GPU COMPUTING

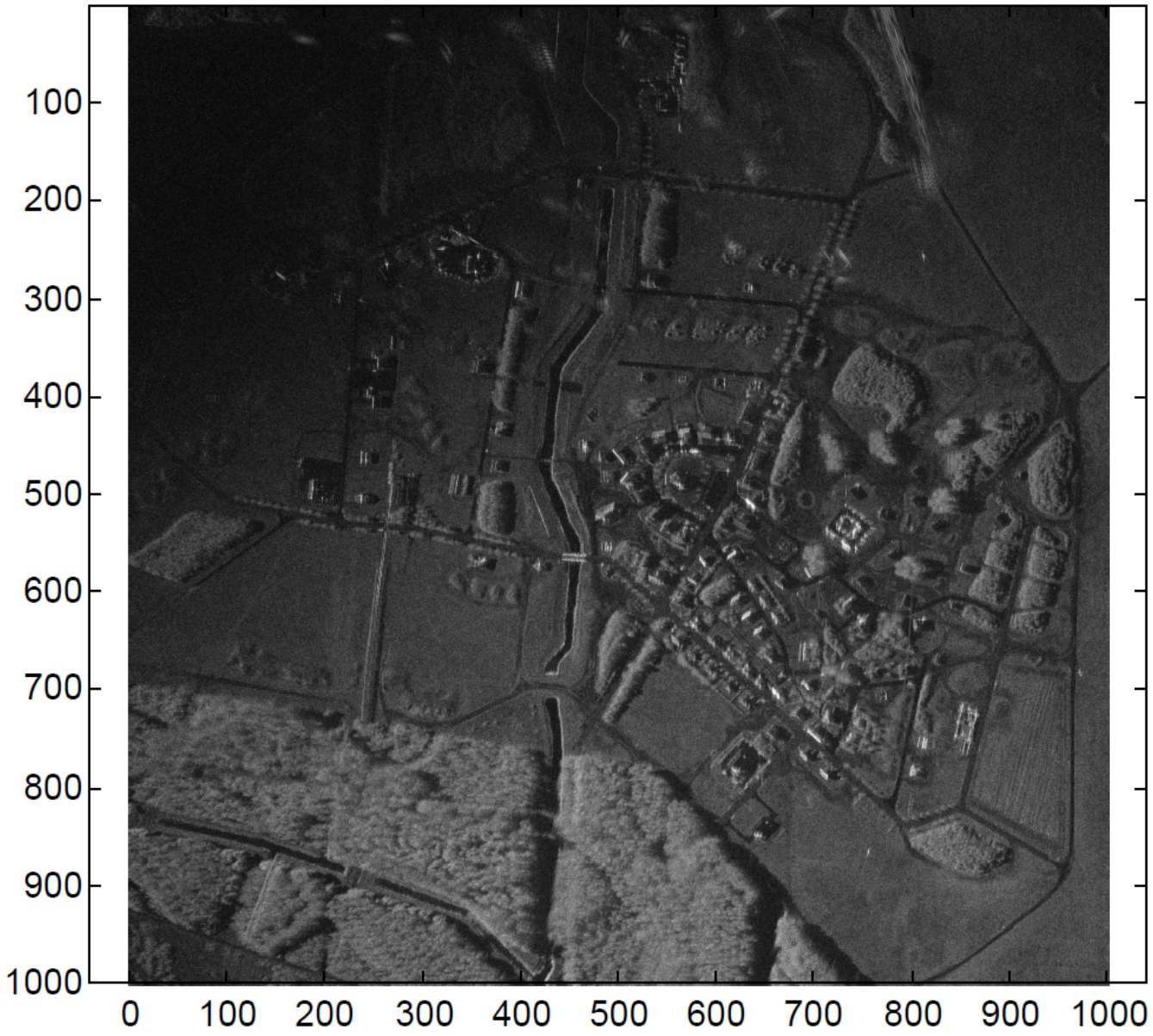
Part 1

Ana Lucia Varbanescu (UvA)

HPC computing

- Big Data, Big Simulation, Big Science
- Challenges
  - Compute and storage
  - Efficiency
    - Performance vs. Energy
- HPC focuses traditionally on performance, but now moving towards efficiency
  - Traditional HPC: complex machines, on-demand
  - Modern HPC: more and more based on existing machines, put together in dense clusters/datacenters
- **HPC is expanding to new application domains.**

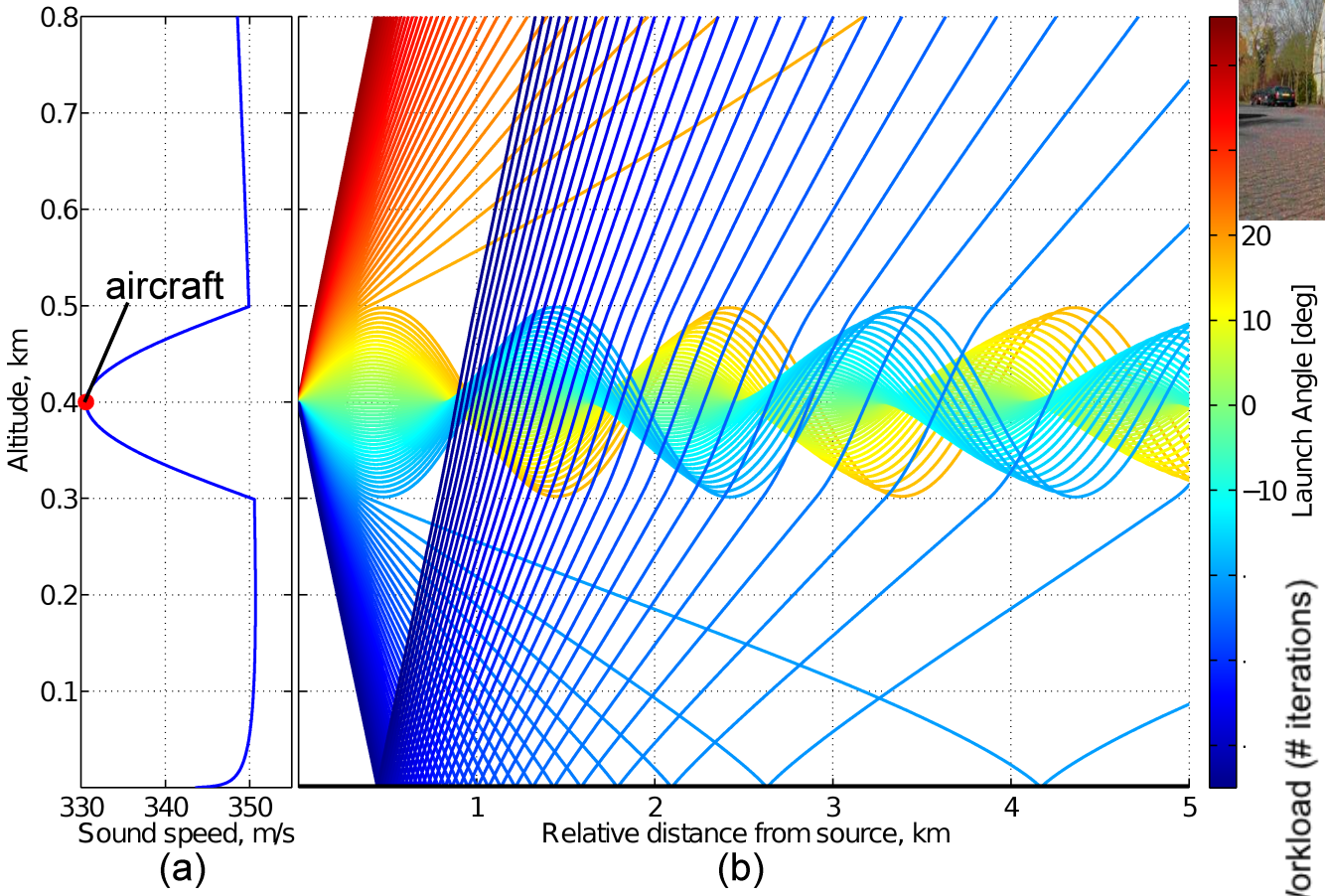
SAP imaging

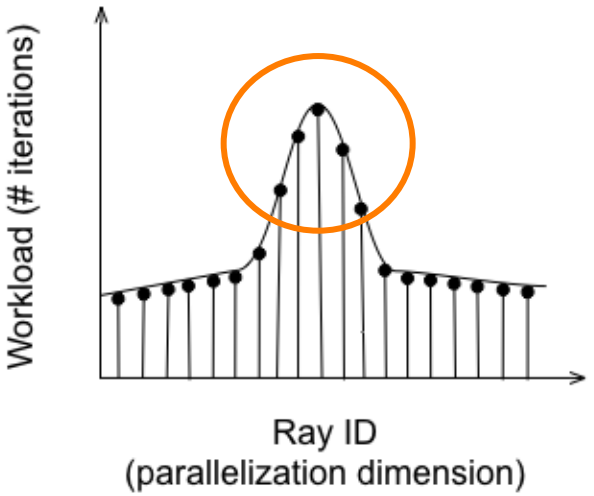

100
200
300
400
500
600
700
800
900
1000
0
100
200
300
400
500
600
700
800
900
1000

Sound ray tracing

- A collaboration with Dutch NLR
- Simulate the sound propagation
  - from an aircraft to receivers
- Assess aircraft flyover noise during the aircraft take-off and approach procedures

Sound ray tracing

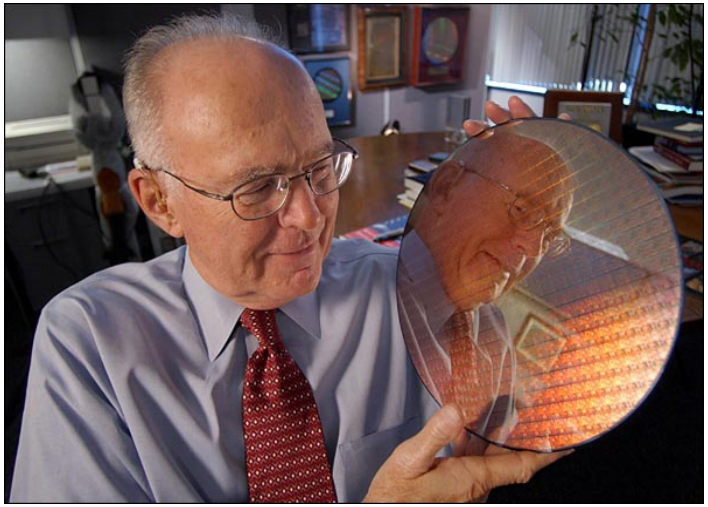


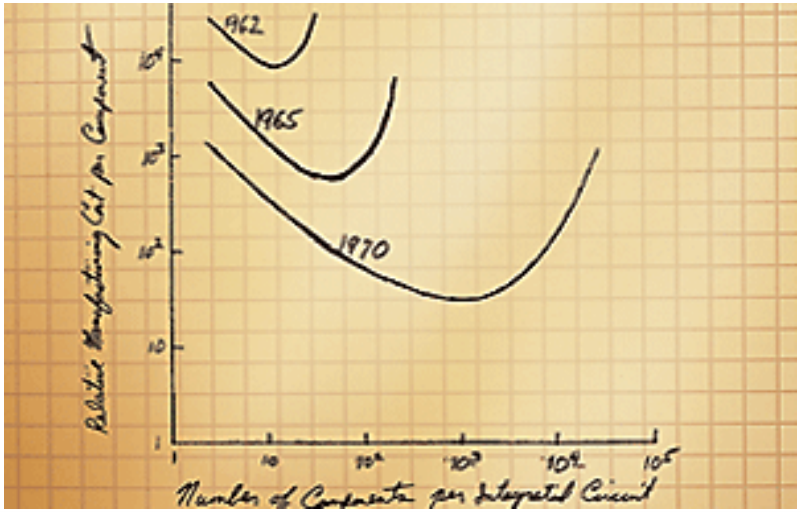


BRIEFLY ON HARDWARE

Moore's Law

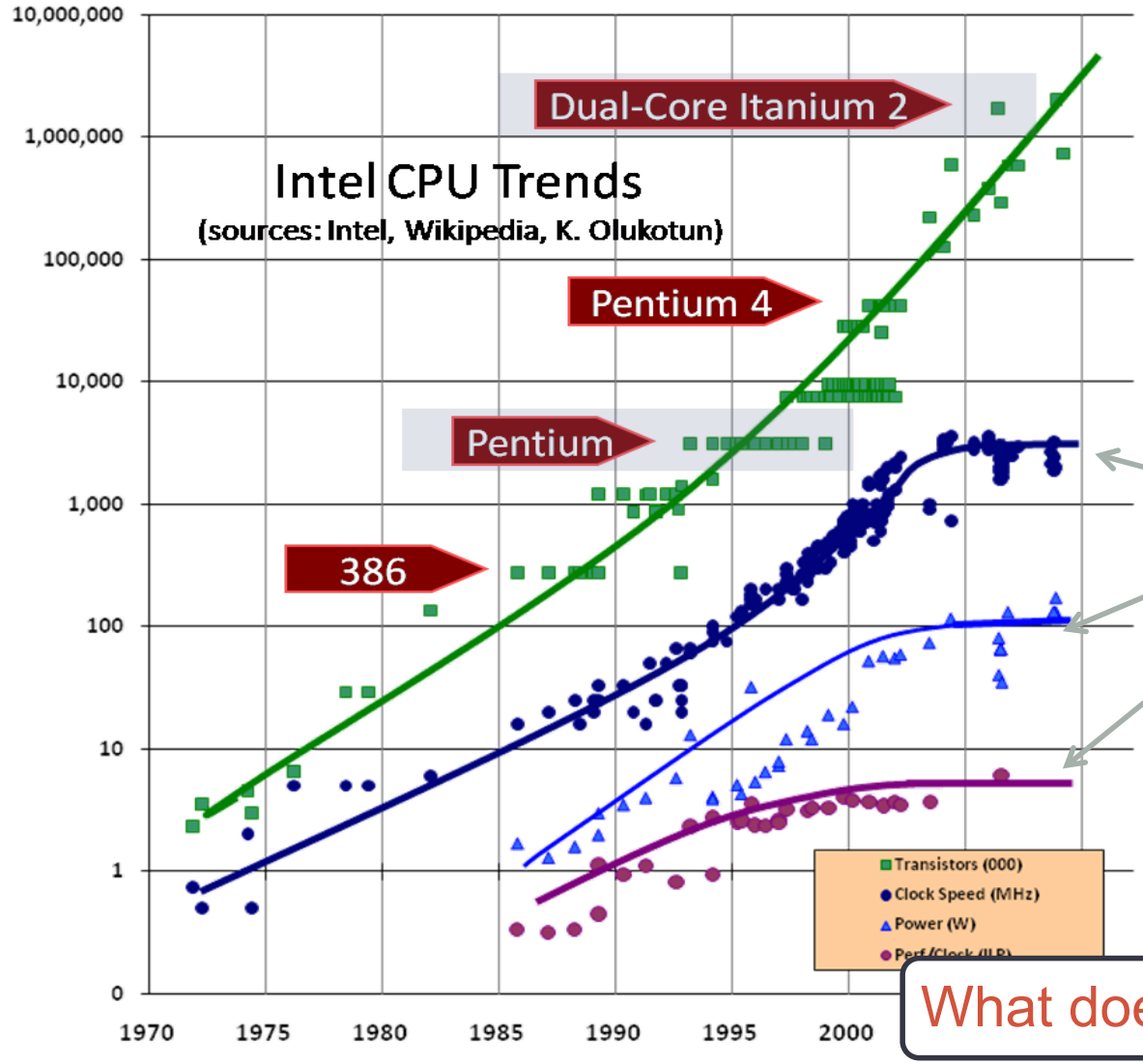
- Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.





"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase...." Electronics Magazine 1965

Evolution of processors



Chip density can increase about 2x every 2 years

BUT

- **Clock speed is not**
- **Power is not**
- **Instruction Level Parallelism is not**

What does this mean in practice?

New ways to use transistors

Improve PERFORMANCE by using parallelism on-chip: multi-core (CPUs) and many-core processors (GPUs).

Parallelism ⇔ HPC

- Parallelism is **mandatory** for **high performance**
  - Yesterday: **clusters (and grids)**
  - Today: **multi-/many-core processors**
  - Tomorrow: **massive multi-scale heterogeneous parallelism = clusters using different types of multi-/many-cores**

>95% of computing systems today are parallel!

Main challenges: learn to program parallel machines and learn to use them efficiently!

Why talk about GPUs?

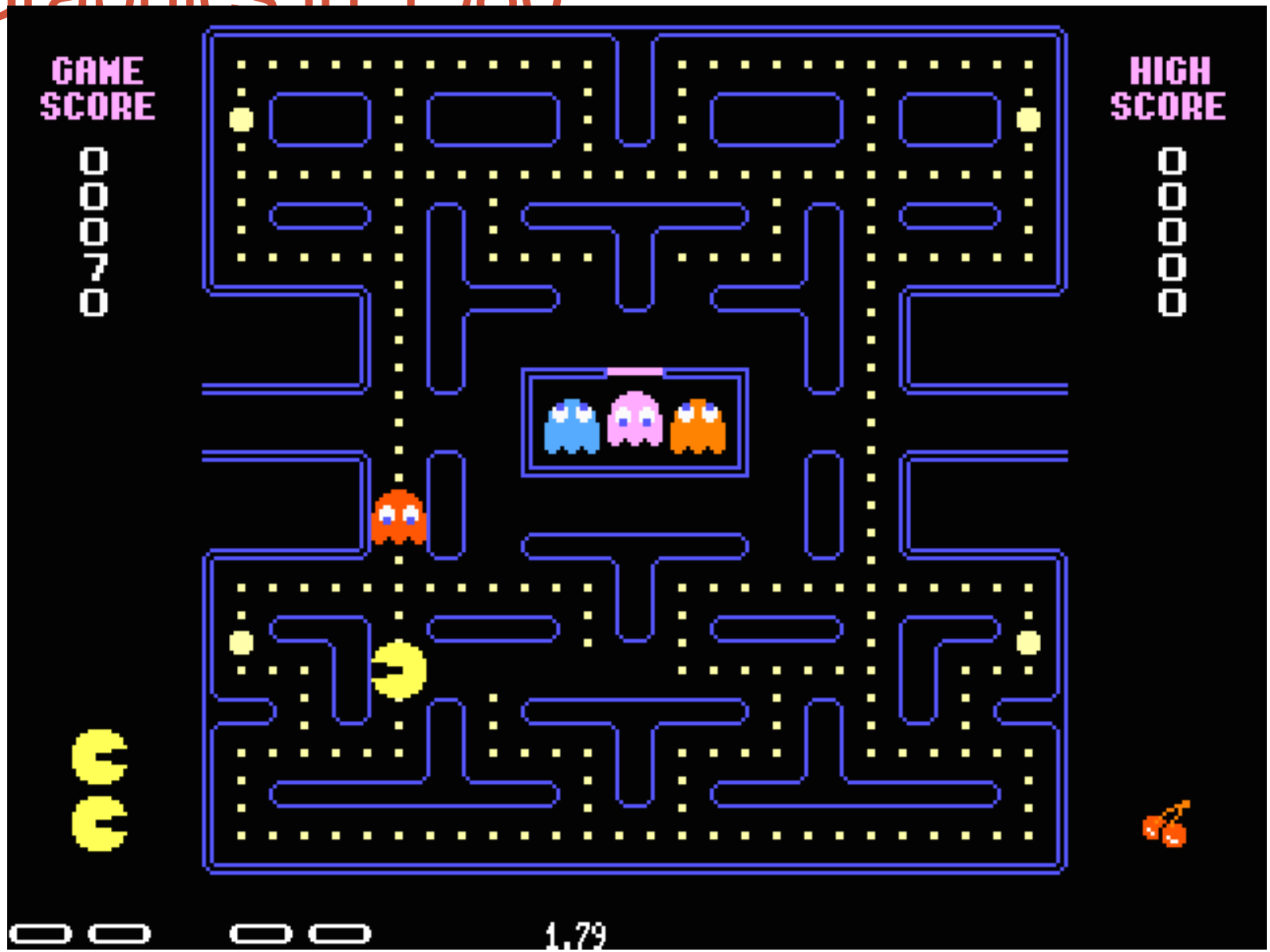
- GPUs are a steady market
  - **Gaming**
  - CAD-like activities
    - Traditional or not …
  - Visualisation
    - Scientific or not …

- GPUs are increasingly used for other types of applications
  - Number crunching in science, finance, image processing
  - (fast) Memory operations in big data processing

GPGPU ?!

Massive parallelism => massive performance

Graphics in 1980



Graphics in 2015

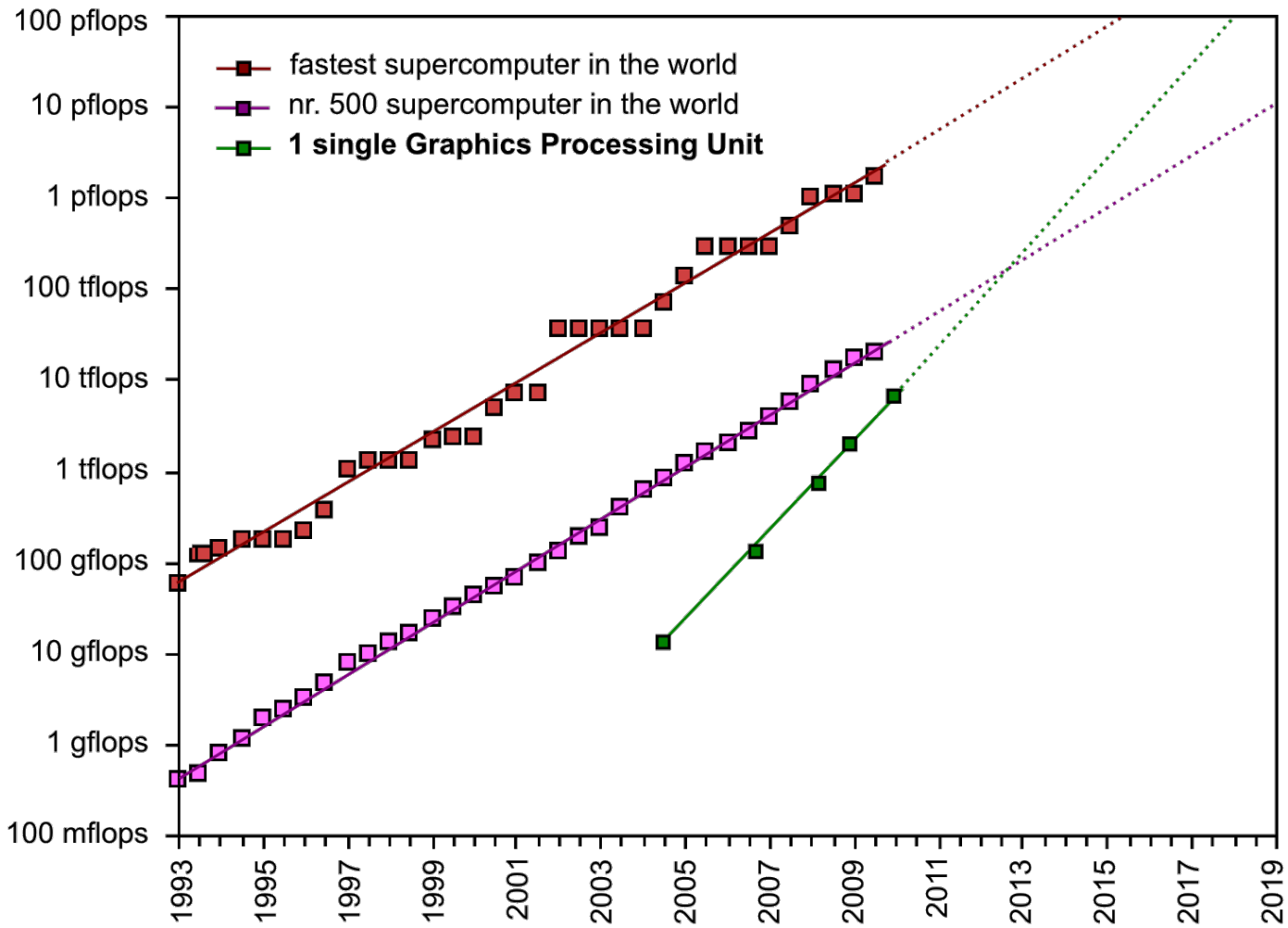
GPUs in movies

- From Ariel in Little Mermaid to Brave

Why GPUs?

- Promise of performance beyond most other architectures
  - CPUs
  - Multi-core CPUs
  - FPGAs
  - …
- They are power efficient
  - 2-5x better than a CPU
- What took us so long?
  - These things are not easy to program …

GPUs vs. supercomputers



TODO List

1. ~~Introduction~~
2. GPGPUs & hardware performance
3. CUDA & application performance
4. Advanced CUDA

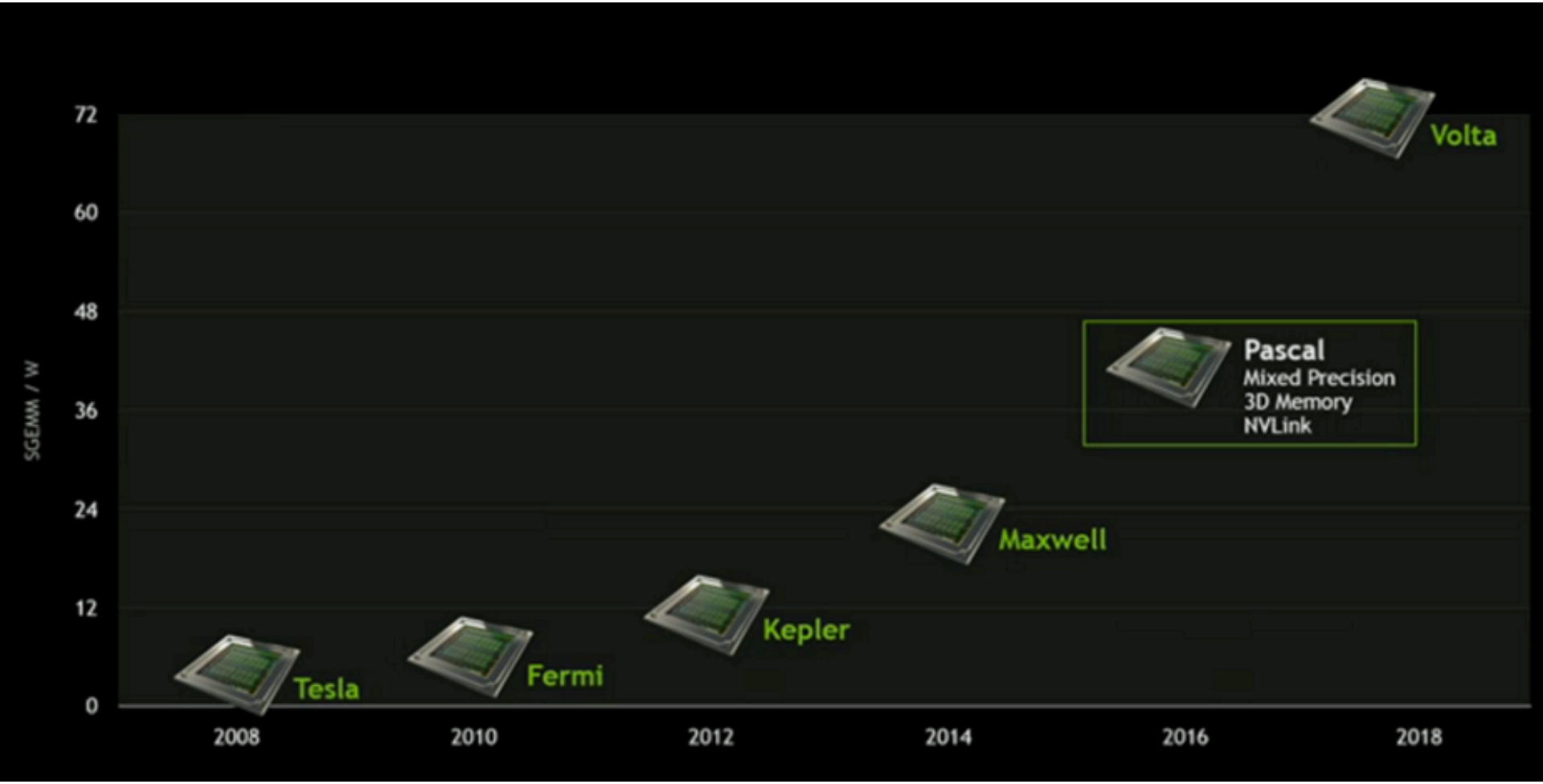
INTRODUCTION TO GPUS

GPU = the processor

GPGPU = general purpose computing on GPUs

(typically refers to non-graphics stuff)

GPUs @ NVIDIA


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

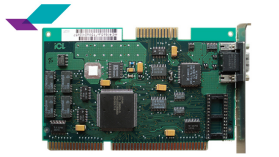
GPUs @ ATI/AMD

Celebrating 30 Years of Graphics Innovation:

The Evolution of AMD Radeon GPUs

1985

ATI develops its **first graphics controller** and **first graphics board product**.



EGA Wonder™ and **VGA Wonder™** debut.

1987

This product wins **8 industry awards** for innovation and design signifying the arrival of the graphics card business as a bona fide industry.

1991

Mach8™ chip and board products are introduced: first products to process graphics independently of the CPU.

Introduction of **Mach32™**: first integrated graphics controller and 32-bit "true color" graphics accelerator in one chip.

1992

1994

Introduction of **Mach64™**: first I graphics boards to accelerate full-motion video.

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.

1996

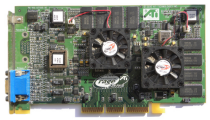
1997

First month to ship **1 million** boards!

Radeon™ graphics technology debuts: leading product for high-end gaming.

2000

1999



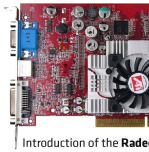
Launch of the **Rage Fury MAXX**, the world's first dual GPU card and introduces alternate frame rendering.

Radeon™ 8500 launches, the first DirectX® 8.1 GPU and our first GPU with programmable pixel and vertex processors.

2001

2002

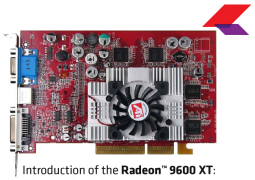
Radeon™ 9700 Pro launches: the world's first DirectX® 9 graphics processor.



Introduction of the **Rade** the world's first high vol low-k chips.

2003

GPUs @ ATI/AMD



Introduction of the **Radeon™ 9600 XT**: the world's first high volume 0.13um low-k chips.

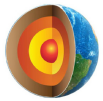
2003

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

s technology debuts: for high-end gaming.

000

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.

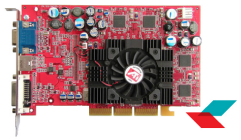
2013

What's Next?
Find out on ***08.23.14***

1999

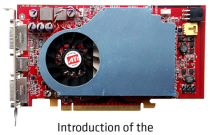
ch of the Rage Fury the world's first dual card and introduces ate frame rendering.

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

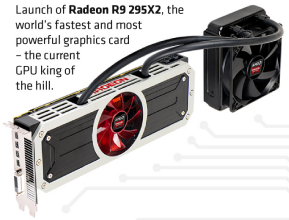
Eyefinity multi-display technology first introduced, enabling seamless connections of up to six ultra HD displays for a stunning, new PC experience.

2012

Launch of **Radeon™ HD 7970 GHz Edition**, the world's fastest and most versatile graphics card.

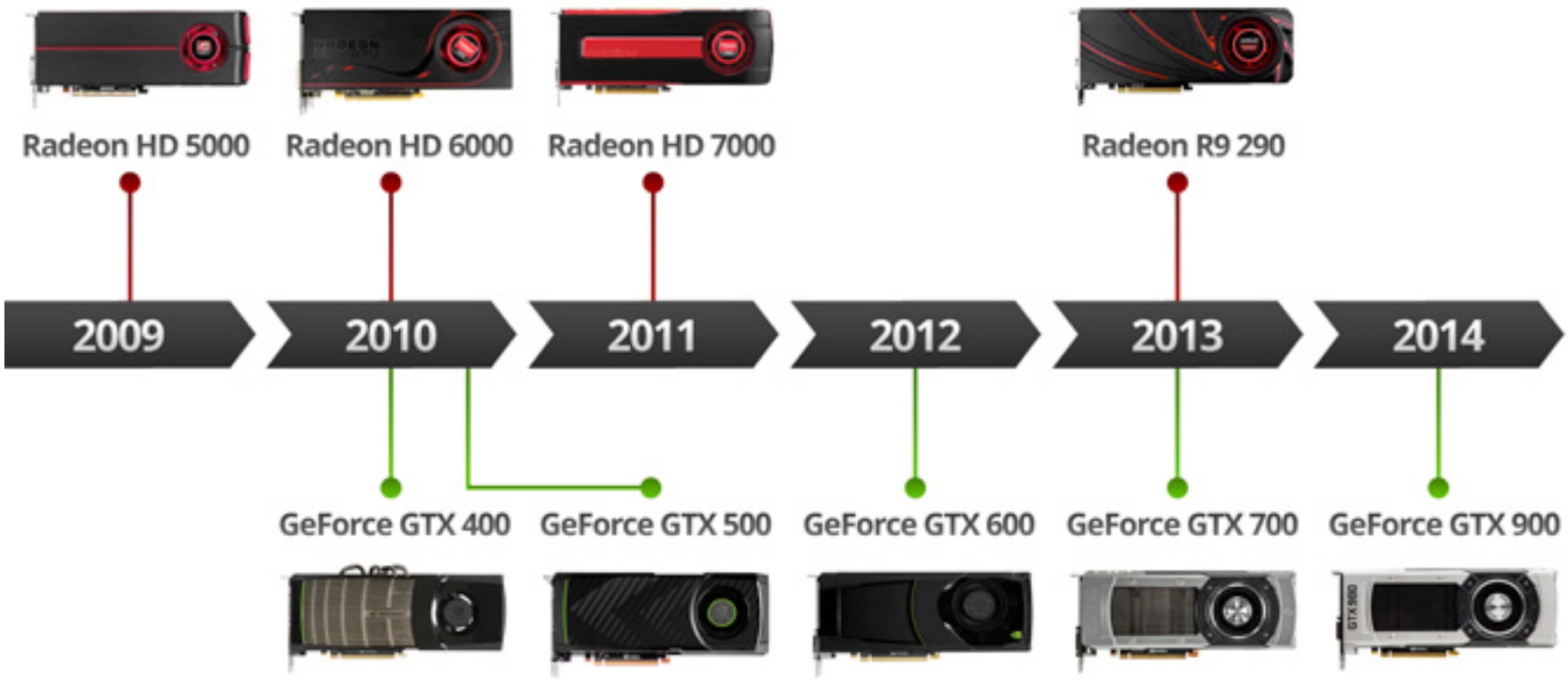
2014

Launch of **Radeon R9 295X2**, the world's fastest and most powerful graphics card – the current GPU king of the hill.

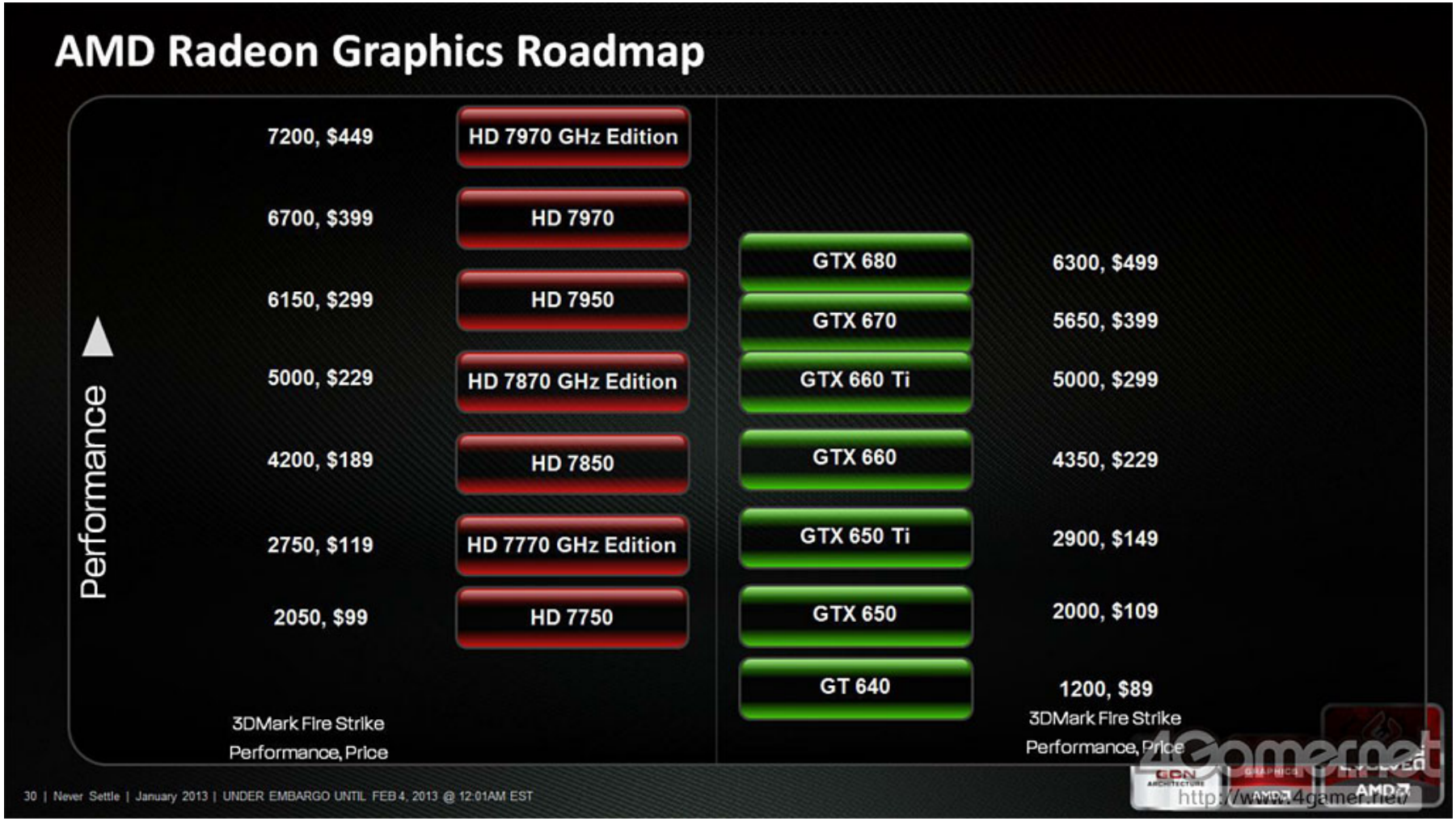


AMD

NVIDIA vs AMD



NVIDIA vs AMD



Radeon HD 7950

Gpu

Clock speed	1,050 MHz
Shading units	1,664
Pixel rate	58.8 GPixel/s
Floating-point performance	3,494 GFLOPS

320 $

GeForce GTX 970

Gpu

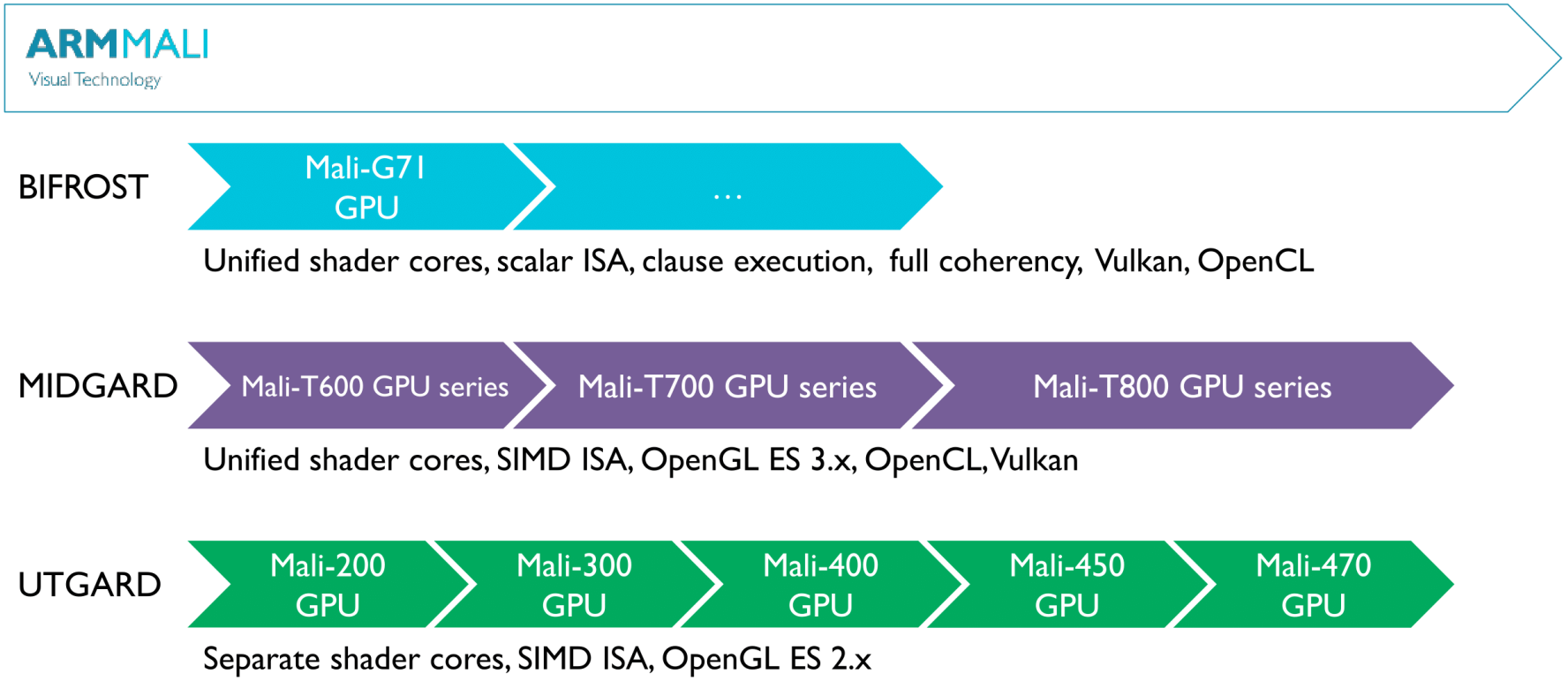
Clock speed	800 MHz
Shading units	1,792
Pixel rate	25.6 GPixel/s
Floating-point performance	2,867 GFLOPS

311 $

As you can see nvidia is great choise under 311$

GPUs @ ARM

ARM Mali Graphics Processor Generations



ON PERFORMANCE

Performance [1]

- Latency/delay
  - The time for one operation (instruction) to finish, L
  - To improve: minimize L
    - Lower is better
- Throughput
  - The number of operations (instructions) per time unit, T
  - To improve: maximize T
    - Higher is better
    - Thus, time per instruction decreases, on average
- Example: 1 man builds a house in 10 days.
  - Latency improvement: …
  - Throughput improvement: …

Performance [2]

- How do we get faster computers?
  - Faster processors and memory
    - Increase clock frequency → latency boost
  - Better memory techniques
    - Use memory hierarchies → latency boost
    - More memory closer to processor → latency boost
  - Better processing techniques
    - Use pipelining → throughput boost
  - More processing units (cores, threads, …)
    - Use parallelism/concurrency → throughput boost (only?)
  - Accelerators
    - Use specialized functional units → latency+throughput boost

Hardware Performance metrics

- Clock frequency [GHz] = absolute hardware speed
  - Memories, CPUs, interconnects
- **Operational speed [GFLOPs]**
  - Operations per second
  - **single** AND **double** precision
- **Memory bandwidth [GB/s]**
  - Memory operations per second
    - Can differ for read and write operations !
  - Differs a lot between different memories on chip
- Power [Watt]
  - The rate of consumption of energy
- Derived metrics
  - FLOP/Byte, FLOP/Watt

Name	FLOPS
yottaFLOPS	10^{24}
zettaFLOPS	10^{21}
exaFLOPS	10^{18}
petaFLOPS	10^{15}
teraFLOPS	10^{12}
gigaFLOPS	10^{9}
megaFLOPS	10^{6}
kiloFLOPS	10^{3}

Theoretical peak performance

Peak = chips * cores * vectorWidth * FLOPs/cycle * clockFrequency

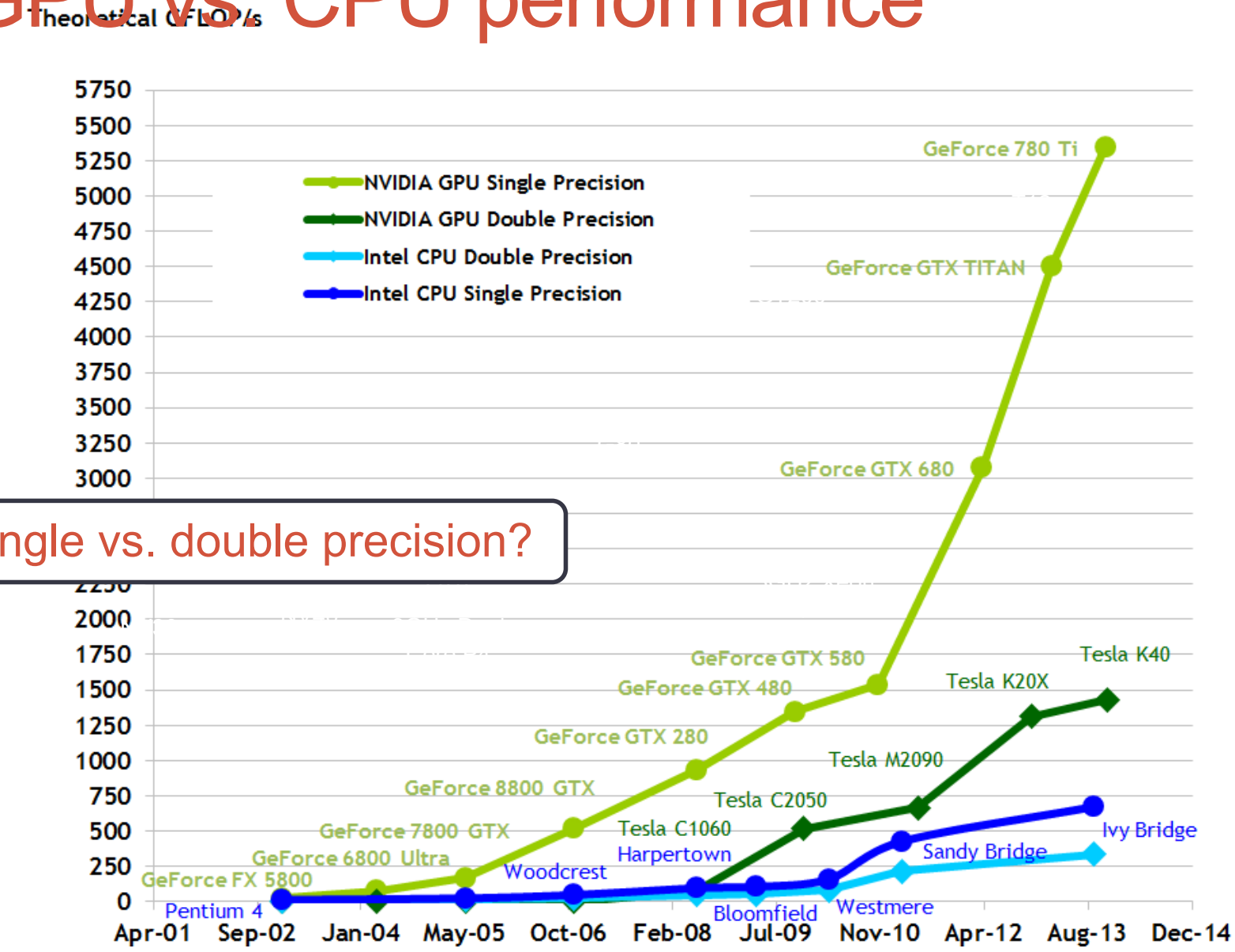
- Examples
  - Intel Core i7 CPU

    2 chips * 4 cores * 4-way vectors * 2 FLOPs/cycle * 2.4 GHz = **154 GFLOPs**
  - NVIDIA GTX 580 GPU

    1 chip * 16 SMs * 32 cores * 2 FLOPs/cycle * 1.544 GhZ = **1581 GFLOPs**
  - AMD HD 6970

    1 chip * 24 SIMD engines * 16 cores * 4-way vectors * 2 FLOPs/cycle * 0.880 GhZ = **2703 GFLOPs**

GPU vs. CPU performance



Single vs. double precision?

Main Memory bandwidth

Throughput = memory bus frequency * bits per cycle * bus width

- Memory clock **!=** CPU clock
- In bits, divide by 8 for GB/s

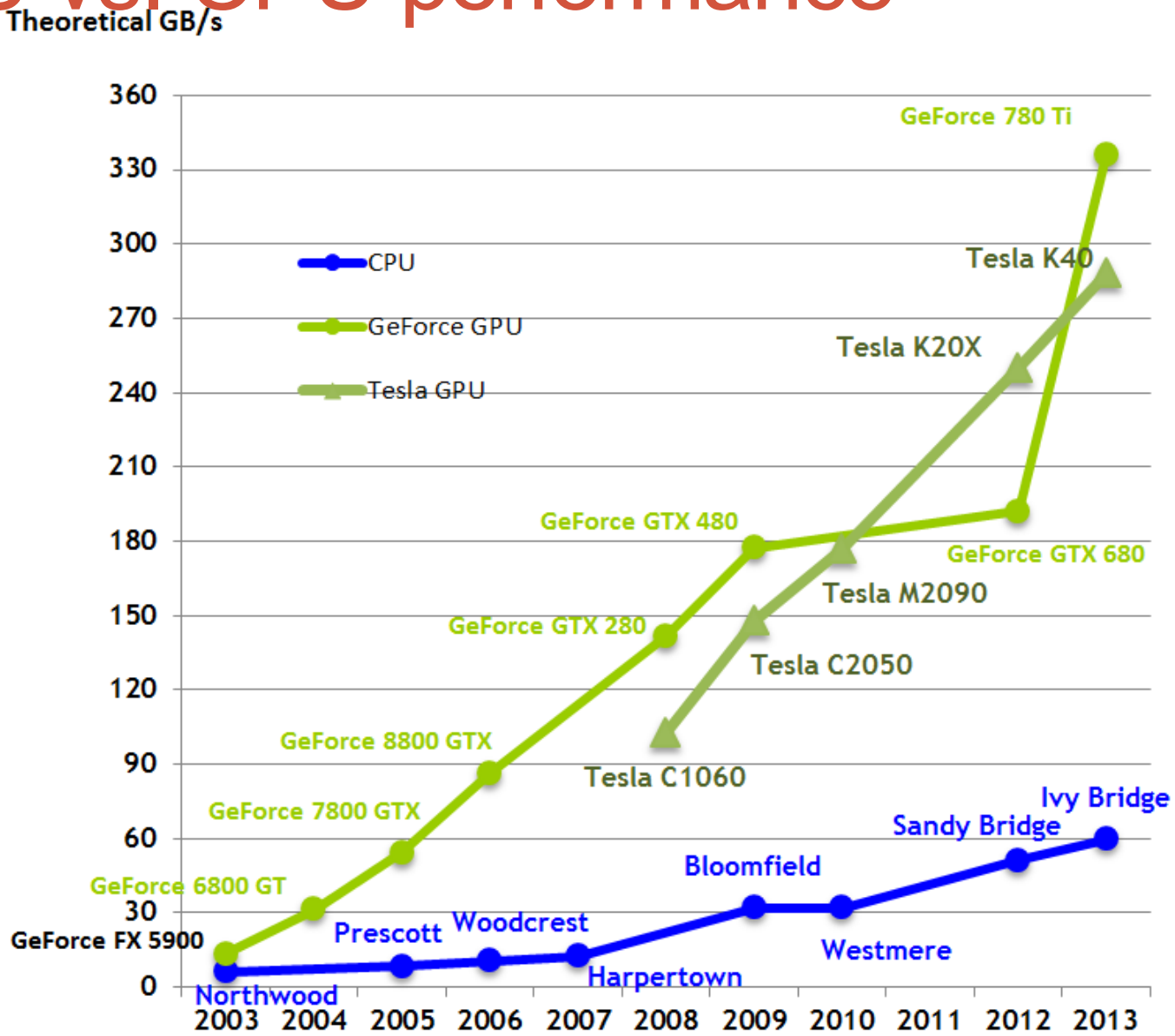
- Examples:
  - Intel Core i7 DDR3: 1.333 * 2 * 64 = **21 GB/s**
  - NVIDIA GTX 580 GDDR5: 1.002 * 4 * 384 = **192 GB/s**
  - ATI HD 6970 GDDR5: 1.375 * 4 * 256 = **176 GB/s**

Memory bandwidths

- On-chip memory can be orders of magnitude faster
  - Registers, shared memory, caches, …
  - E.g., AMD HD 7970 L1 cache achieves **2 TB/s** (vs. 176GB/s for main memory)

- Other memories: depends on the interconnect
  - Intel's technology: QPI (Quick Path Interconnect)
    - 25.6 GB/s
  - AMD's technology: HT3 (Hyper Transport 3)
    - 19.2 GB/s
  - Accelerators: PCI-e 2.0
    - 8 GB/s

GPU vs. CPU performance



Power

- Chip manufactures specify Thermal Design Power (TDP)
- We can measure dissipated power
  - Whole system
  - Typically (much) lower than TDP
- Power efficiency
  - FLOPS / Watt

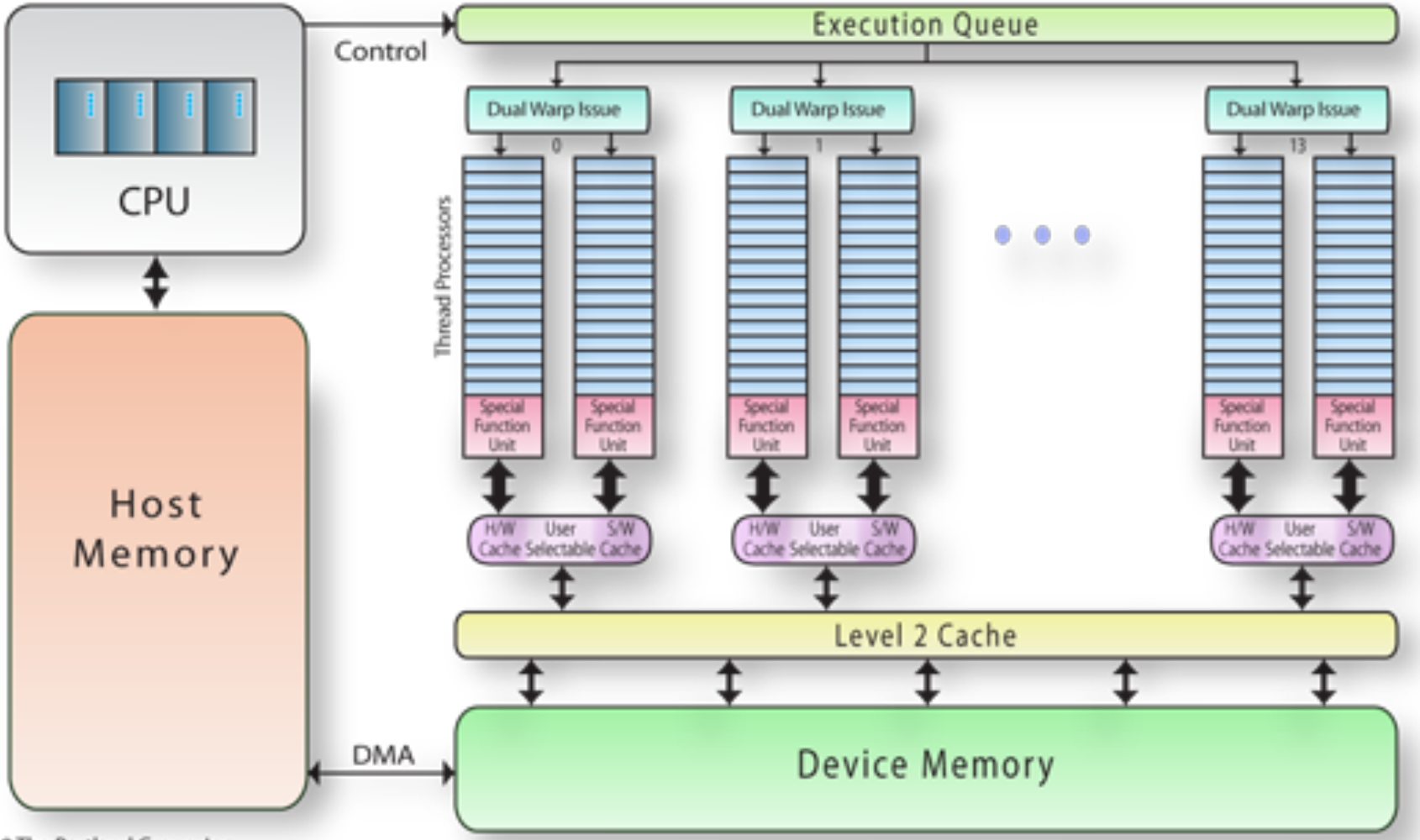
- Examples (with theoretical peak and TDP)
  - Intel Core i7: 154 / 160 = **1.0 GFLOPs/W**
  - NVIDIA GTX 580: 1581 / 244 = **6.3 GFLOPs/W**
  - ATI HD 6970: 2703 / 250 = **10.8 GFLOPs/W**

Absolute hardware performance

- Only achieved in the optimal conditions:
  - Processing units 100% used
  - All parallelism 100% exploited
  - All data transfers at maximum bandwidth
- In real life
  - No application is like this
  - Can we reason about “real” performance?

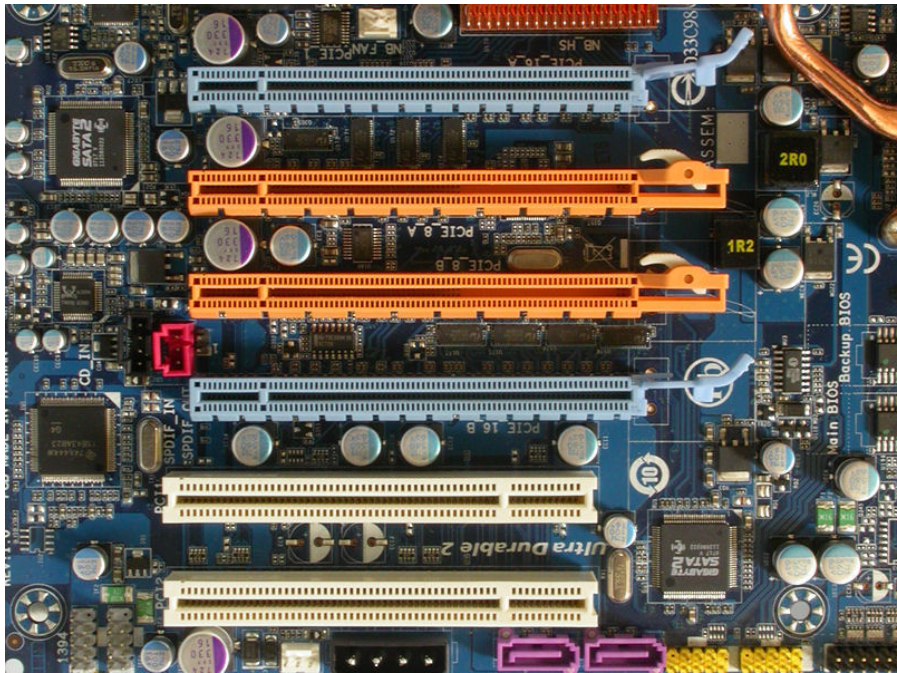
HIGH-LEVEL OPERATIONAL VIEW

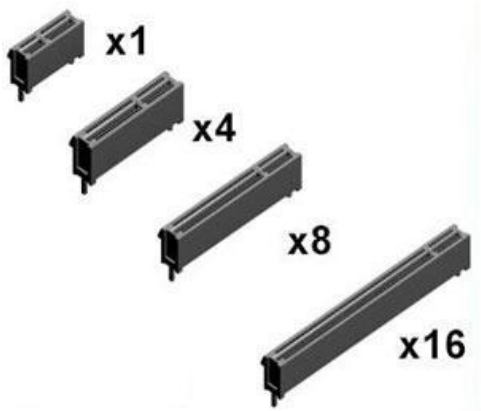
A GPU Architecture



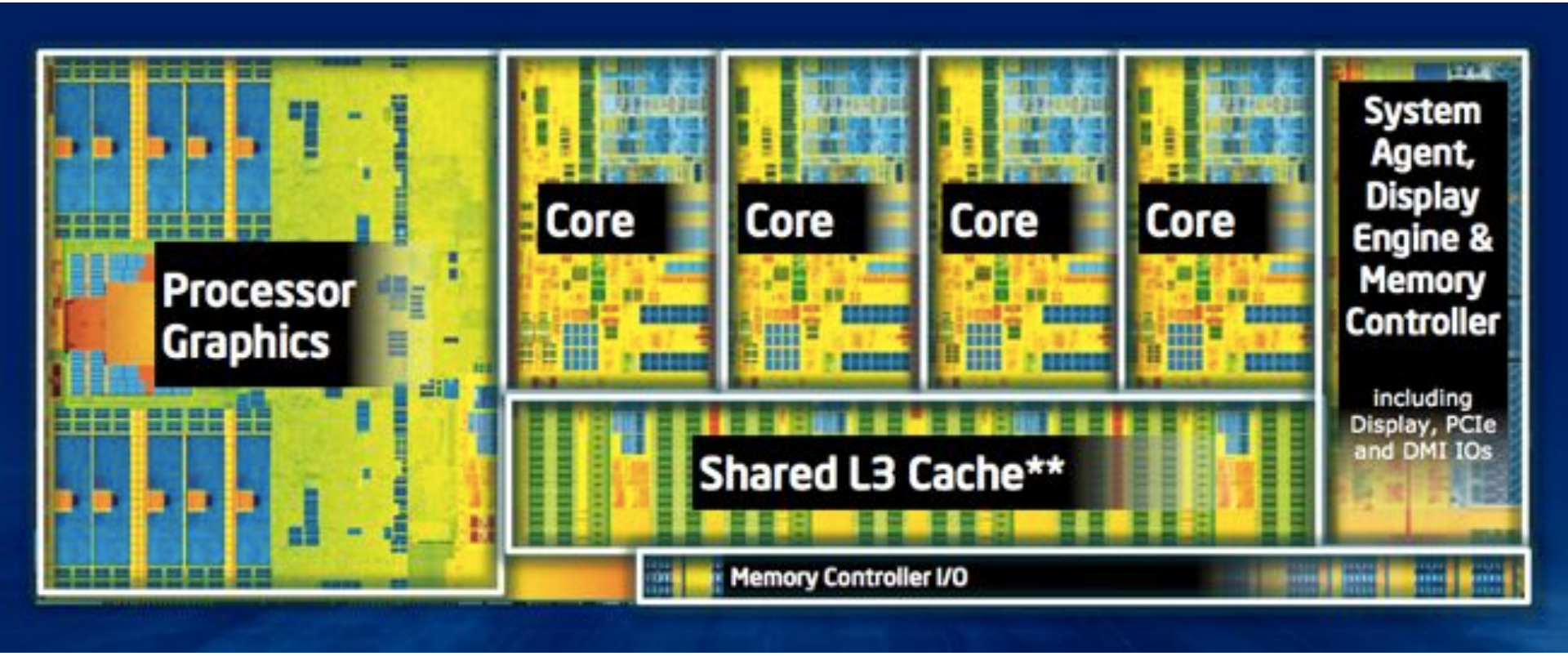
Integration into host system

- 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

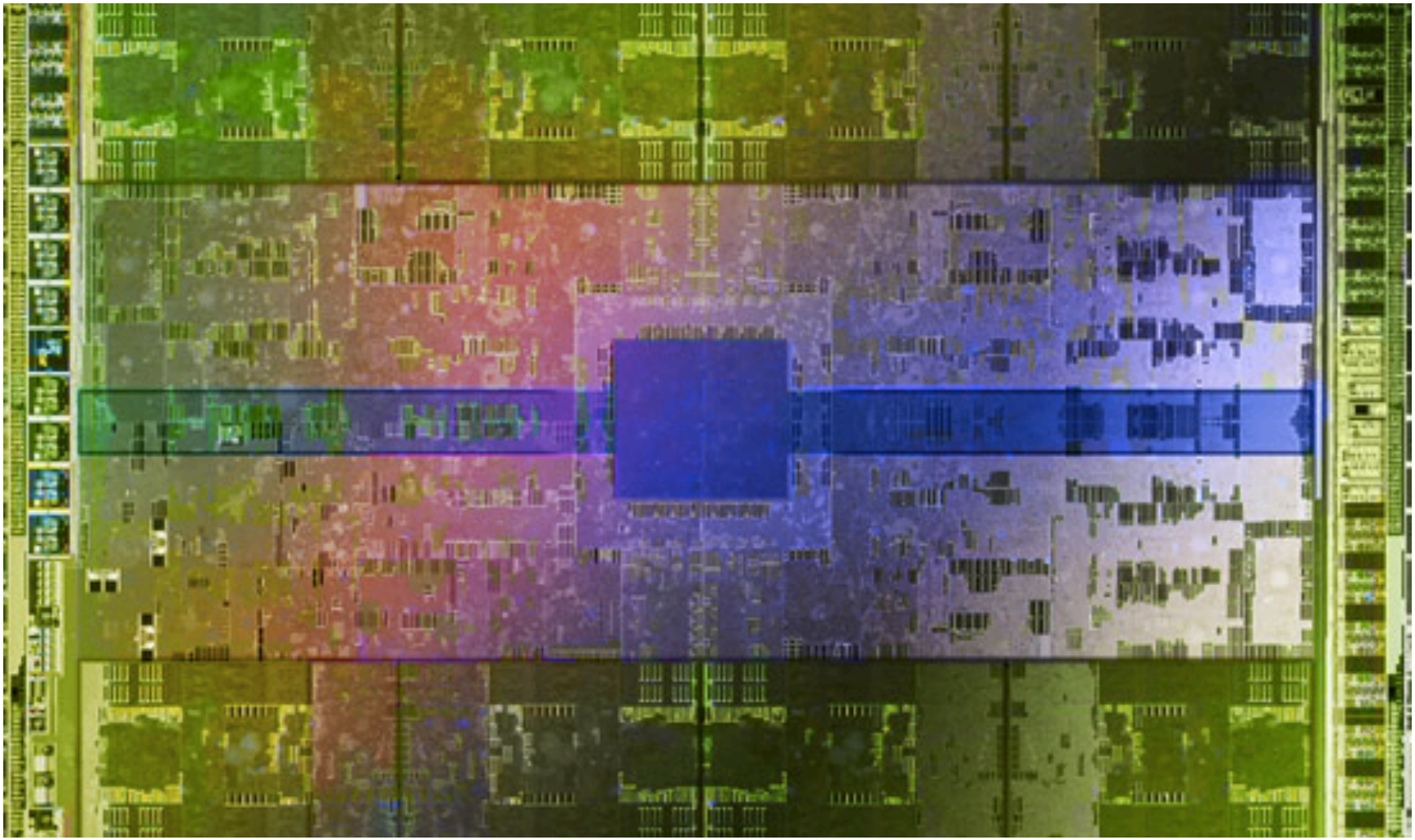




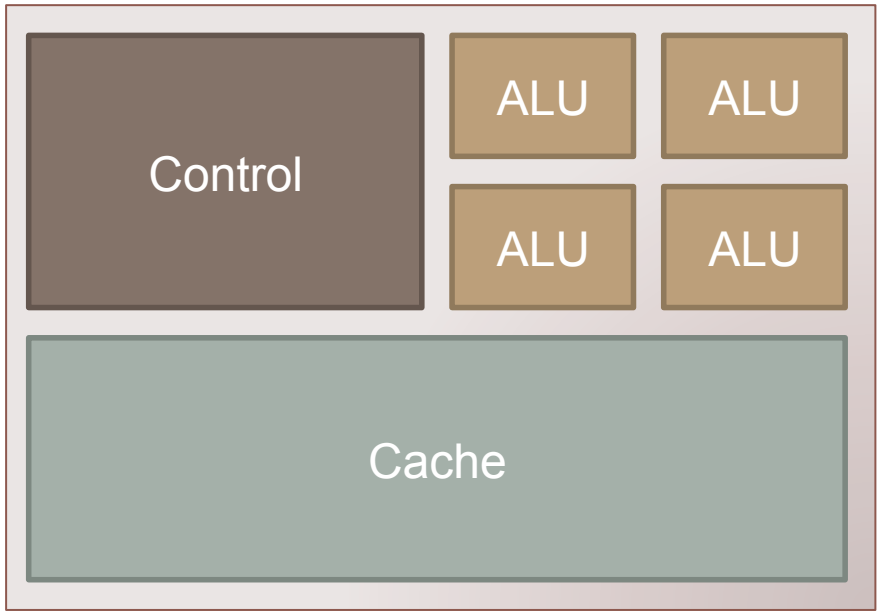
A CPU die


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

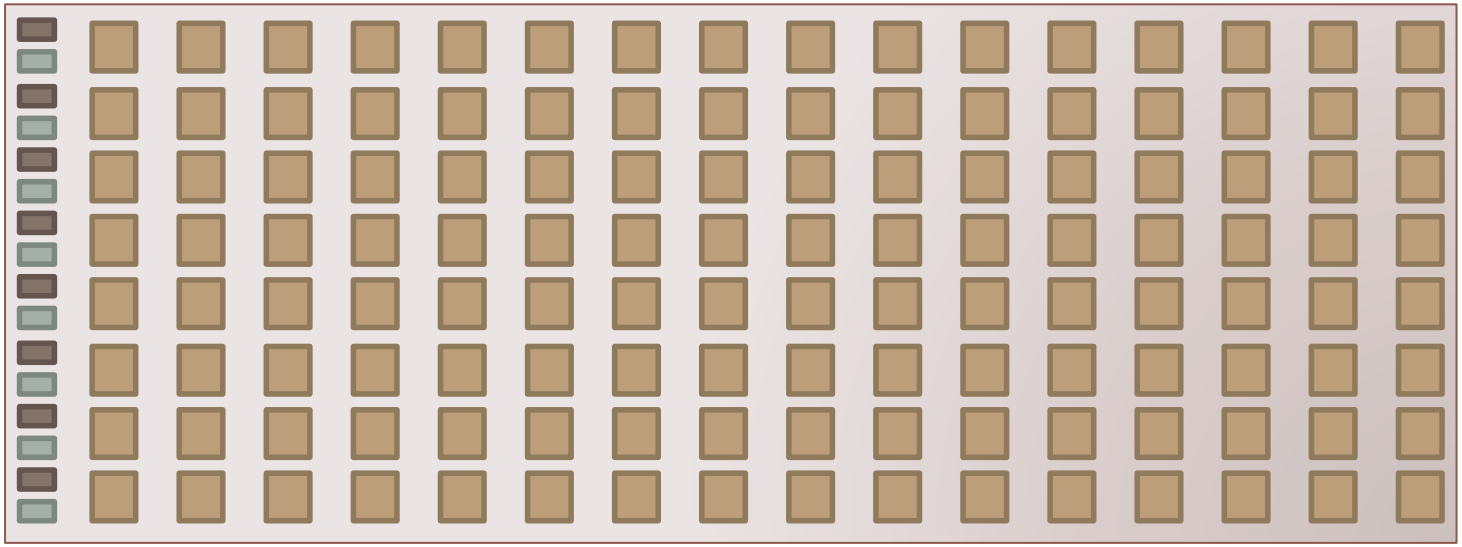


CPU

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

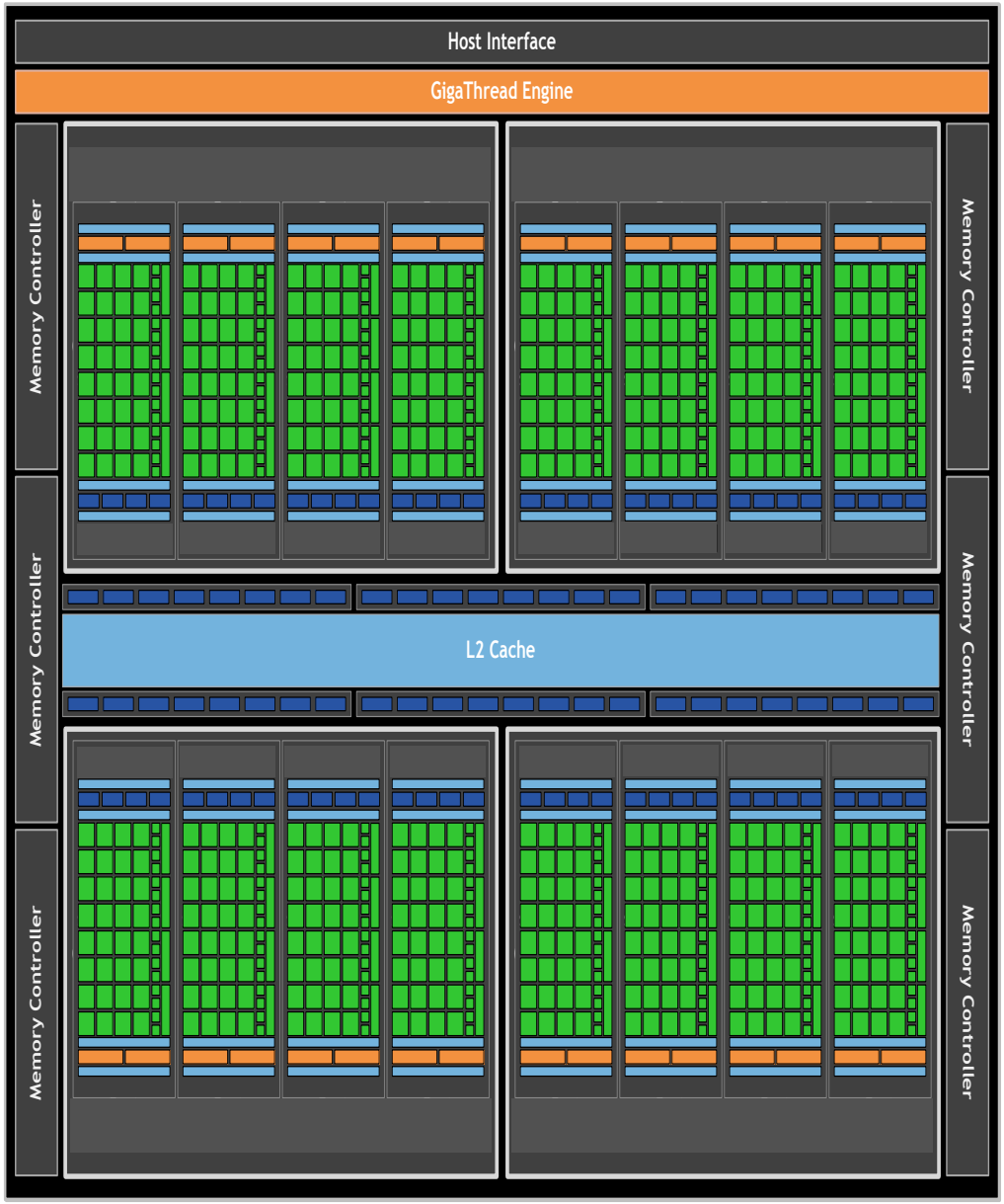
CPU vs. GPU

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

NVIDIA GPUS ARCHITECTURE

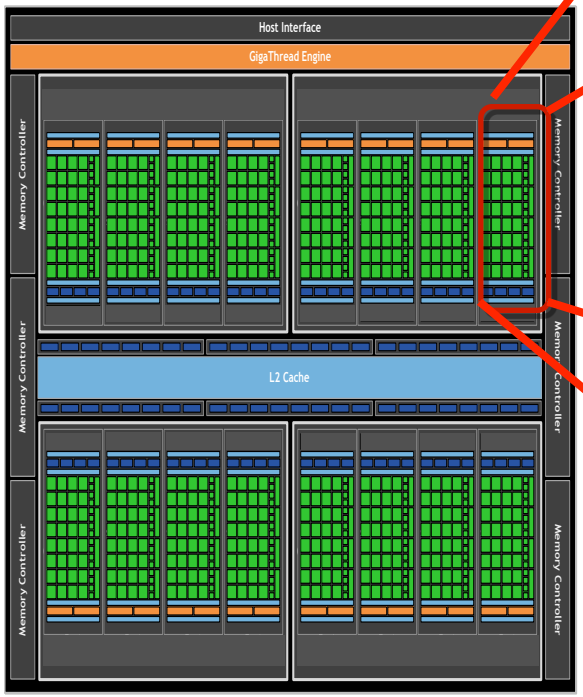
Fermi

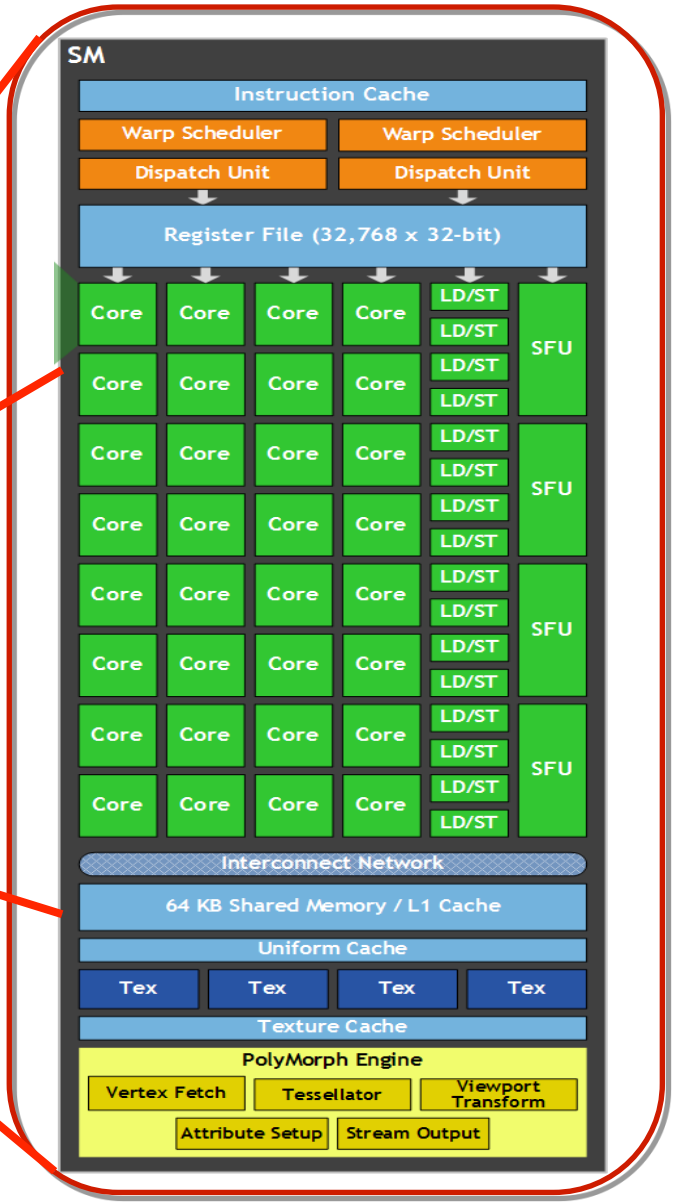
- Consumer: GTX 480, 580
- HPC: Tesla C2050
  - More memory, ECC
  - 1.0 Tlop SP
  - 515 GFlop SP
- 16 streaming multiprocessors (SM)
  - GTX 580: 16
  - GTX 480: 15
  - C2050: 14
- SMs are independent
- 768 KB L2 cache



Fermi Streaming Multiprocessor (SM)

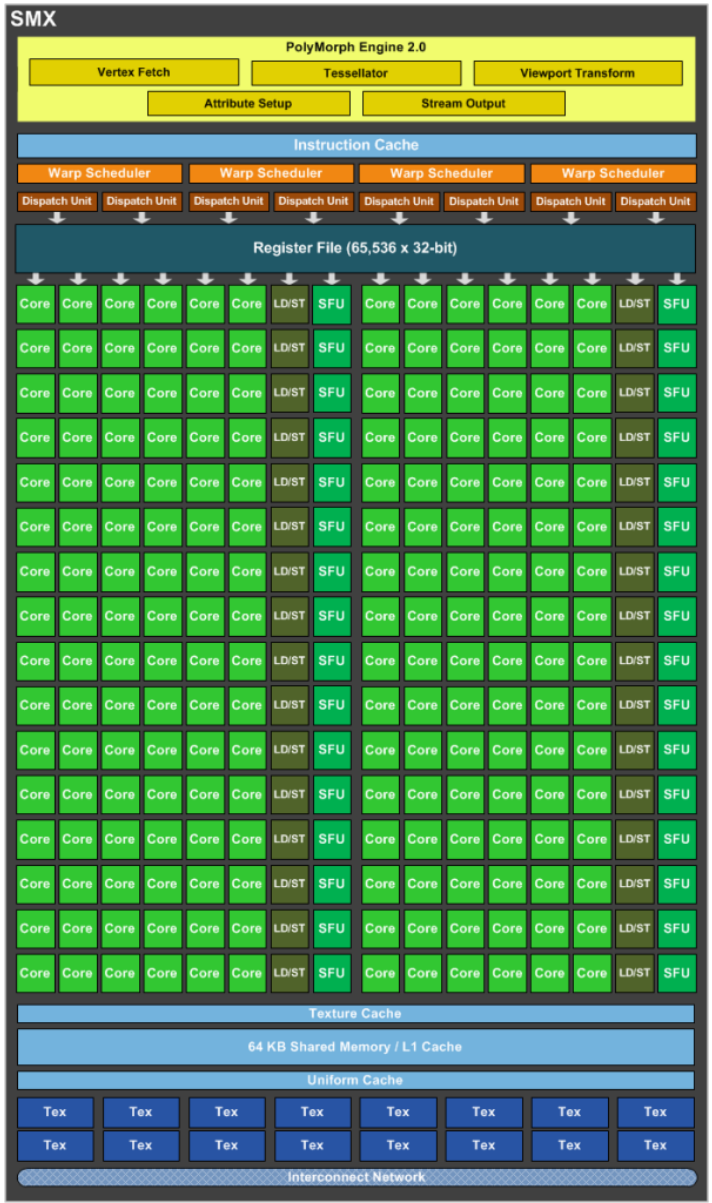
- 32 cores per SM (512 cores total)
- 64KB configurable
  L1 cache / shared memory
- 32,768 32-bit registers





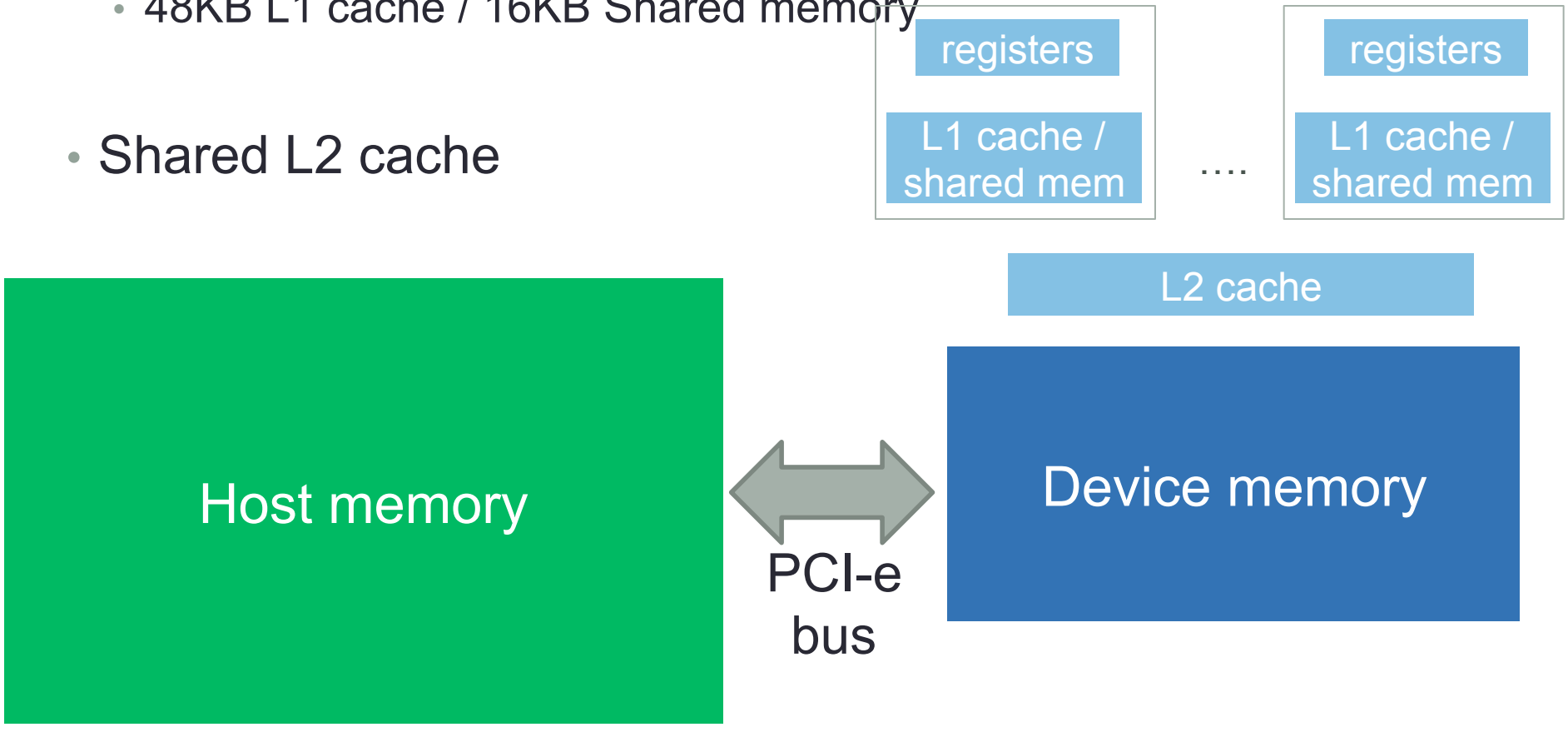
Kepler: SMX

- 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



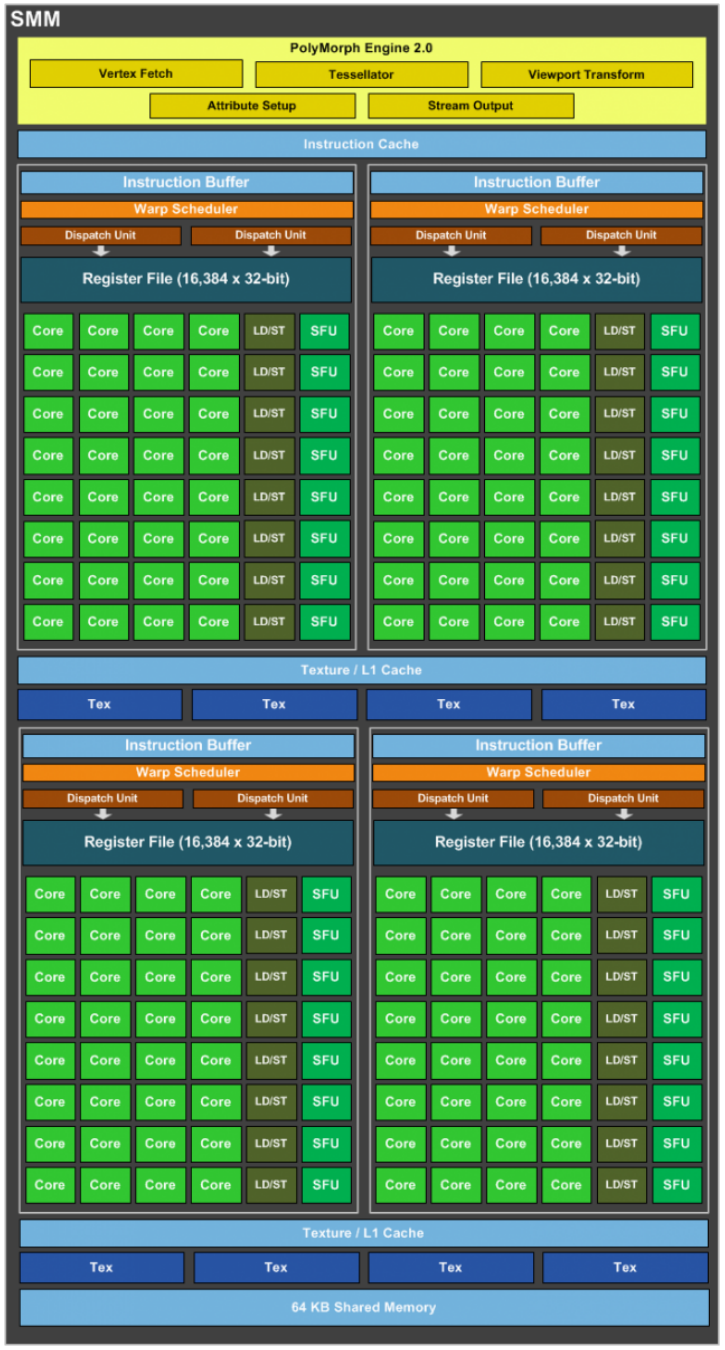
Memory architecture (since Fermi)

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



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
- Focus on DP
- Energy efficiency

Evolution in numbers

GPU / Form Factor	Kepler	Maxwell	Pascal	Pascal
	GK110 / PCIe	GM200 / PCIe	GP100 / SXM2	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 MANY-CORES

Parallelism

- Threads
  - Independent units of computation
  - Expected to execute in parallel
  - Write once, instantiate many times
- Concurrent execution
  - Threads execute in the same time if there are sufficient resources
- Assume a processor P with 10 cores and an application A with:
  - 10 threads: how long does A take?
  - 20 threads: how long does A take?
  - 33 threads: how long does A take?

Parallelism

- Synchronization = a thread's execution must depend on other threads
  - Barrier = all threads wait to get to barrier before they continue
  - Shared variables = more threads RD/WR them
    - Locks = threads can use locks to protect the WR sections
  - Atomic operation = operation completed by a single thread at a time
- Thread scheduling = the order in which the threads are executed on the machine
  - User-based: programmer decides
  - OS-based: OS decides (e.g., Linux, Windows)
  - Hardware-based: hardware decides (e.g., GPUs)

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

PROGRAMMING GPUS IN CUDA

Kernel = the parallel program

Device code = manage the parallel program

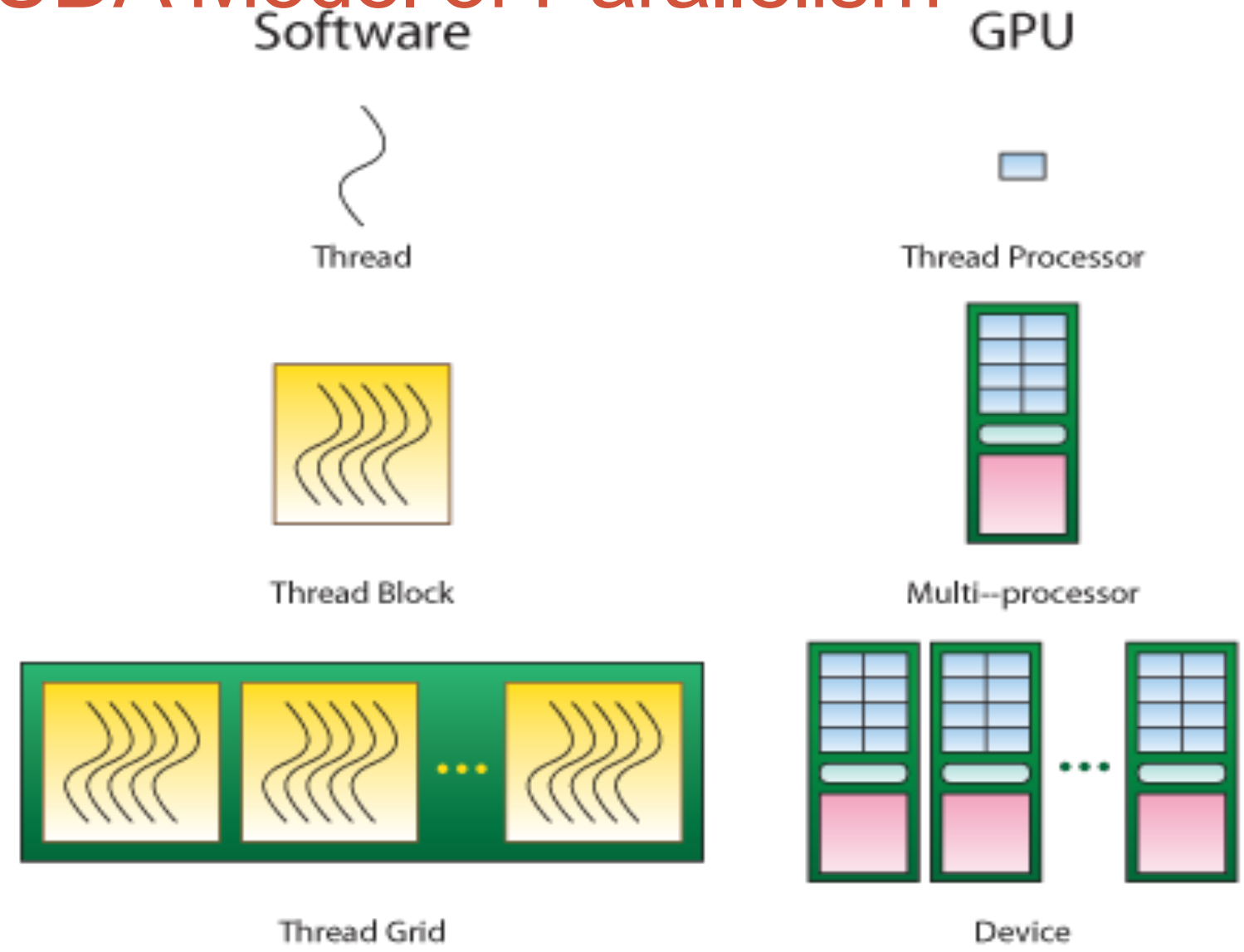
CUDA

- CUDA: Scalable parallel programming
  - C/C++ extensions
    - Other wrappers exist
- Straightforward mapping onto hardware
  - Hierarchy of threads (to map to cores)
    - Configurable at logical level
  - Various memory spaces (to map to physical spaces)
    - Usable via variable scopes
- Scale to 1000s of cores & 100,000s of threads
  - GPU threads are lightweight
  - GPUs need 1000s of threads for full utilization

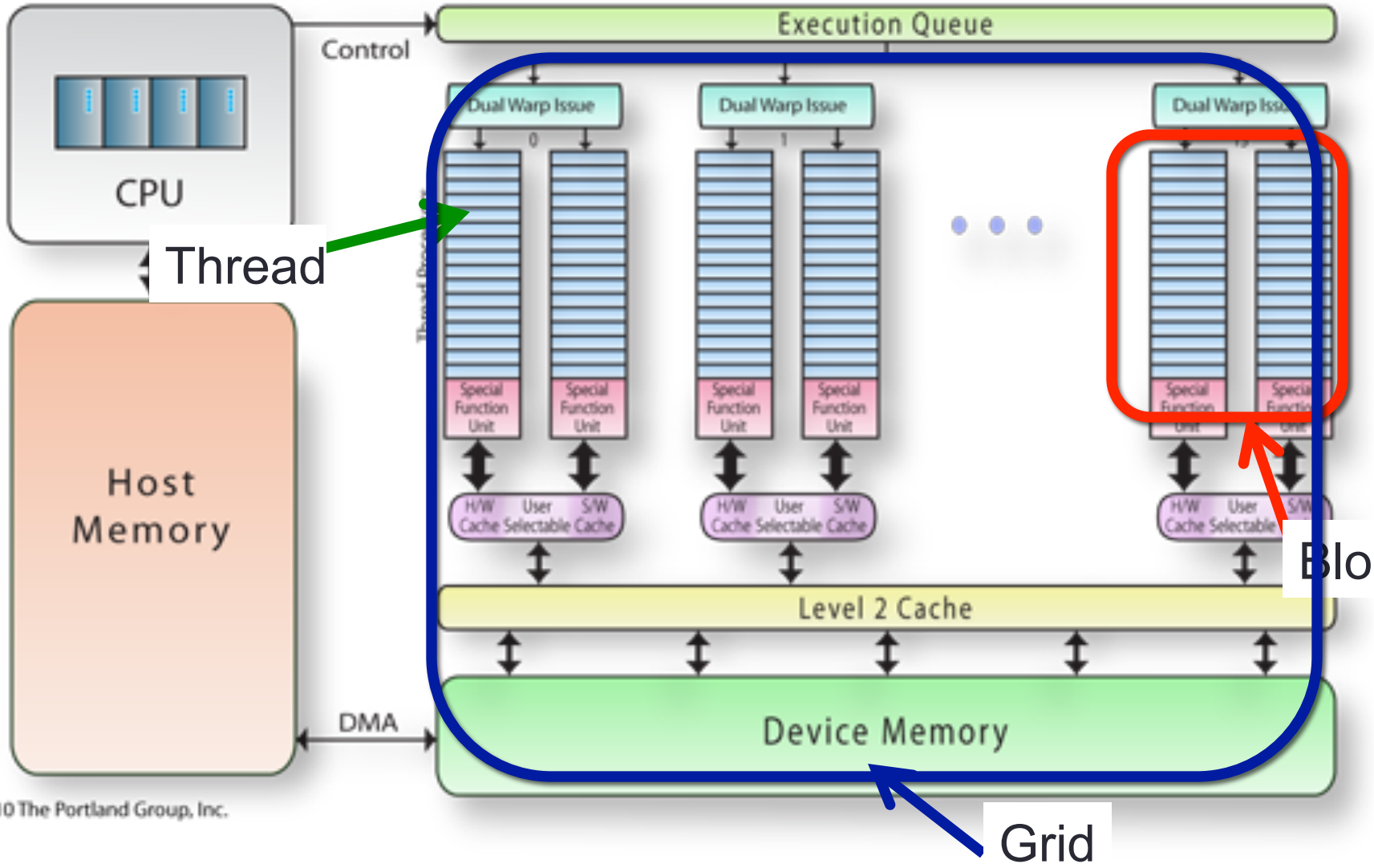
CUDA Model of Parallelism

- 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
- Threads are scheduled onto physical hardware without pre-emption
  - threads/blocks launch & run to completion
  - blocks must be independent

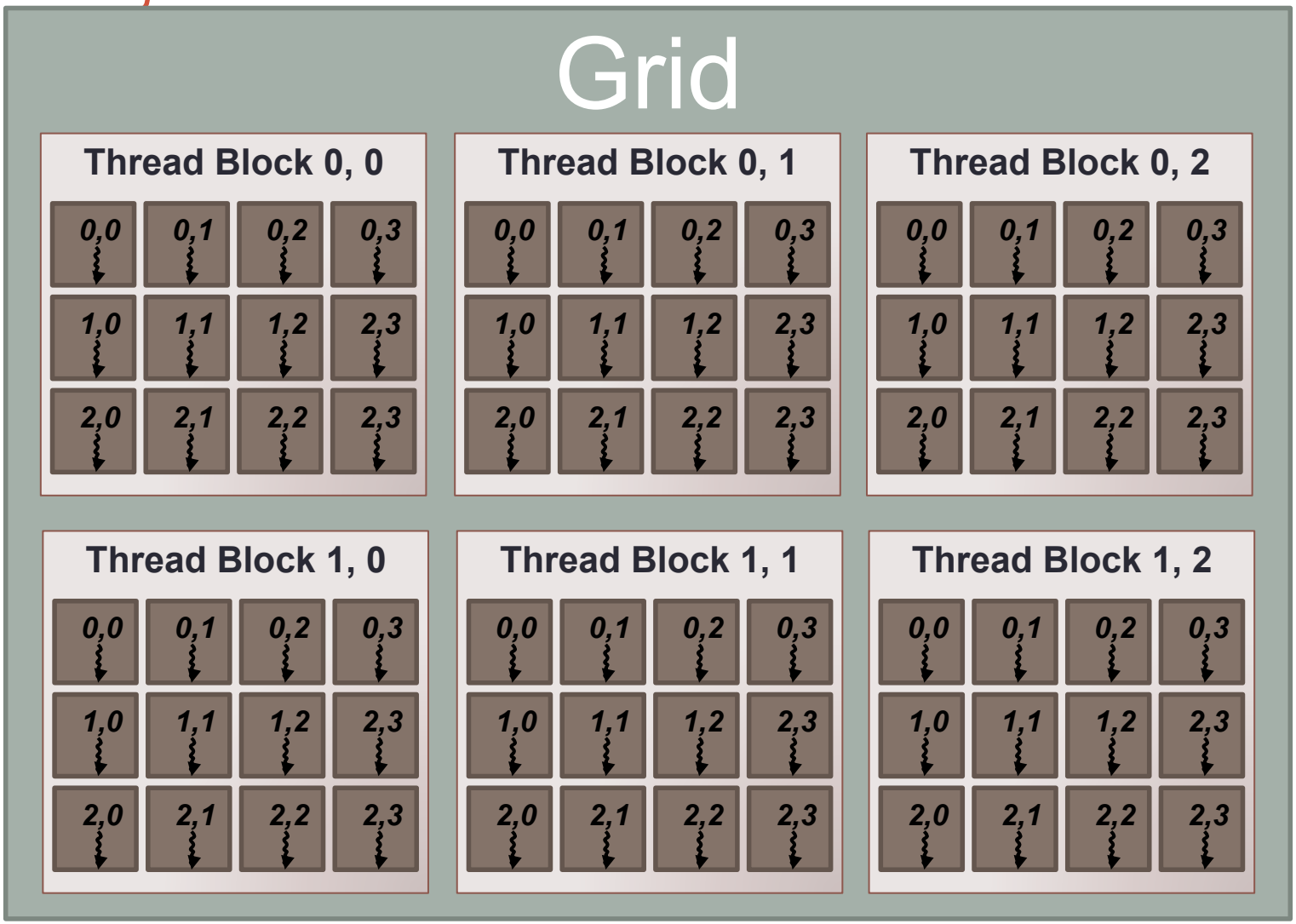
CUDA Model of Parallelism



Hierarchy of threads



Grids, Thread Blocks and Threads

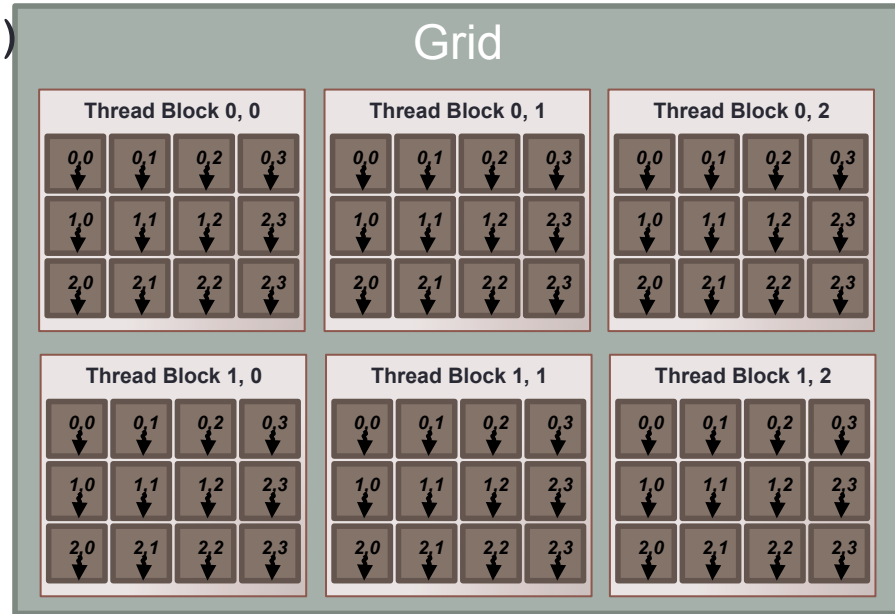


Kernels and grids

- Launch kernel (12 x 6 = 72 instances)

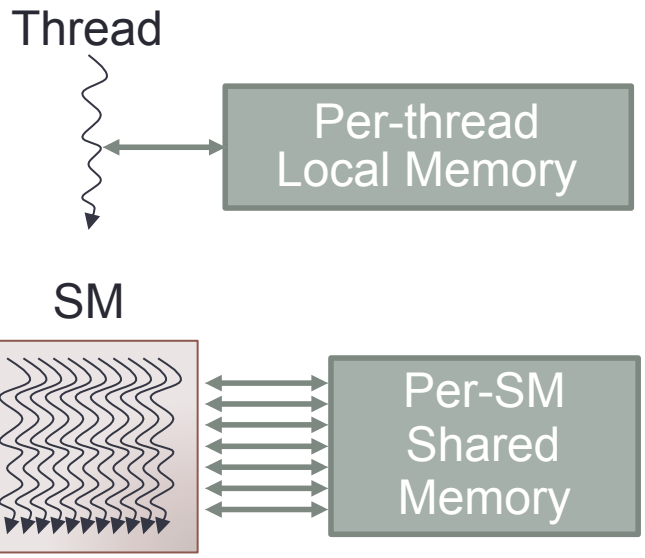
```
myKernel<<<numBlocks,threadsPerBlock>>>(…);
```

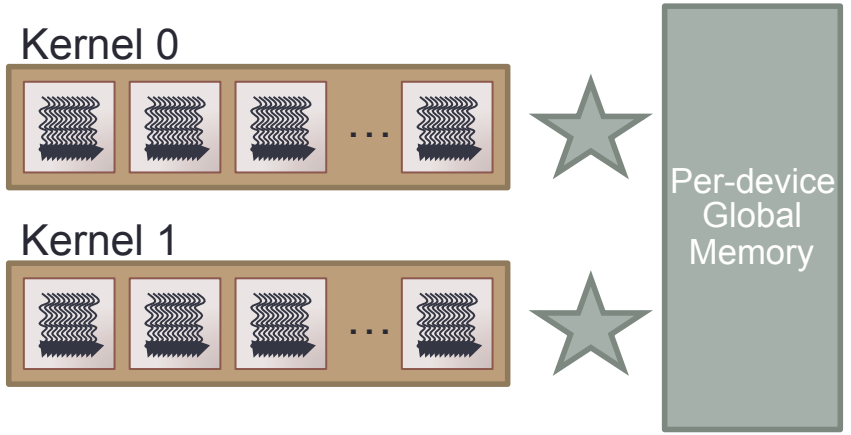
  - `dim3 threadsPerBlock(3,4);`
    - `threadsPerBlock.x = 3`
    - `threadsPerBlock.y = 4`
    - `Each thread:`
      `(threadIdx.x, threadIdx.y)`
  - `dim3 numBlocks(2,3);`
    - `blockDim.x = 2`
    - `blockDim.y=3`
    - `Each block :`
      `(blockIdx.x,blockIdx.y)`



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
```

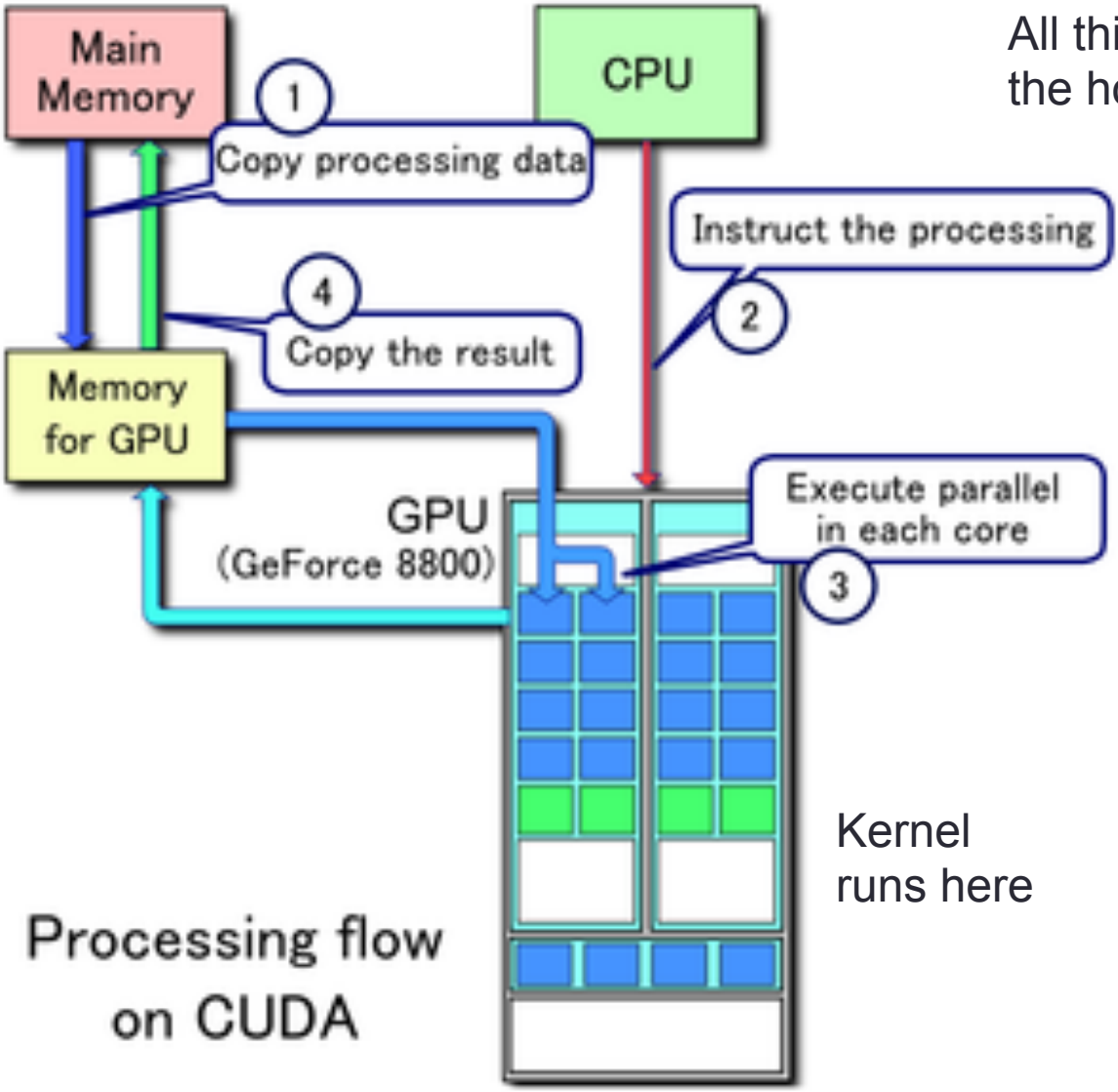
Shared memory

- 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

Processing flow



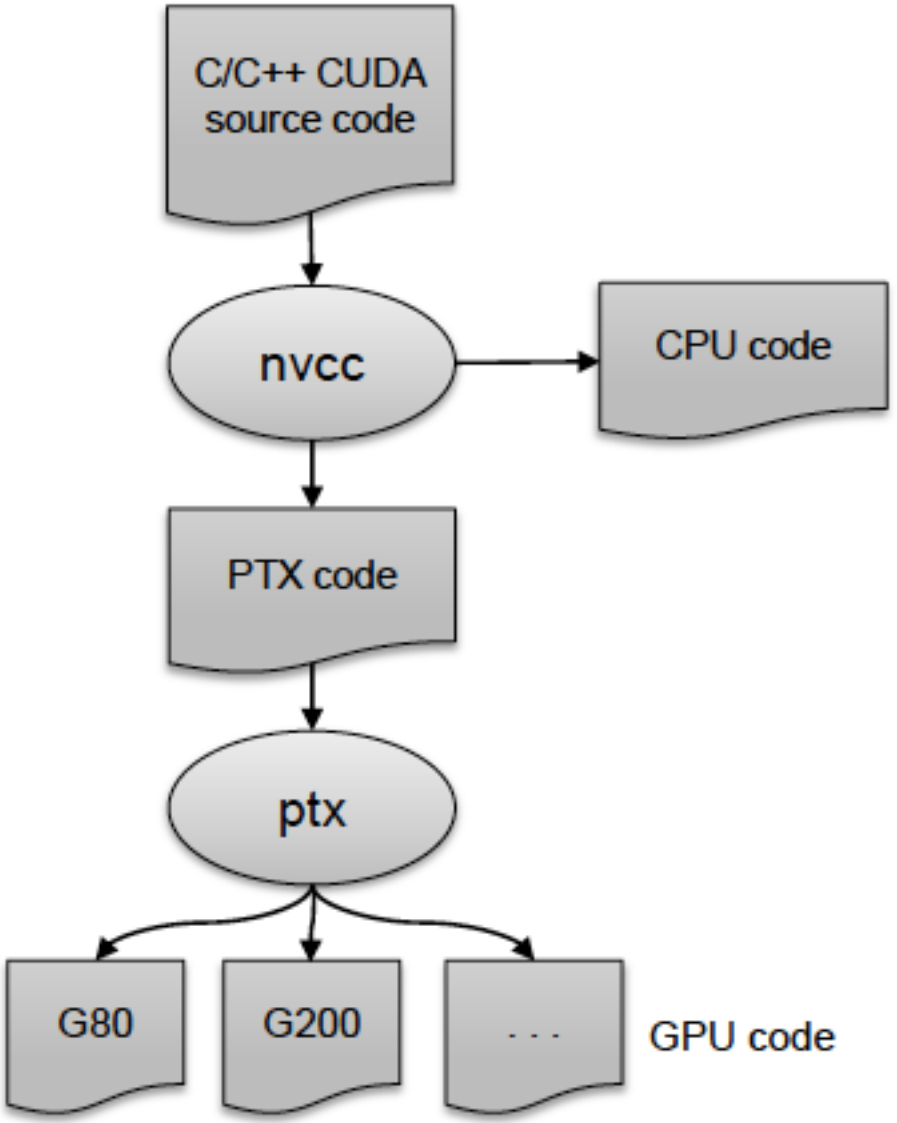
All this happens from the host code.

Kernel runs here

Image courtesy of Wikipedia

Compiling CUDA

- nvcc is a compiler driver
- Separates source code into:
  - device code (runs on GPU)
    - further processed by NVIDIA compi
  - host code (runs on CPU)
    - further processed by host compiler



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];
    }
}
```

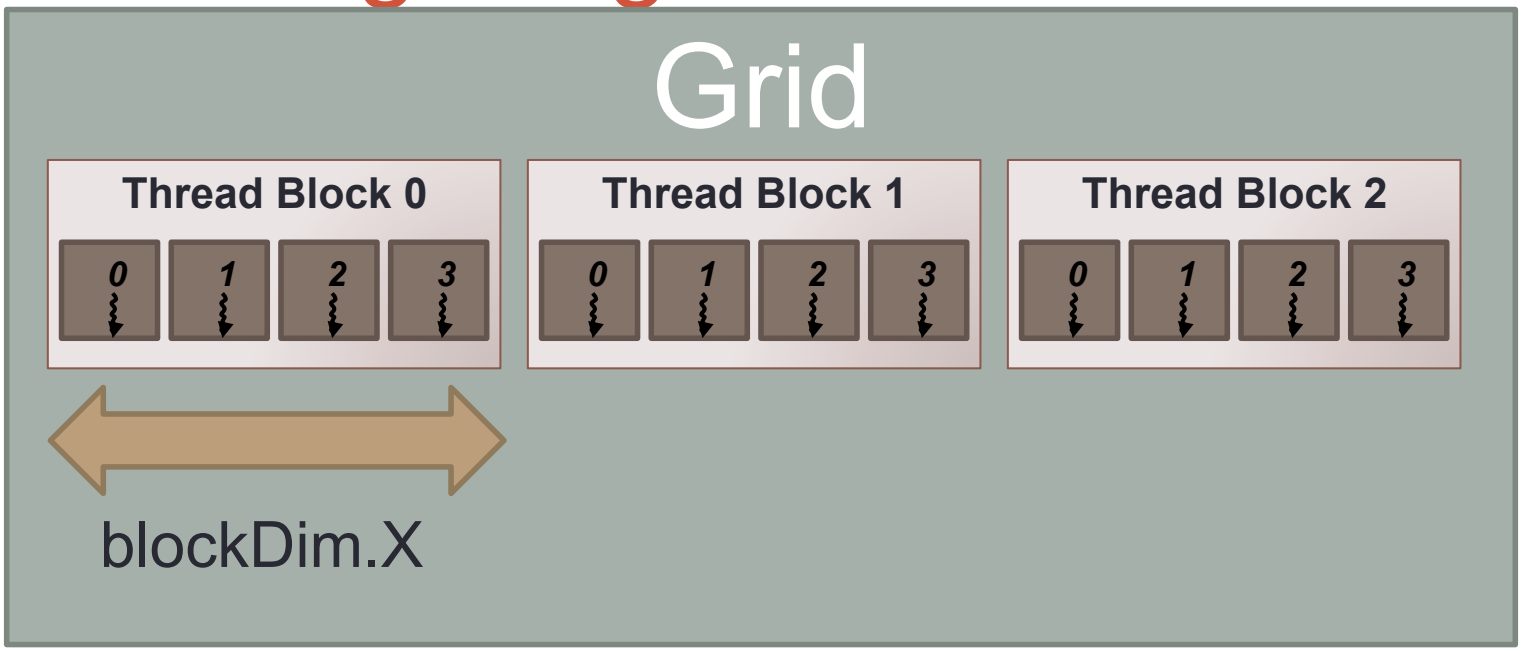
How do we parallelize this?

- 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];
}
```

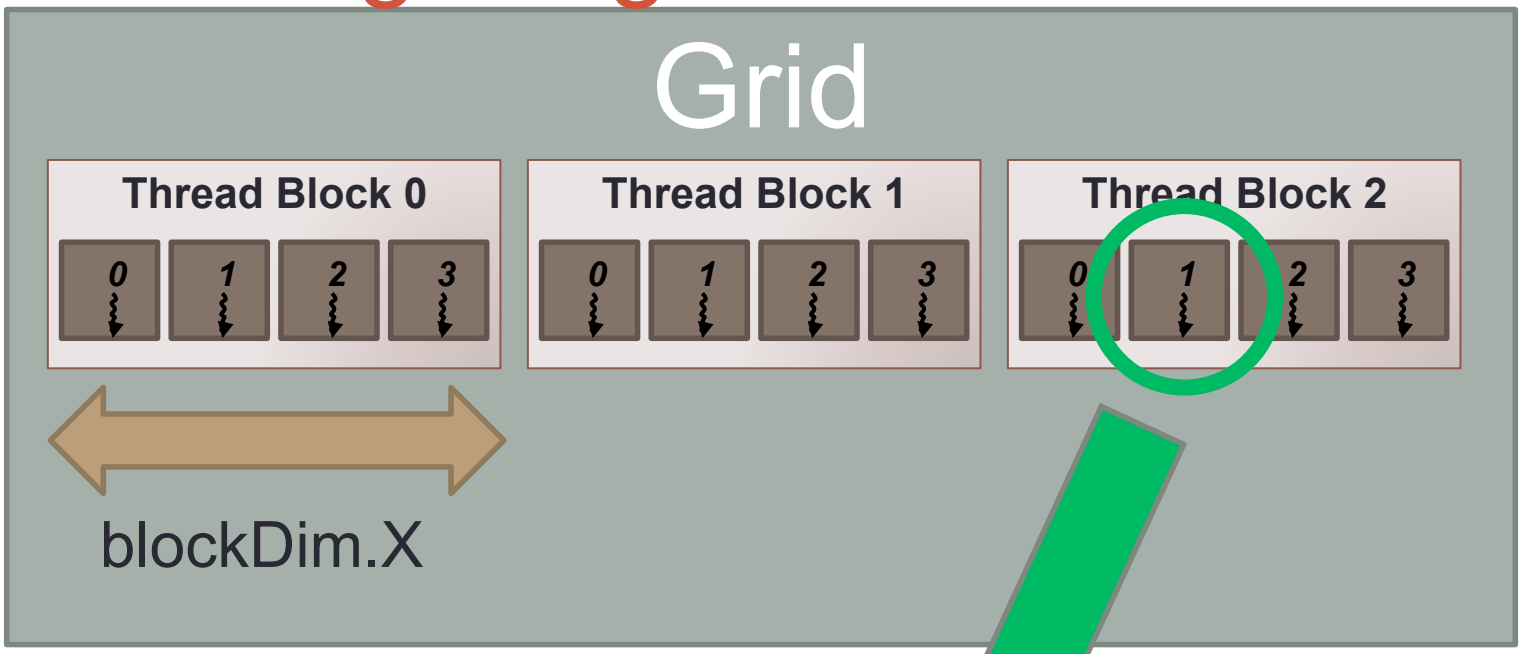
Calculating the global thread index



"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

(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

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

Host code

(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

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

Host code

(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!

ADVANCED CONCEPTS

Thread Scheduling

- Order of threads within a block is undefined!
  - Threads are grouped in warps (32 threads/warp)
    - AMD calls it “a wavefront” (64 threads/wavefront)
- Order in which thread blocks are mapped and scheduled is undefined!
  - Blocks run to completion on one SM without preemption
  - Can run in any order
    - Any possible interleaving of blocks should be valid
  - Can run concurrently OR sequentially

Global synchronization

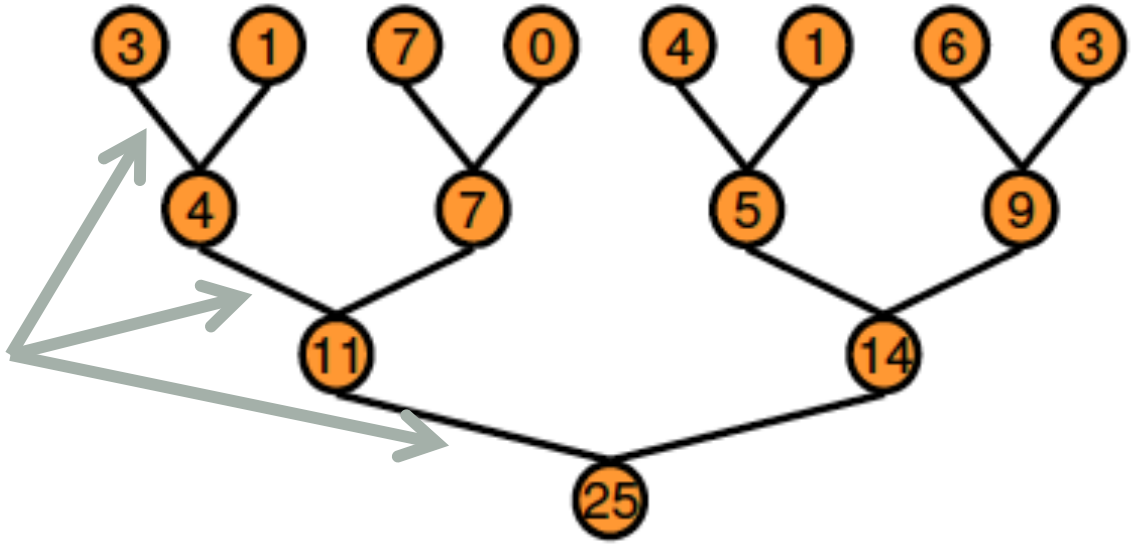
- We launch many more blocks than physical SM's.
- Each block might/should have more threads than the SM's cores

```
__global__ void my_kernel() {
    step1; // compute some values in a global array
    // wait for *all* threads to finish
    __my_global_barrier();
    step2; // use the array
}

int main() {
  dim3 blockSize(32, 32);
  dim3 gridSize(100, 100, 100);
  my_kernel<<<gridDim, blockDim>>>();
}
```

An example: parallel reduction

- Given an array with data, “reduce” it to a single value
  - The sum of all elements
  - The min/max of all elements
- Sequentially: O(n)
- In parallel?
  - Tree-based algo.
    - O(log n)
  - Requires a barrier after each step

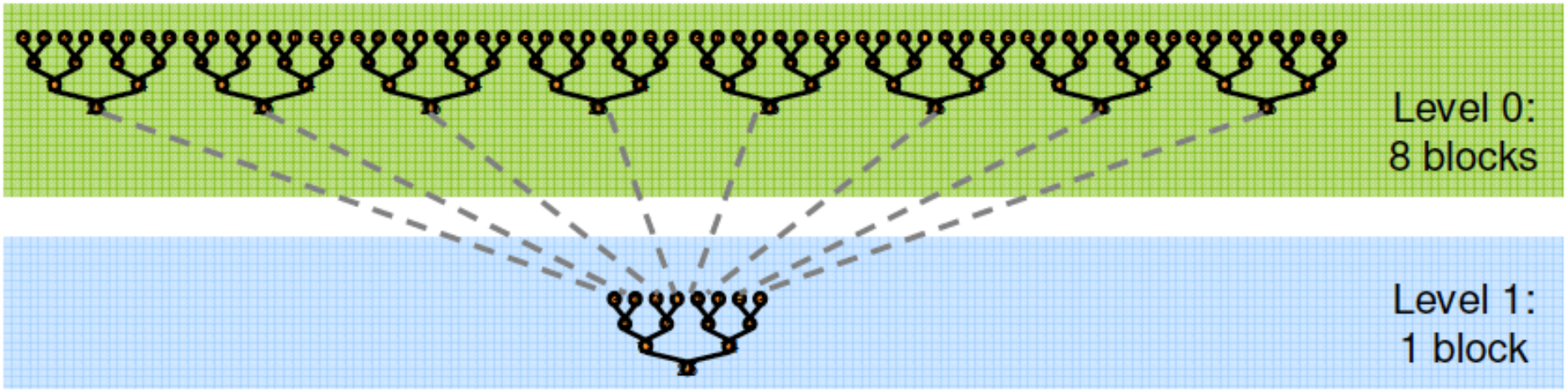


Parallel reduction in CUDA*

- One element per thread
- We need to use multiple blocks
  - Large arrays
  - Good GPU utilization
- We need global synchronization
  - Synchronization inside blocks is possible.
  - Synchronization between blocks is not possible!
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

*http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Parallel reduction in CUDA*



- Other optimizations
  - Use shared memory
  - Increase granularity
  - Avoid branching
  - Improve data access patterns

*http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Memory consistency

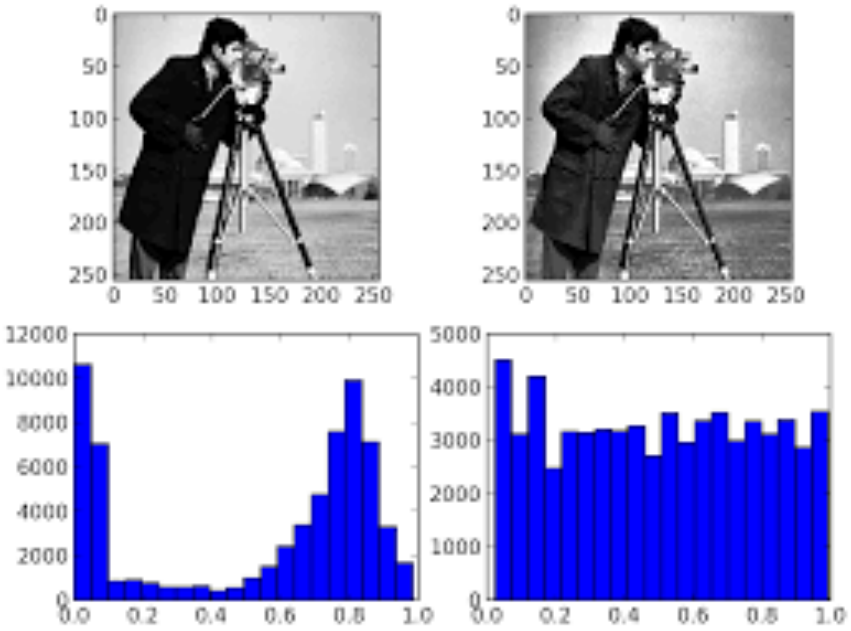
- Device (global) memory is not serially consistent
  - No ordering guarantees in shared/global memory Rd/Wr
- Share data between streaming multiprocessors
  - Potential write hazards!
- Use **atomics** to avoid data races for global (and shared) memory variables!
- Evolution:
  - Fermi has reasonable atomics for both shared and global memory
  - Kepler increases *global memory atomics* performance vs. Fermi
  - Maxwell uses native support for shared memory atomics
    - Much faster than Fermi and Kepler

Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
  - Ordering is still arbitrary
- Different types of atomic instructions
  - Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Both for device memory and shared memory
- Much more expensive than load + operation + store

An example: image histogram

- The histogram of an image: the distribution of the pixels in the image.
  - In practice: count the pixels of each color
  - Useful image feature detection for image recognition.



An example: image histogram

```
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;  // incorrect!
}
```

An example: image histogram

```
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter atomically

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
```

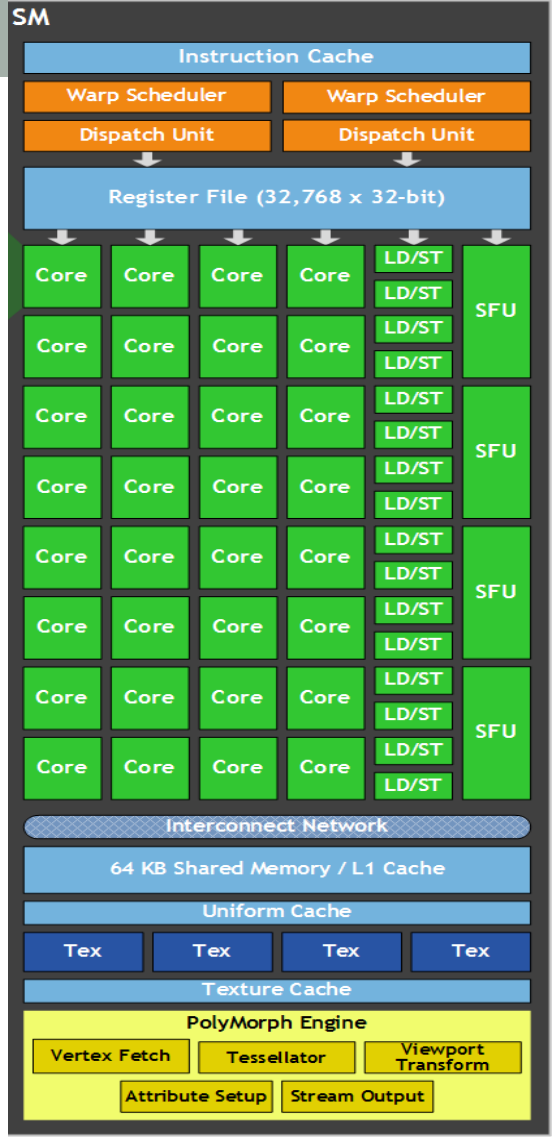
CUDA: OCCUPANCY

Thread Scheduling

- Order of threads within a block is undefined!
  - Threads are grouped in warps (32 threads/warp)
    - AMD calls it “a wavefront” (64 threads/wavefront)
- Order in which thread blocks are mapped and scheduled is undefined!
  - Blocks run to completion on one SM without preemption
  - Can run in any order
    - Any possible interleaving of blocks should be valid
  - Can run concurrently OR sequentially

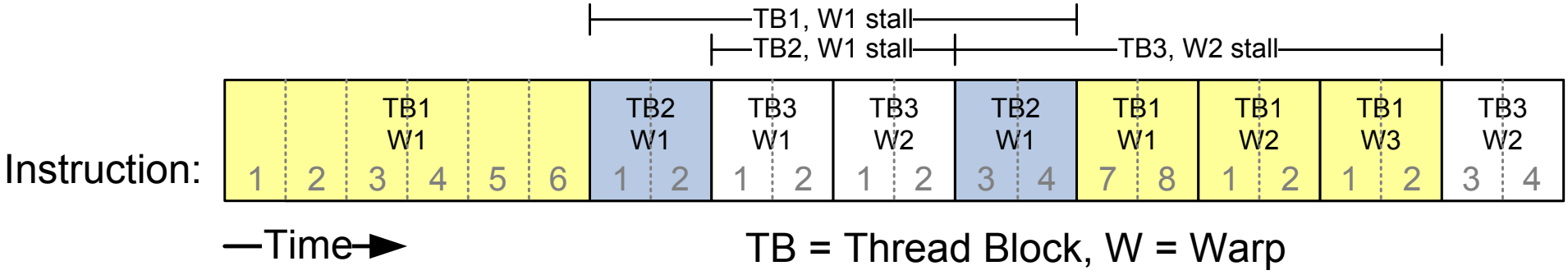
Warps = 32 threads

- Threads are scheduled in warps
  - AMD calls them “wavefronts”
- One warp => on one SM
  - Same SM till completion
- Scheduling
  - GigaThread Unit : schedules blocks per SM’s
  - Inside SM: warp scheduler(s) + instruction dispatcher
    - Replace warps that are stalled by warps waiting to compute
    - Very fast context switching



Thread Scheduling

- SMs implement zero-overhead warp scheduling
  - A warp is a group of 32 threads that runs concurrently on an SM
  - At any time, the number of warps concurrently executed by an SM is limited by its number of cores.
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected



Stalling warps

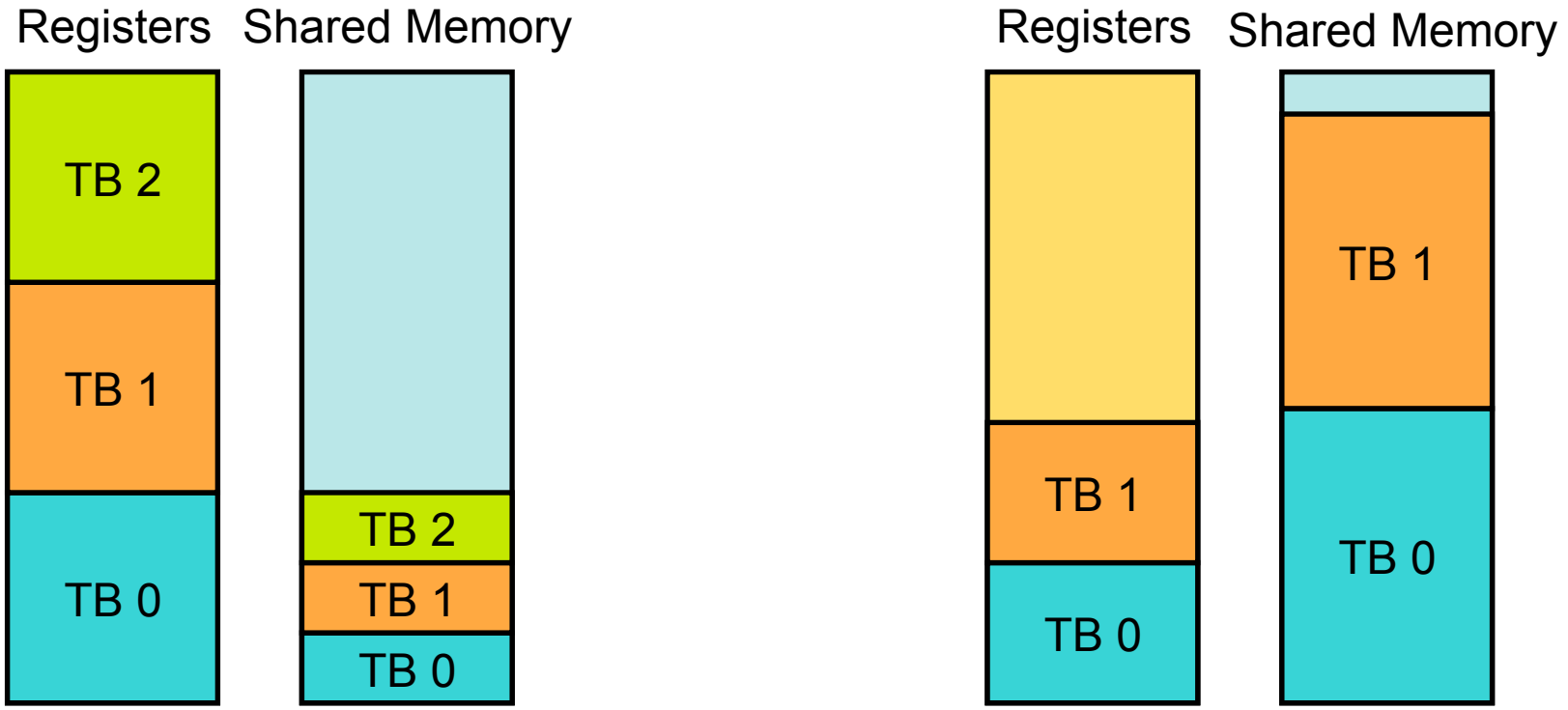
- What happens if all warps are stalled?
  - No instruction issued → performance lost
- Most common reason for stalling?
  - Waiting on global memory
- If your code reads global memory every couple of instructions
  - You should try to maximize **occupancy**

Occupancy

Occupancy = Active Warps / Maximum Active Warps

- Remember: resources are allocated for the entire block!
- Resources are finite
  - Utilizing too many resources per thread may limit the occupancy
- Potential occupancy limiters:
  - Register usage
  - Shared memory usage
  - Block size

Resource Limits



- Pool of registers and shared memory per SM
  - Each thread block grabs registers & shared memory
  - If one or the other is fully utilized => no more thread blocks

How do you know what you're using?

- Use compiler flags to get register and shared memory usage
  - "`nvcc -Xptxas -v`"
- Use the NVIDIA Profiler
- Plug those numbers into CUDA Occupancy Calculator

- Maximize occupancy for improved performance
  - Empirical rule! Don't overuse!

CUDA_Occupancy_calculator.xlsm - Microsoft Excel

Home | Insert | Page Layout | Formulas | Data | Review | View

Security Warning Macros have been disabled. Options...

MyRegCount | 25

1.) Select Compute Capability (click):	1.3	(Help)
2.) Enter your resource usage:		
Threads Per Block	128	(Help)
Registers Per Thread	25	
Shared Memory Per Block (bytes)	640	
(Don't edit anything below this line)		
3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	512	(Help)
Active Warps per Multiprocessor	16	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	50%	

Physical Limits for GPU Compute Capability:	1.3
Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block	
Warps	4
Registers	3584
Shared Memory	1024

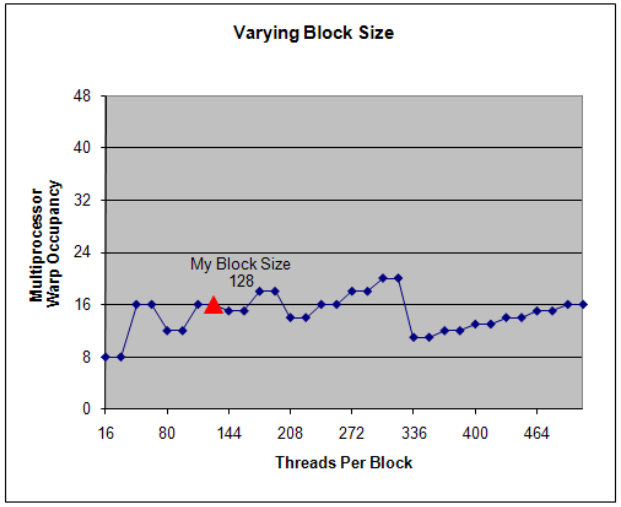
These data are used in computing the occupancy data in blue

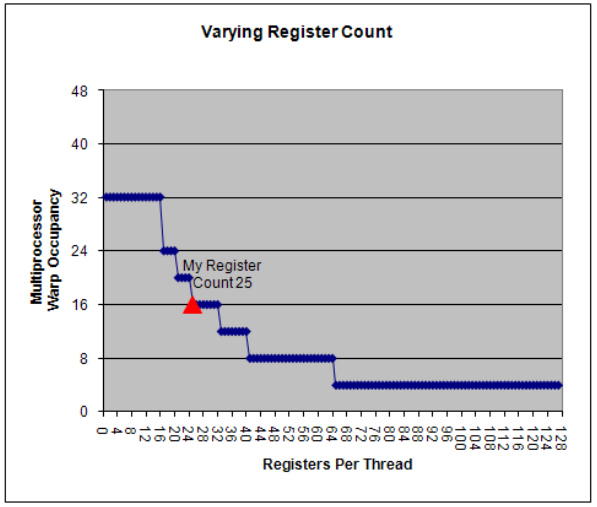
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	8
Limited by Registers per Multiprocessor	4
Limited by Shared Memory per Multiprocessor	16

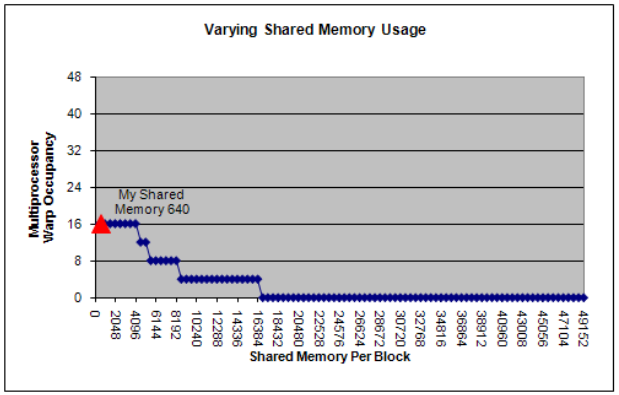
Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator	
Version:	2.0
Copyright and License	

The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







Calculator | Help | GPU Data | Copyright & License

Ready

Thread divergence - penalty?

- Depends on the amount of divergence
  - Worst case: 1/32 performance
    - When each thread does something different
- Depends on whether branching is data- or ID- dependent
  - If ID – consider grouping threads differently
  - If data – consider sorting
- Non-diverging warps => NO performance penalty
  - In this case, branches are not expensive …

CUDA: THREAD DIVERGENCE

Thread divergence

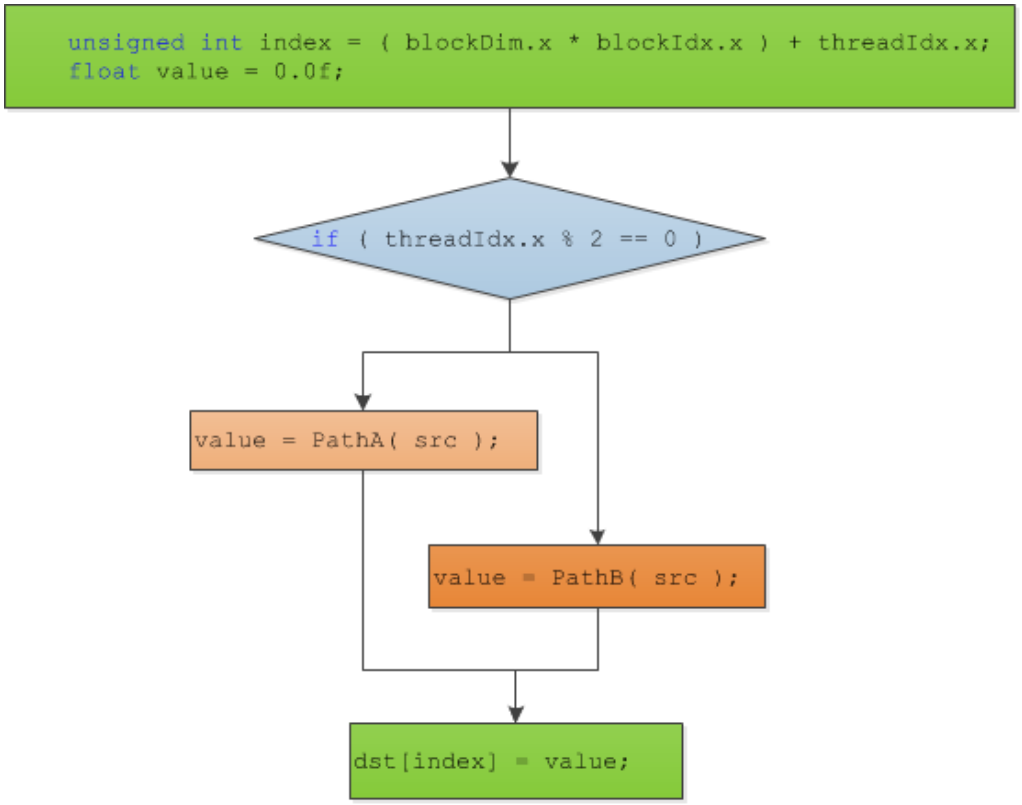
“I heard GPU branching is expensive. Is this true?”

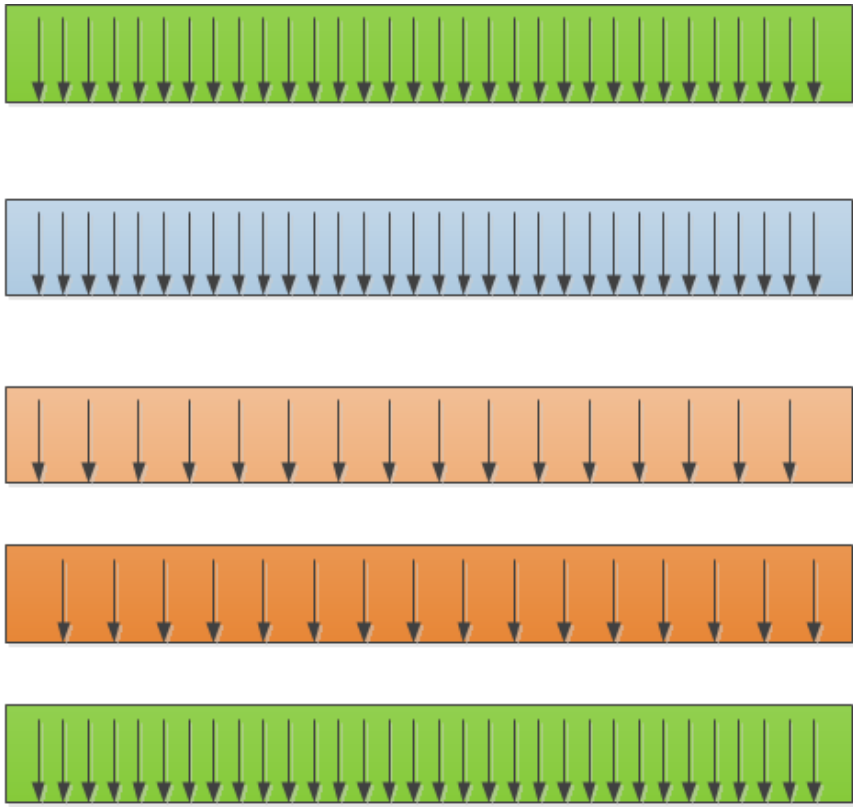
```
__global__ void Divergence(float* dst,float* src )
{
    float value = 0.0f;

    if ( threadIdx.x % 2 == 0 )
// active threads : 50%
        value = src[0] + 5.0f;
    else
// active threads : 50%
        value = src[0] – 5.0f;

    dst[index] = value;
}
```

Execution

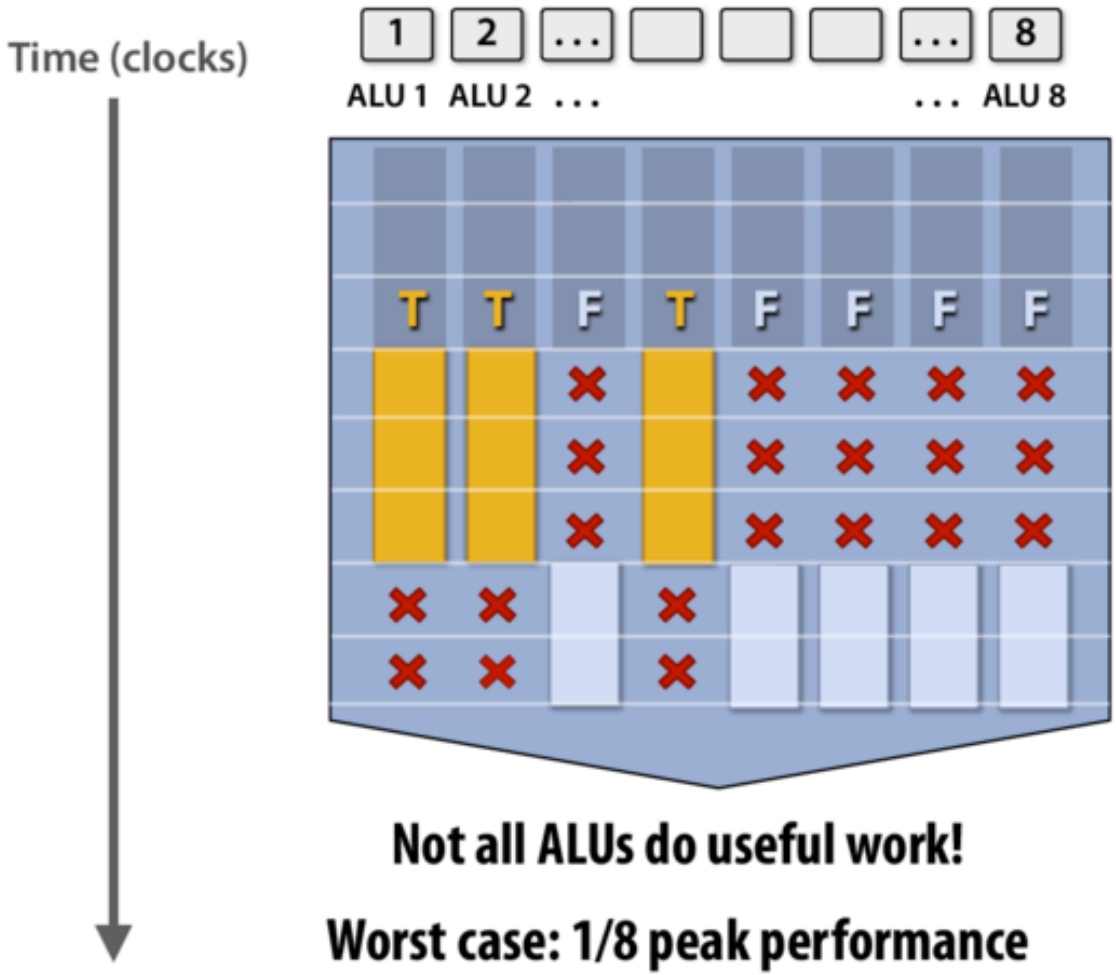




Worst case performance loss:

50% compared with the non divergent case.

Another example



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

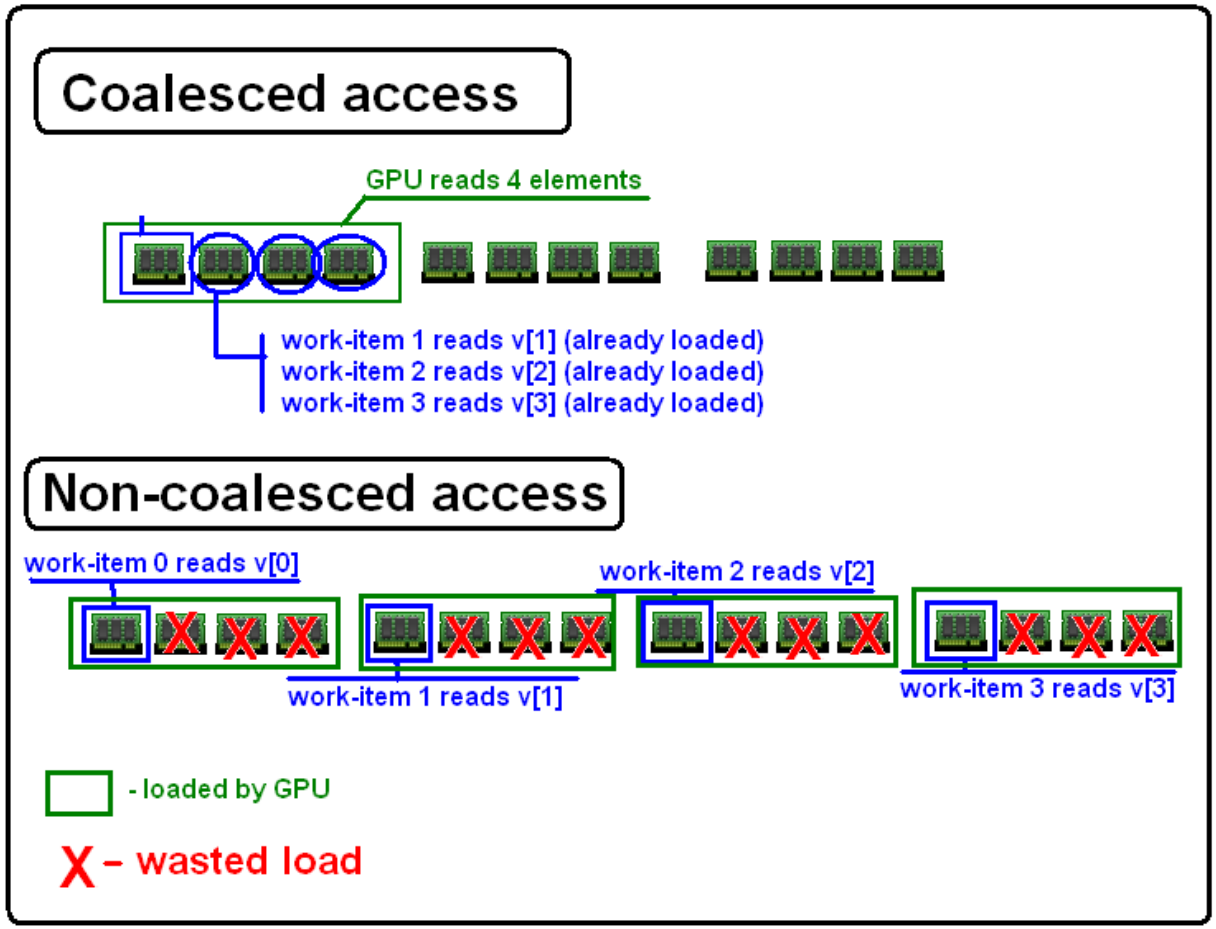
Performance penalty?

- Depends on the amount of divergence
  - Worst case: 1/32 performance
    - When each thread does something different
- Depends on whether branching is data- or ID- dependent
  - If ID – consider grouping threads differently
  - If data – consider sorting
- Non-diverging warps => NO performance penalty
  - In this case, branches are not expensive …

CUDA: MEMORY COALESCING

Memory Coalescing

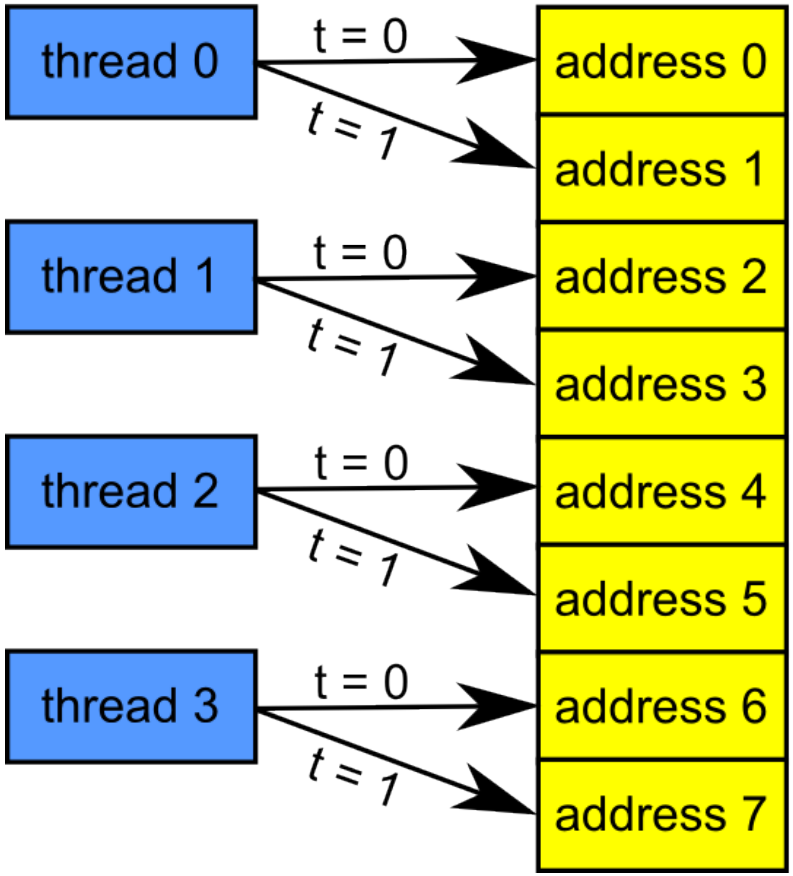
- **Memory coalescing** refers to combining multiple **memory** accesses into a single transaction



Caching vs. Coalescing

traditional multi-core

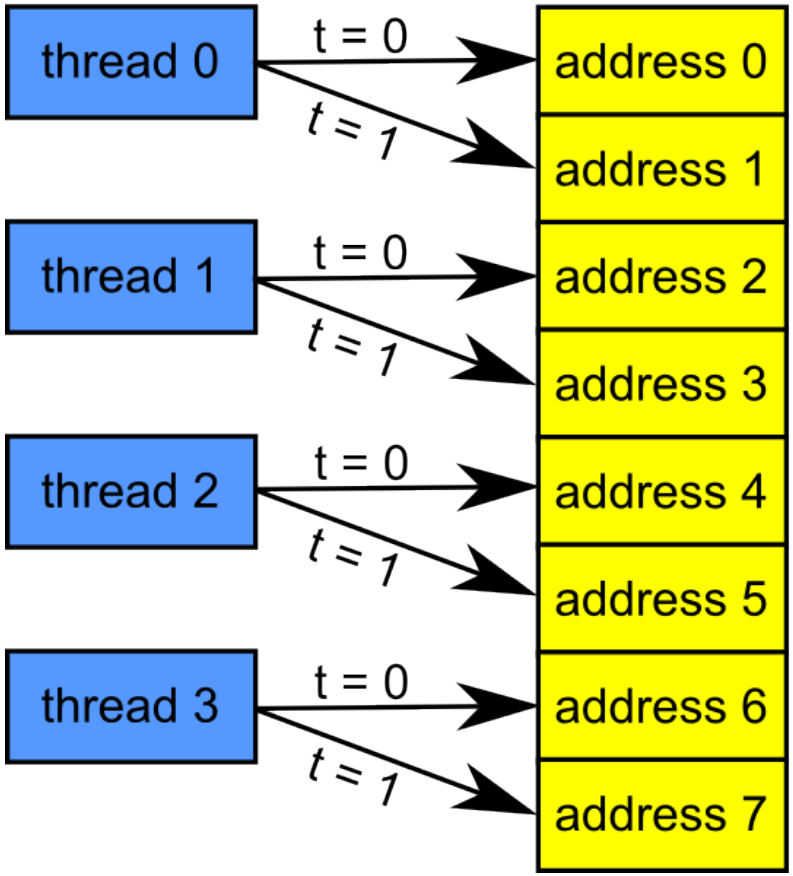
optimal memory access pattern



Caching

Caching vs. Coalescing

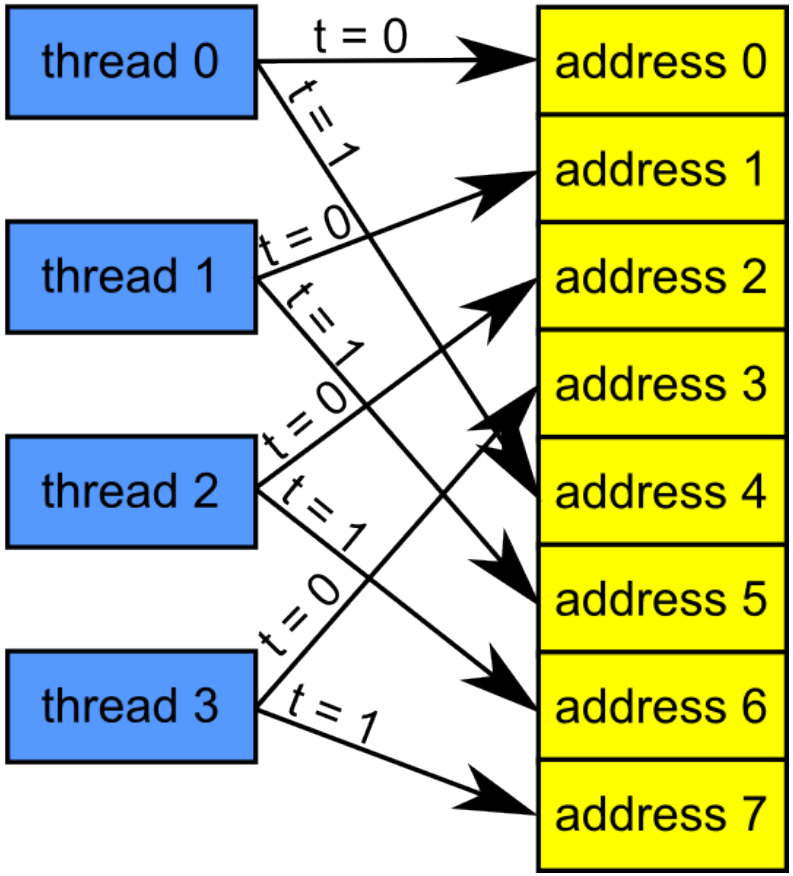
traditional multi-core
optimal memory access pattern



Caching

vs.

many-core GPU
optimal memory access pattern



Coalescin

Consider the stride of your accesses

	Input[0]	Input[1]	Input[2]	Input[3]	Input[4]	Input[5]	Input[6]	Input[7]
Stride1	T0	T1	T2	T3	T4	T5	T6	T7
Stride2		T0		T1		T2		T3
“random”			T7		T1			T4

```
__global__ void foo(int* input, float3* input2) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  // Stride 1, full bandwidth used!
  int a = input[i];
  // Stride 2, 50% of the bandwidth is wasted
  int b = input[2*i+1];
  // “Random” stride – ?? up to 7/8 bandwidth wasted
  int c = input[f(i)];
}
```

Example: Array of Structures (AoS)

```
Struct AoS{
    int key;
    int value;
    int flag;
};

record *d_AoS_data;
cudaMalloc((void**)&d_AoS_data, ...);

kernel {
 threadID = blockDim.x * blockIdx.x + threadIdx.x;
 // …
 d_AoS_data[threadID].value += i; // wastes bandwidth!
 // …
}
```

Example: Structure of Arrays (SoA)

```
Struct SoA {
    int* keys;
    int* values;
    int* flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);

kernel {
  threadID = blockDim.x * blockIdx.x + threadIdx.x;
…
  d_SoA_data.values[threadID] += i; // full
 bandwidth!
… }
```

Memory Coalescing*

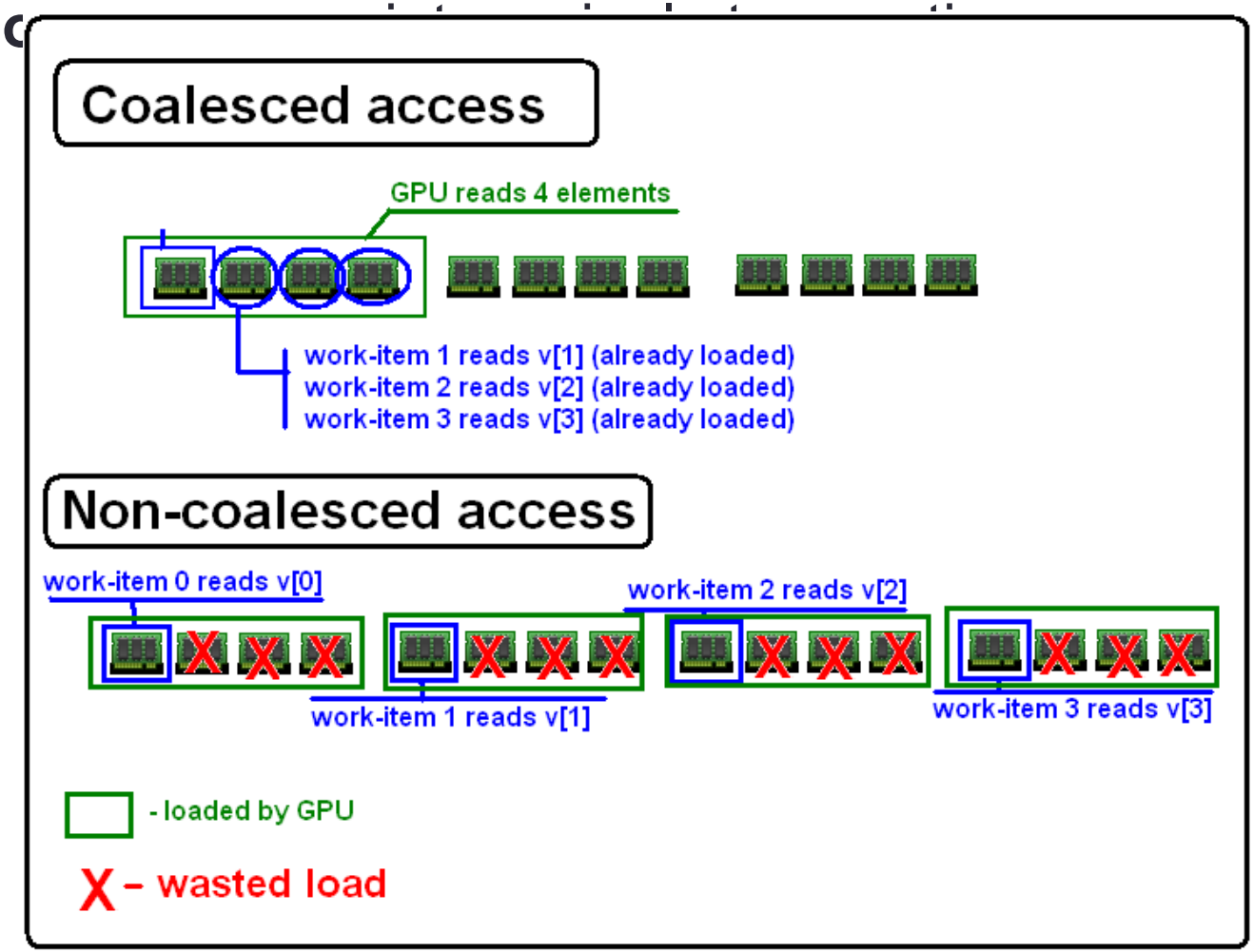
- Group memory accesses in as few memory transactions as possible.
  - 128-byte or 32-byte long lines
- Stride 1 access patterns are preferred!
  - Other patterns can still get benefits
- Structure of arrays is often better than array of structures
- Unpredictable/ irregular access patterns
  - Case-by-case performance impact
- No coalescing => performance loss ~10x or more !
  - Caching might improve this impact …

*http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3nJ0kBsWe

CUDA: MEMORY COALESCING

Memory Coalescing

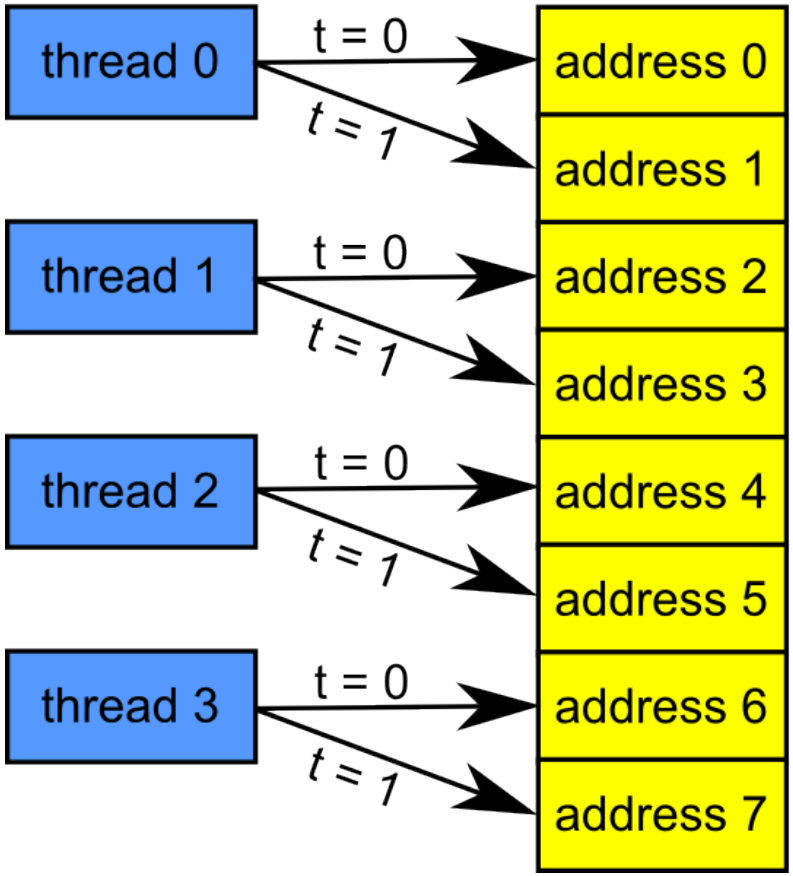
- **Memory coalescing** refers to combining multiple **memo**



Caching vs. Coalesci

traditional multi-core

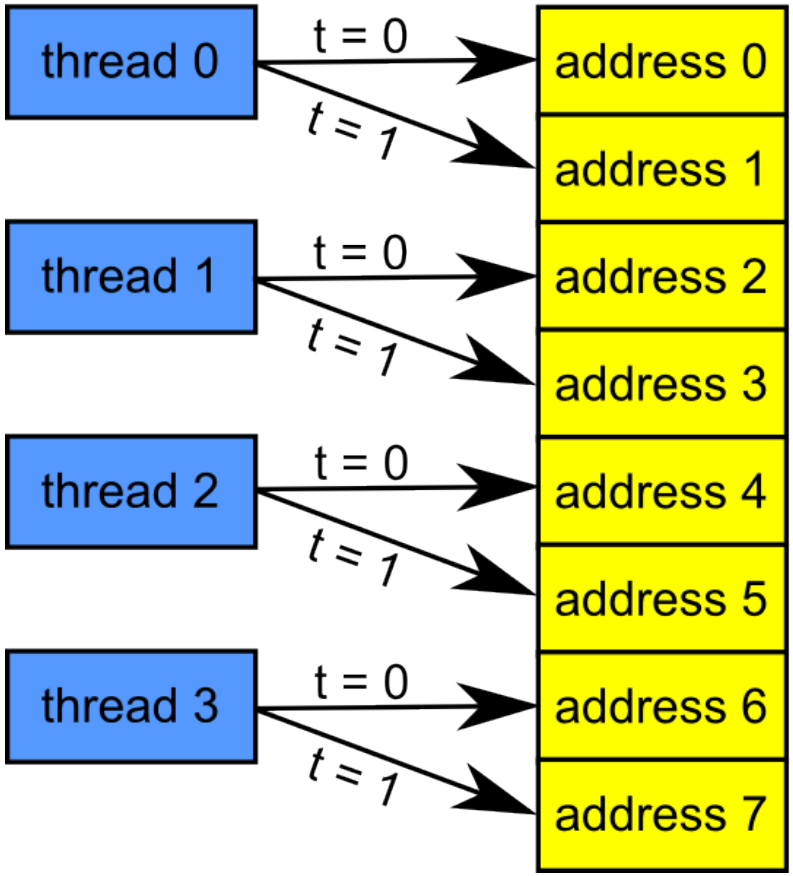
optimal memory access pattern



Caching

Caching vs. Coalescing

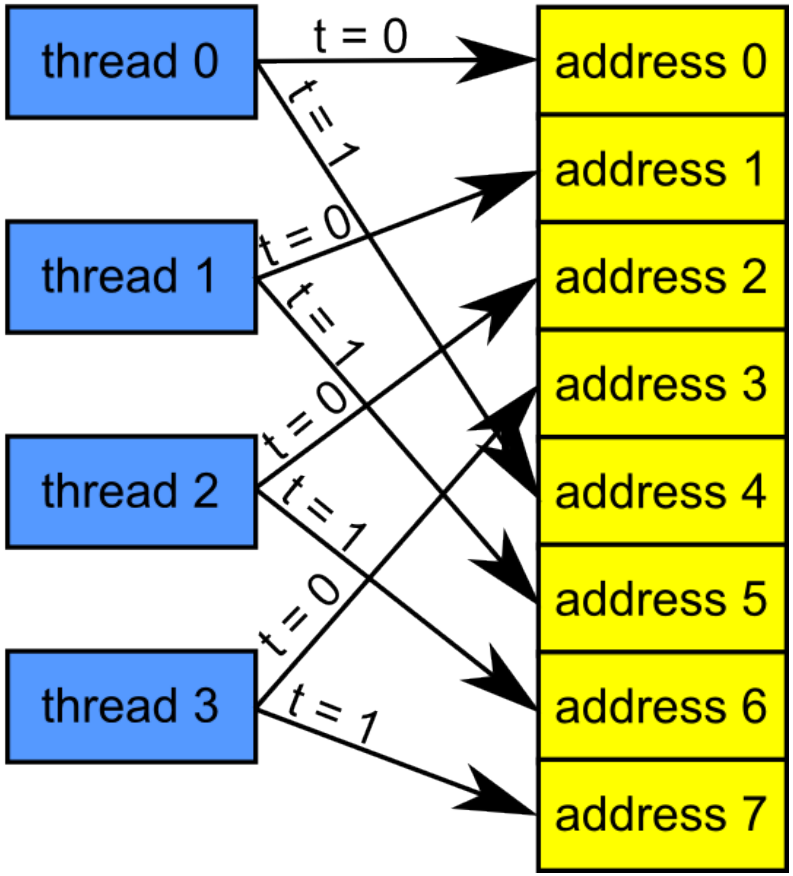
traditional multi-core
optimal memory access pattern



Caching

vs.

many-core GPU
optimal memory access pattern



Coalescin

Consider the stride of your accesses

	Input[0]	Input[1]	Input[2]	Input[3]	Input[4]	Input[5]	Input[6]	Input[7]
Stride1	T0	T1	T2	T3	T4	T5	T6	T7
Stride2		T0		T1		T2		T3
“random”			T7		T1			T4

```
__global__ void foo(int* input, float3* input2) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  // Stride 1, full bandwidth used!
  int a = input[i];
  // Stride 2, 50% of the bandwidth is wasted
  int b = input[2*i+1];
  // “Random” stride – ?? up to 7/8 bandwidth wasted
  int c = input[f(i)];
}
```

Example: Array of Structures (AoS)

```
Struct AoS{
    int key;
    int value;
    int flag;
};

record *d_AoS_data;
cudaMalloc((void**)&d_AoS_data, ...);

kernel {
 threadID = blockDim.x * blockIdx.x + threadIdx.x;
 // …
 d_AoS_data[threadID].value += i; // wastes bandwidth!
 // …
}
```

Example: Structure of Arrays (SoA)

```
Struct SoA {
    int* keys;
    int* values;
    int* flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);

kernel {
  threadID = blockDim.x * blockIdx.x + threadIdx.x;
…
  d_SoA_data.values[threadID] += i; // full
 bandwidth!
… }
```

Memory Coalescing*

- Group memory accesses in as few memory transactions as possible.
  - 128-byte or 32-byte long lines
- Stride 1 access patterns are preferred!
  - Other patterns can still get benefits
- Structure of arrays is often better than array of structures
- Unpredictable/ irregular access patterns
  - Case-by-case performance impact
- No coalescing => performance loss 10 – 30x !

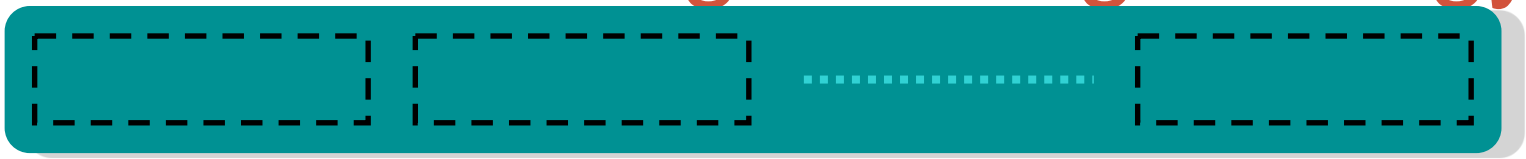
*http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#axzz3nJ0kBs

CUDA: USING SHARED MEMORY

Using shared memory

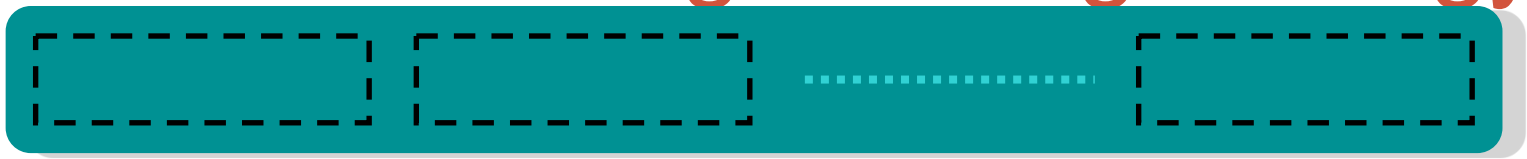
- Equivalent with providing software caching
  - Explicit: Load data to be re-used in shared memory
  - Use it for computation
  - Explicit: Store results back to global memory
- All threads in a block share memory
  - Load/Store: using all threads
  - Barrier: __syncthreads
    - Guard against using uninitialized data – not all threads have finished loading data to shared memory
    - Guard against corrupting live data – not all threads have finished computing

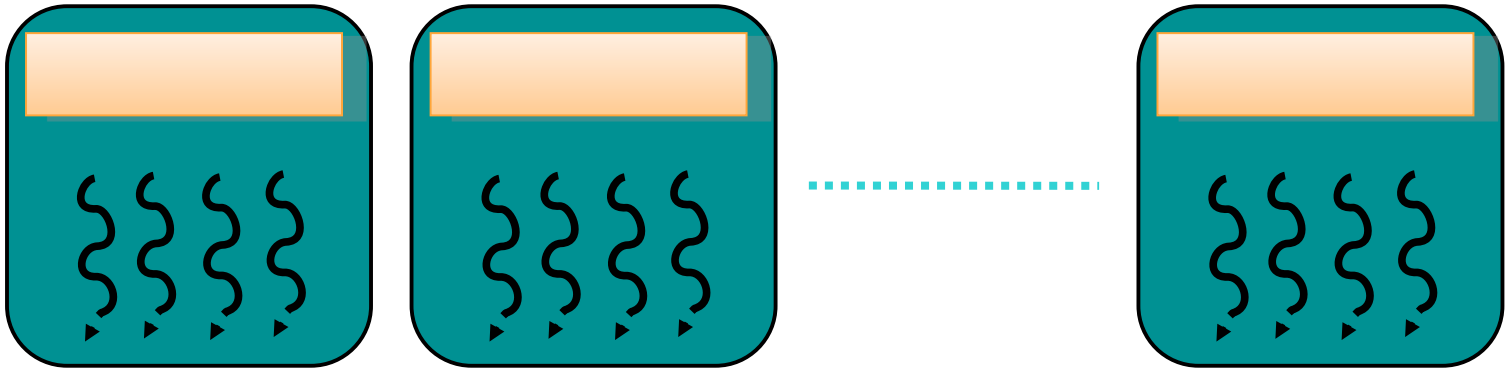
A Common Programming Strategy



- Partition data into subsets that fit into shared memory

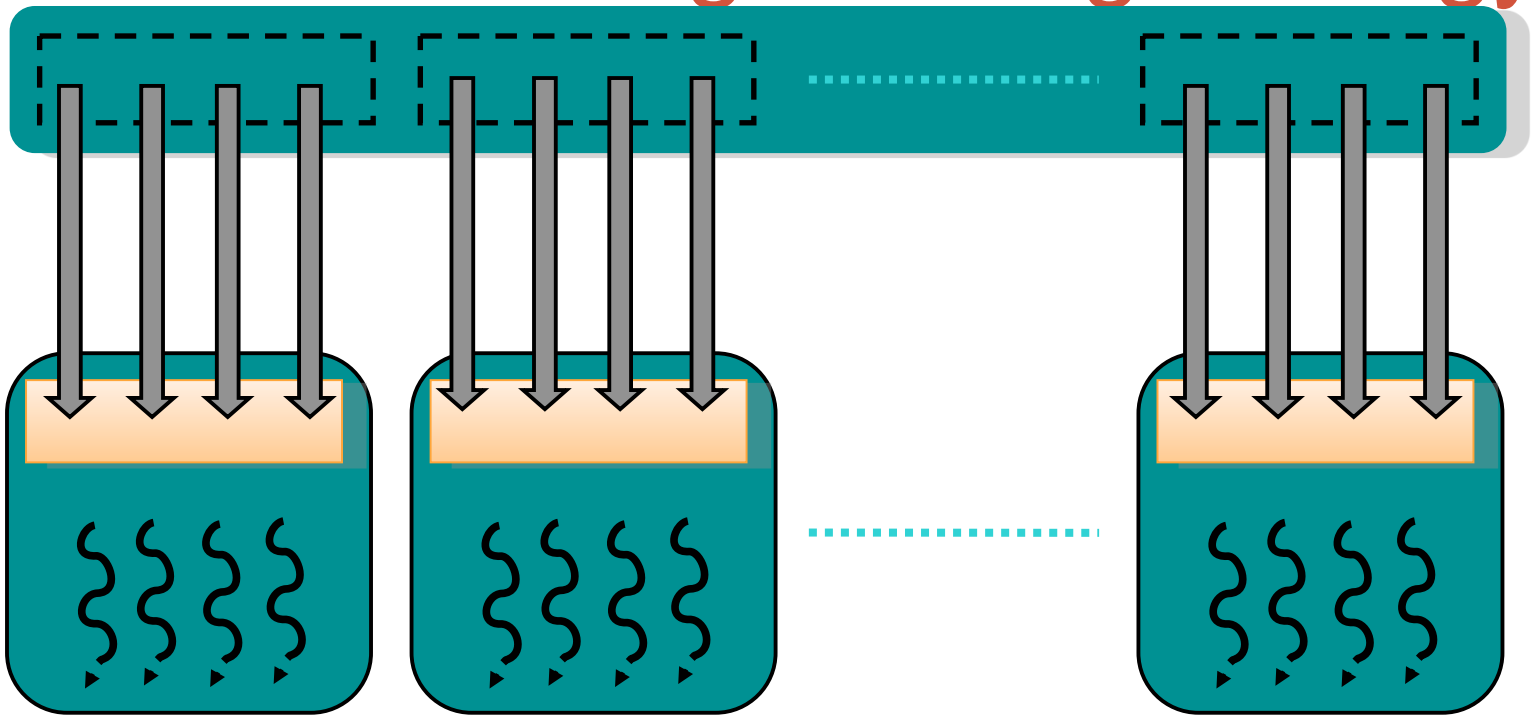
A Common Programming Strategy





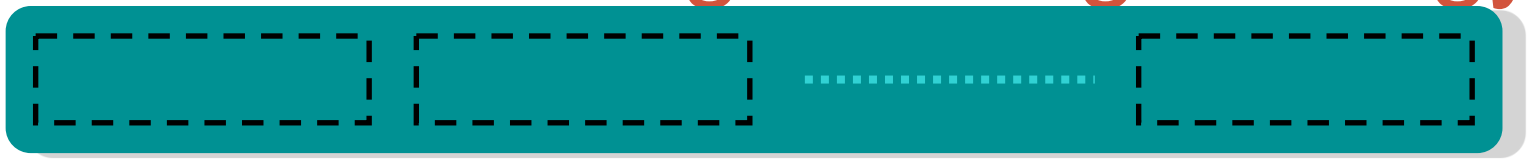
- Handle each data subset with one thread block

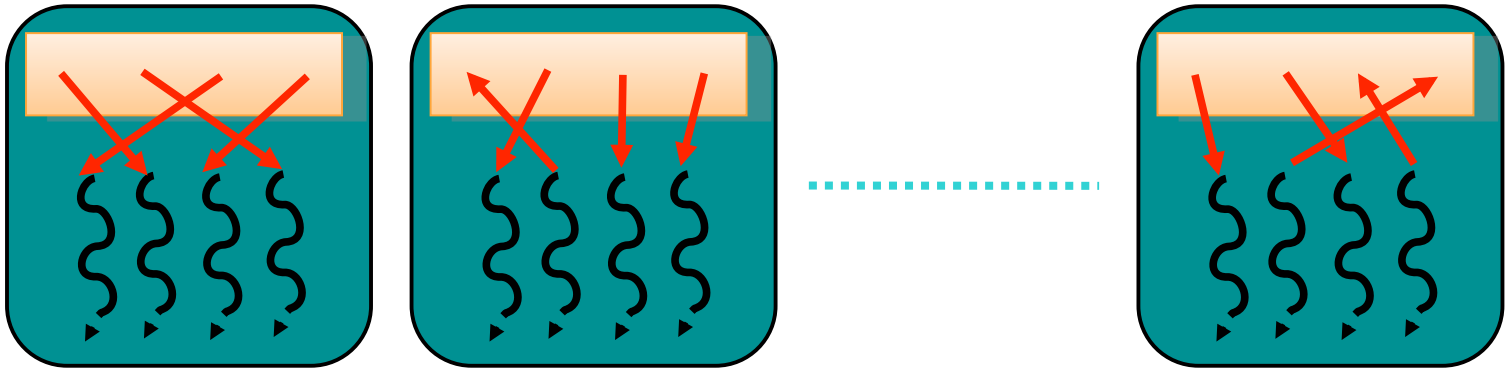
A Common Programming Strategy



- Load the subset from device memory to shared memory, using multiple threads to exploit memory-level parallelism

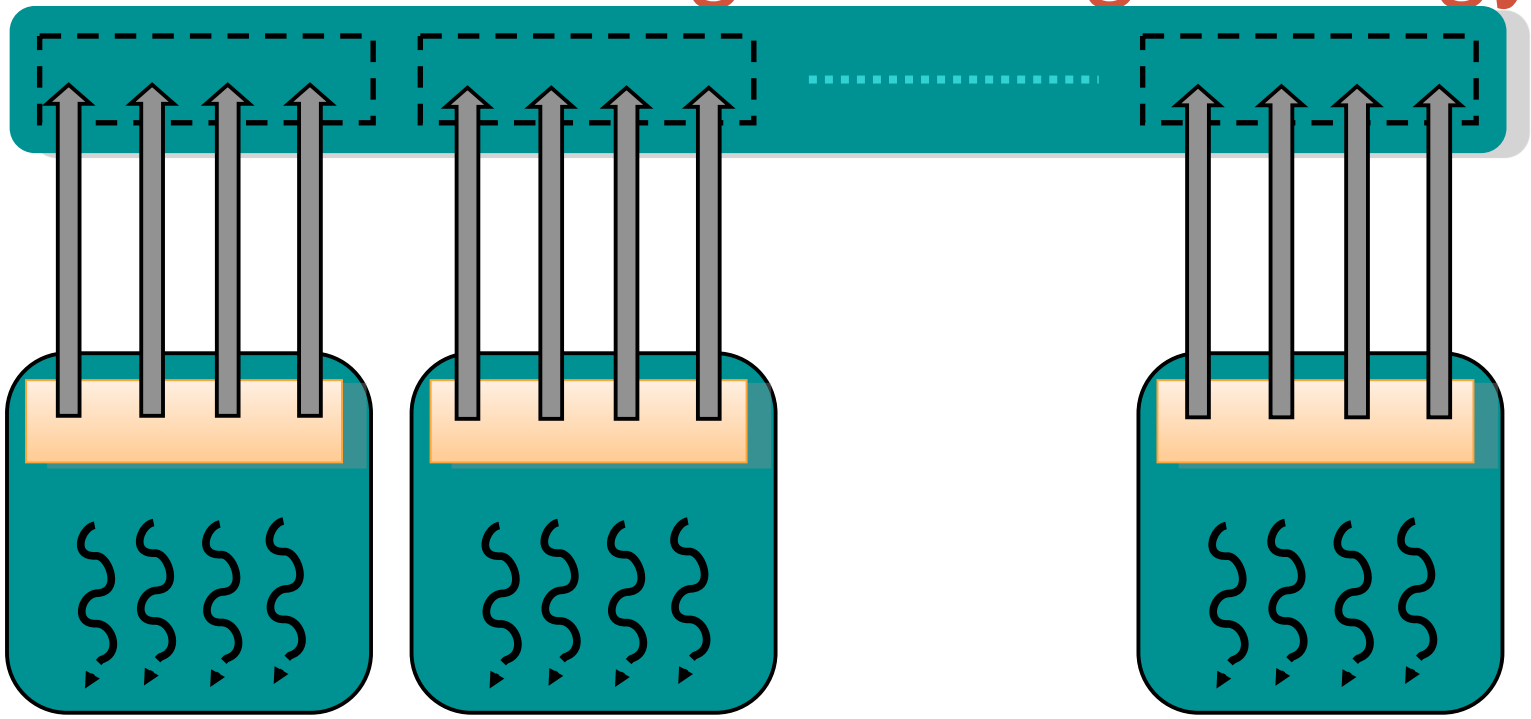
A Common Programming Strategy





- Perform the computation on the subset from shared memory

A Common Programming Strategy



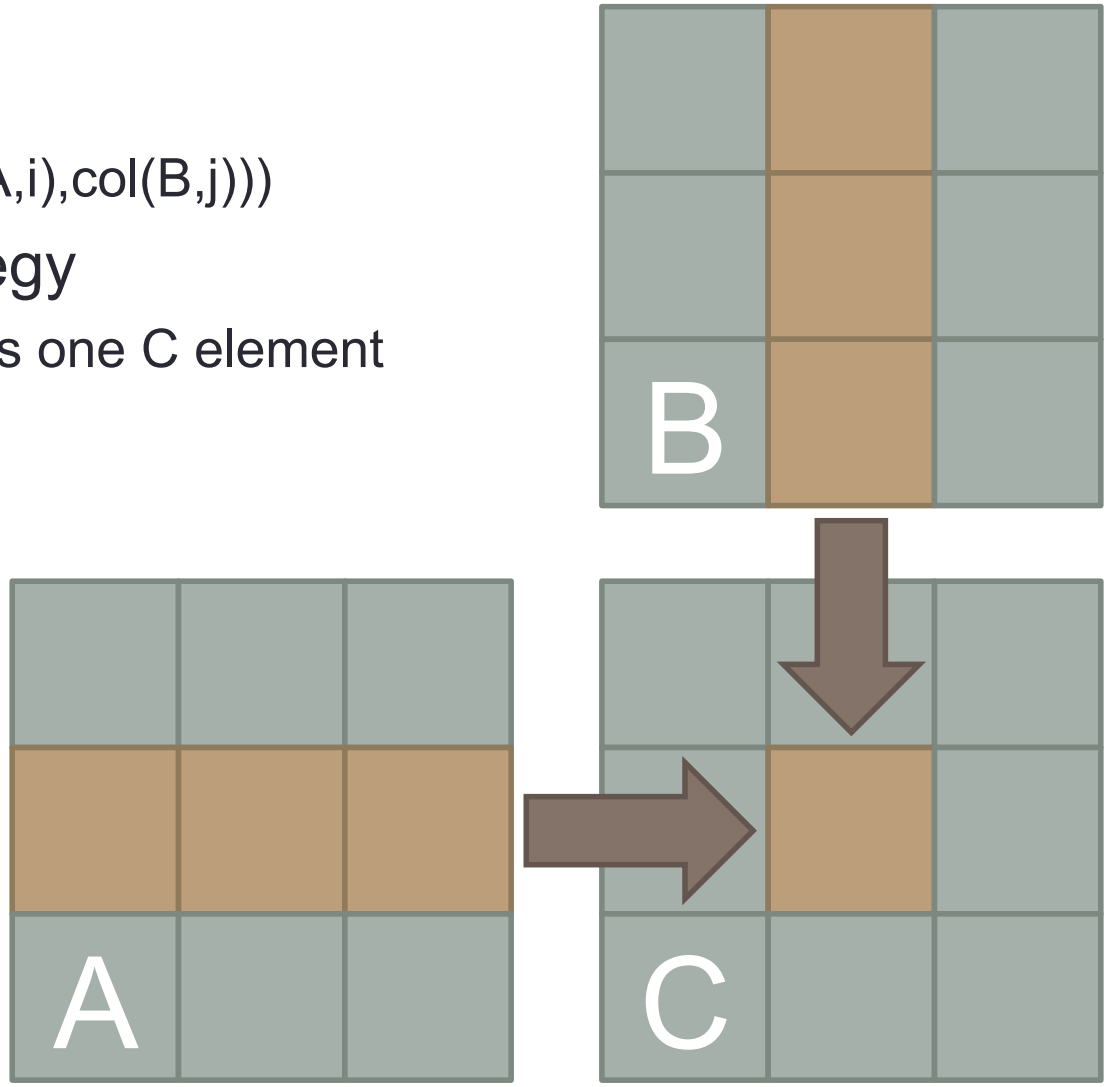
- Copy the result from shared memory back to device memory

Caches vs. Shared Memory

- Since Fermi, NVIDIA GPUs feature BOTH hardware **L1 caches** and **shared memory** per SM
  - They share the same space
    - ¾ Cache + ¼ Shared Memory OR
    - ¼ Cache + ¾ Shared Memory
- L1 Cache
  - Hardware caching enabled
    - The HW decides what goes in or out and when
- Shared memory
  - Software manages what goes in/out
  - Allows more complex access patterns to be cached

Example: Matrix multiplication

- C = A * B
  - C(i,j) = sum(dot(row(A,i),col(B,j)))
- Parallelization strategy
  - Each thread computes one C element
  - 2D kernel

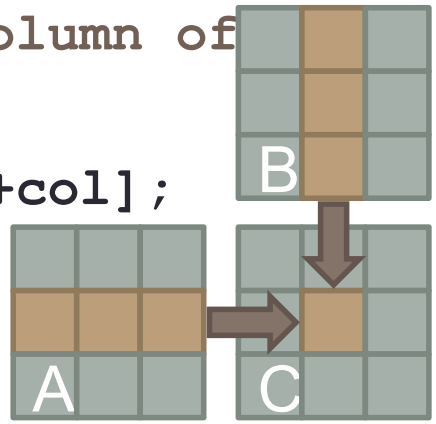


Matrix multiplication implementation

```
__global__ void mat_mul(float *a, float *b,
                        float *c, int width)
{
  // calc row & column index of output element
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  int col = blockIdx.x*blockDim.x + threadIdx.x;

 float result = 0;

  // do dot product between row of a and column of
  for(int k = 0; k < width; k++) {
      result += a[row*width+k] * b[k*width+col];
 }
  c[row*width+col] = result;
}
```

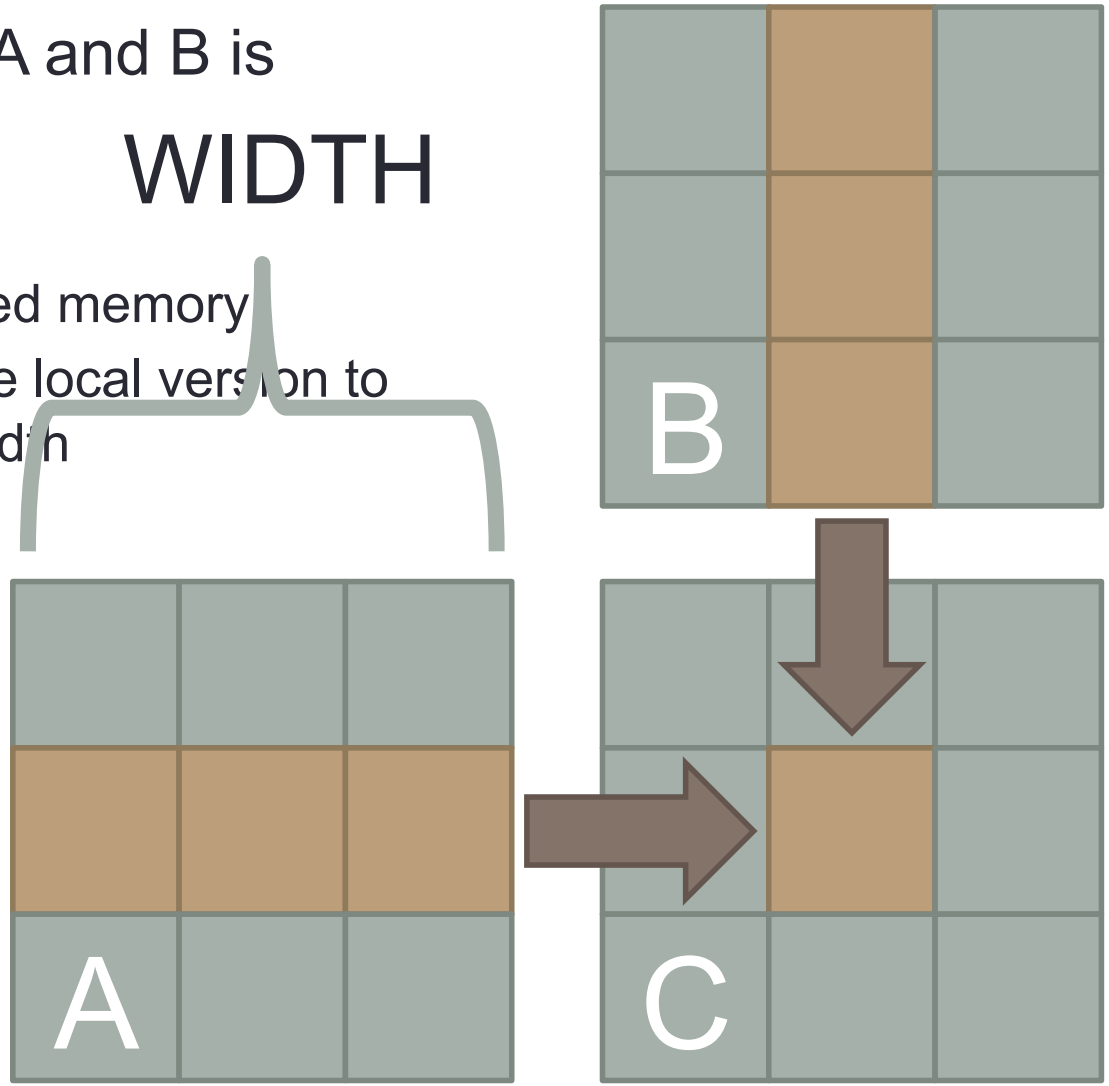


Matrix multiplication performance

Loads per dot product term	2 (a and b) = 8 bytes
FLOPS	2 (multiply and add)
AI	2 / 8 = 0.25
Performance GTX 580	1581 GFLOPs
Memory bandwidth GTX 580	192 GB/s
Attainable performance	192 * 0.25 = 48 GFLOPS
Maximum efficiency	3.0 % of theoretical peak

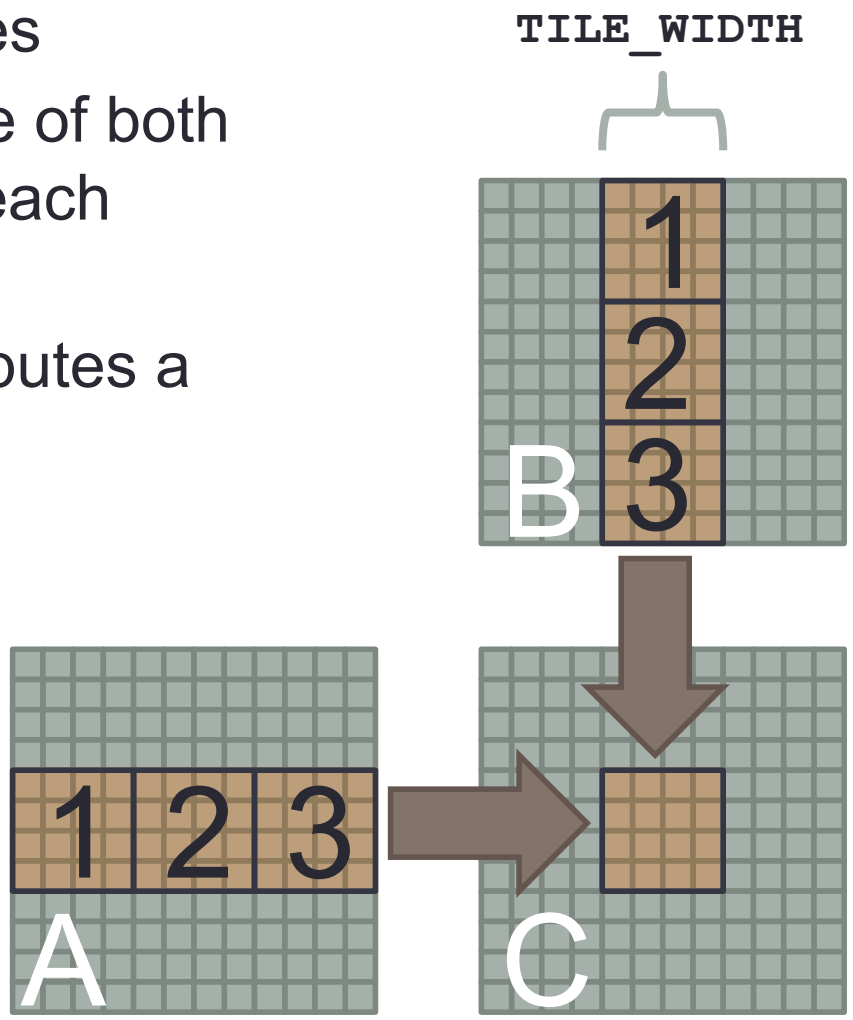
Data reuse

- Each input element in A and B is read WIDTH times
- IDEA:
  - Load elements into shared memory
  - Have several threads use local version to improve memory bandwidth



Using shared memory

- Partition kernel loop into phases
- In each thread block, load a tile of both matrices into shared memory each phase
- Each phase, each thread computes a partial result



Matrix multiply with shared memory

```
__global__ void mat_mul(float *a, float *b,
                        float *c, int width) {
  // shorthand
  int tx = threadIdx.x, ty = threadIdx.y;
  int bx = blockIdx.x,  by = blockIdx.y;

  // allocate tiles in shared memory
  __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
  __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

  // calculate the row & column index from A,B
  int row = by*blockDim.y + ty;
  int col = bx*blockDim.x + tx;

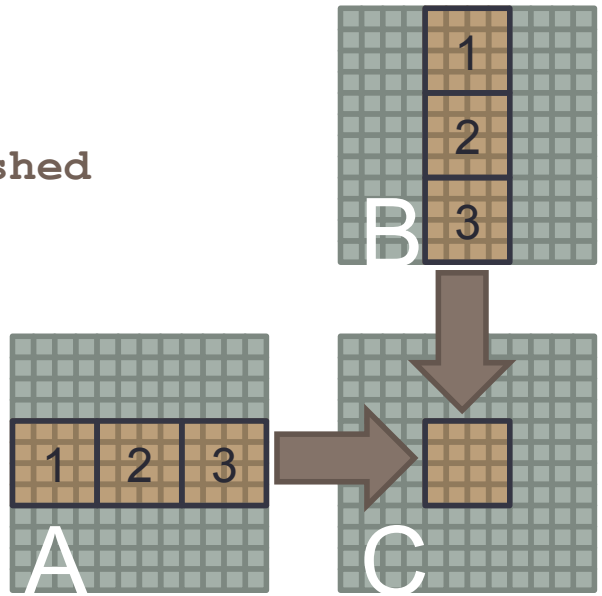
 float result = 0;
```

Matrix multiply with shared memory

```
  // loop over input tiles in phases, p = crt. phase
  for(int p = 0; p < width/TILE_WIDTH; p++) {
    // collaboratively load tiles into shared memory
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
// barrier: ALL writes to shared memory finished
    __syncthreads();

    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; k++) {
      result += s_a[ty][k] * s_b[k][tx];
    }
// barrier: ALL reads of shared memory finished
    __syncthreads();
  }

  c[row*width+col] = result;
}
```



Use of Barriers in mat_mul

- Two barriers per phase:
  - __syncthreads after all data is loaded into shared memory
  - __syncthreads after all data is read from shared memory
    - Second __syncthreads in phase p guards the load in phase p+1

- Formally, __synchthreads is a barrier for shared memory for a block of threads:

"void __syncthreads();

waits until all threads in the thread block have reached this point **and** all global and shared memory accesses made by these threads prior to __syncthreads() are visible to all threads in the block."

Matrix multiplication performance

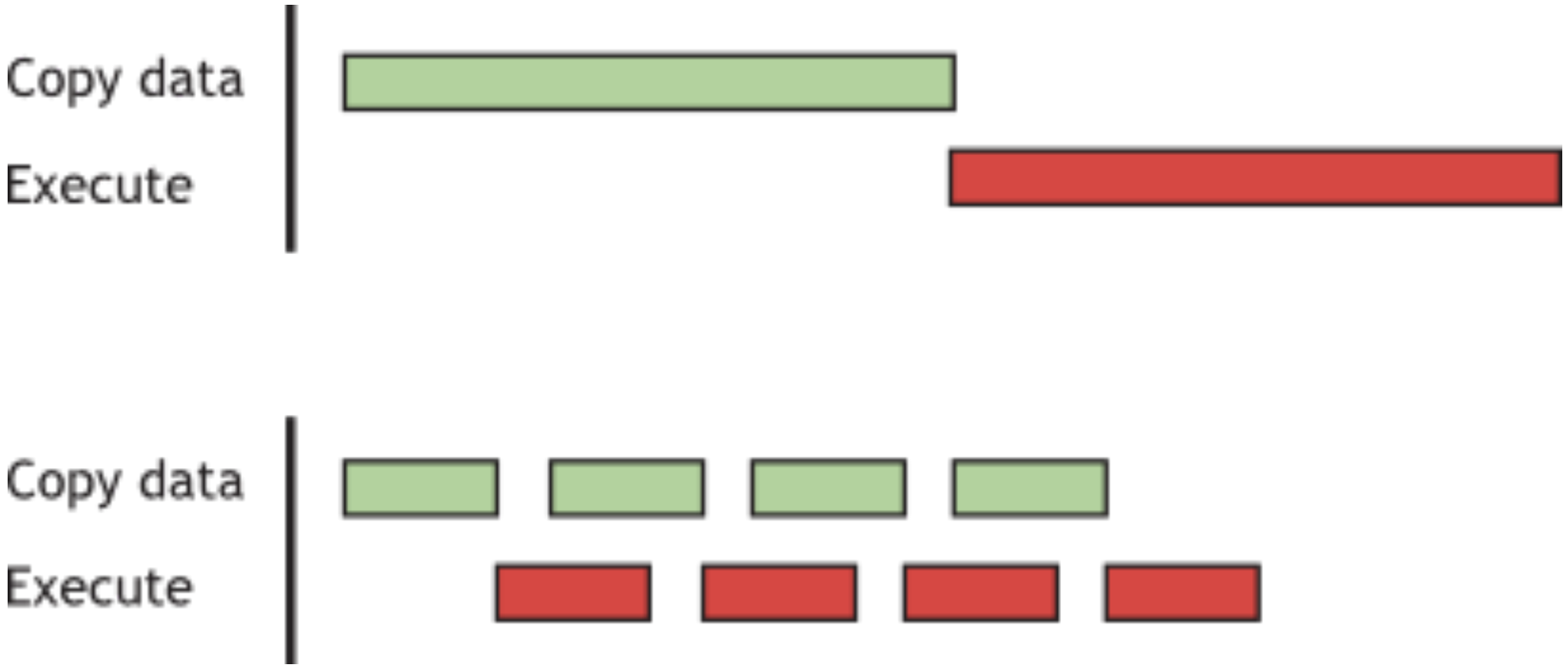
	Original	shared memory
Global loads	$2N^3$ * 4 bytes	($2N^3$ / TILE_WIDTH) * 4 bytes
Total ops	$2N^3$	$2N^3$
AI	0.25	0.25 * TILE_WIDTH

Performance GTX 580	1581 GFLOPs
Memory bandwidth GTX 580	192 GB/s
AI needed for peak	1581 / 192 = **8.23**
TILE_WIDTH required to achieve peak	0.25 * TILE_WIDTH = **8.23**, TILE_WIDTH = 32.9

CUDA: STREAMS

Overlap Computation and Communication

- Main idea: while executing a kernel, bring data in for the next kernel:



What are streams?

- **Stream** = a sequence of operations that execute on the device in the order in which they are issued by the host code.

- Same stream: In-Order execution
- Different streams: Out-of-Order execution

- Default stream = Synchronizing stream
  - No operation in the default stream can begin until all previously issued operations in any stream on the device have completed.
  - An operation in the default stream must complete before any other operation in any stream on the device can begin.

Default stream: example

```
cudaMemcpy(d_a, a, numBytes,cudaMemcpyHostToDevice);
 increment<<<1,N>>>(d_a);
 CpuFunction(b);
cudaMemcpy(a, d_a, numBytes,cudaMemcpyDeviceToHost);
```

- All operations happen in the same stream
- Device (GPU)
  - Synchronous execution
    - all operations execute (in order), one after the previous has finished
  - Unaware of `CpuFunction()`
- Host (CPU)
  - Launches `increment` and regains control
  - *May* execute `CpuFunction` *before* increment has finished
  - Final copy starts *after* both `increment` and `CpuFunction()` have finished

Non-default streams

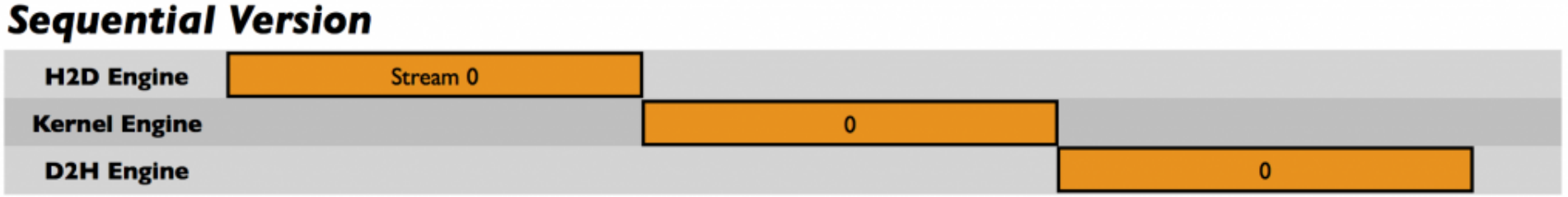
- Enable asynchronous execution and overlaps
  - Require special creation/deletion of streams
    - `cudaStreamCreate(&stream1)`
    - `cudaStreamDestroy(stream1)`
  - Special memory operations
    - `cudaMemcpyAsync(deviceMem, hostMem, size, cudaMemcpyHostToDevice, stream1)`
  - Special kernel parameter (the 4th one)
    - `increment<<<1, N, 0, stream1>>>(d_a)`
- Synchronization
  - All streams
    - `cudaDeviceSynchronize()`
  - Specific stream:
    - cudaStreamSyncrhonize(stream1)

Computation vs. communication

```
//Single stream, numBytes = 16M, numElements = 4M
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
kernel<<blocks,threads>>(d_a, firstElement);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

C1060 (pre-Fermi): 12.9ms

C2050 (Fermi): 9.9ms

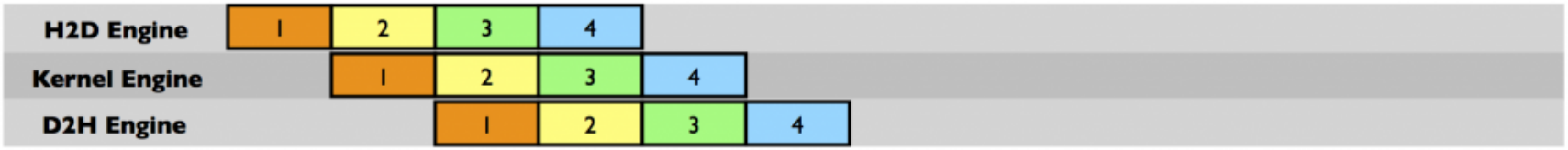


Computation-communication overlap[1]*

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
stream[i]);
  kernel<<blocks,threads,0,stream[i]>>(d_a, offset);
  cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
stream[i]);
}
```

C1060 (pre-Fermi): 13.63 ms (worse than sequential)

C2050 (Fermi): 5.73 ms (better than sequential)



https://github.com/parallel-forall/code-samples/blob/master/series/cuda-cpp/overlap-data-transfers/async

Computation-communication overlap[2]*

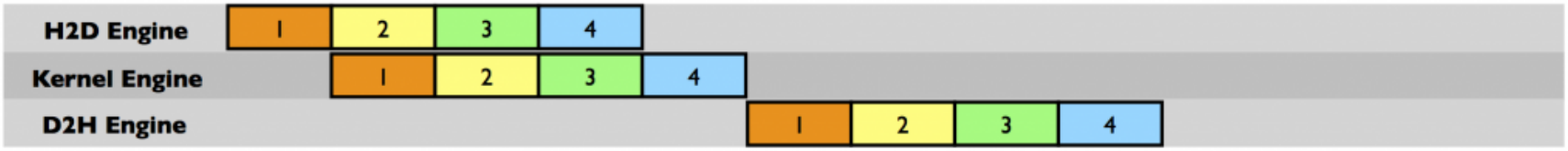
```
for (int i = 0; i < nStreams; ++i) offset[i]=i * streamSize;
for (int i = 0; i < nStreams; ++i)
   cudaMemcpyAsync(&d_a[offset[i]], &a[offset[i]], streamBytes,
      cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < nStreams; ++i)
      kernel<<blocks,threads,0,stream[i]>>(d_a, offset);

for (int i = 0; i < nStreams; ++i)
   cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
      cudaMemcpyDeviceToHost, stream[i]);
```

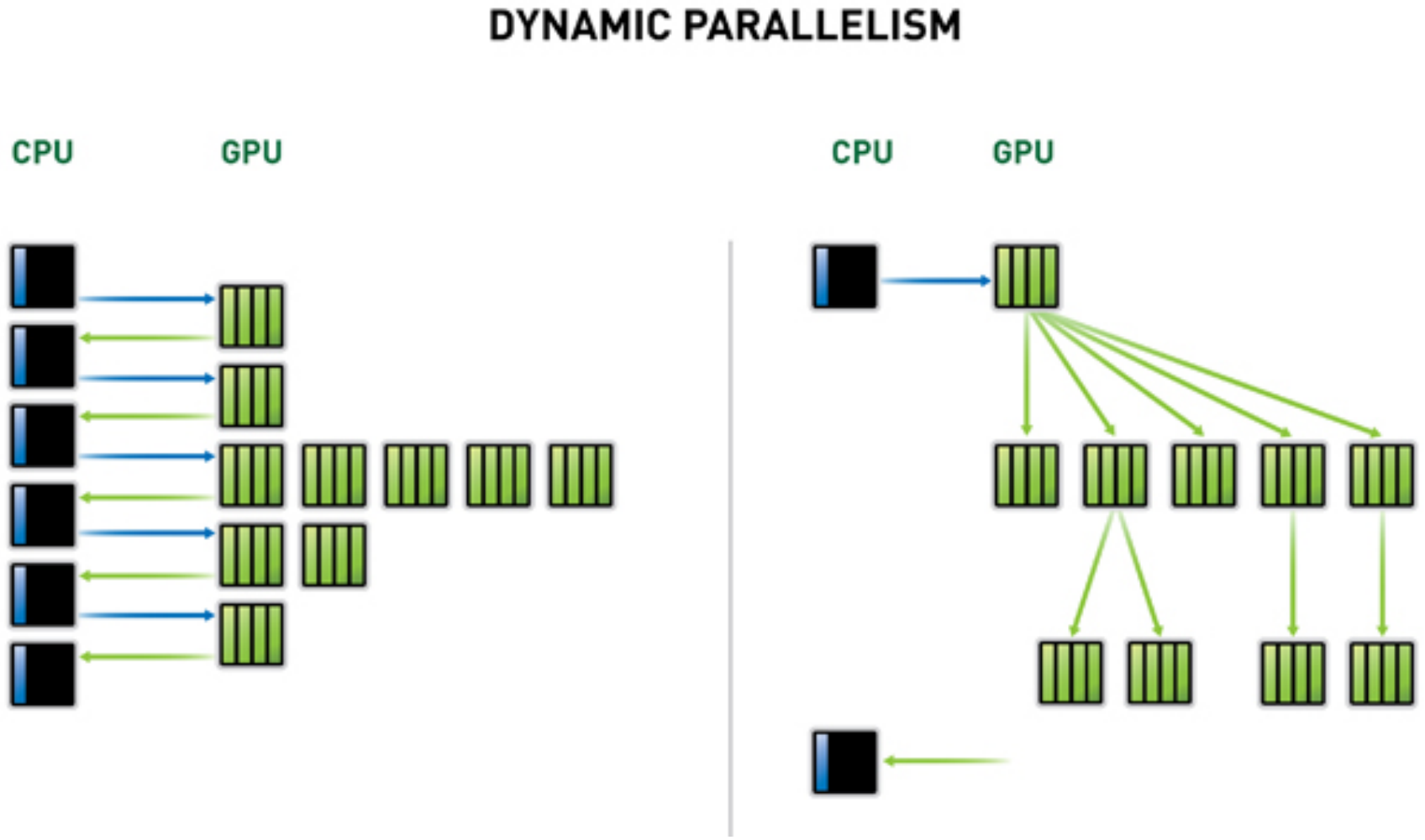
C1060 (pre-Fermi): 8.84 ms (better than sequential)

C2050 (Fermi): 7.59 ms (better than sequential, worse than v1)

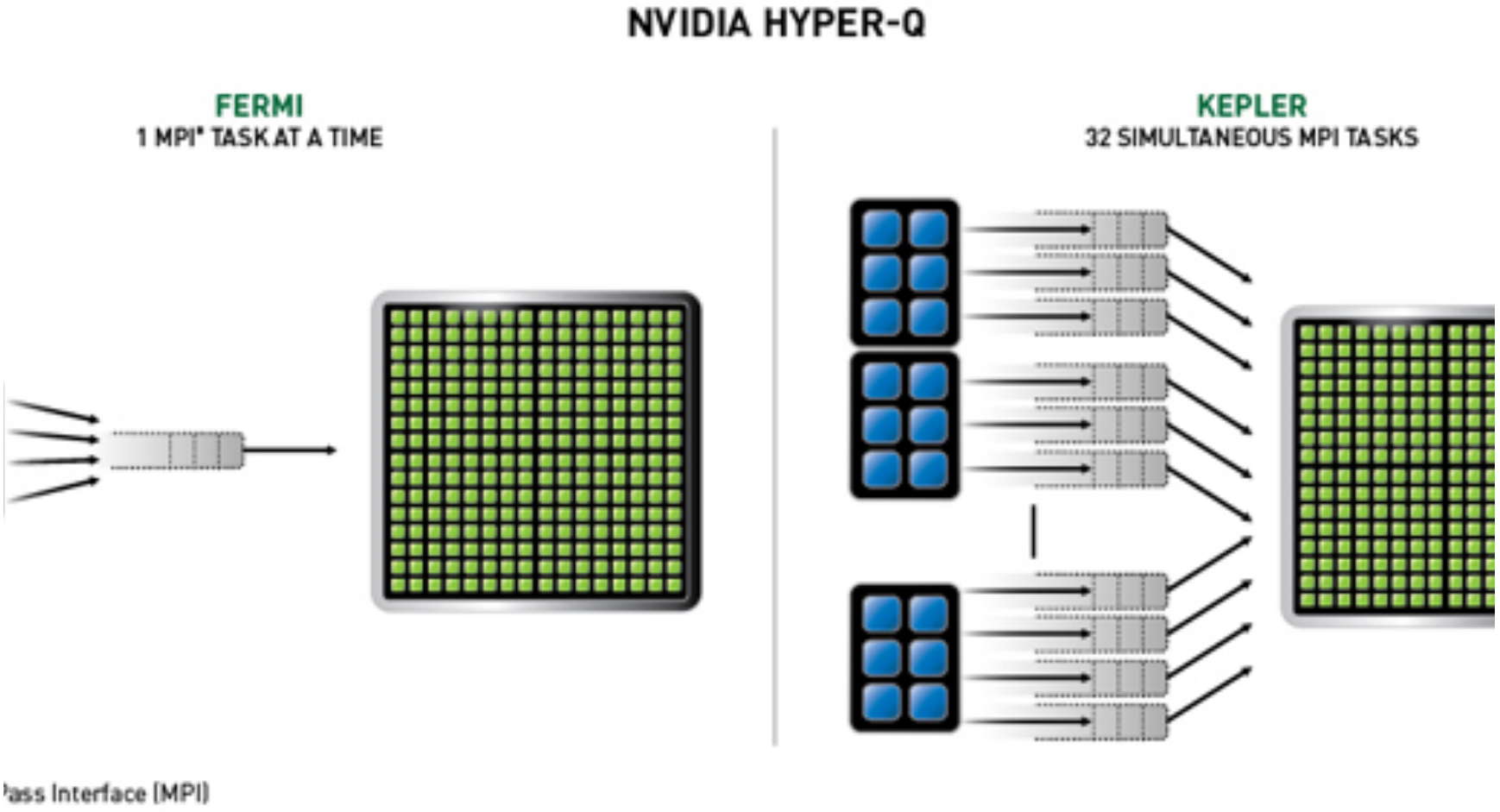


CUDA: MANY OTHER FEATURES

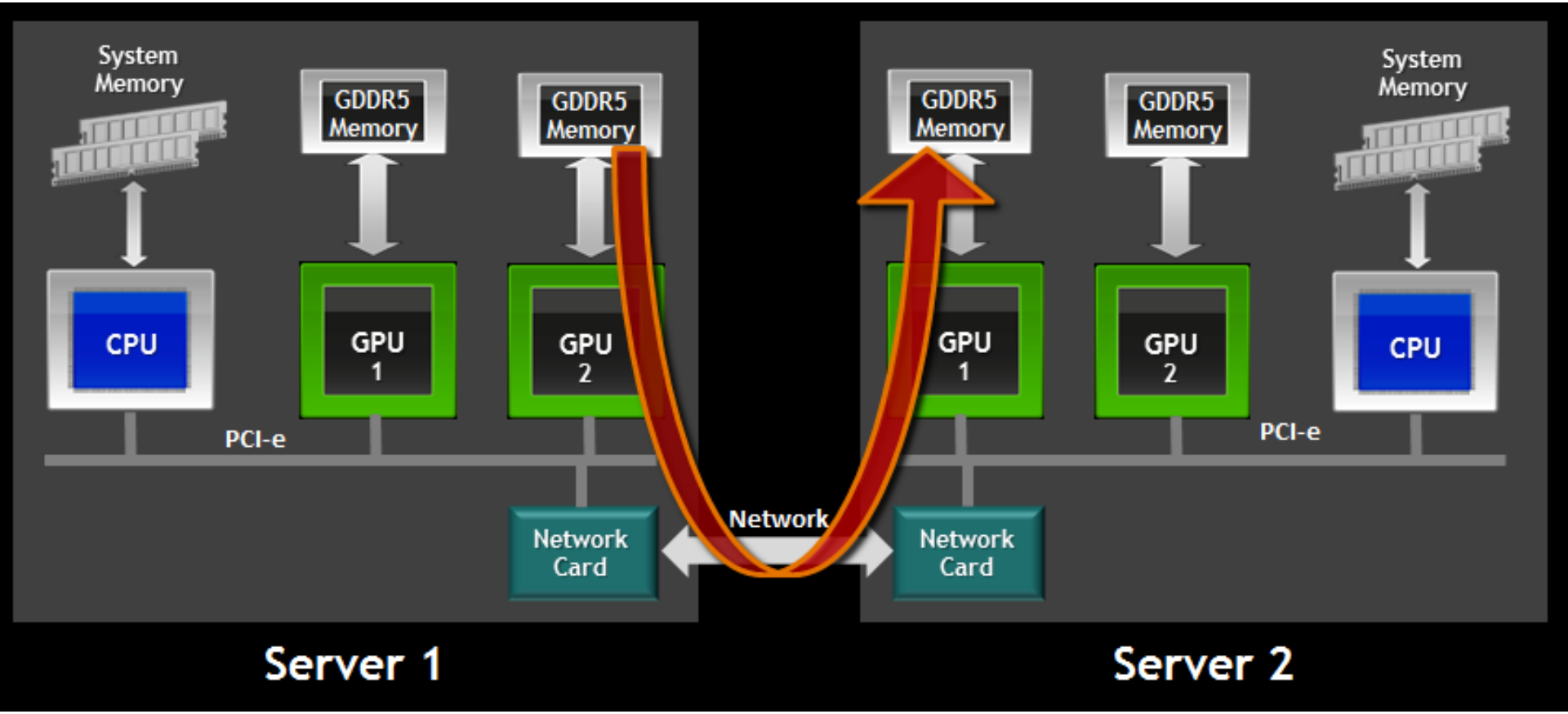
Kepler: Dynamic parallelism



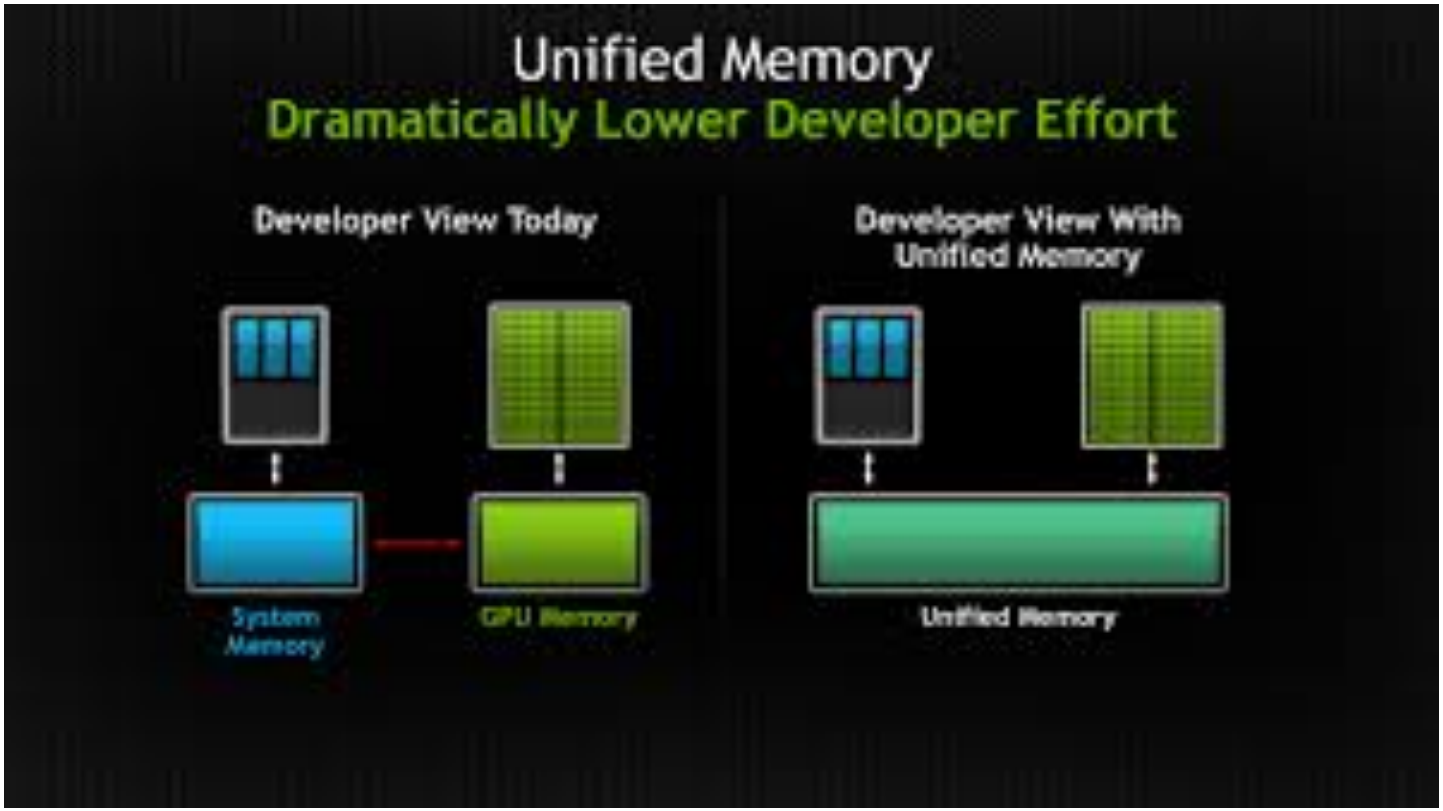
Kepler: Hyper-Q



Kepler GK110: GPU Direct



Unified memory



SUMMARY

Take home message

- GPUs are massively parallel architectures with limited flexibility, but very high throughput
- Pro's:
  - Much higher compute capabilities
  - Higher bandwidth
- Con's
  - Limited on-card memory
  - Low-bandwidth communication with host
- Debate-able
  - Programmability & productivity

Open research questions

- Shall we port all applications on GPUs?
  - If yes – can we automate the process?
  - If not – can we decide how to select?
- Shall we use GPUs in large-scale systems?
- Shall we use heterogeneous CPU+GPU systems?
- Can we improve the GPU design …
  - For HPC?
  - For other application domains?

Questions? Comments? Suggestions?

- A.L.Varbanescu@uva.nl

… also if you want to work on GPU-related projects OR in a team that works on heterogeneous computing.

… All you have to do is ask ☺