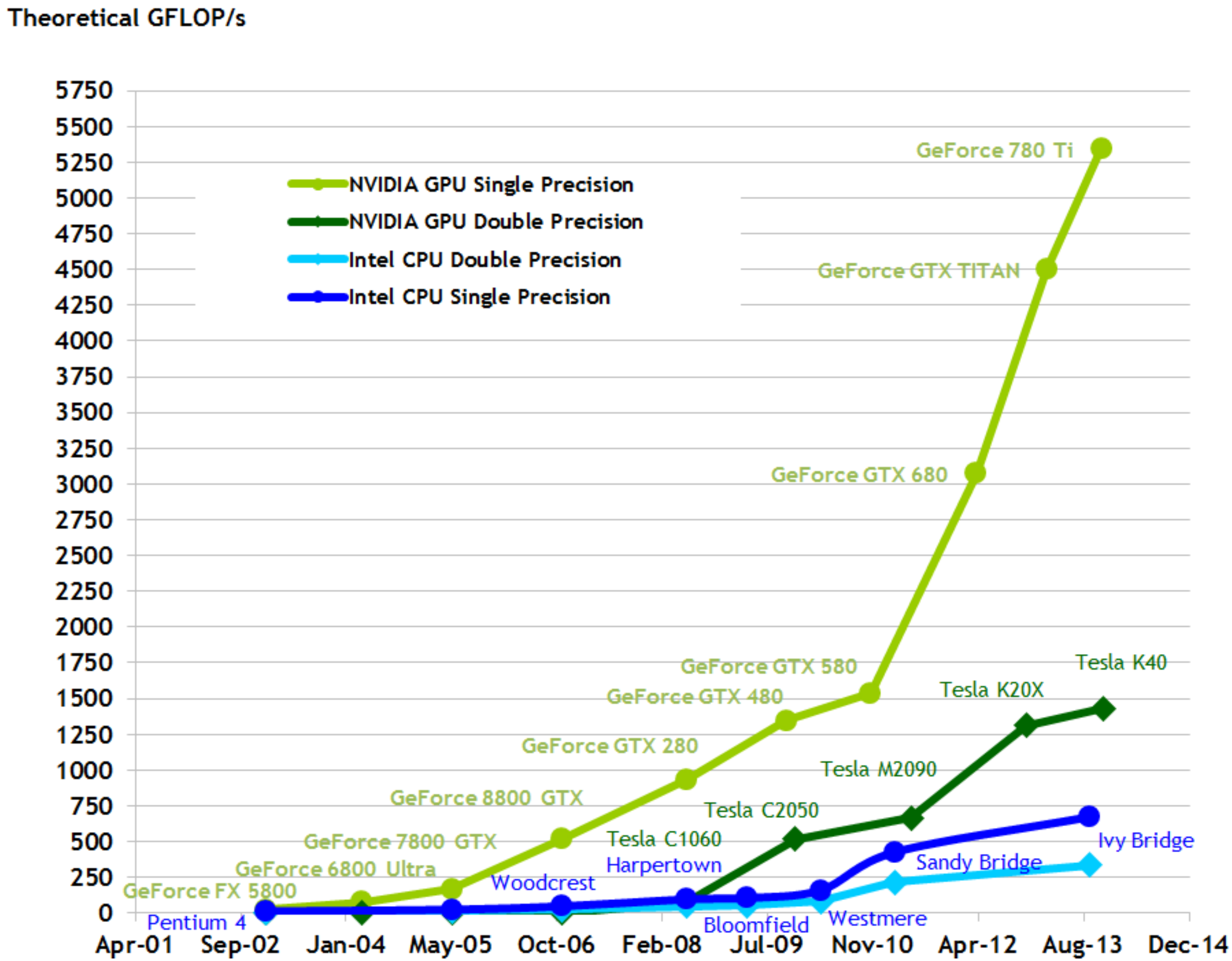
Map application onto graphics domain

CUDA (2007)

Programmable shaders, around 1998

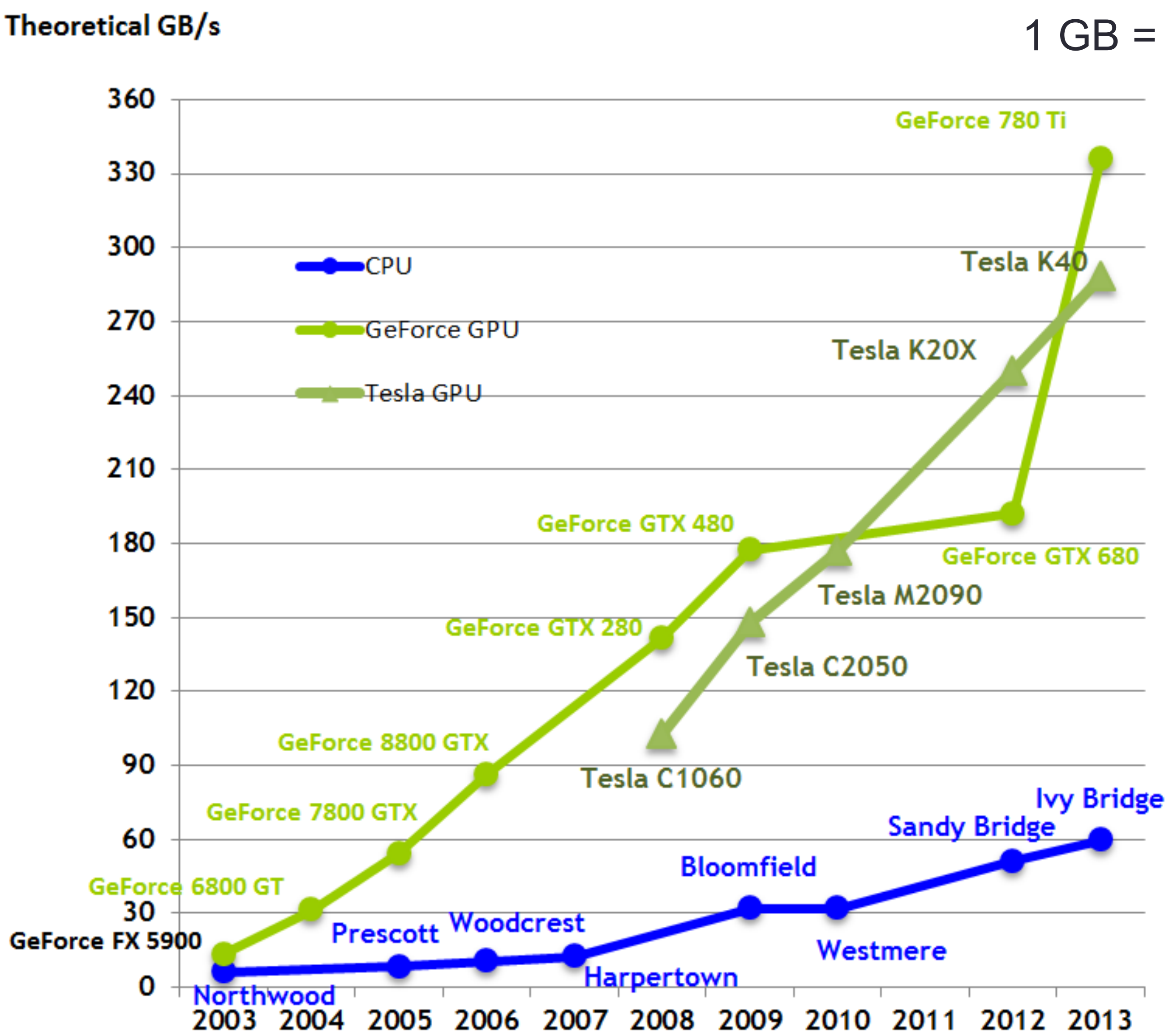
GPU vs. CPU performance

1 GFLOP = 10^9 ops

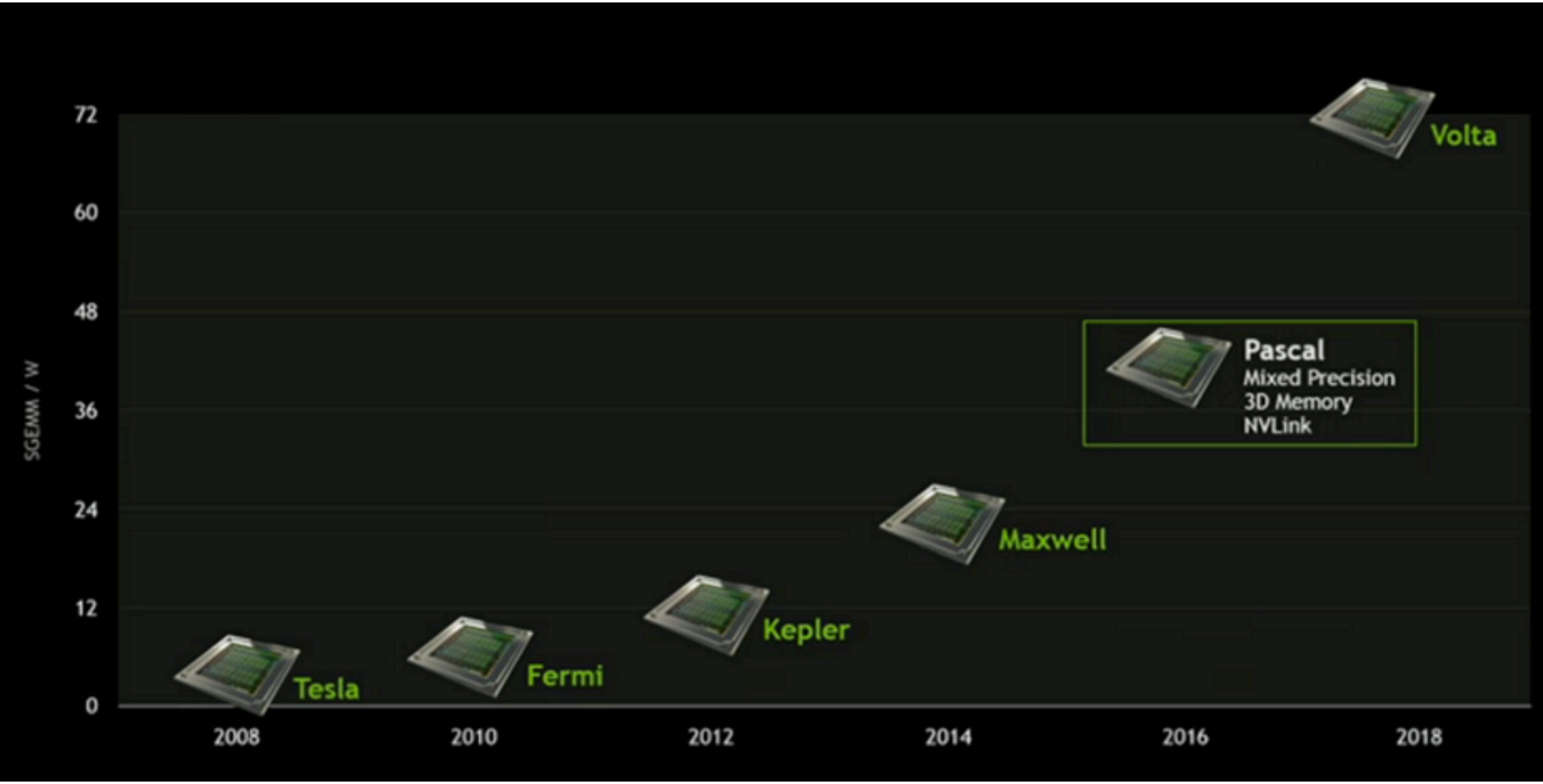


GPU vs. CPU performance

1 GB = 8 x10^9 bits

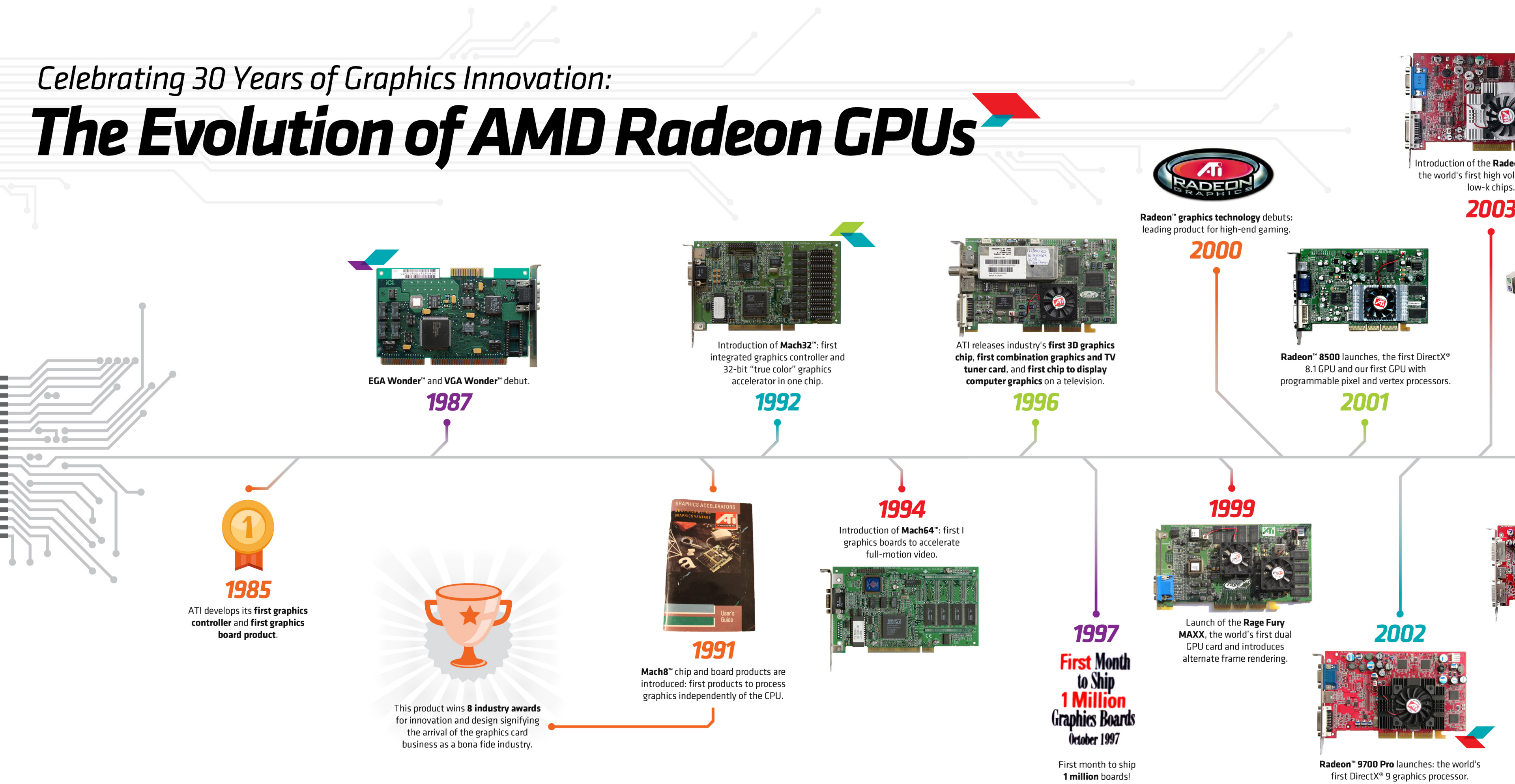


GPUs @ NVIDIA

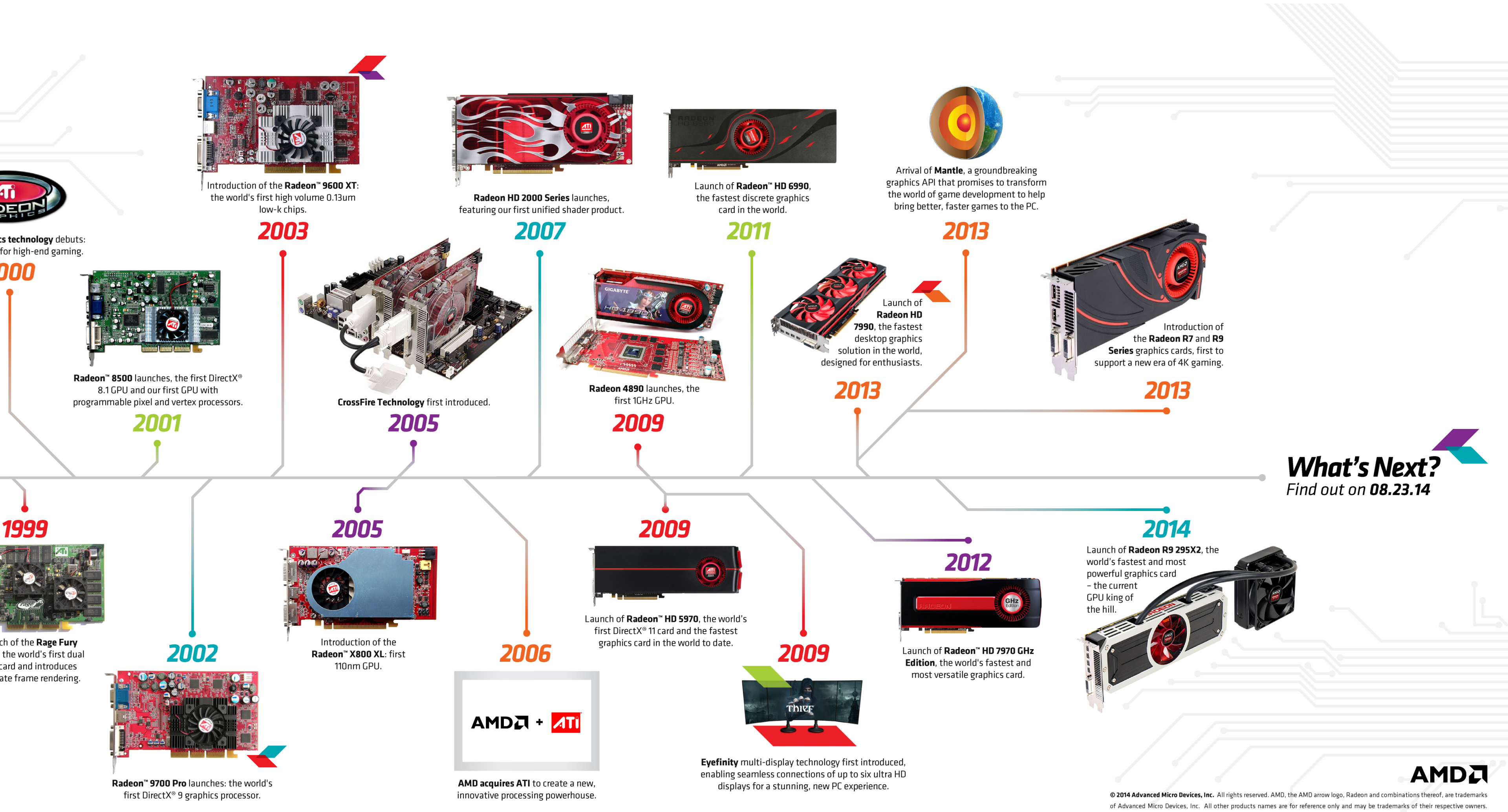


SGEMM / W
72
60
48
36
24
12
0
2008
2010
2012
2014
2016
2018
Tesla
Fermi
Kepler
Maxwell
Pascal
Mixed Precision
3D Memory
NVLink
Volta

GPUs @ ATI/AMD

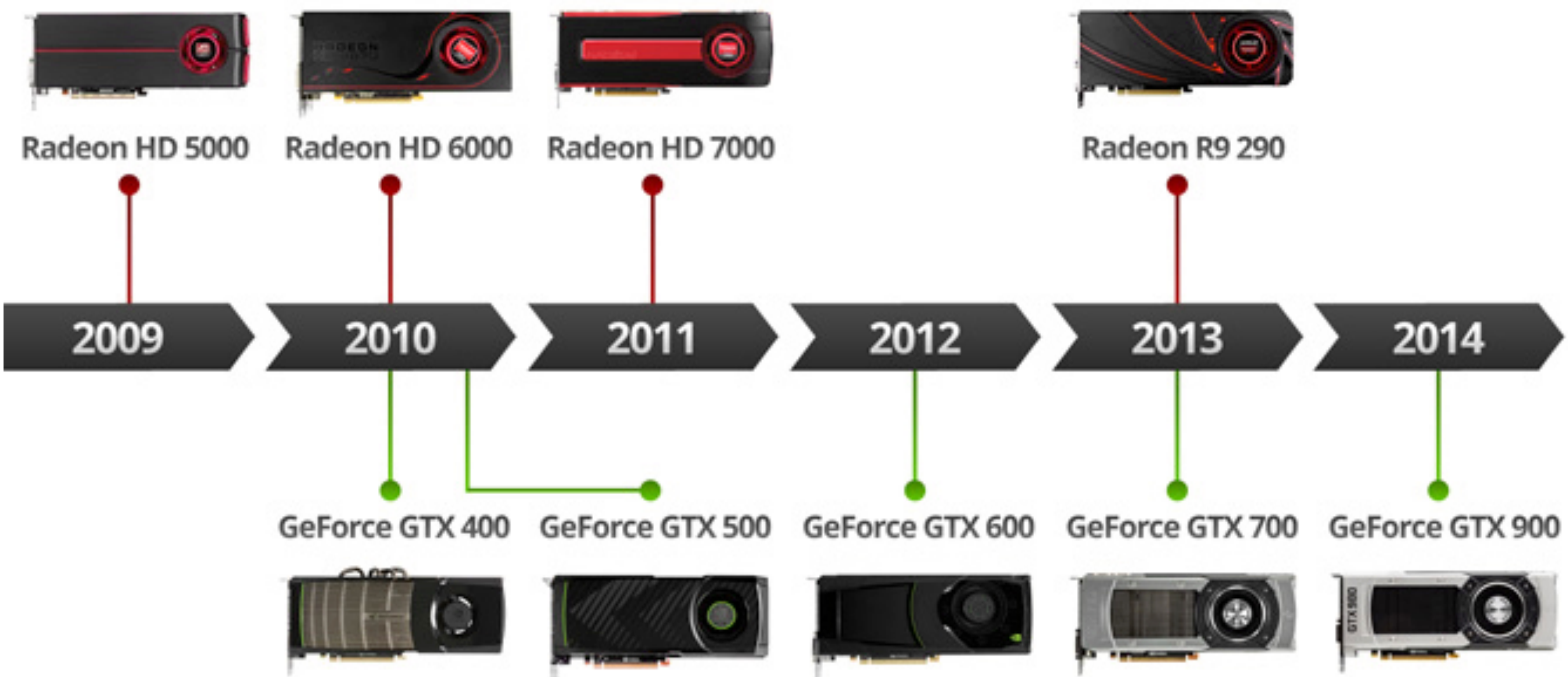

Celebrating 30 Years of Graphics Innovation:
The Evolution of AMD Radeon GPUs
1985
ATI develops its first graphics controller and first graphics board product.
This product wins 8 industry awards for innovation and design signifying the arrival of the graphics card business as a bona fide industry.
1987
EGA Wonder™ and VGA Wonder™ debut.
1991
Mach8™ chip and board products are introduced: first products to process graphics independently of the CPU.
1992
Introduction of Mach32™: first integrated graphics controller and 32-bit "true color" graphics accelerator in one chip.
1994
Introduction of Mach64™: first graphics boards to accelerate full-motion video.
1996
ATI releases industry's first 3D graphics chip, first combination graphics and TV tuner card, and first chip to display computer graphics on a television.
1997
First Month to Ship 1 Million Graphics Boards October 1997
First month to ship 1 million boards!
1999
Launch of the Rage Fury MAXX, the world's first dual GPU card and introduces alternate frame rendering.
2000
Radeon™ graphics technology debuts: leading product for high-end gaming.
2001
Radeon™ 8500 launches, the first DirectX® 8.1 GPU and our first GPU with programmable pixel and vertex processors.
2002
Radeon™ 9700 Pro launches: the world's first DirectX® 9 graphics processor.
2003

GPUs @ ATI/AMD



...cs technology debuts: ...for high-end gaming.
...000
Introduction of the Radeon™ 9600 XT: the world's first high volume 0.13um low-k chips.
2003
Radeon HD 2000 Series launches, featuring our first unified shader product.
2007
Launch of Radeon™ HD 6990, the fastest discrete graphics card in the world.
2011
Arrival of Mantle, a groundbreaking graphics API that promises to transform the world of game development to help bring better, faster games to the PC.
2013
Radeon™ 8500 launches, the first DirectX® 8.1 GPU and our first GPU with programmable pixel and vertex processors.
2001
CrossFire Technology first introduced.
2005
Radeon 4890 launches, the first 1GHz GPU.
2009
Launch of Radeon HD 7990, the fastest desktop graphics solution in the world, designed for enthusiasts.
2013
Introduction of the Radeon R7 and R9 Series graphics cards, first to support a new era of 4K gaming.
2013
What's Next?
Find out on 08.23.14
1999
...ch of the Rage Fury ... the world's first dual ...card and introduces ...ate frame rendering.
2002
Radeon™ 9700 Pro launches: the world's first DirectX® 9 graphics processor.
2005
Introduction of the Radeon™ X800 XL: first 110nm GPU.
2006
AMD + ATi
AMD acquires ATI to create a new, innovative processing powerhouse.
2009
Launch of Radeon™ HD 5970, the world's first DirectX® 11 card and the fastest graphics card in the world to date.
2009
Eyefinity multi-display technology first introduced, enabling seamless connections of up to six ultra HD displays for a stunning, new PC experience.
2012
Launch of Radeon™ HD 7970 GHz Edition, the world's fastest and most versatile graphics card.
2014
Launch of Radeon R9 295X2, the world's fastest and most powerful graphics card – the current GPU king of the hill.
AMD
© 2014 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD arrow logo, Radeon and combinations thereof, are trademarks of Advanced Micro Devices, Inc. All other products names are for reference only and may be trademarks of their respective owners.

NVIDIA vs AMD



GPUs @ ARM

ARM Mali Graphics Processor Generations

ARMMALI
Visual Technology

BIFROST
- Mali-G71 GPU
- …

Unified shader cores, scalar ISA, clause execution, full coherency, Vulkan, OpenCL

MIDGARD
- Mali-T600 GPU series
- Mali-T700 GPU series
- Mali-T800 GPU series

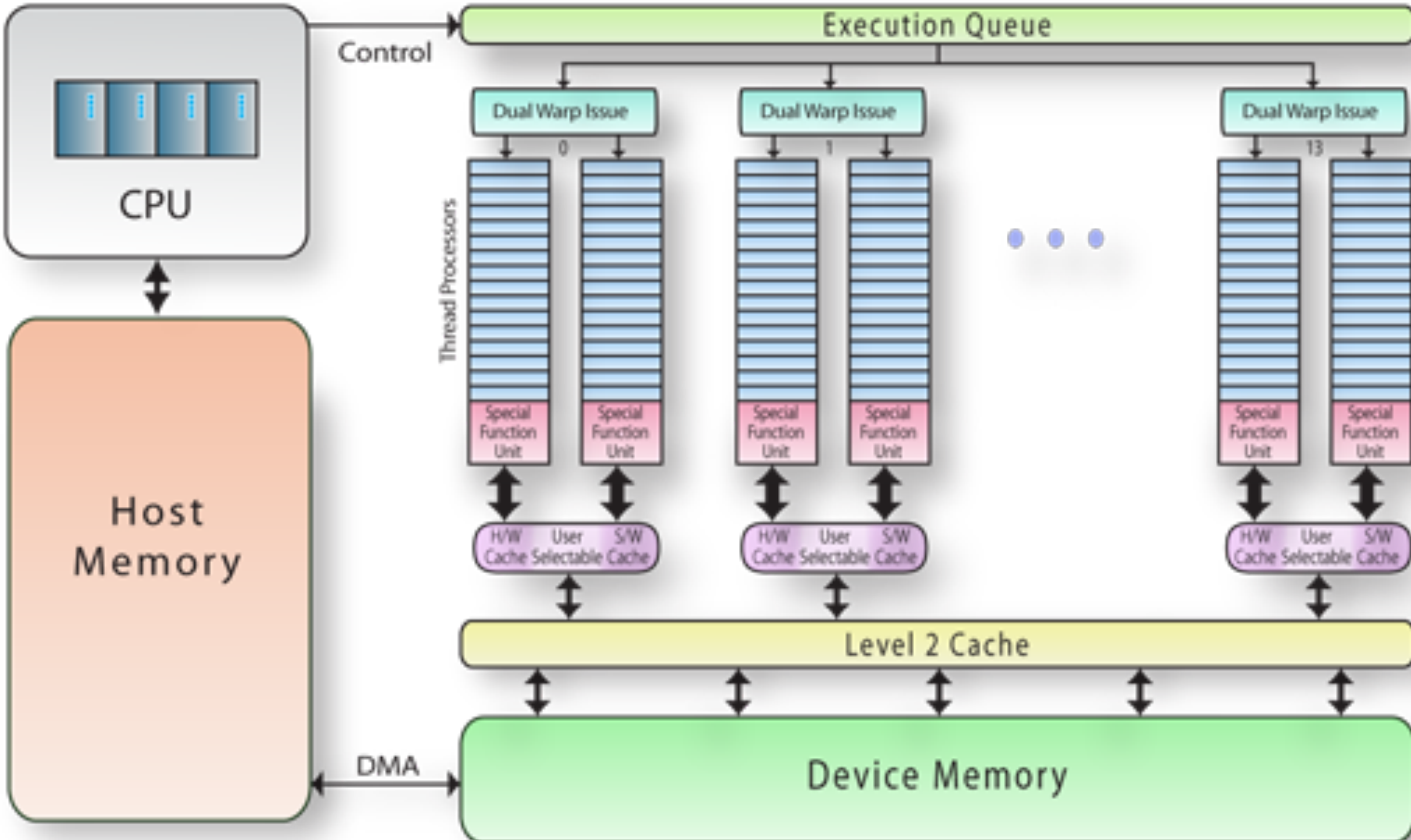
Unified shader cores, SIMD ISA, OpenGL ES 3.x, OpenCL, Vulkan

UTGARD
- Mali-200 GPU
- Mali-300 GPU
- Mali-400 GPU
- Mali-450 GPU
- Mali-470 GPU

Separate shader cores, SIMD ISA, OpenGL ES 2.x

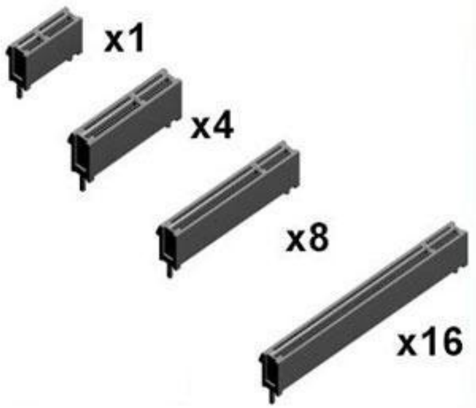
HIGH-LEVEL OPERATIONAL VIEW

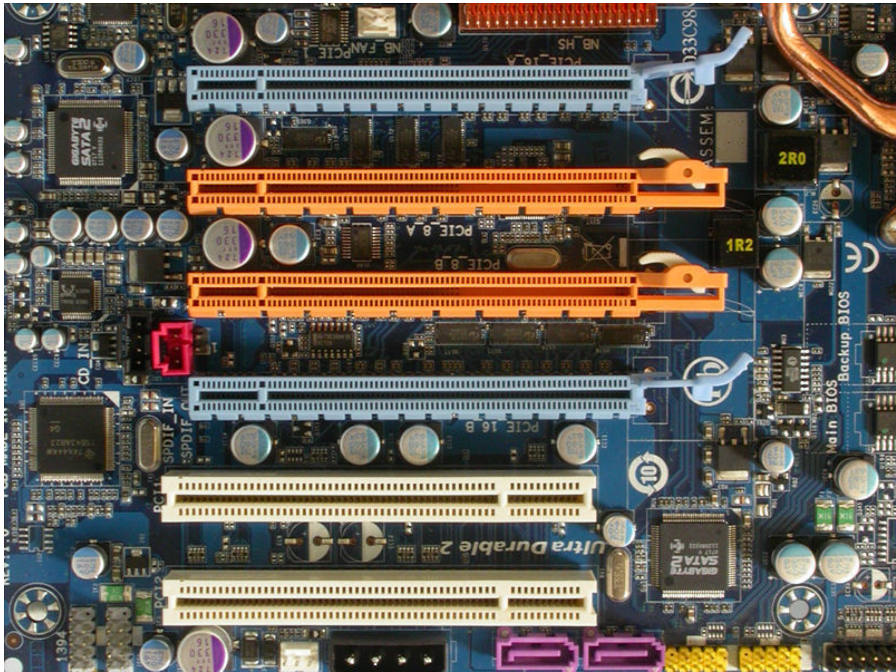
A GPU Architecture



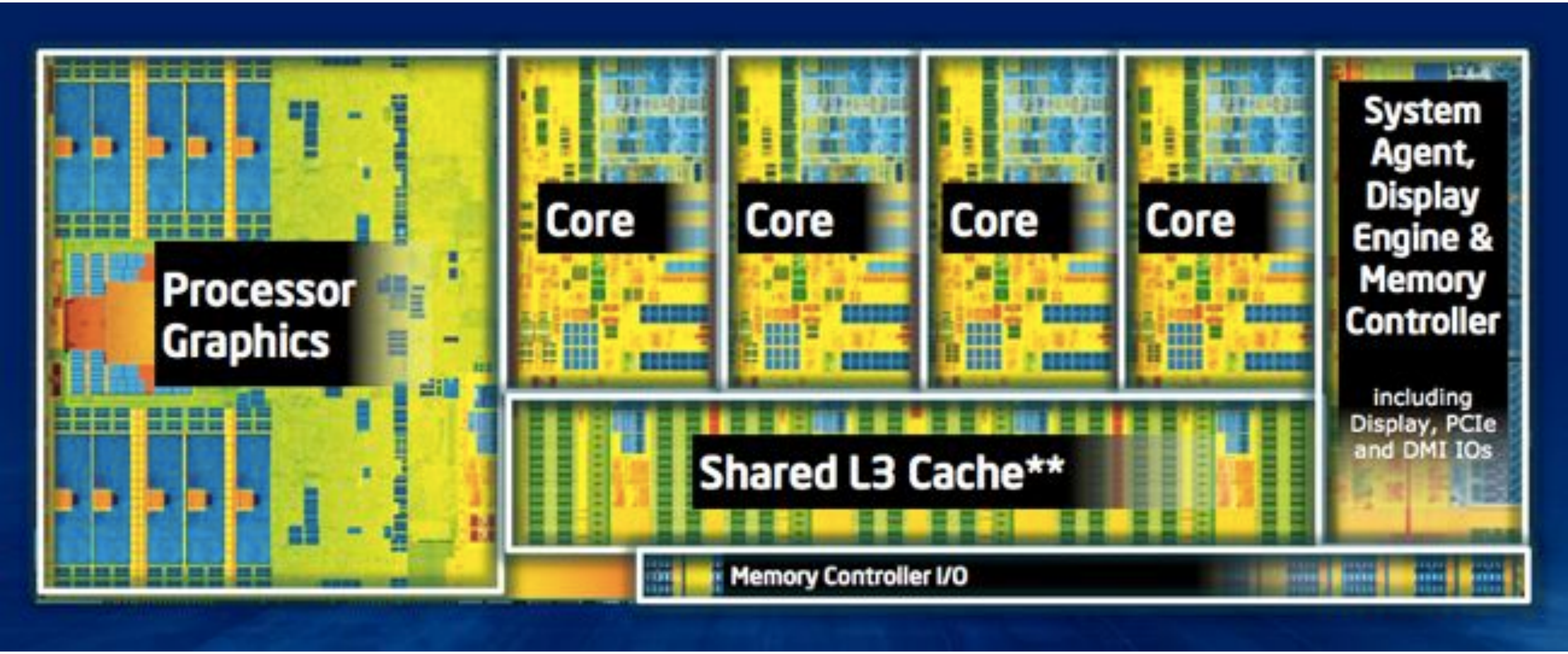
Integration into host system

- Typically PCI Express 2.0
- Theoretical speed 8 GB/s
  - Effective ≤ 6 GB/s
  - In reality: 4 – 6 GB/s
- V3.0 recently available
  - Double bandwidth
  - Less protocol overhead

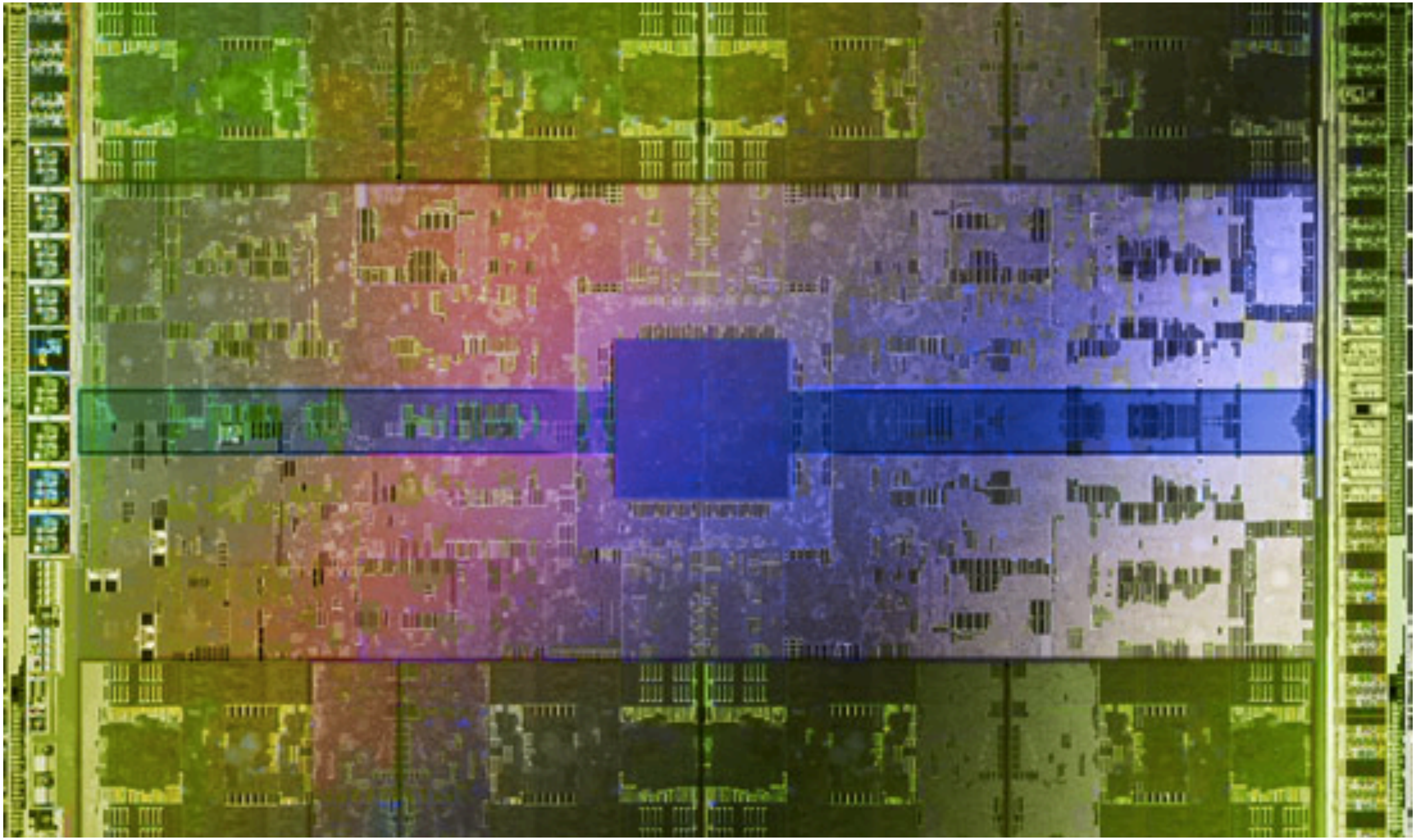




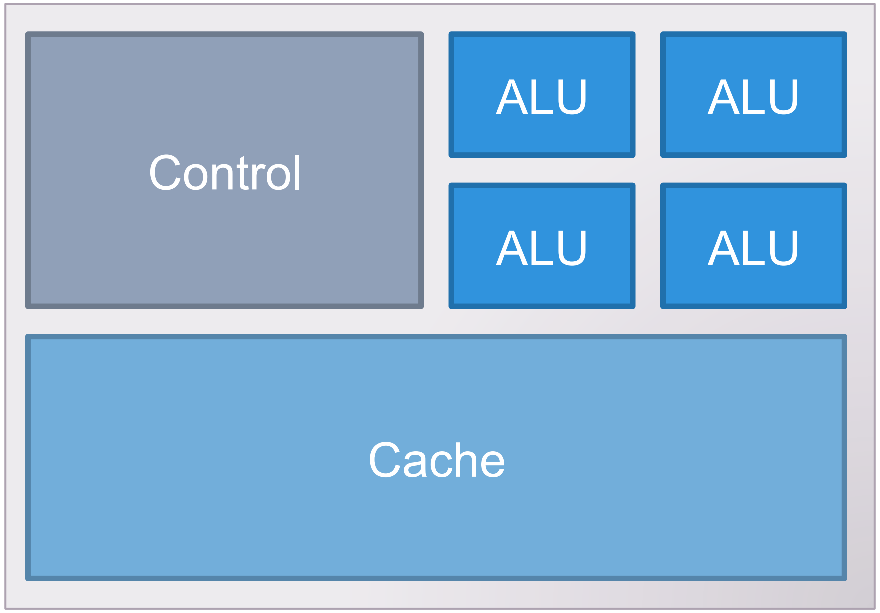
A CPU die


Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O

A GPU die: Fermi



CPU vs. GPU



CPU

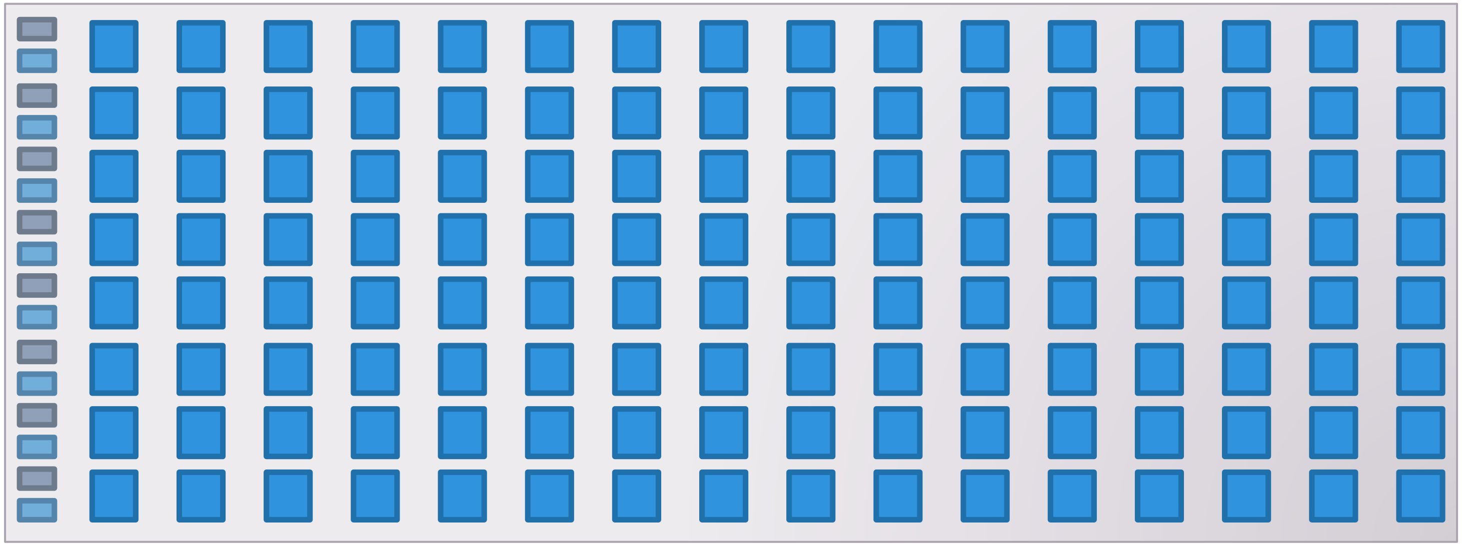
Few complex cores

Lots of on-chip memory

Lots of control logic

GPU

many simple cores, little memory, little control



Why so different?

- Different goals produce different designs!
  - CPU must be good at everything
  - GPUs focus on massive parallelism
    - Less flexible, more specialized
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, etc.)
  - multithreading can hide latency => no big caches
  - share control logic across many threads

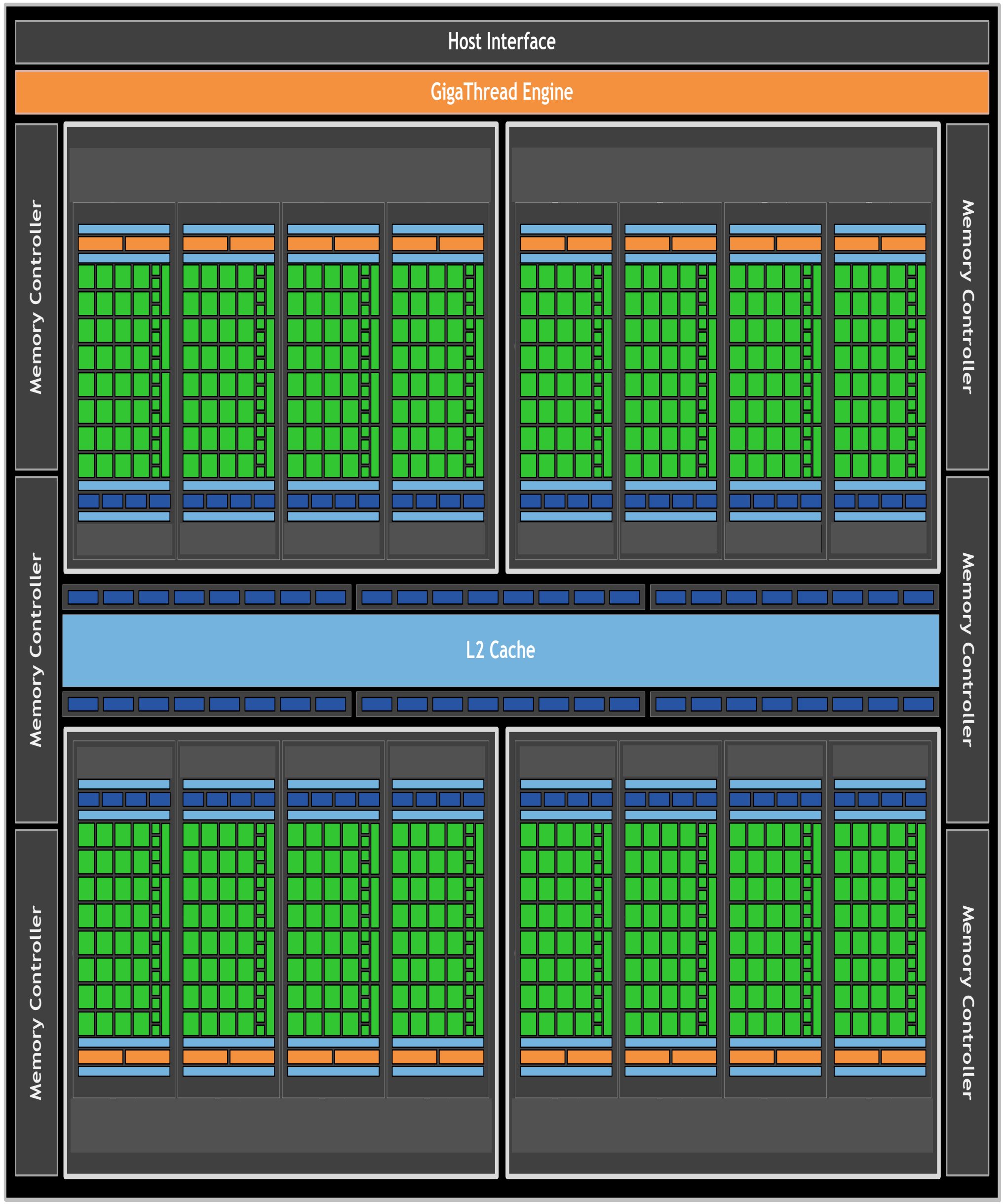
CPU vs. GPU

- Movie
- The Mythbusters
  - Jamie Hyneman & Adam Savage
  - Discovery Channel
- Appearance at NVIDIA's NVISION 2008

NVIDIA GPUS ARCHITECTURE

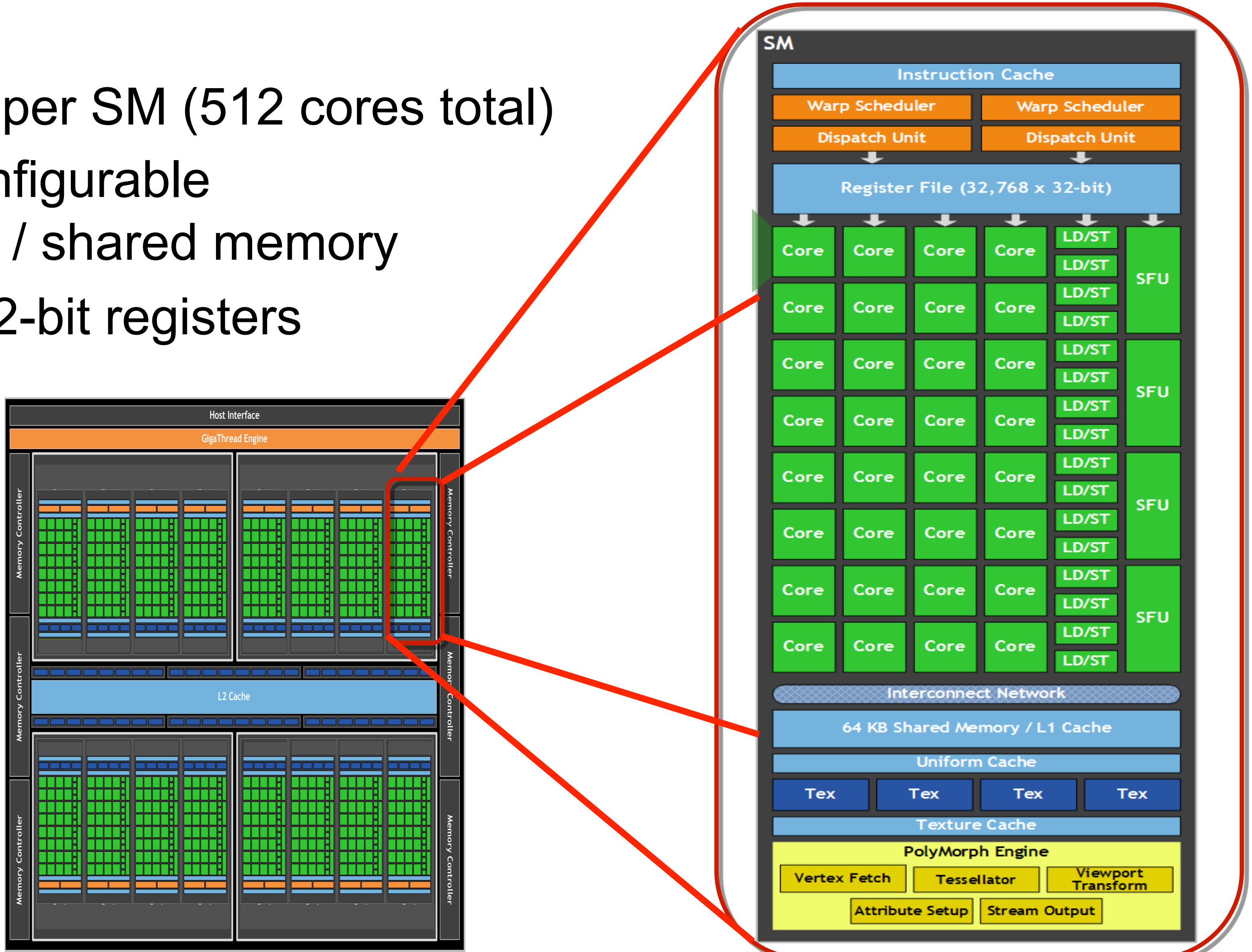
Fermi

- Consumer: GTX 480, 580
- HPC: Tesla C2050
  - More memory, ECC
  - 1.0 Tlop SP
  - 515 GFlop SP
- 16 streaming multiprocessors (SM)
  - GTX 580: 16
  - GTX 480: 15
  - C2050: 14
- SMs are independent
- 768 KB L2 cache



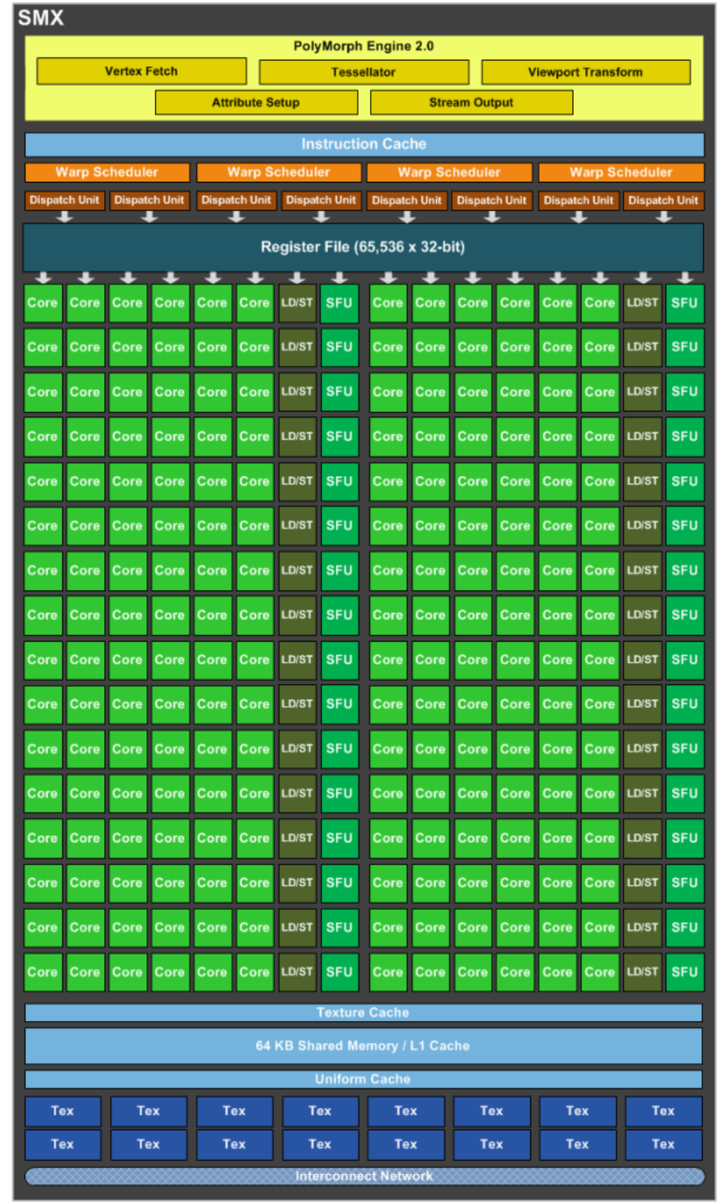
Fermi Streaming Multiprocessor (SM)

- 32 cores per SM (512 cores total)
- 64KB configurable
  L1 cache / shared memory
- 32,768 32-bit registers



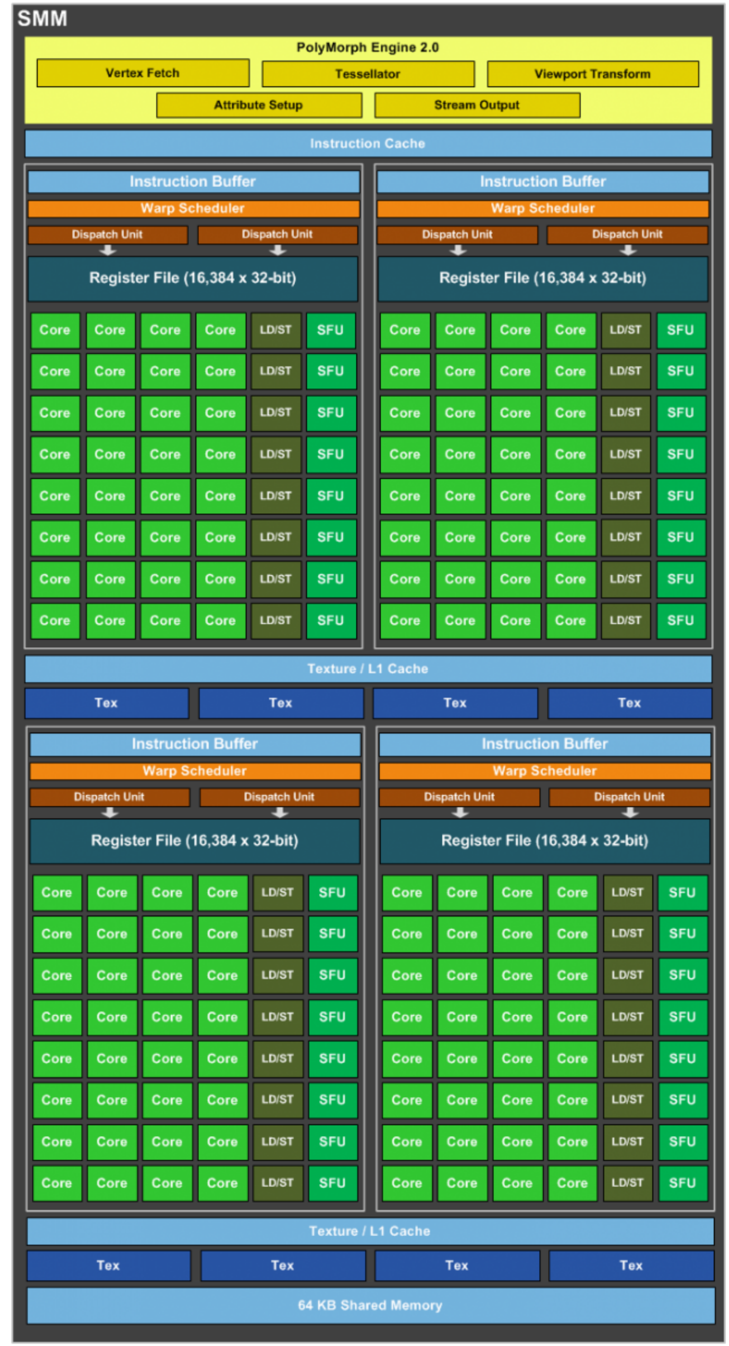
Kepler: SMX

- Consumer:
  - GTX680, GTX780, GTX-Titan
- HPC
  - Tesla K10..K40, K80
- SMX features
  - 192 CUDA cores
    - 32 in Fermi
  - 32 Special Function Units (SFU)
    - 4 for Fermi
  - 32 Load/Store units (LD/ST)
    - 16 for Fermi
- 3x Perf/Watt improvement
- 4x more texture memory



Maxwell: SMM

- Consumer:
  - GTX 970, GTX 980, …
- HPC:
  - Tesla M40
- SMM Features:
  - 4 subblocks of 32 cores
  - Dedicated L1/LM per 64 cores
  - Dispatch/decode/registers per 32 cores
- L2 cache: 2MB (~3x vs. Kepler)
- 40 texture units
- Lower power consumption



Pascal: SMP

- 64 single-precision (FP32) CUDA Cores.
  - Maxwell = 128
  - Kepler = 192
- Support for FP16
- 32 DP32 cores
- Independent L1
- Shared memory
  - 64KB/SM
- 4MB of shared L2

Evolution in numbers

GPU / Form Factor	Kepler GK110 / PCIe	Maxwell GM200 / PCIe	Pascal GP100 / SXM2	Pascal GP100 / PCIe
SMs	15	24	56	56
FP32 CUDA Cores / SM	192	128	64	64
FP32 CUDA Cores / GPU	2880	3072	3584	3584
FP64 CUDA Cores / SM	64	4	32	32
FP64 CUDA Cores / GPU	960	96	1792	1792
Base Clock	745 MHz	948 MHz	1328 MHz	1126 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1303 MHz
Single precision GFLOPS	5040	6844	10608	9340
Double precision GFLOPS	1680	213	5304	4670

Evolution in numbers

GPU / Form Factor	Kepler GK110 / PCIe	Maxwell GM200 / PCIe	Pascal GP100 / SXM2	Pascal GP100 / PCIe
Texture Units	240	192	224	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	3072-bit HBM2 (12GB) 4096-bit HBM2 (16GB)
Memory Bandwidth	288 GB/s	288 GB/s	732 GB/s	549 GB/s (12GB) 732 GB/s (16GB)
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	12 GB or 16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts	250 Watts
Transistors	7.1 billion	8 billion	15.3 billion	15.3 billion
GPU Die Size	551 mm^2	601 mm^2	610 mm^2	610 mm^2
Manufacturing Process	28-nm	28-nm	16-nm	16-nm

PROGRAMMING GPUS IN CUDA

Kernel = the parallel program

Device code = manage the parallel program

(NVIDIA) GPUs

- Architecture
  - Many (100s) slim cores
  - Sets of (32 or 192) cores grouped into "multiprocessors" with shared memory
    - SM(X) = stream multiprocessors
  - Work as accelerators
- Memory
  - Shared L2 cache
  - Per-core caches + shared caches
  - Off-chip global memory
- Programming
  - Symmetric multi-threading
  - Hardware scheduler

GPU Parallelism

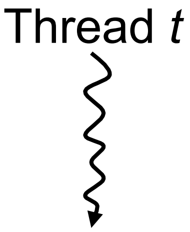
- Data parallelism (fine-grain)
- **SIMT** (Single Instruction **Multiple Thread**) execution
  - Many threads execute concurrently
    - Same instruction
    - Different data elements
    - HW automatically handles divergence
  - Not same as SIMD because of multiple register sets, addresses, and flow paths*
- Hardware multithreading
  - HW resource allocation & thread scheduling
    - Excess of threads to hide latency
    - Context switching is (basically) free

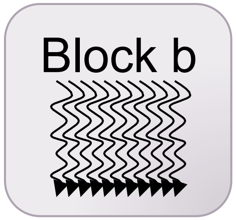
*http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html

CUDA

- CUDA: Compute Unified Device Architecture
  - C/C++ extensions
    - Other wrappers exist
- Straightforward mapping onto hardware
  - Hierarchy of threads (map to cores)
    - Configurable at logical level
  - Various memory spaces (map to physical mem. spaces)
    - Usable via variable scopes
- SIMT: single instruction multiple threads
  - Have 1000s threads running concurrently
  - Hardware multi-threading
    - GPU threads are lightweight

CUDA: Hierarchy of threads

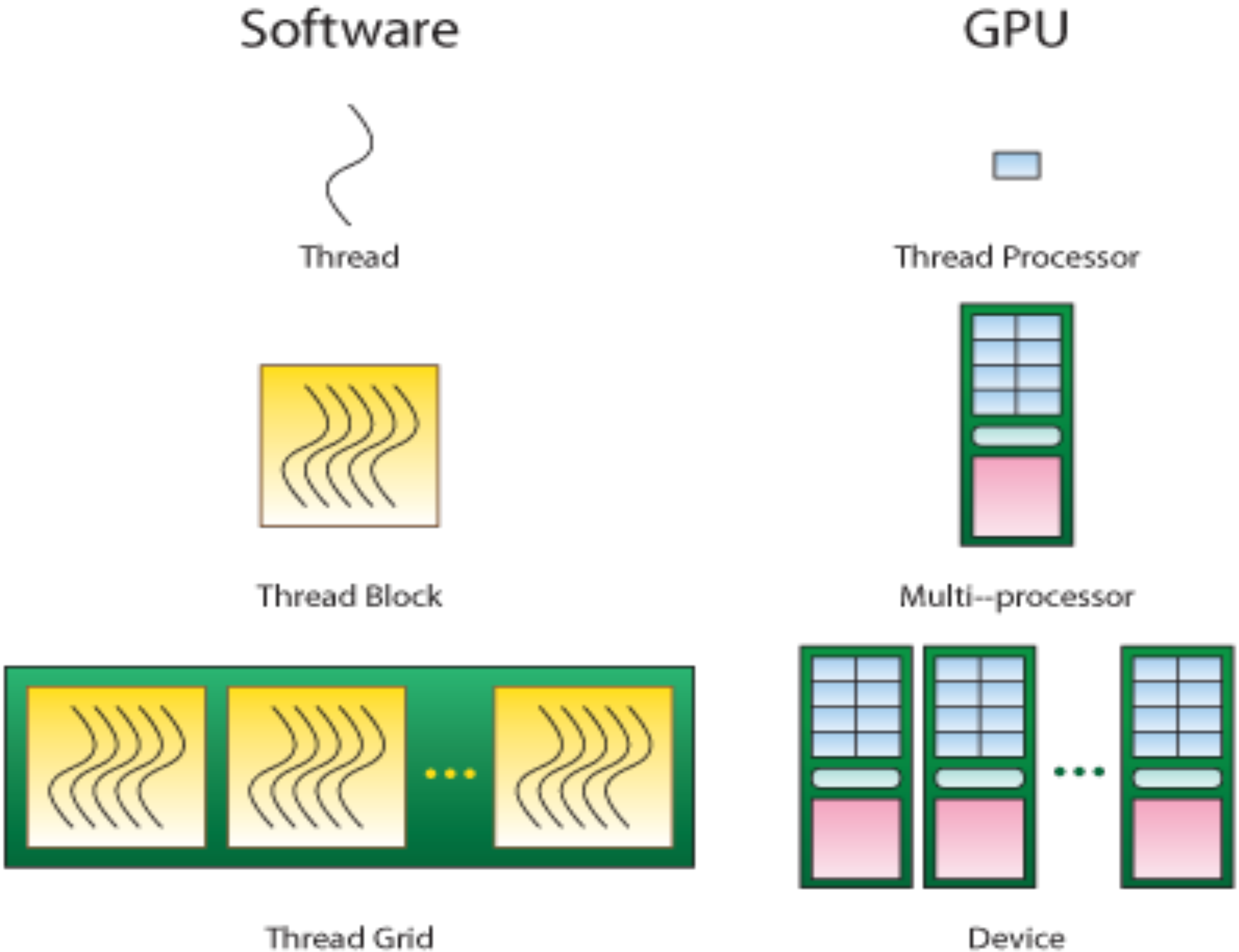




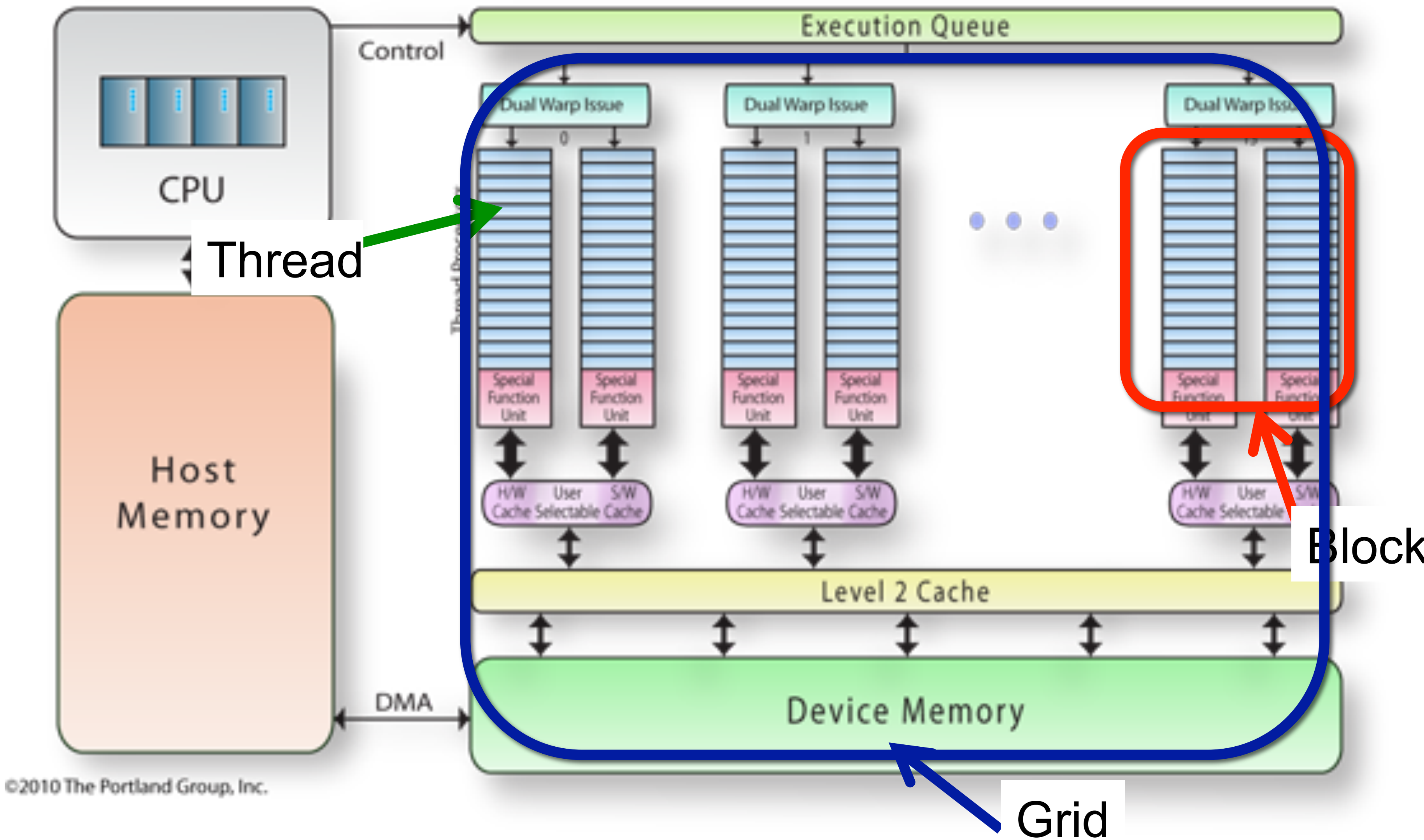
- Each thread executes the kernel code
  - One thread runs on one CUDA core
- Threads are logically grouped into thread blocks
  - Threads in the same block can cooperate
  - Threads in different blocks cannot cooperate
- All thread blocks are logically organized in a Grid
  - 1D or 2D or 3D
  - Threads and blocks have unique IDs

- A grid specifies in how many instances the kernel is being run

CUDA Model of Parallelism



Hierarchy of threads

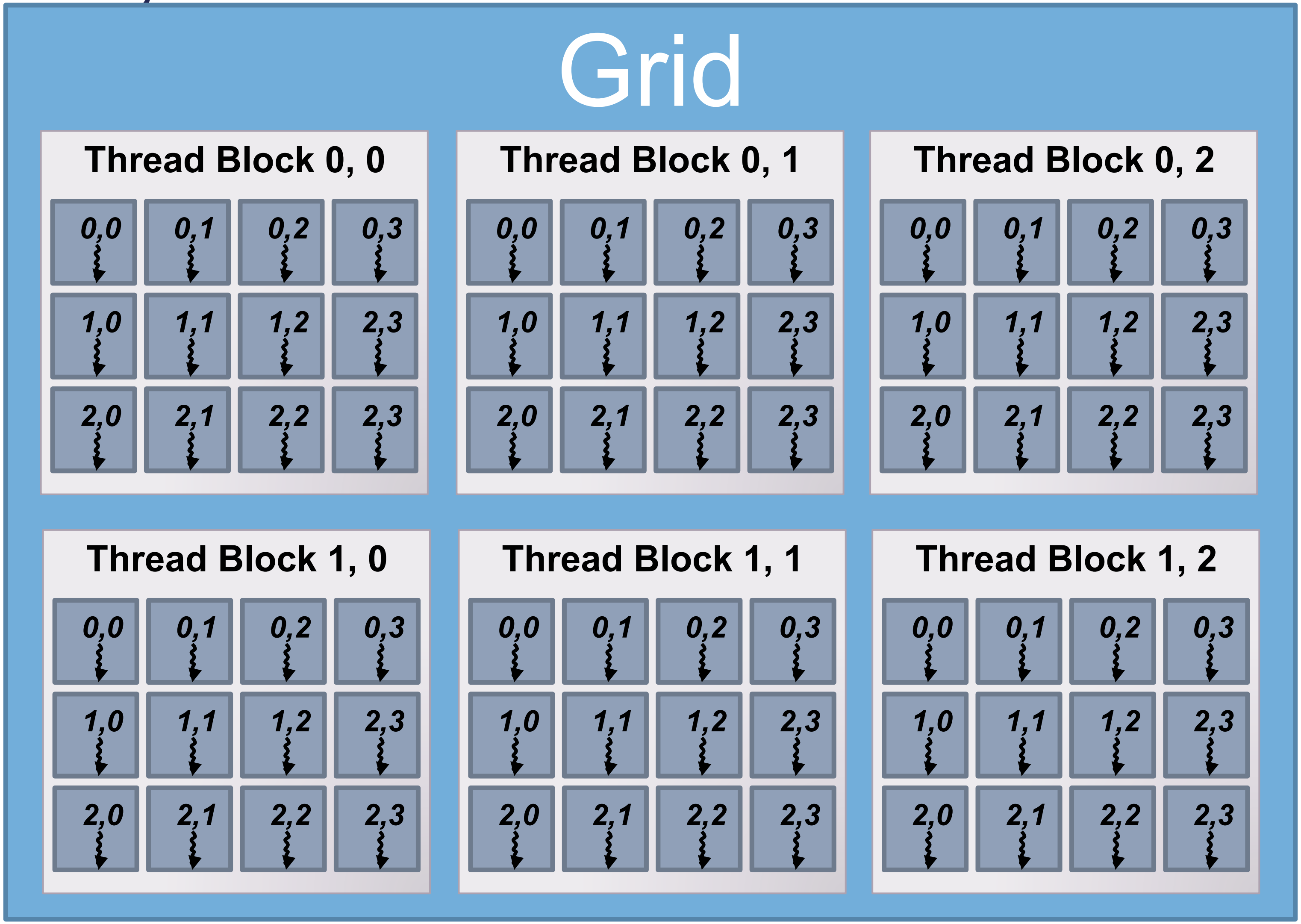


CUDA Model of Parallelism

- CUDA virtualizes the physical hardware
  - A block is a virtualized streaming multiprocessor
    - threads, shared memory
  - A thread is a virtualized scalar processor
    - registers, PC, state

- Execution model:
  - Threads execute in warps (32 threads per warp)
    - Called “wavefronts” by AMD (64 threads)
  - **All threads in a warp execute the same code**
    - On different data
  - Blocks = multiple warps
    - Scheduled **independently** on the same **SM**

Grids, Thread Blocks and Threads

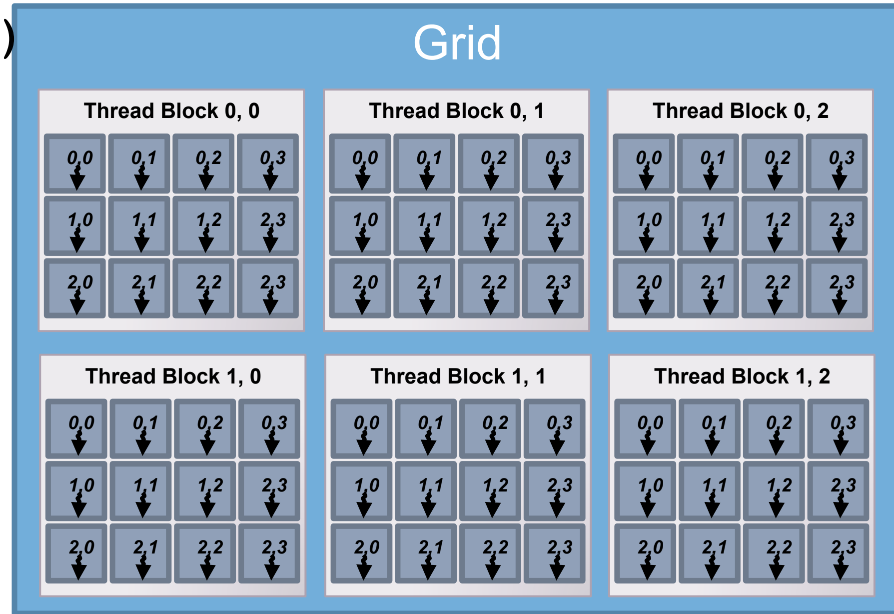


Kernels and grids

- Launch kernel (12 x 6 = 72 instances)

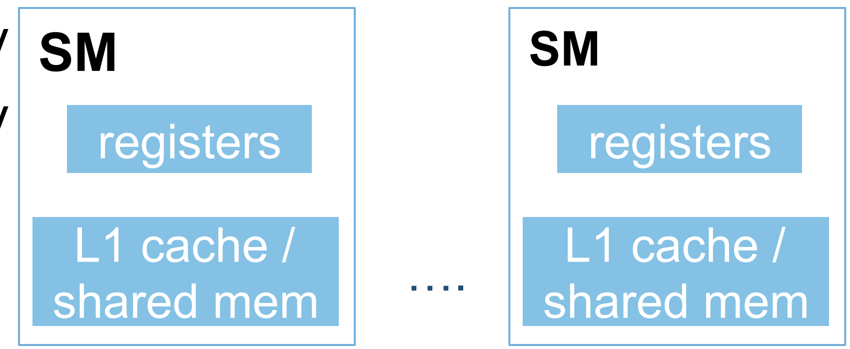
```
myKernel<<<numBlocks,threadsPerBlock>>>(…);
```

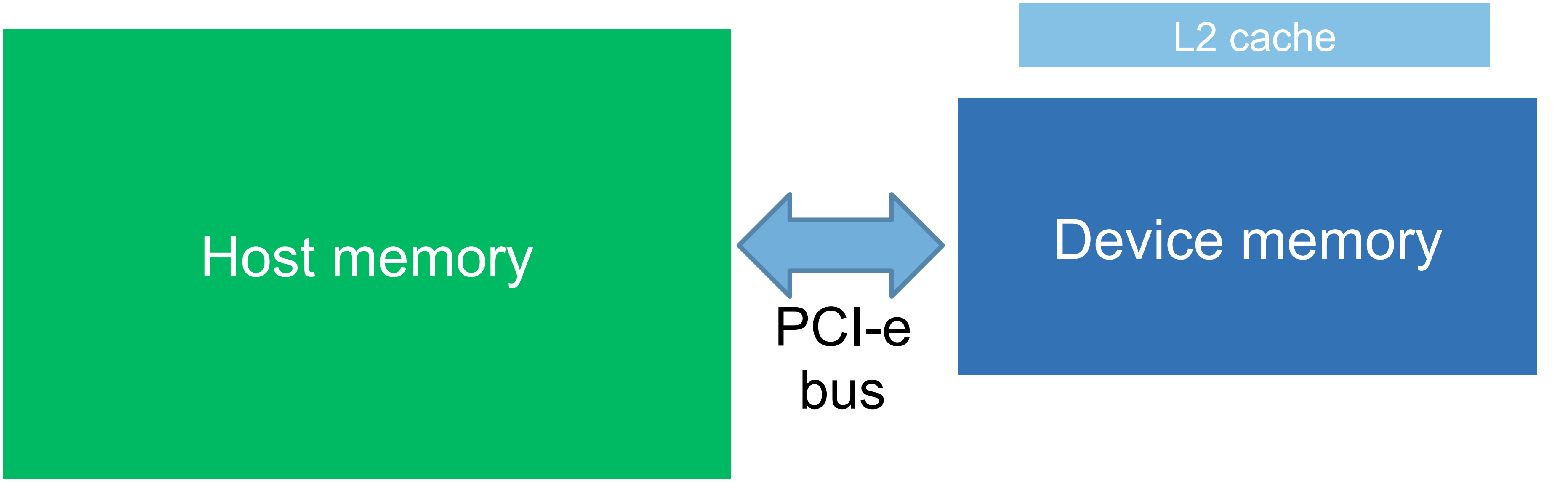
  - dim3 threadsPerBlock(3,4);
    - threadsPerBlock.x = 3
    - threadsPerBlock.y = 4
    - Each thread: (threadIdx.x, threadIdx.y)
  - dim3 numBlocks(2,3);
    - blockDim.x = 2
    - blockDim.y=3
    - Each block : (blockIdx.x,blockIdx.y)



Memory architecture (since Fermi)

- Configurable L1 cache per SM
  - 16KB L1 cache / 48KB Shared memory
  - 48KB L1 cache / 16KB Shared memory
- Shared L2 cache
- Global memory





Device (Global) Memory

- CPU and GPU have separate memory spaces
  - Data is moved across PCI-e bus
  - Use functions to allocate/set/copy memory on GPU
  - Very similar to corresponding C functions
- Pointers are just addresses
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa

Additional memories

- Textures
  - Read-only
  - Data resides in device memory
  - Different read path, includes specialized caches
- Constant memory
  - Data resides in device memory
  - Manually managed
  - Small (e.g., 64KB)
  - Assumes all threads in a block read the same addresses
    - Serializes otherwise

ECC (Error-Correcting Code)

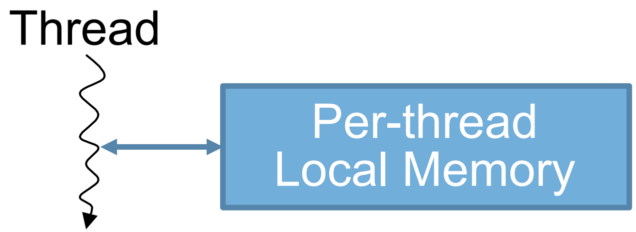
- Expensive hardware mechanism for memory error protection
- ECC is a must have for many computing applications

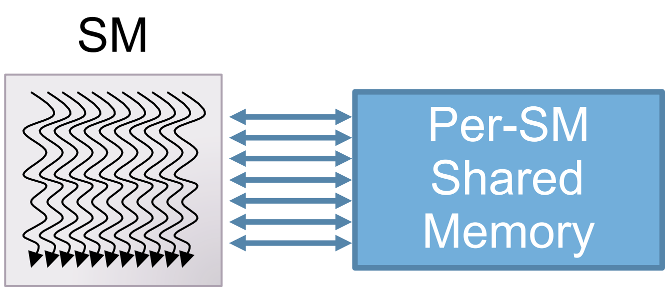
On GPUs:

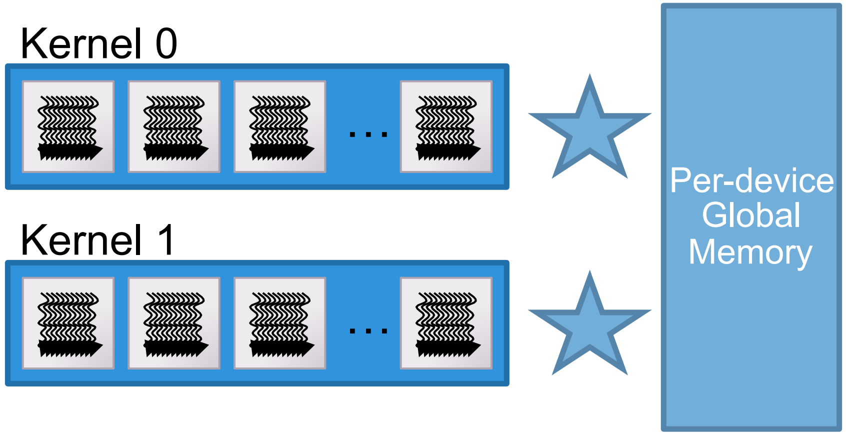
- All major internal memories are ECC protected
  - Register file, L1 cache, L2 cache
- DRAM protected by ECC on Tesla only
  - Some newer generations of GTX cards feature this too
  - Protection can be enabled/disabled

Multiple Device Memory Scopes

- Per-thread private memory
  - Each thread has its own local memory
  - Stacks, other private data, **registers**
  - Accessible to a single thread only
- Per-SM shared memory
  - Small memory close to the processor, low latency
  - Accessible to threads in the same block.
- Device memory
  - GPU frame buffer
  - Accessible to any thread







Memory spaces: Registers

Example:

```
__global__ void aKernel(float *C, float *A, float *B) {
    int tx = threadIdx.x; //local variable in registers
    float local_sum[4]; //small compile-time sized array
in registers
```

Registers:

- Thread-local scalars or small constant size arrays are stored as registers
- Implicit in the programming model
- Behavior is very similar with local variables
- Not persistent: kernel ends, data is lost

Memory spaces: Global memory

Example:

```
__global__ void matmul_kernel(float *C, //C points to global memory
                              float *A, //A points to global memory
                              float *B) //B points to global memory
```

Global memory

- **Allocated by the host program using** cudaMalloc()
- **Initialized by the host program using** cudaMemcpy()**or previous kernels**
- Persistent = the values are retained between kernels
- Not coherent, writes by other threads might not be visible until kernel has finished

Memory spaces: Constant

Example

```
__constant__ float speed_of_light= 0.299792458; //scalars can be initialized directly
__constant__ float2 vertices[NUM_VERTICES]; //initialized by a host function
__global__ void cn_pnpoly(uint8_t* bitmap, float2* points, intn) {
...
for (intj=0; j<NUM_VERTICES; k = j++) {
float2 vj= vertices[j]; //index j does not depend on threadIdx
```

Constant memory:

- **Statically defined by the host program using** __constant__**qualifier**
- Defined as a global variable, visible only within the same translation unit
- **Initialized by the host program using** cudaMemcpyToSymbol()
- **Read-only to the GPU, cannot be accessed directly by the host**
- **Values are cached in a special cache optimized for broadcast access by multiple threads simultaneously, access should not depend on** threadIdx

Memory spaces: Shared

Example:

```
__global__ void matmul_kernel(float *C, float *A, float *B) {
__shared__ float sh_A[tile_size][tile_size]; //2D array in shared memory
for (k = 0; k < WIDTH; k += tile_size) {
__syncthreads(); //wait for all threads in the block
sA[ty][tx] = A[y*WIDTH + k + tx]; //fill shared memory with values
__syncthreads(); //wait again
```

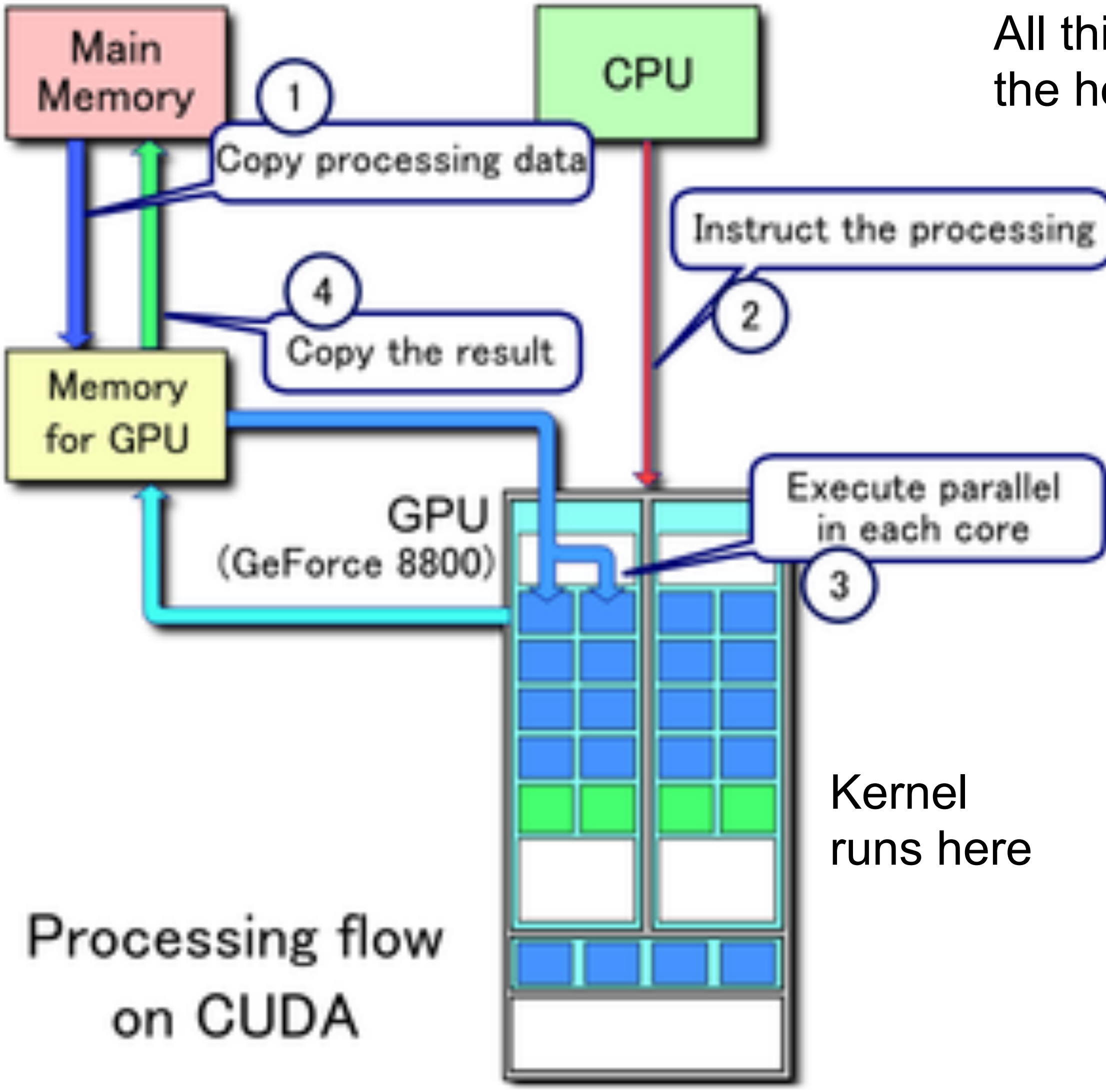
Shared memory

- Variables have to be declared using __shared__qualifier, size known at compile time
- In the scope of thread block, all threads in a thread block see the same piece of memory
- Not initialized, threads have to fill shared memory with meaningful values
- Not persistent, after the kernel has finished, values in shared memory are lost
- Not coherent, __syncthreads()is required to make writes visible to other threads within the thread block

Using CUDA

- Two parts of the code:
  - Device code = GPU code = kernel(s)
    - Sequential program
    - Write for 1 thread, execute for all
  - Host code = CPU code
    - Instantiate grid + run the kernel
    - Memory allocation, management, deallocation
    - C/C++/Java/Python/…
- Host-device communication
  - Explicit / implicit via PCI/e
  - Minimum: data input/output

CUDA processing flow



All this happens from the host code.

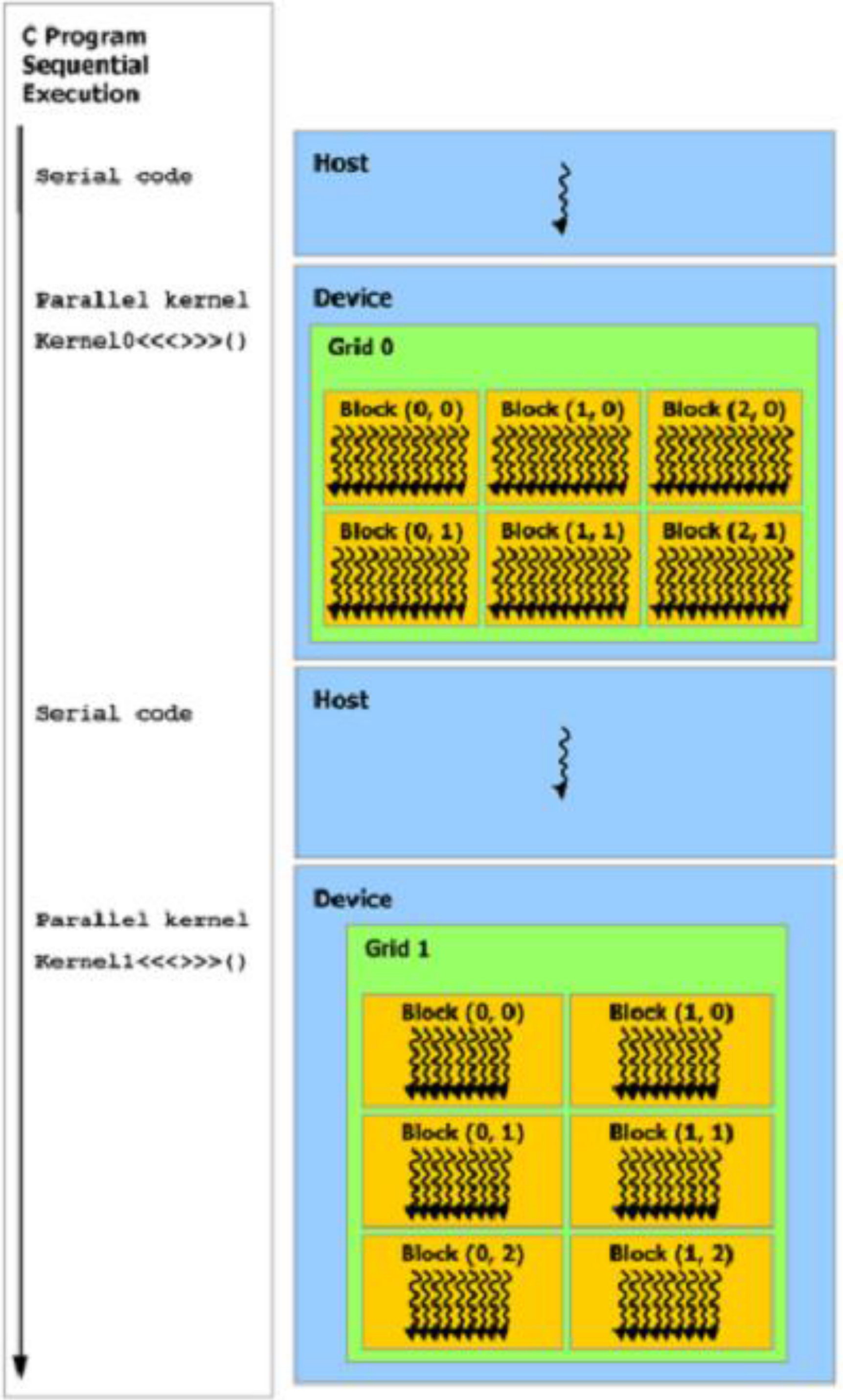
Kernel runs here

Image courtesy of Wikipedia

Execution flow

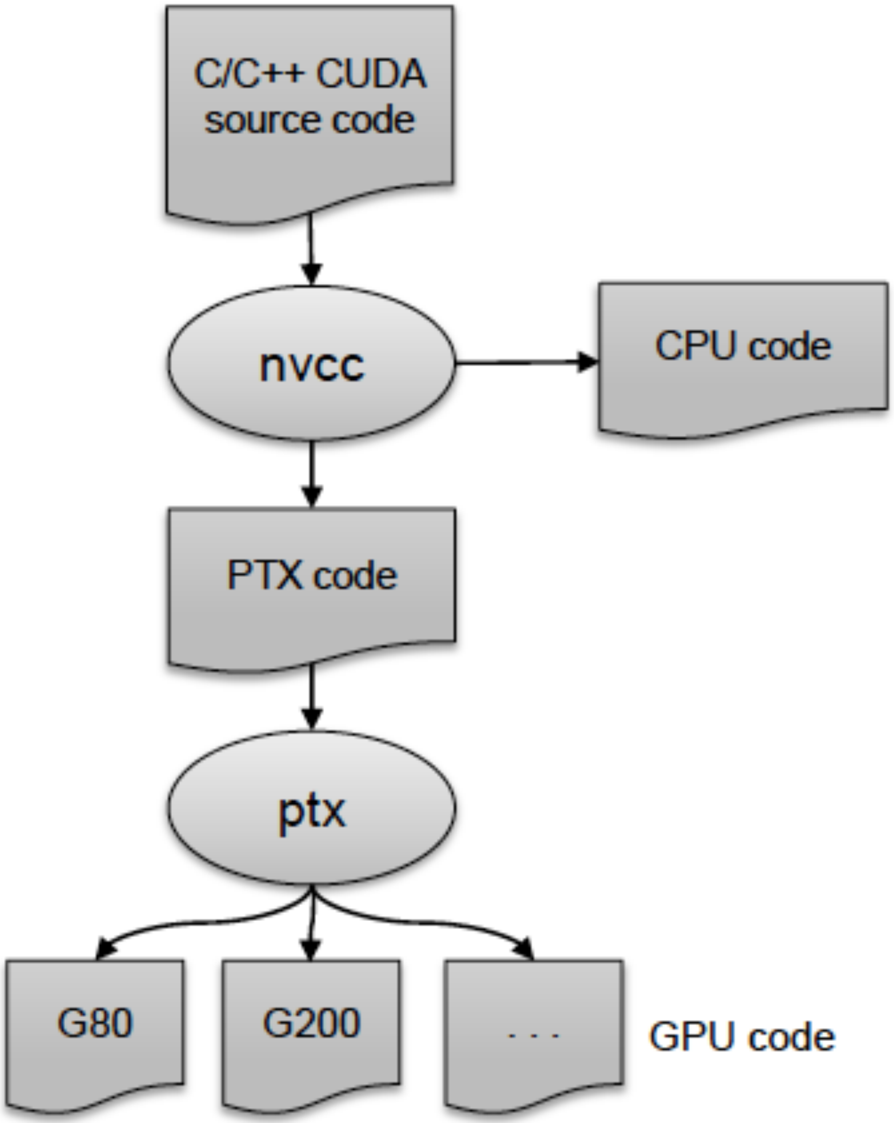
- Serial code executes on CPU
- Parallel code executes on GPU

1. Transfer data CPU→GPU
2. CPU calls GPU kernel
3. GPU kernel operates on device data
4. Transfer data GPU→CPU
5. [Repeat from 1]



Compiling CUDA

- Use nvcc
- nvcc separates source code:
  - **device code** (runs on GPU)
    - further processed by NVIDIA compiler
  - **host code** (runs on CPU)
    - further processed by host compile (g++, cl.exe)



CUDA: kernels and launch

- Function qualifiers:

```
__global__ void my_kernel() { }
__device__ float my_device_func() { }
```

- Execution configuration:

```
dim3 gridDim(100, 50);  // 5000 thread blocks
dim3 blockDim(4, 8, 8); // 256 threads per block (1.3M
  total)
my_kernel <<<gridDim, blockDim>>> (...); // Launch kernel
```

- Built-in variables and functions valid in device code:

```
dim3 gridDim;   // Grid   dimension
dim3 blockDim;  // Block  dimension
dim3 blockIdx;  // Block  index
dim3 threadIdx; // Thread index

void syncthreads(); // Thread synchronization
```

CUDA: memory allocation/release

- All memory buffers – CPU **and** GPU must be allocated
- Host (CPU) manages device (GPU) memory:
  - **`cudaMalloc(void **pointer, size_t nbytes)`**
  - **`cudaMemset(void *pointer, int val, size_t count)`**
  - **`cudaFree(void* pointer)`**

CUDA: Data Copies

```
cudaMemcpy(void *dst, void *src,
           size_t nbytes,
           enum cudaMemcpyKind direction);
```

- blocks CPU thread until all bytes have been copied
- doesn't start copying until previous CUDA calls complete

- **`enum {`**
```
  cudaMemcpyHostToDevice,
  cudaMemcpyDeviceToHost,
  cudaMemcpyDeviceToDevice
  } cudaMemcpyKind
```

- Non-blocking copies are also available
  - **`cudaMemcpyAsync`**
  - DMA transfers, overlap computation and communication

CUDA: dummy example

```
int n = 1024;
int nbytes = n * sizeof(int);
int* dataCPU = (int *)malloc(nbytes);
int* dataGPU;

cudaMalloc(&dataGPU, nbytes);
cudaMemset(dataGPU, 0, nbytes);

cudaMemcpy(dataGPU, dataCPU, nbytes,
                        cudaMemcpyHostToDevice);
myKernel<<<n/128,128>>>(n, dataGPU);
cudaMemcpy(dataCPU, dataGPU, nbytes,
                        cudaMemcpyDeviceToHost);
cudaFree(dataGPU);
free(dataCPU);
```

EXAMPLE: VECTOR-ADD

Programming many-cores

= parallel programming:

- Choose/design algorithm
- Parallelize algorithm
  - Expose enough layers of parallelism
  - Minimize communication, synchronization, dependencies
  - Overlap computation and communication
- Implement parallel algorithm
  - Choose parallel programming model
  - (?) Choose many-core platform
- Tune/optimize application
  - Understand performance bottlenecks & expectations
  - Apply platform specific optimizations
  - (?) Apply application & data specific optimizations

First CUDA program

- Determine mapping of operations and data to threads
- Write kernel(s)
  - Sequential code
  - Written per-thread
- Determine block geometry
  - Threads per block, blocks per grid
  - Number of grids (>= number of kernels)
- Write host code
  - Memory initialization and copying to device
  - Kernel(s) launch(es)
  - Results copying to host
- Optimize the kernels

Vector add: sequential

```
void vector_add(int size, float* a, float* b, float* c) {
    for(int i=0; i<size; i++) {
        c[i] = a[i] + b[i];
    }
}
```

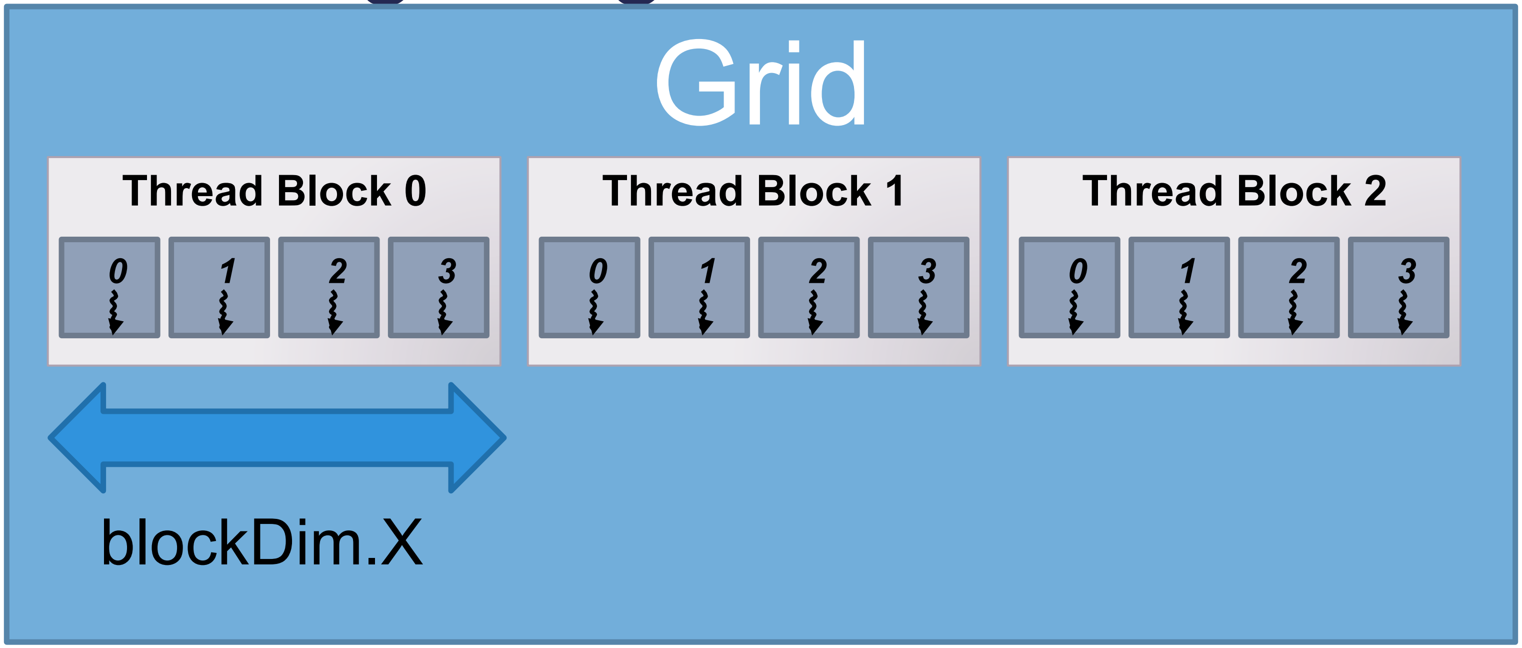
How do we parallelize this?

- What does each thread compute?
  - One addition per thread
  - Each thread deals with *different* elements
  - How do we know which element?
    - Compute a mapping of the grid to the data
      - Any mapping will do!

Vector add: Kernel

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = ?
    C[i] = A[i] + B[i];
}
```

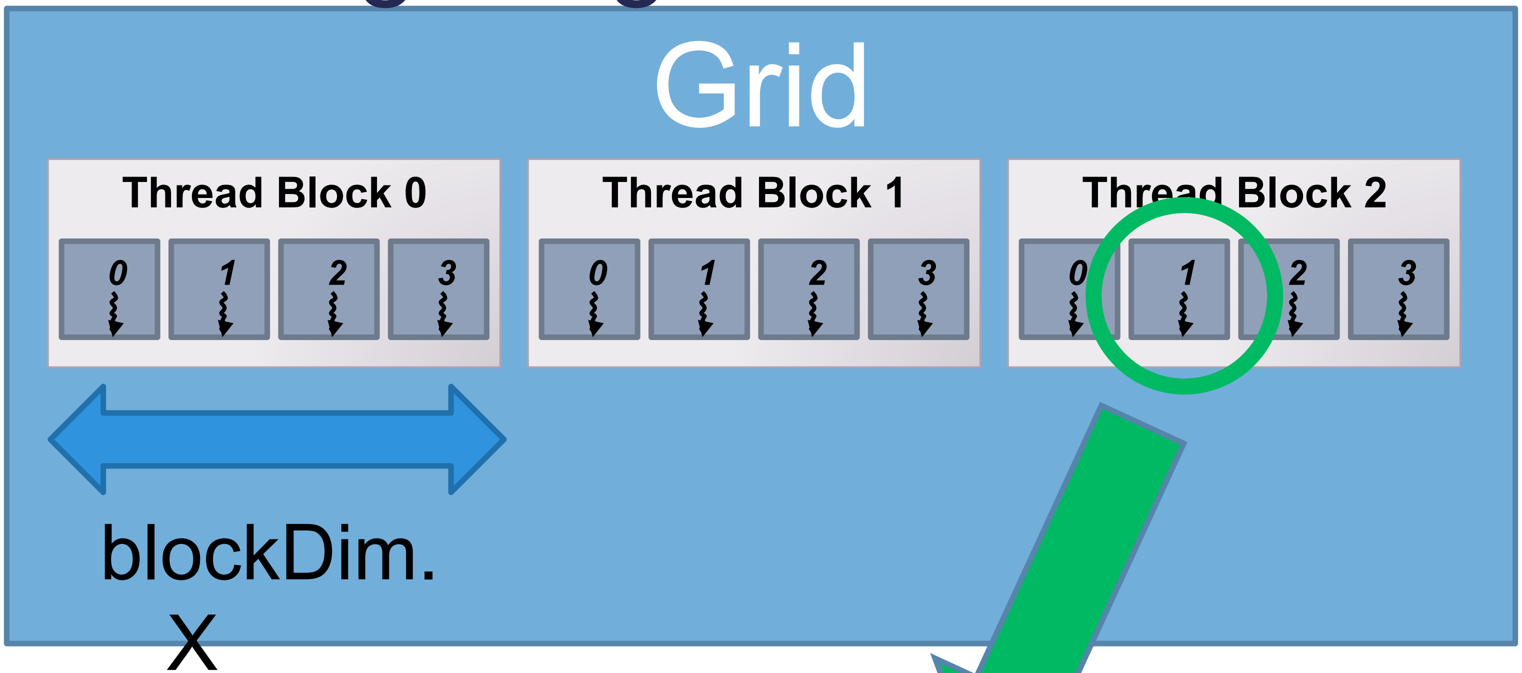
Calculating the global thread index



- "global" thread index:

```
blockDim.x * blockIdx.x + threadIdx.x;
```

Calculating the global thread index



- "global" thread index:

```
blockDim.x * blockIdx.x + threadIdx.x;
```

```
    4       *      2       +      1      = 9
```

Vector add: Kernel

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Done with the kernel!

Vector add: Launch kernel

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

GPU code

```
int main() {
   // initialization code here ...
  N = 5120;
   // launch N/256 blocks of 256 threads each
   vector_add<<< N/256, 256 >>>(deviceA, deviceB, deviceC);
   // cleanup code here ...
}
```

Host code

(can be in the same file)

Vector add: Launch kernel

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

GPU code

What if N = 5000?

Host code

```
int main() {
  // initialization code here ...
 N = 5000;
  // launch N/256 blocks of 256 threads each
  vector_add<<< N/256, 256 >>>(deviceA, deviceB, deviceC);
  // cleanup code here …}
```

(can be in the same file)

Vector add: Launch kernel

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i<N) C[i] = A[i] + B[i];
}
```

GPU code

What if N = 5000?

```
int main() {
  // initialization code here ...
  N = 5000;
  // launch N/256 blocks of 256 threads each
  vector_add<<< N/256+1,256 >>>(deviceA, deviceB,
deviceC);
  // cleanup code here … }
```

Host code

(can be in the same file)

Vector add: Host

```
int main(int argc, char** argv) {
  float *hostA, *deviceA, *hostB, *deviceB, *hostC,
*deviceC;
  int size = N * sizeof(float);

  // allocate host memory
  hostA = malloc(size);
  hostB = malloc(size);
  hostC = malloc(size);

  // initialize A, B arrays here...

  // allocate device memory
  cudaMalloc(&deviceA, size);
  cudaMalloc(&deviceB, size);
  cudaMalloc(&deviceC, size);
```

Vector add: Host

```
 // transfer the data from the host to the device
 cudaMemcpy(deviceA, hostA, size,
cudaMemcpyHostToDevice);
 cudaMemcpy(deviceB, hostB, size,
cudaMemcpyHostToDevice);

 // launch N/256 blocks of 256 threads each
 vector_add<<<N/256, 256>>>(deviceA, deviceB,
deviceC);

 // transfer the result back from the GPU to the
host
 cudaMemcpy(hostC, deviceC, size,
cudaMemcpyDeviceToHost);
}
```

Done with the host code!