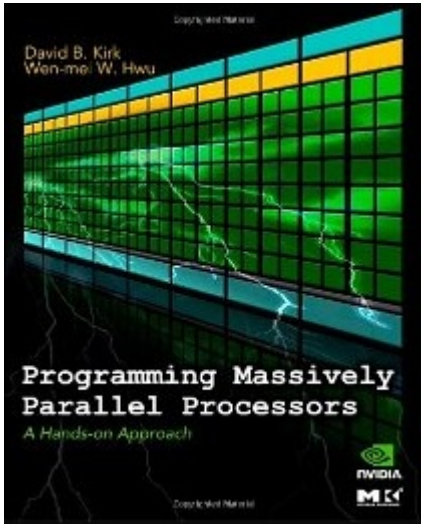
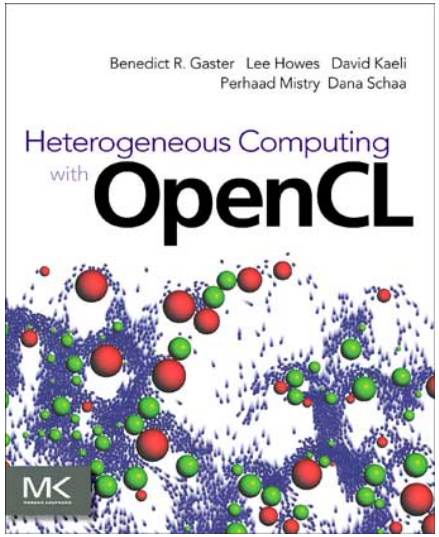
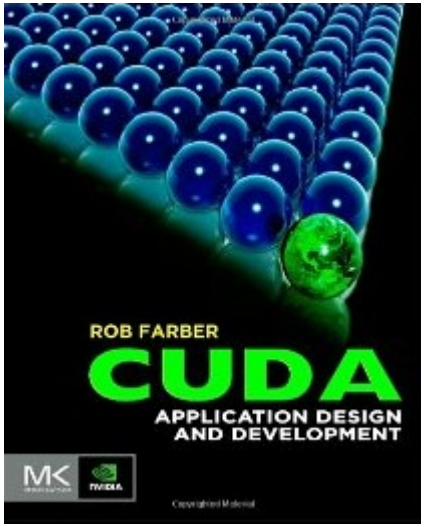
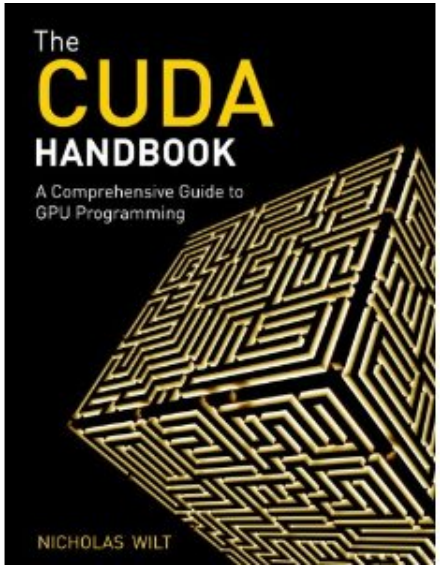
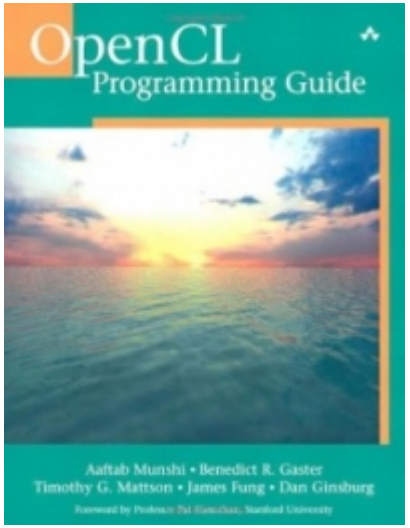
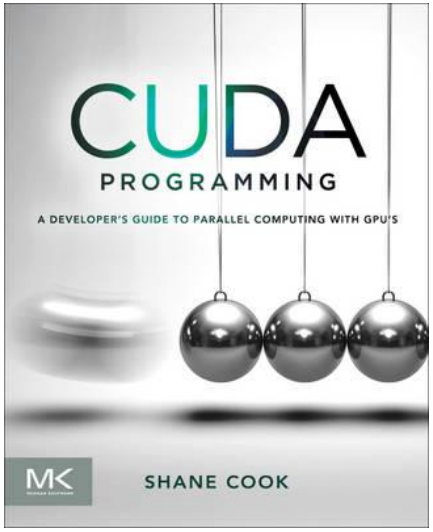












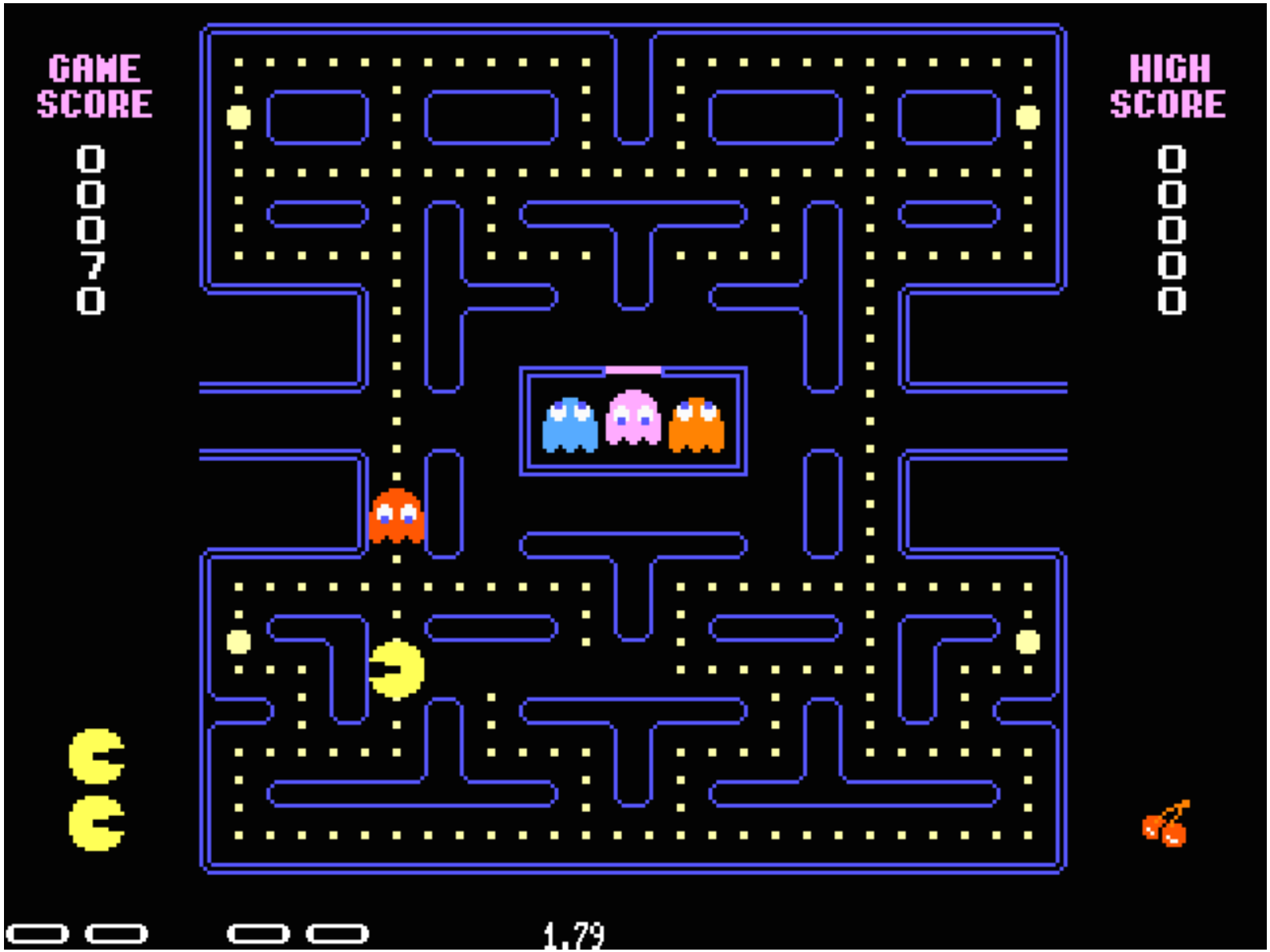
UNIVERSITY OF AMSTERDAM

GPU PROGRAMMING

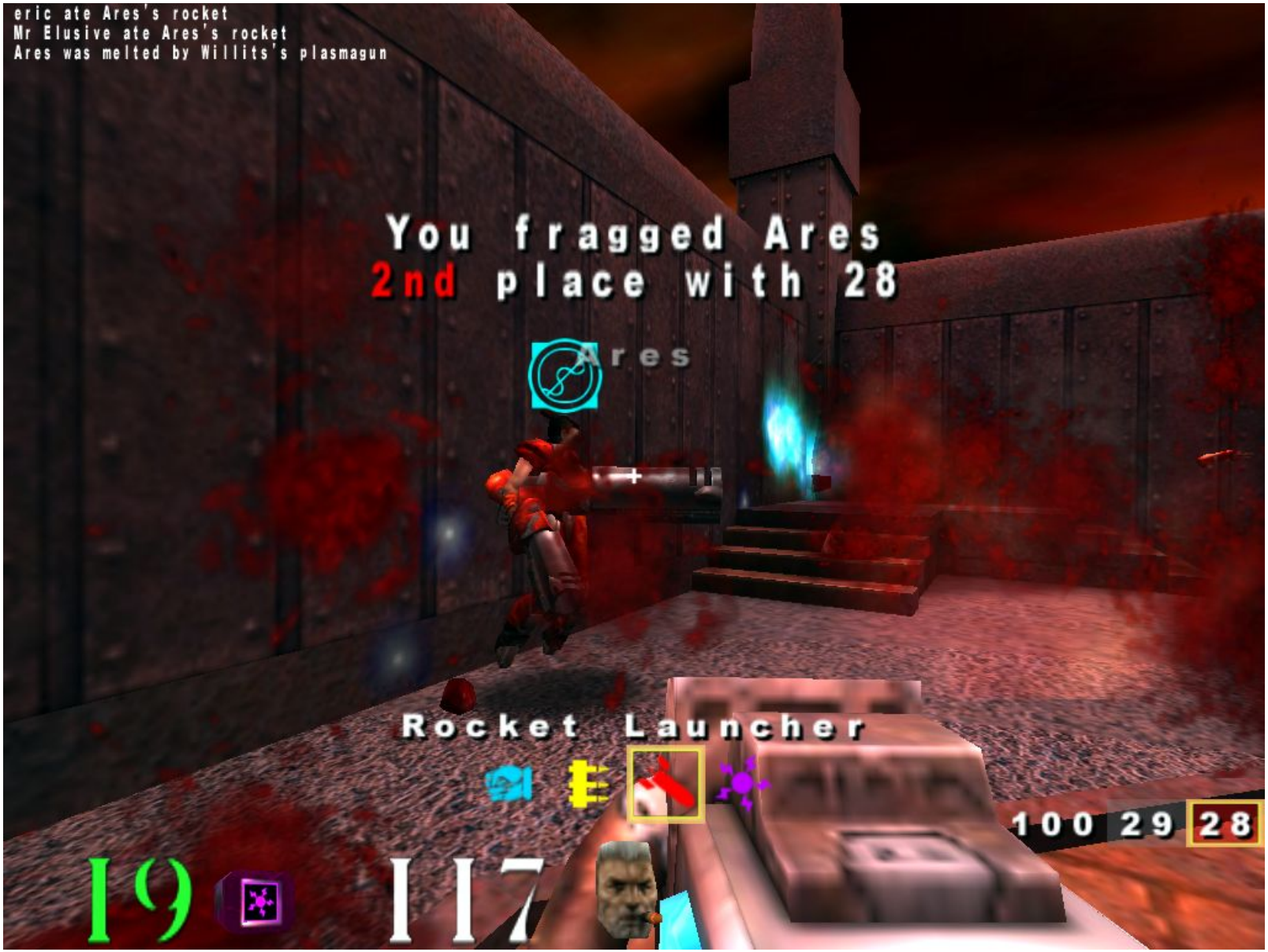
HPC courses
UvA, June 2014

Ana Lucia Varbanescu
A.L.Varbanescu@uva.nl

Graphics in 1980



Graphics in 2000



Realism of modern GPUs

ITV accused of using ArmA 2 game footage in IRA doc

Updated: WHOOPS

By Kate Solomon

September 27th | Tell us what you think [4 comments]

Like 91 Send Tweet 29 +1 2

ITV seems to have broadcast a documentary about Colonel Gaddafi's support for the IRA passing gameplay action taken from *ArmA 2* off as genuine documentary footage.

IRA footage? Er, no

Update: ITV has now sent us a comment on the situation. A spokesman said, "The events featured in *Exposure: Gaddafi and the IRA* were genuine but it would appear that during the editing process the correct clip of the 1988 incident was not selected and other footage was mistakenly included in the film by producers. This was an unfortunate case of human error for which we apologise."

http://www.youtube.com/watch?v=bJDeipvpjGQ&feature=player_embedded#t=49s

Courtesy techradar.com

TODO List

1. Multi and many-core hardware
2. GPGPUs
3. CUDA
4. Advanced CUDA
5. (some) OpenCL

-1

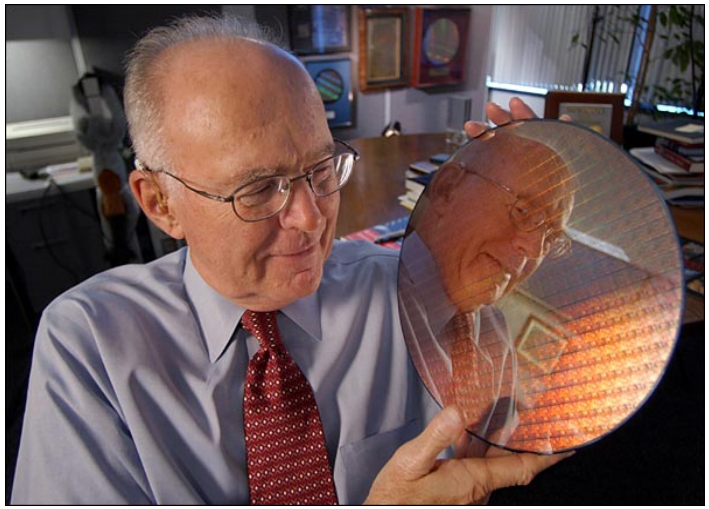
Why many-cores?

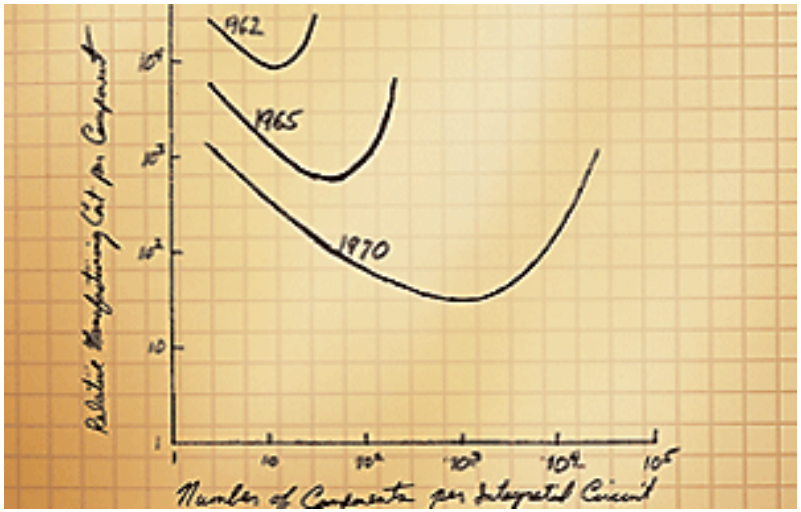
Multi-cores = Intel processors with multiple, homogeneous cores

Many-cores = GPUs & alikes

Moore's Law

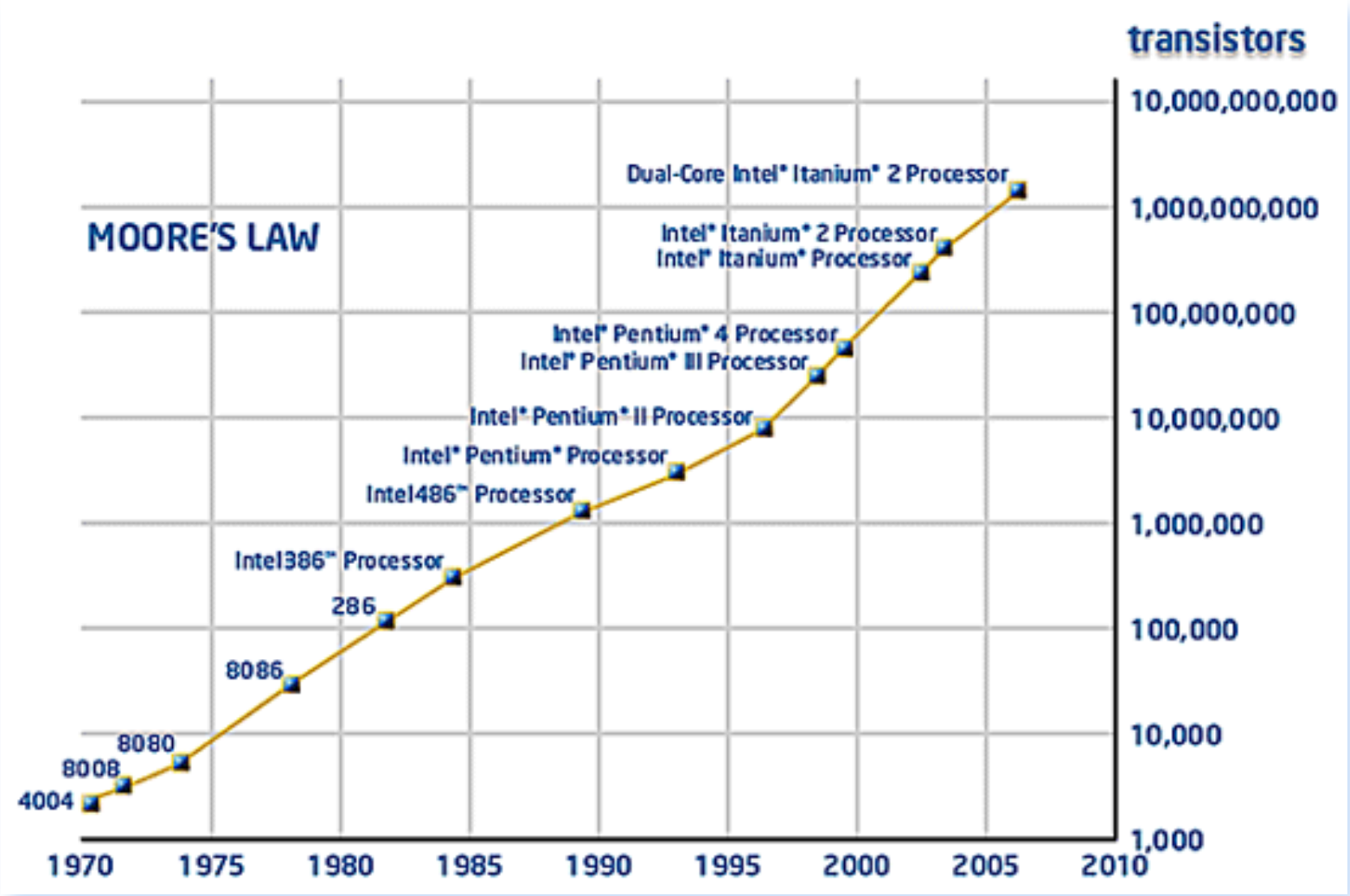
- Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.



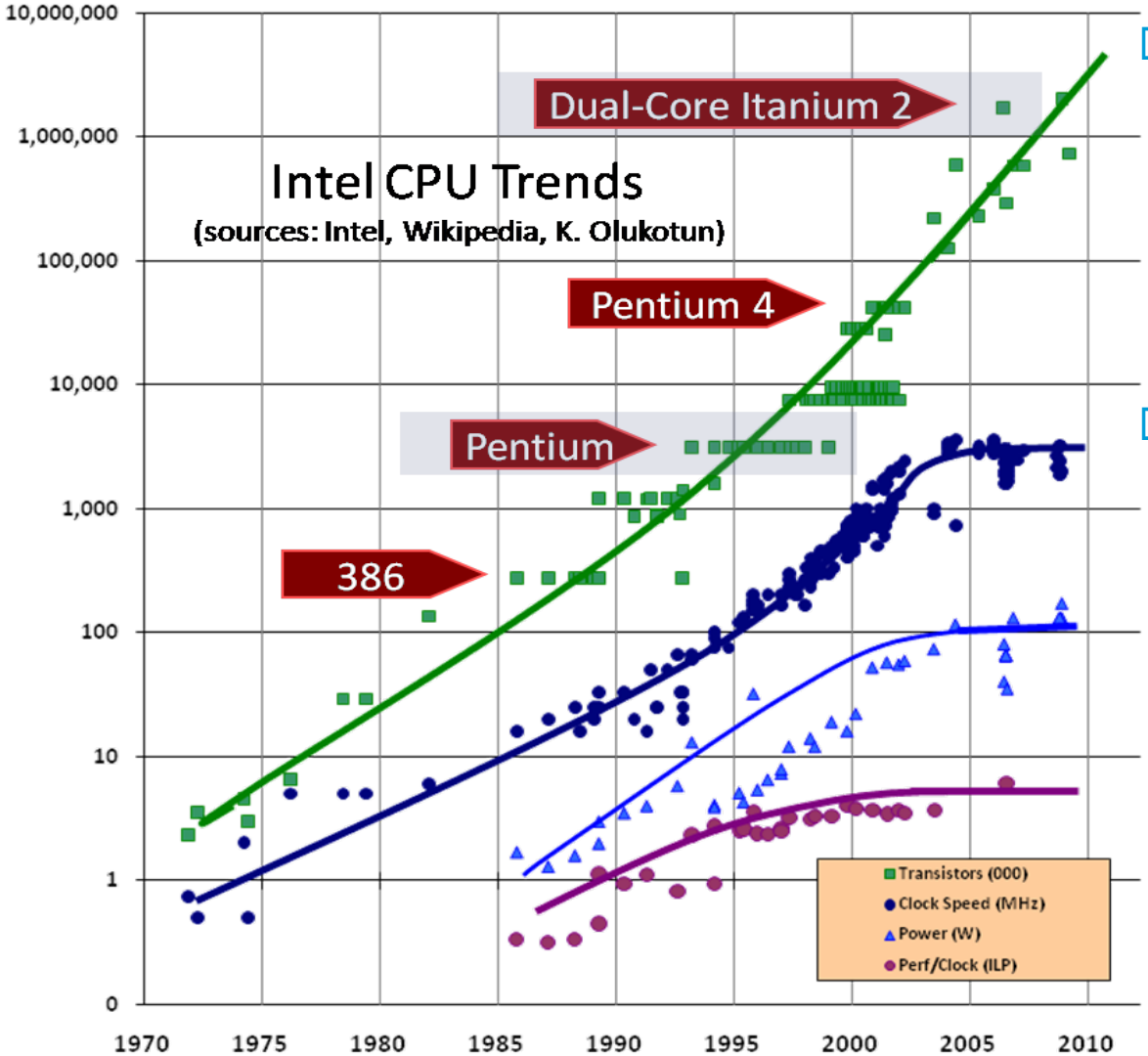


"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase...." Electronics Magazine 1965

Transistor Counts (Intel)



Revolution in Processors



- Chip density is continuing to increase about 2x every 2 years
- BUT
  - Clock speed is not
  - ILP is not
  - Power is not

New ways to use transistors

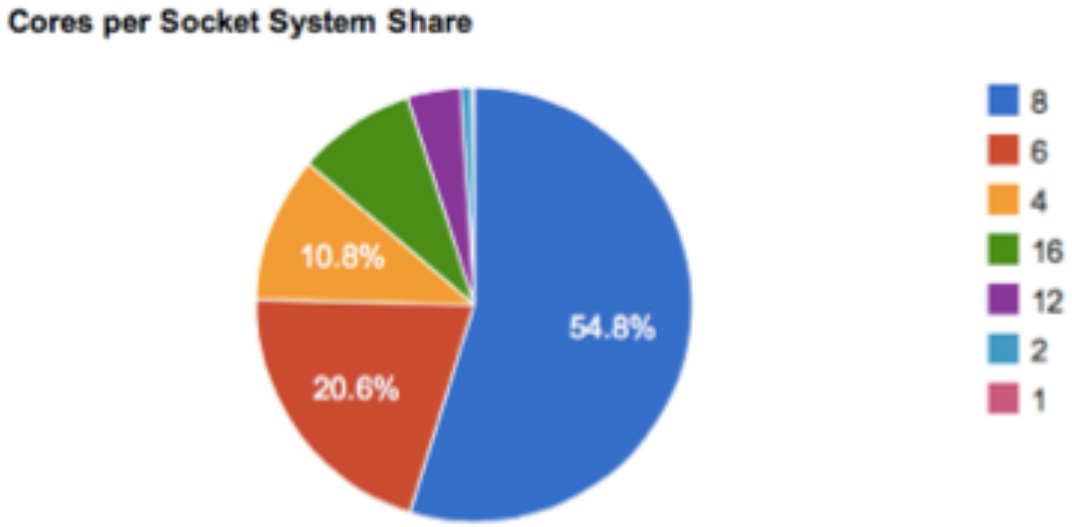
- Parallelism on-chip: multi-core processors
- "Multicore revolution"
  - Every machine will soon be a parallel machine
  - What about performance?
- Can applications use this parallelism?
  - YES, many have to be rewritten from scratch!
- Will all programmers have to be parallel programmers?
  - YES, implicit or explicit!

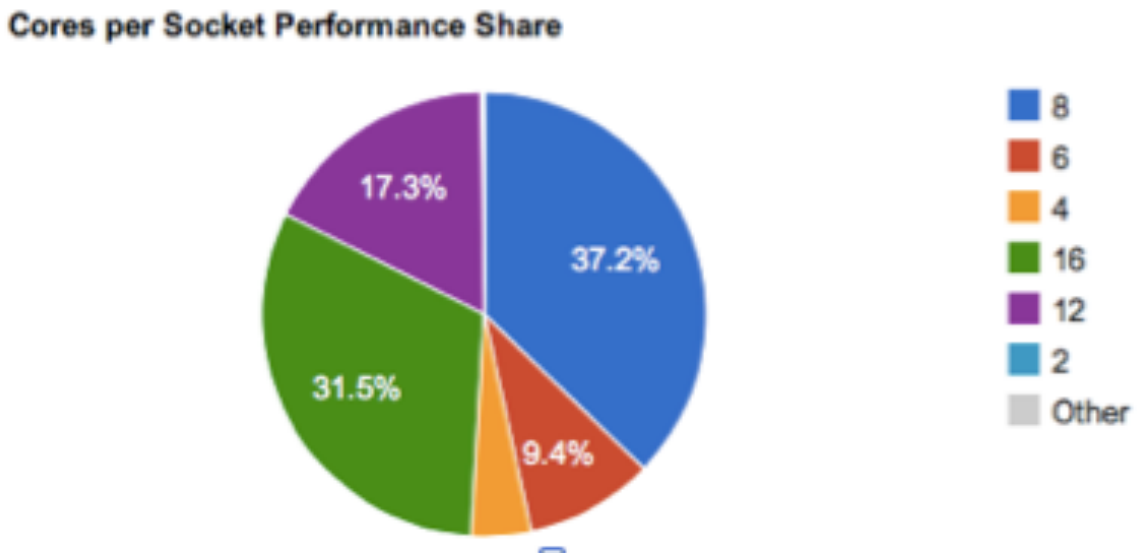
Top500 [1/4]

- State of the art in HPC (top500.org)
  - Trial for all new HPC architectures

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National University of Defense Technology China **195 cores/node!**	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3120000	33862.7	54902.4	17808
			Accelerated!			
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560640	17590.0	27112.5	8209
			Accelerated!			
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1572864	17173.2	20132.7	7890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705024	10510.0	11280.4	12660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786432	8586.6	10066.3	3945

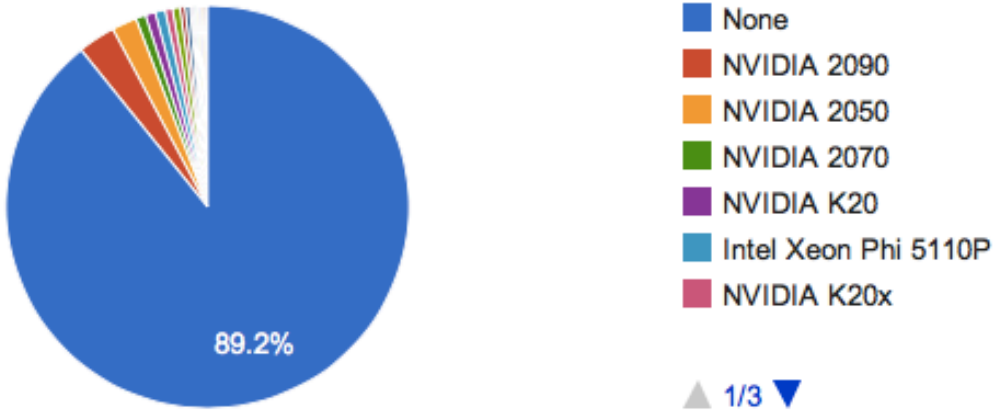
Top500: cores per socket



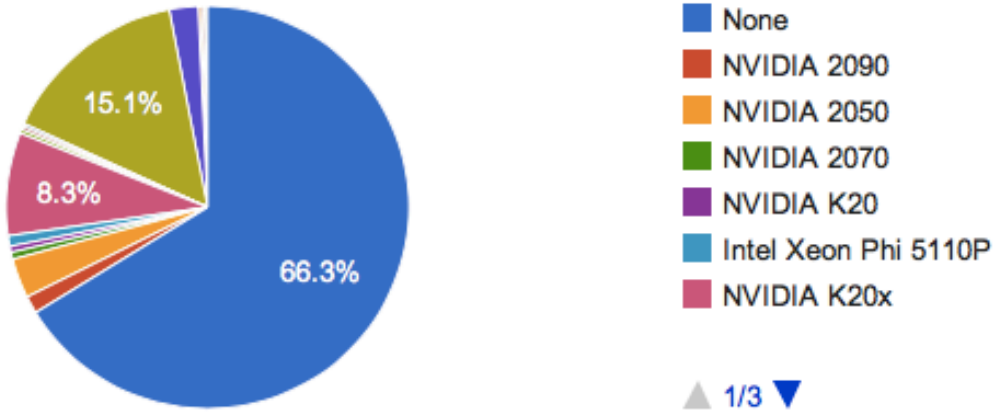


Top500: Accelerators

Accelerator/Co-Processor System Share



Accelerator/Co-Processor Performance Share



China's Tianhe-1A

#10 in top500 list – June 2013 (#1 in Top500 in November 2010)

4.701 pflops peak

2.566 pflops max

14,336 Xeon X5670 processors

7168 Nvidia Tesla M2050 GPUs x 448 cores = 3,211,264 cores

China's Tianhe-2

#1 in Top500 – June 2013

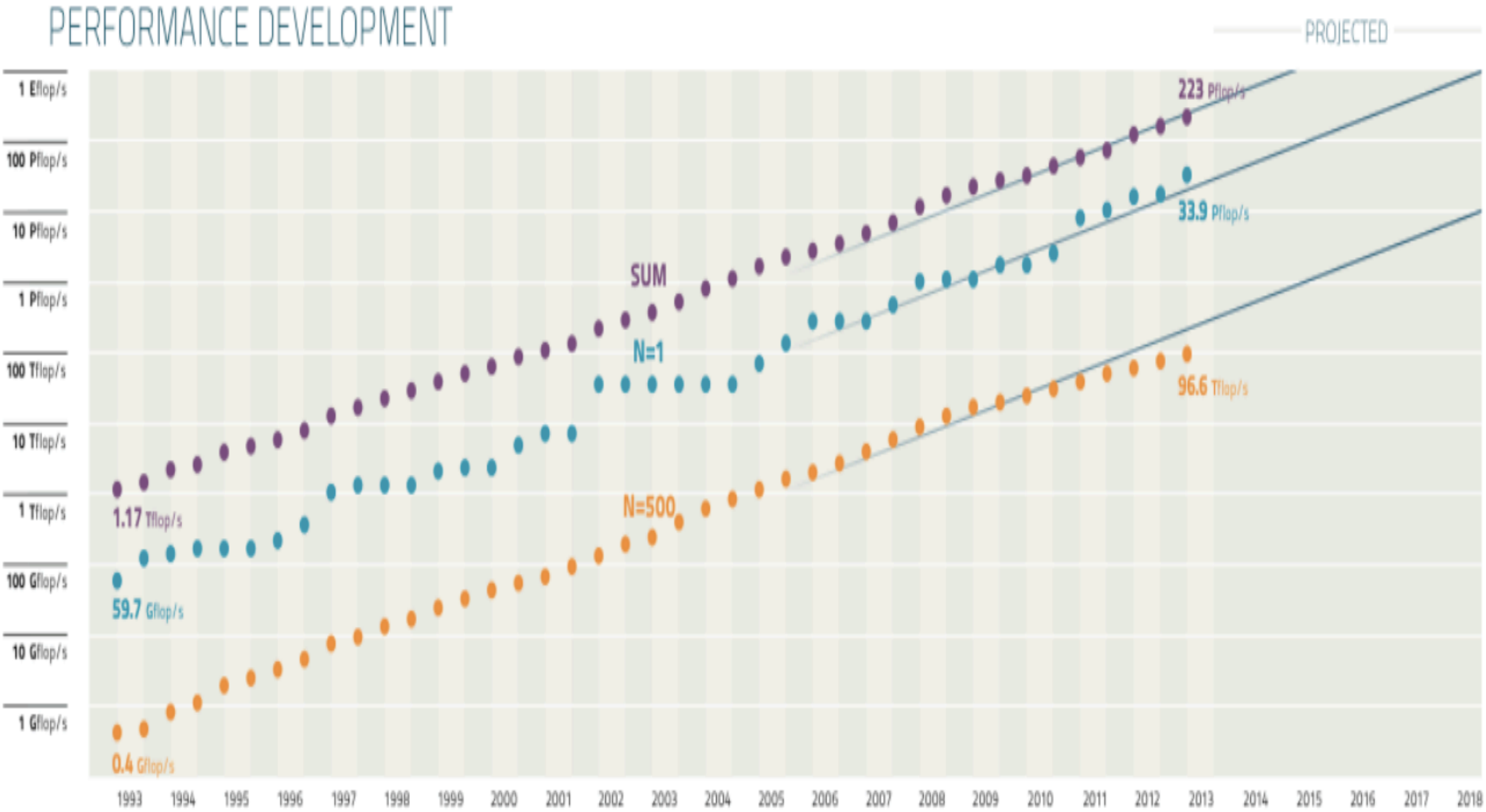
54.902 pflops peak

33.862 pflops max

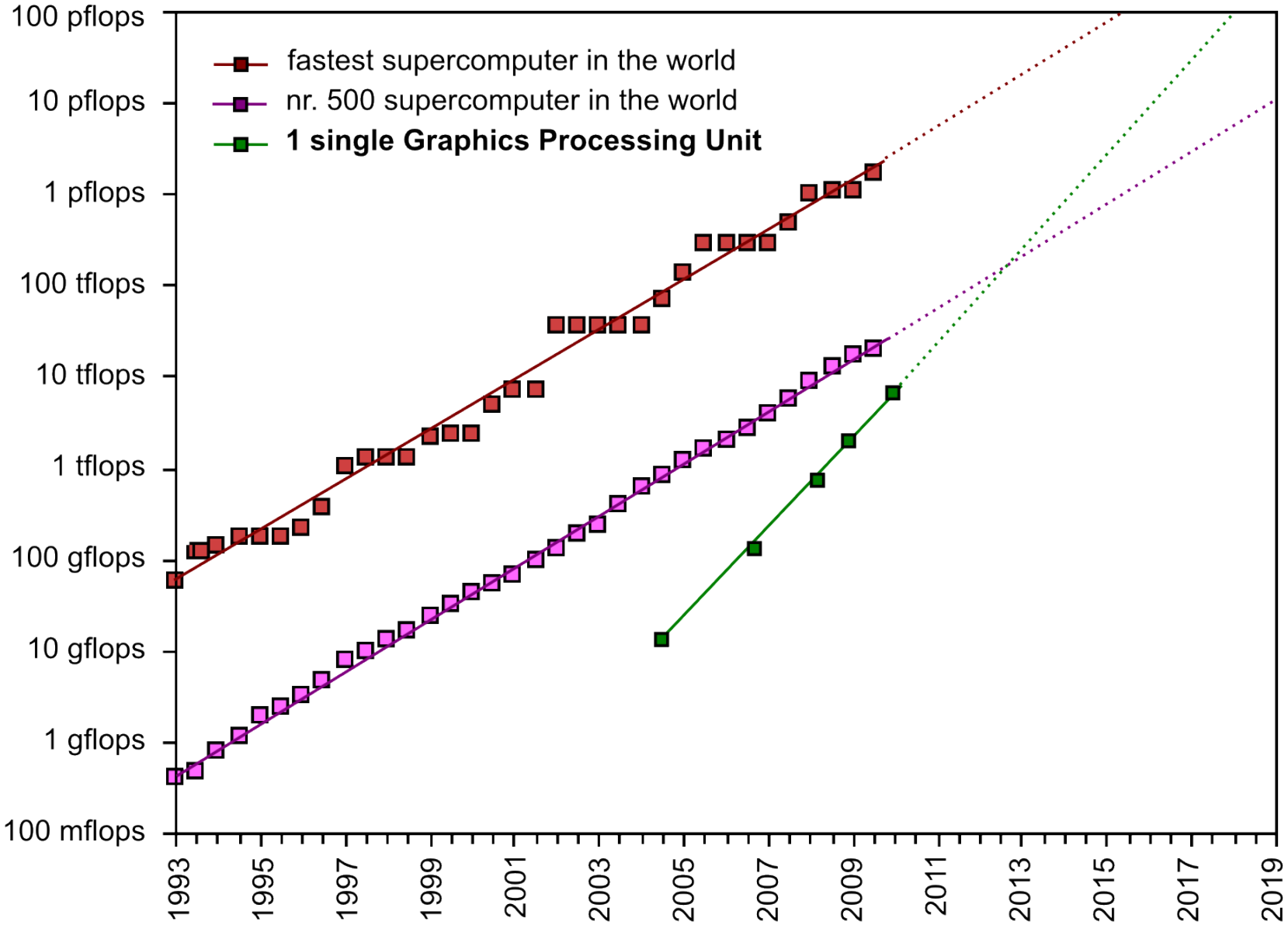
16.000 nodes = 16.000 x (2 x Xeon IvyBridge + 3 x Xeon Phi)

= 3.120.000 cores (=> 195 cores/node)

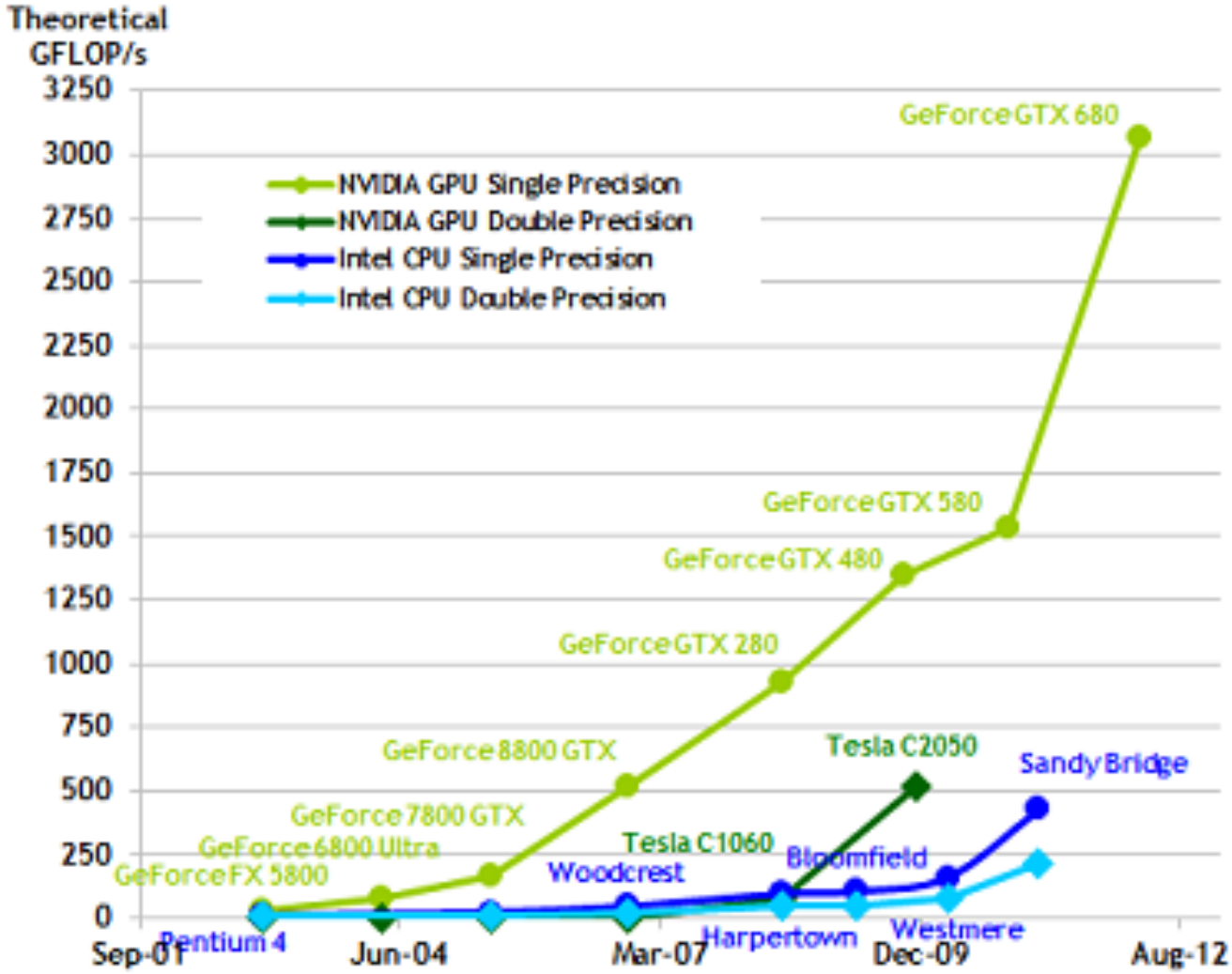
Top500: prediction



GPUs vs. Top500

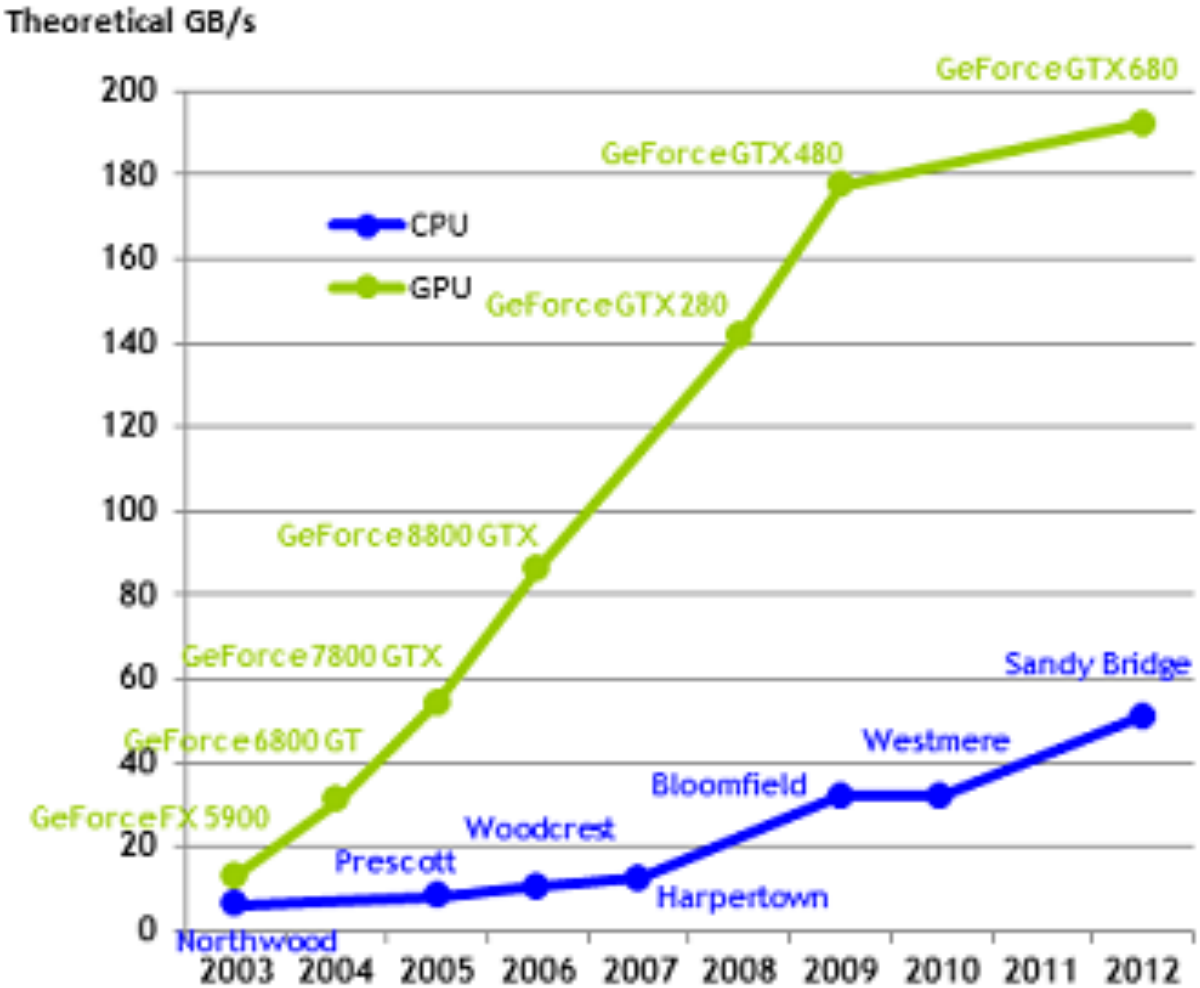


GPUs vs. CPUs



Floating-Point Operations per Second - NVIDIA CUDA C Programming Guide Version 4.2 - 4/5/2012 - copyright NVIDIA Corporation 2012

GPUs vs CPUs



Memory Bandwidth for the CPU and GPU - NVIDIA CUDA C Programming Guide - Version 4.2 - 4/5/2012 - copyright NVIDIA Corporation 2012

Why do we need many-cores?

- Performance
  - Large scale parallelism
- Power Efficiency
  - Use transistors more efficiently
- Price (GPUs)
  - Game market is huge, bigger than Hollywood
  - Mass production, economy of scale
  - “spotty teenagers” pay for our HPC needs!
- Prestige
  - Reach ExaFLOP by 2019

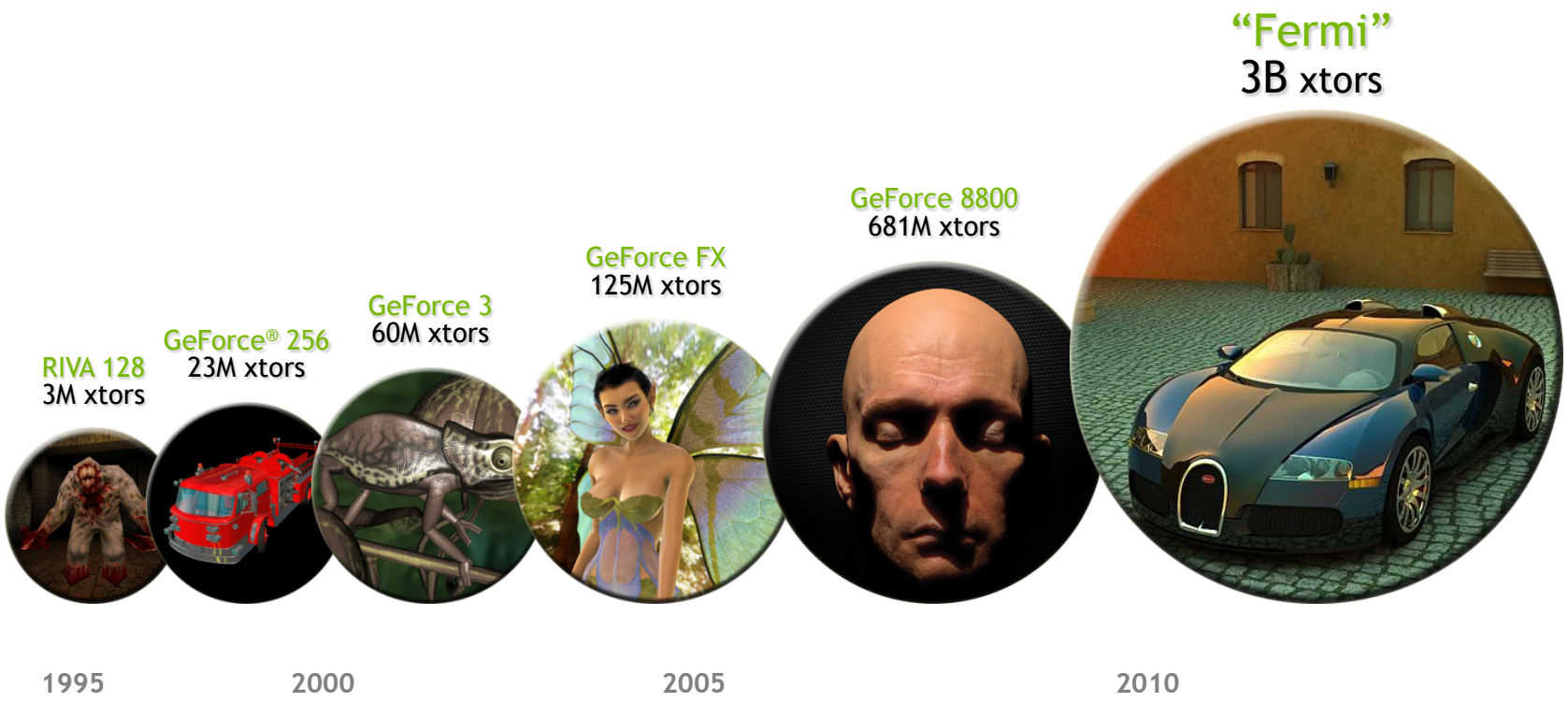
0

History

GPUs = the hardware

GPGPU = general purpose GPU

GPGPU History

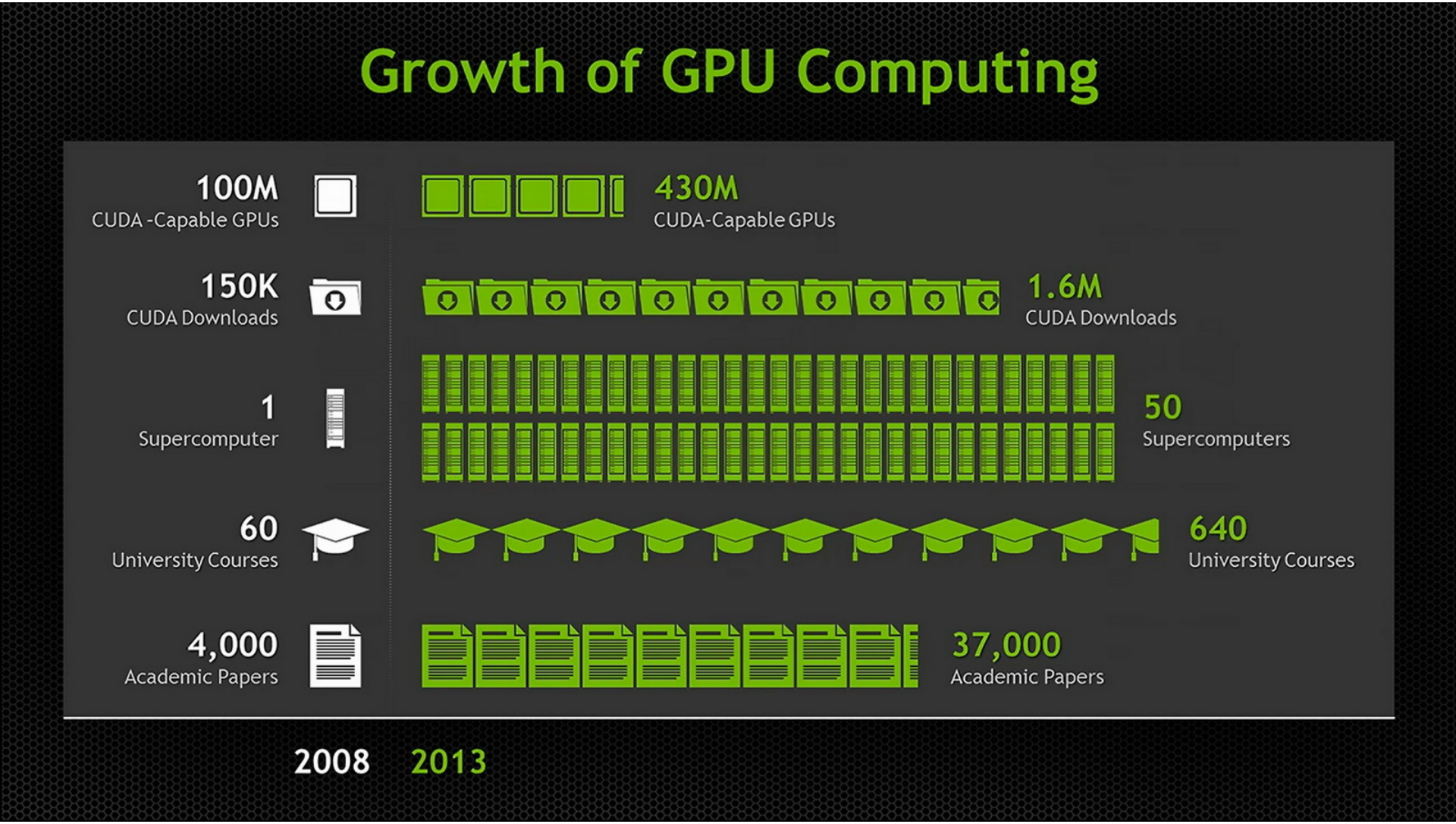


- Current generation: NVIDIA Kepler
  - 7.1B transistors
  - More cores, more parallelism, more performance

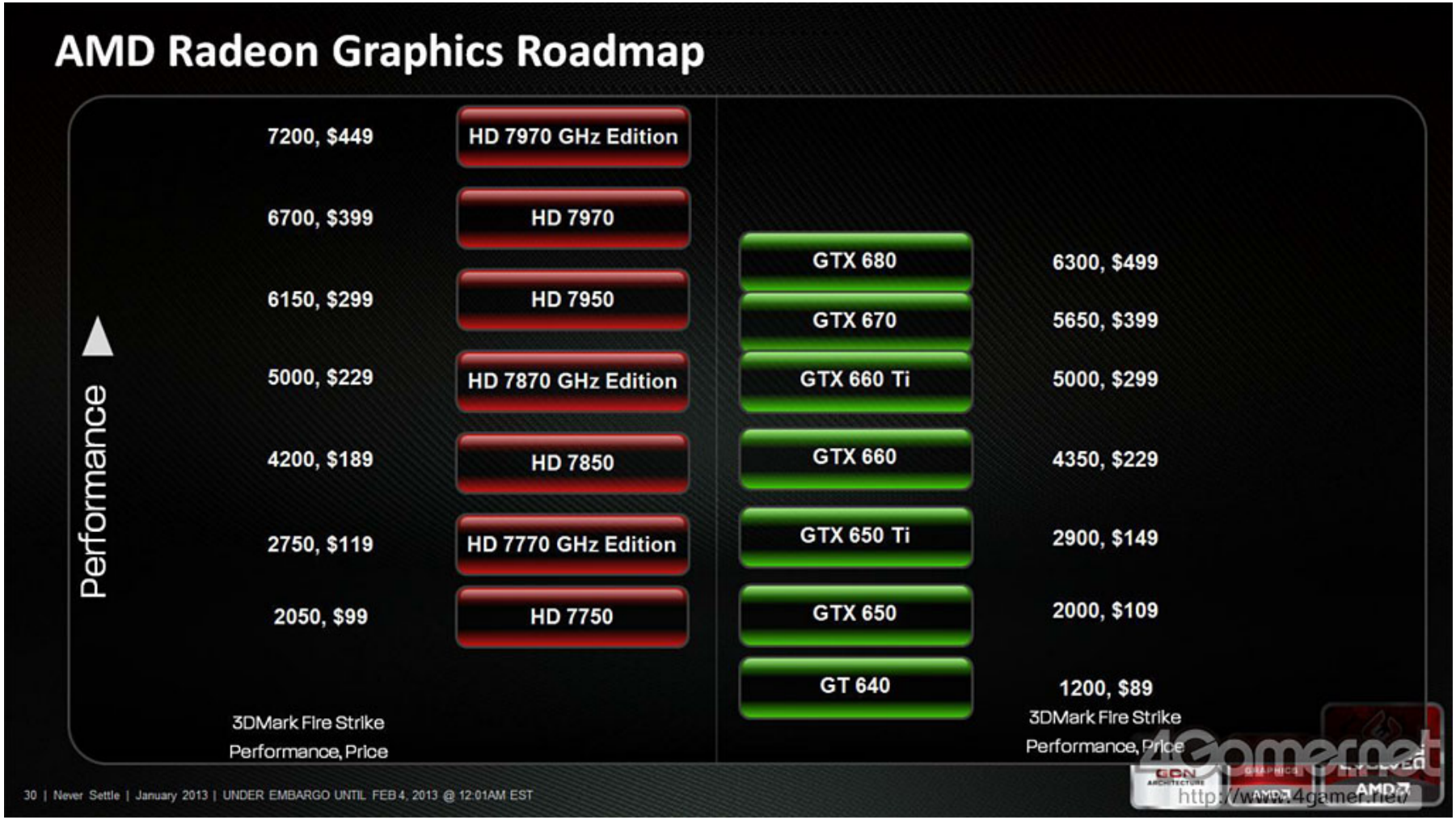
GPGPU History

- Use Graphics primitives for HPC
  - Ikonas [England 1978]
  - Pixel Machine [Potmesil & Hoffert 1989]
  - Pixel-Planes 5 [Rhoades, et al. 1992]
- Programmable shaders, around 1998
  - DirectX / OpenGL
  - Map application onto graphics domain!
- GPGPU
  - Brook (2004), Cuda (2007), OpenCL (Dec 2008), ...

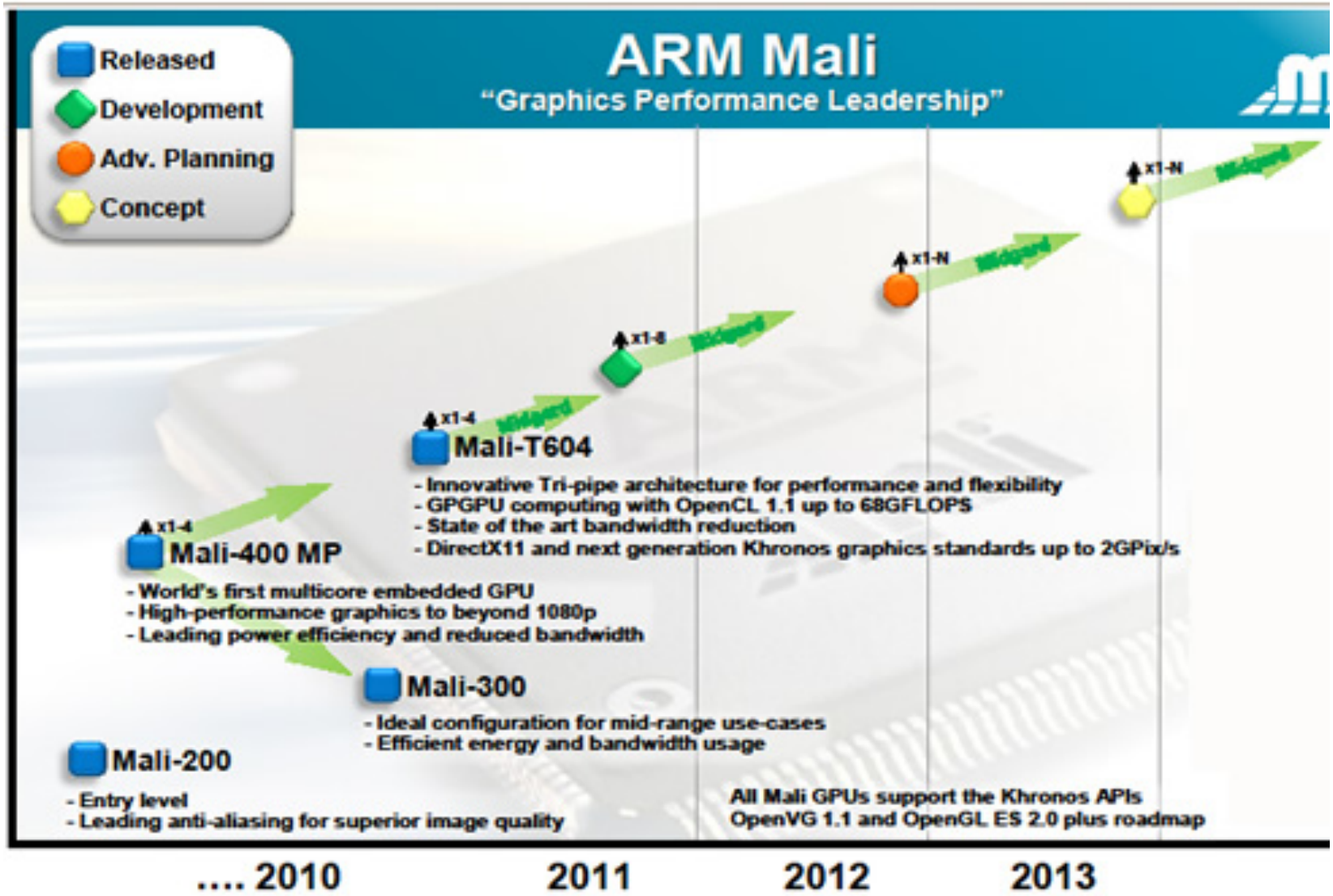
Another GPGPU history



GPUs @ AMD


AMD Radeon Graphics Roadmap
Performance
7200, $449
HD 7970 GHz Edition
6700, $399
HD 7970
6150, $299
HD 7950
5000, $229
HD 7870 GHz Edition
4200, $189
HD 7850
2750, $119
HD 7770 GHz Edition
2050, $99
HD 7750
3DMark Fire Strike
Performance, Price
GTX 680
6300, $499
GTX 670
5650, $399
GTX 660 Ti
5000, $299
GTX 660
4350, $229
GTX 650 Ti
2900, $149
GTX 650
2000, $109
GT 640
1200, $89
3DMark Fire Strike
Performance, Price
30 | Never Settle | January 2013 | UNDER EMBARGO UNTIL FEB 4, 2013 @ 12:01AM EST
http://www.4gamer.net/

GPU @ ARM

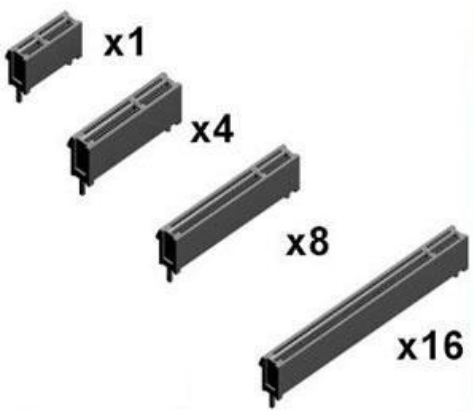

Released
Development
Adv. Planning
Concept
ARM Mali
"Graphics Performance Leadership"
x1-N
x1-N
x1-8
x1-4
Mali-T604
- Innovative Tri-pipe architecture for performance and flexibility
- GPGPU computing with OpenCL 1.1 up to 68GFLOPS
- State of the art bandwidth reduction
- DirectX11 and next generation Khronos graphics standards up to 2GPix/s
x1-4
Mali-400 MP
- World's first multicore embedded GPU
- High-performance graphics to beyond 1080p
- Leading power efficiency and reduced bandwidth
Mali-300
- Ideal configuration for mid-range use-cases
- Efficient energy and bandwidth usage
Mali-200
- Entry level
- Leading anti-aliasing for superior image quality
All Mali GPUs support the Khronos APIs
OpenVG 1.1 and OpenGL ES 2.0 plus roadmap
.... 2010
2011
2012
2013

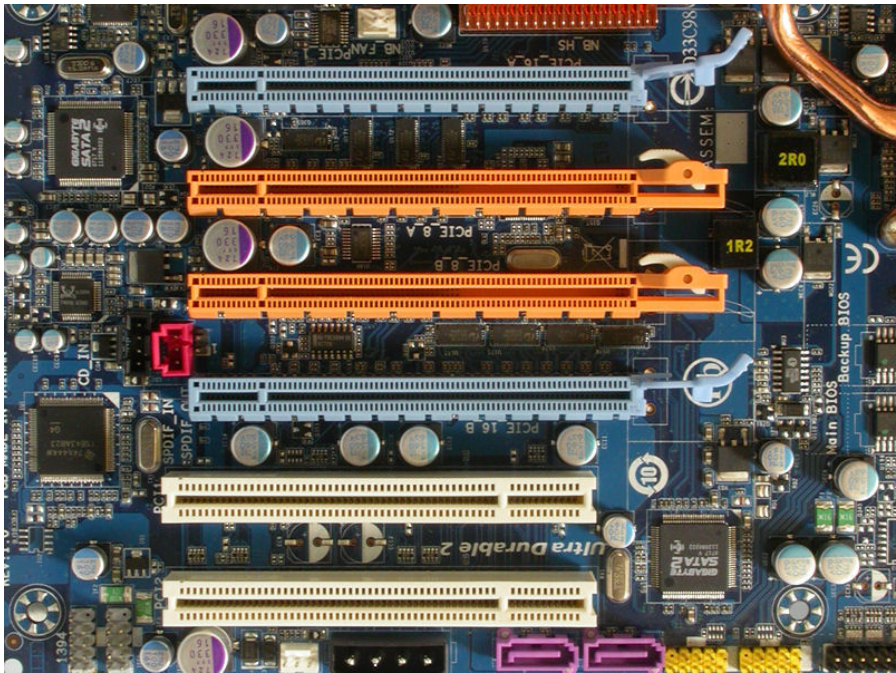
1

The hardware

Integration into host system

- Typically PCI Express 2.0 x16
- Theoretical speed 8 GB/s
  - protocol overhead → 6 GB/s
- In reality: 4 – 6 GB/s
- V3.0 recently available
  - Double bandwidth
  - Less protocol overhead





Lessons from the graphics pipeline

- Throughput is the main focus
  - must paint every pixel within frame time
  - scalability
- Create, run, and retire lots of threads very rapidly
  - measured 14.8 billion thread/s on increment() kernel
- Use multithreading to hide latency
  - 1 stalled thread is OK if 100 are ready to run

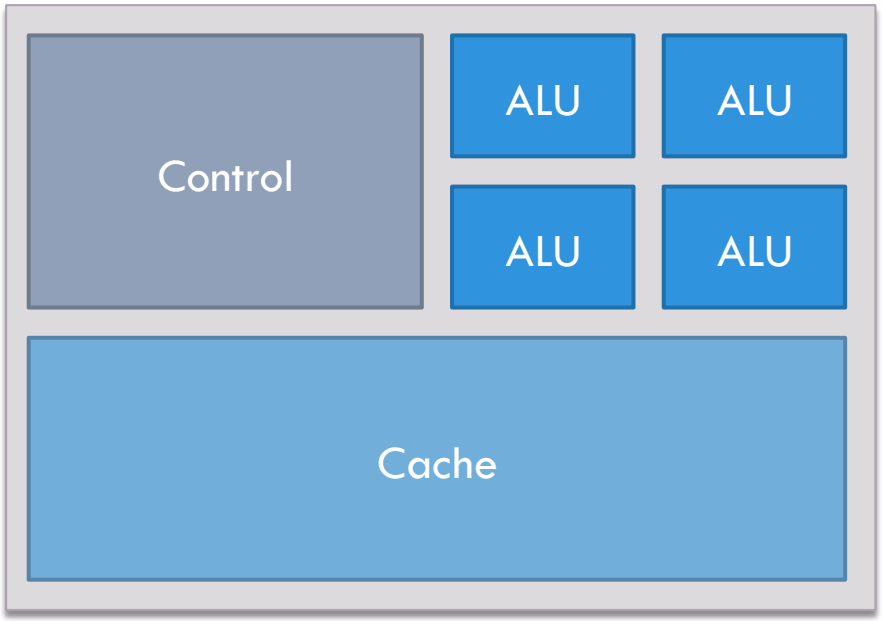
Key GPU architectural ideas

- Data parallel, like a vector machine
  - There, 1 thread issues parallel vector instructions
- **SIMT** (Single Instruction Multiple Thread) execution
  - Many threads work on a vector, each on a different element
  - They all execute the same instruction
  - HW automatically handles divergence
- Hardware multithreading
  - HW resource allocation & thread scheduling
  - HW relies on threads to hide latency
  - Context switching is (practically) free

CPU vs. GPU

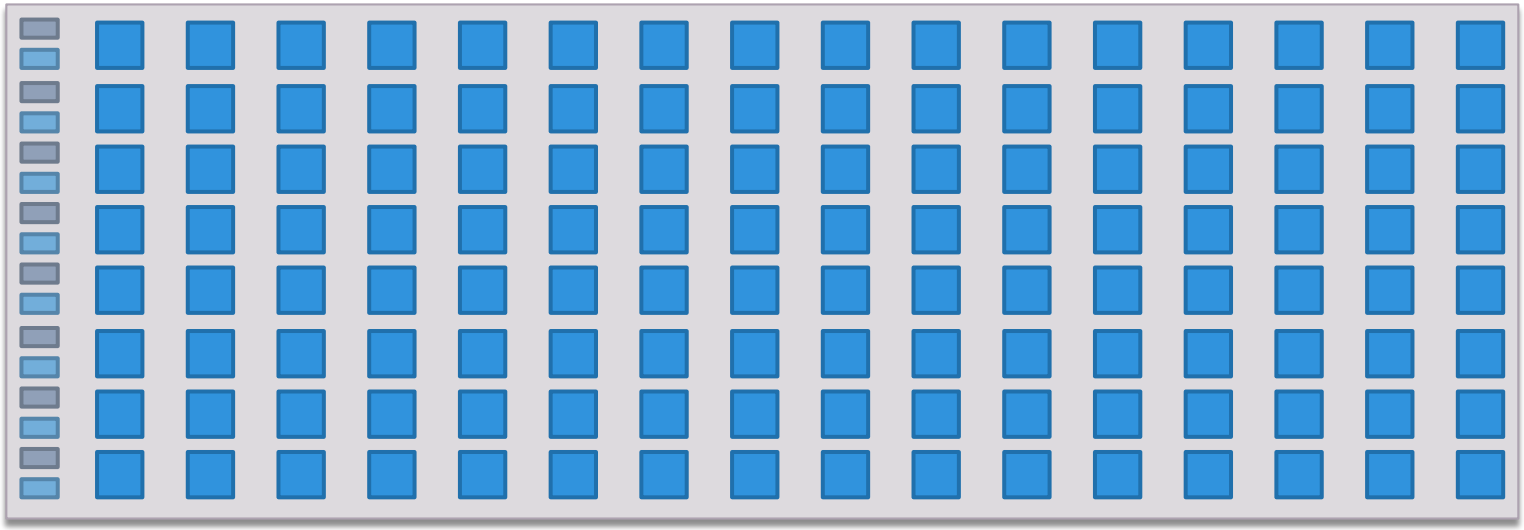
- Different goals produce different designs
  - GPU assumes work load is highly parallel
  - CPU must be good at everything, parallel or not
- CPU: minimize latency experienced by 1 thread
  - big on-chip caches
  - sophisticated control logic
- GPU: maximize throughput of all threads
  - # threads in flight limited by resources => lots of resources (registers, etc.)
  - multithreading can hide latency => no big caches
  - share control logic across many threads

Chip area CPU vs GPU

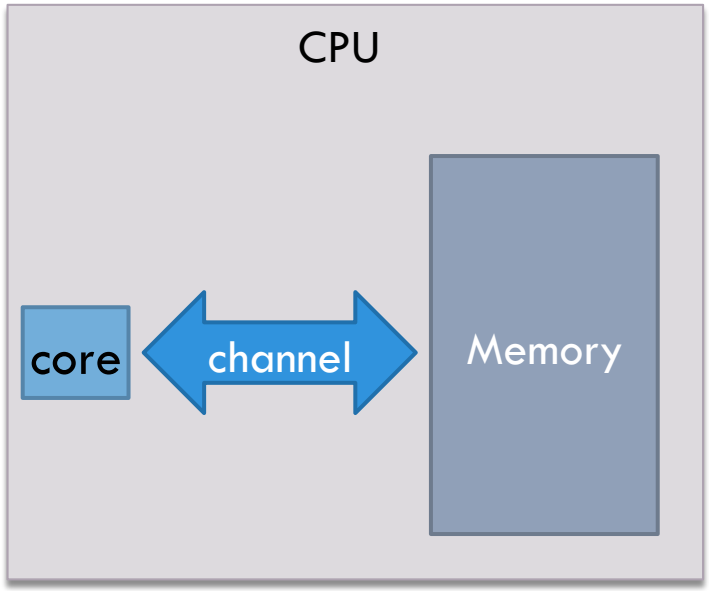


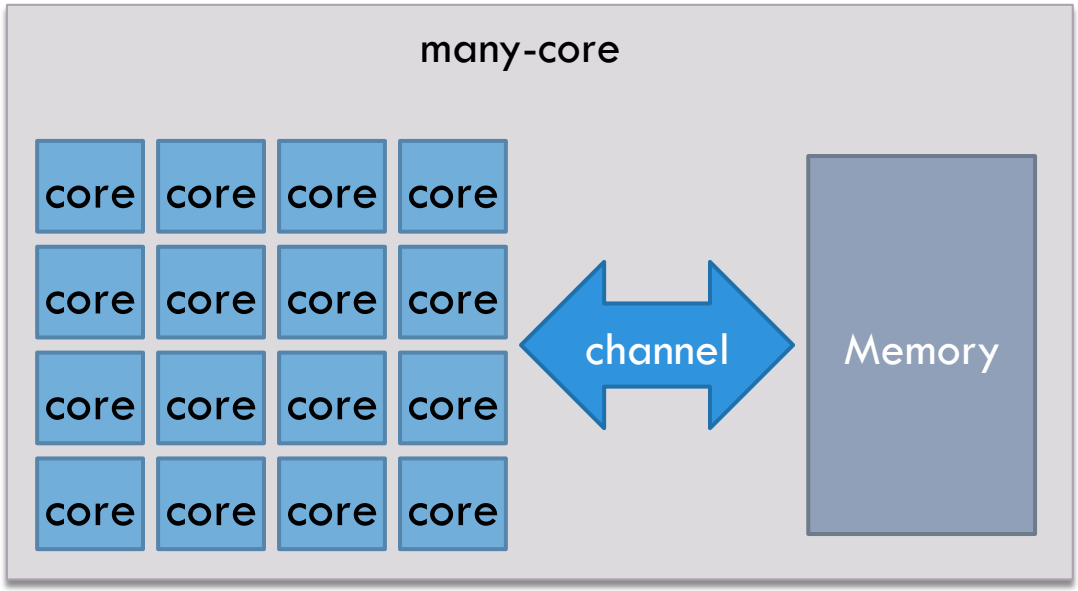
CPU

GPU



It's all about the memory





CPU vs GPU

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

ATI GPUs

Latest generation ATI

- Southern Islands
- 1 chip: HD 7970
  - 2048 cores
  - 264 GB/sec memory bandwidth
  - 3.8 Tflops single, 947 Gflops double precision
  - Maximum power: 250 Watts
  - 399 euros!
- 2 chips: HD 7990
  - 4096 cores, 7.6 Tflops
- Note: the entire 36-node DAS-4 TUD cluster has 2.2 Tflops

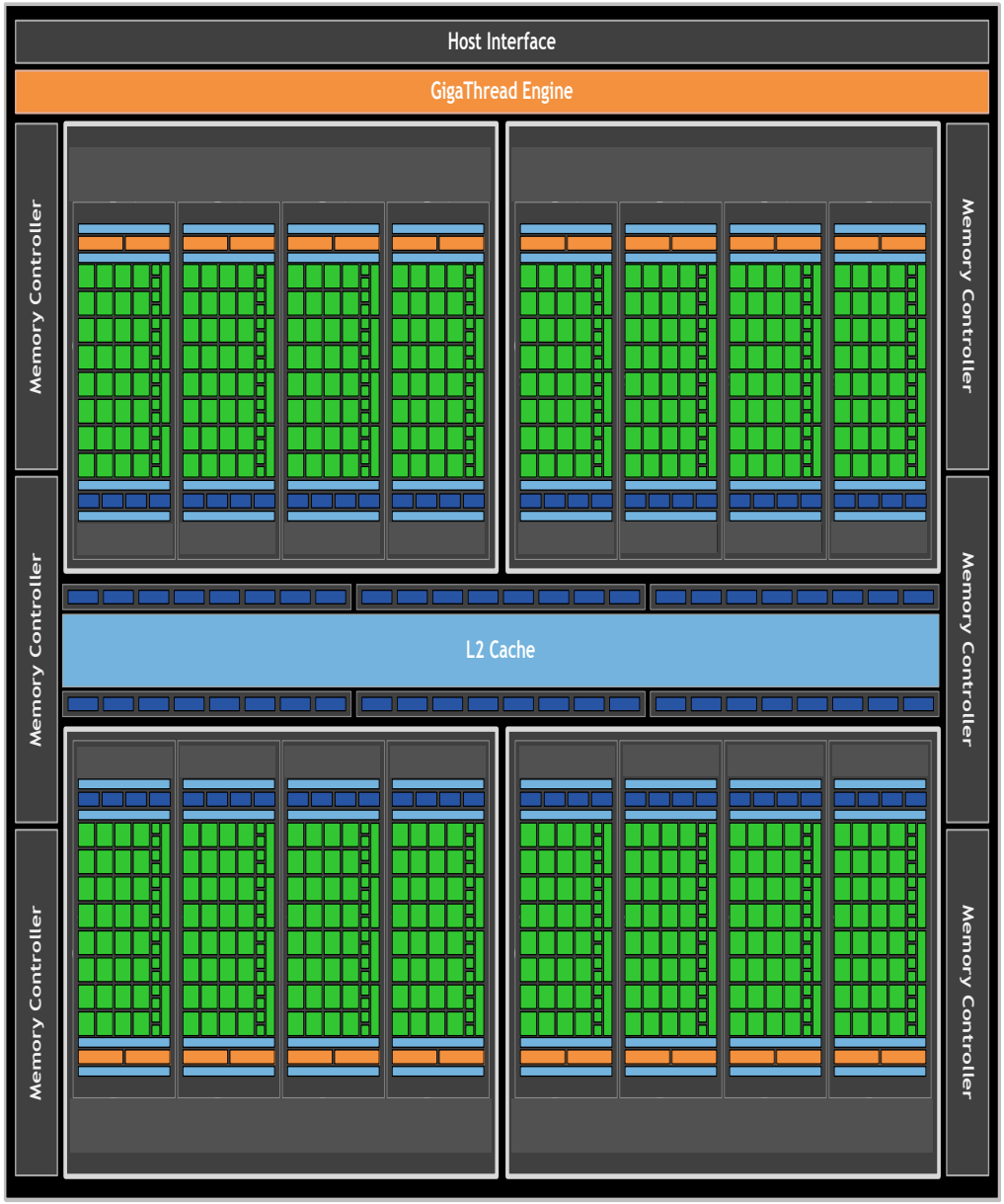
ATI programming models

- Low-level: CAL (assembly)
- High-level: Brook+
  - Originally developed at Stanford University
  - Streaming language
  - Performance is not great
- Now
  - OpenCL
- Near future
  - HSA – Heterogeneous System Architecture
  - HSAIL – HSA Intermediate Language
  - Targeted at Fusion devices, single source code

GPU Hardware: NVIDIA

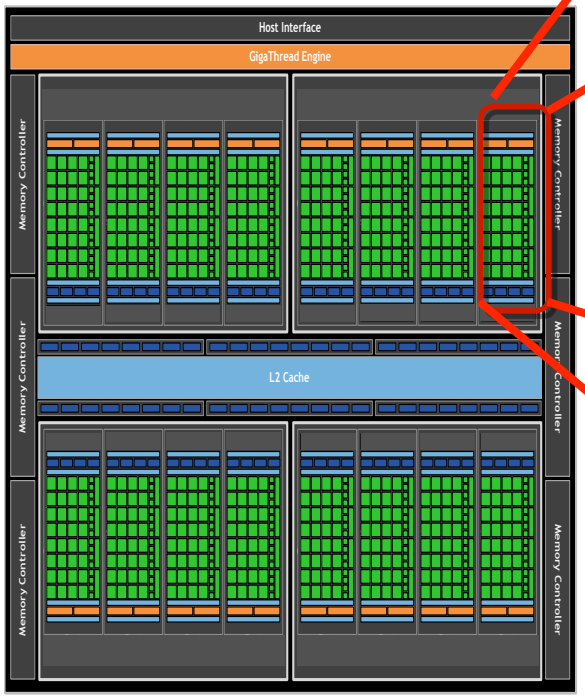
Fermi

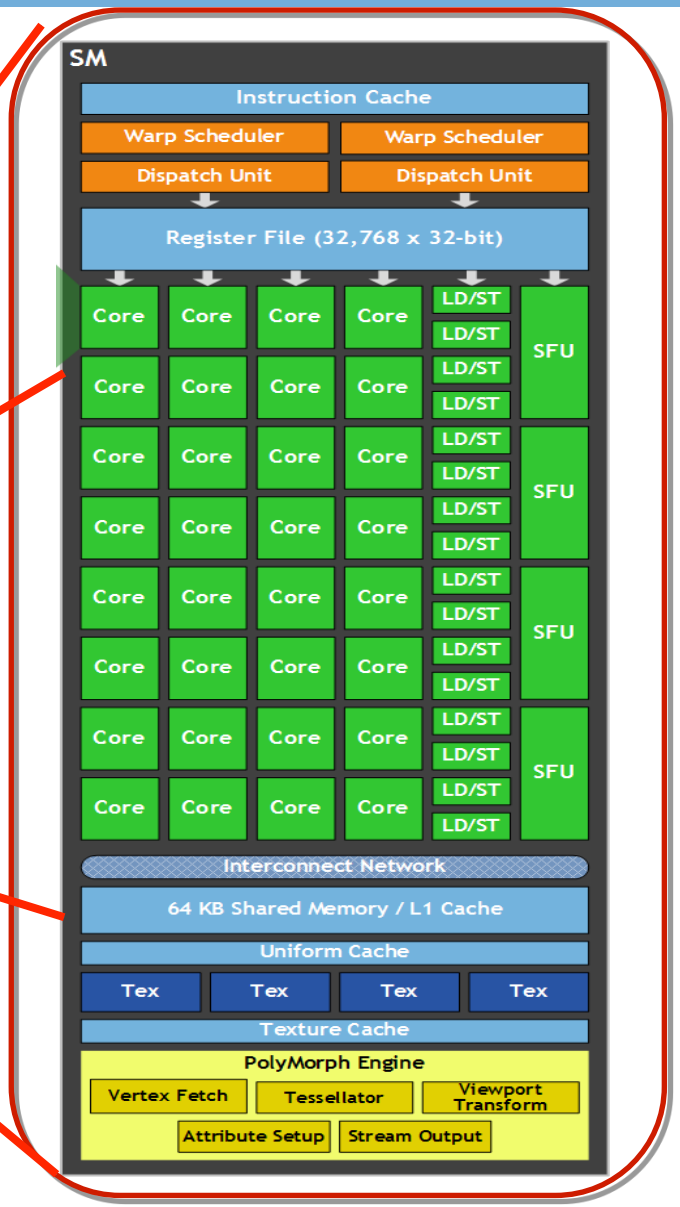
- Consumer: GTX 480, 580
- HPC: Tesla C2050
  - More memory, ECC
  - 1.0 Tlop SP
  - 515 GFlop SP
- 16 streaming multiprocessors (SM)
  - GTX 580: 16
  - GTX 480: 15
  - C2050: 14
- SMs are independent
- 768 KB L2 cache



Fermi Streaming Multiprocessor (SM)

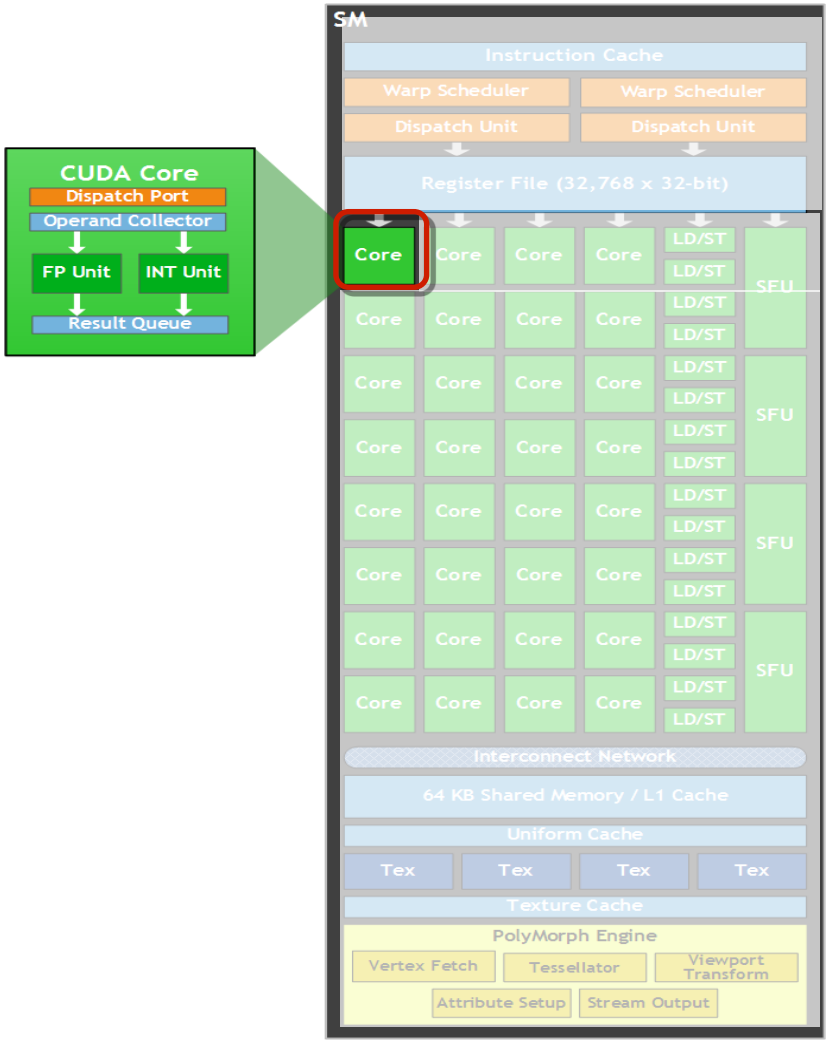
- 32 cores per SM (512 cores total)
- 64KB configurable L1 cache / shared memory
- 32,768 32-bit registers





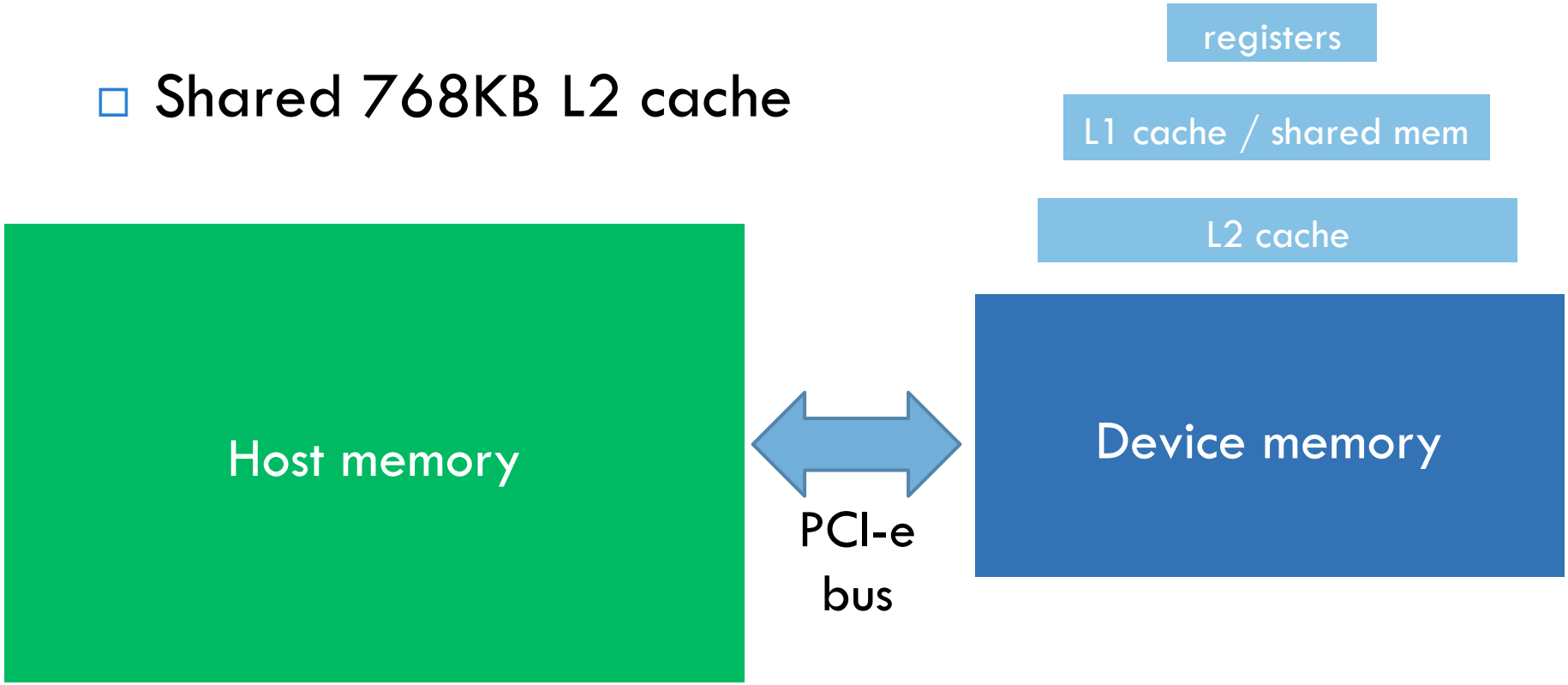
CUDA Core Architecture

- Decoupled floating point and integer data paths
- Double precision throughput is 50% of single precision
- Integer operations optimized for extended precision
  - 64 bit and wider data element size
- Predication field for all instructions
- Fused-multiply-add



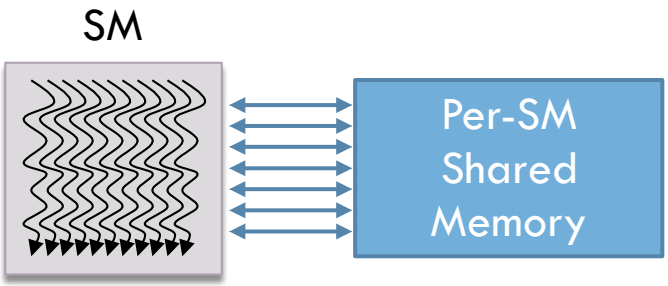
Memory Hierarchy

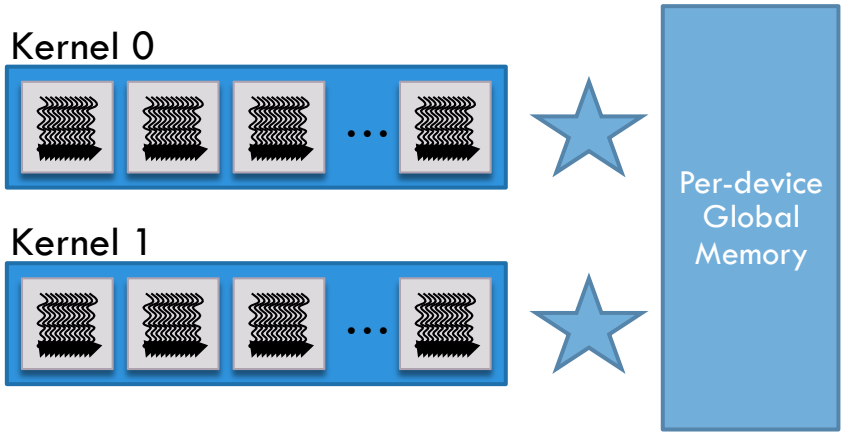
- Configurable L1 cache per SM
  - 16KB L1 cache / 48KB Shared
  - 48KB L1 cache / 16KB Shared
- Shared 768KB L2 cache



Multiple Memory Scopes

- Per-thread private memory
  - Each thread has its own local memory
  - Stacks, other private data, **registers**
- Per-SM shared memory
  - Small memory close to the processor, low latency
- Device memory
  - GPU frame buffer
  - Can be accessed by any thread in any SM





Atomic Operations

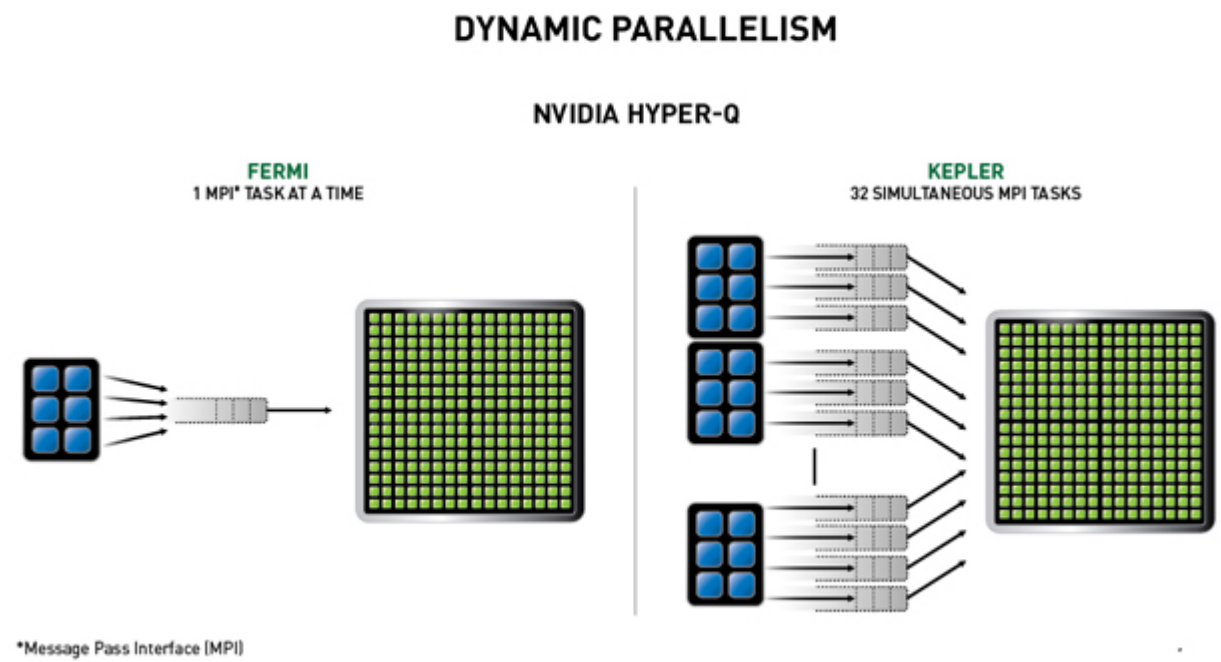
- Device memory is not coherent!
- Share data between streaming multiprocessors
- Read / Modify / Write
- Fermi increases atomic performance by 5x to 20x
  - Still, much slower than non-atomic access

ECC (Error-Correcting Code)

- All major internal memories are ECC protected
  - Register file, L1 cache, L2 cache
- DRAM protected by ECC (on Tesla only)
- ECC is a must have for many computing applications

NVIDIA Kepler

- New core - SMX (successor of SM)
  - 192 single precision FMAs per cycle
    - 4x compared to Fermi's 48
  - GTX 680 has 8 cores
- Dynamic parallelism
- HyperQ



Getting technical

	Fermi	GF104	Kepler	GCN	Units
Threads	48	48	64	40	
Work-items	1536	1536	2048	2560	
SP FLOP/cycle	64	96	384	128	
Register File	128	128	256	256	KB
Shared Memory	64	64	64	64	KB
L1D				16	KB
Shared Memory BW	64	64	128*	128	B/cycle
L1D BW				64	B/cycle
Register File/Work-item	85.33	85.33	128	102.4	B
Shared Memory/Work-item	42.67	42.67	32	50	B
L1/Work-item				25	B
Shared Memory BW/FLOP	1	0.67	0.33*	1	B/FLOP
L1D BW/FLOP				0.5	B/FLOP

Table 1. GPU Core Computational and Memory Resources

2

CUDA: Programming NVIDIA GPUs

CUDA

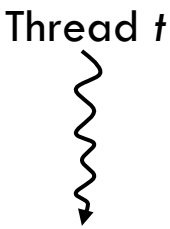
- CUDA: Scalable parallel programming
  - C/C++ extensions
  - Higher level extensions, too
- Provide straightforward mapping onto hardware
  - Good fit to GPU architecture
  - Maps well to multi-core CPUs too
- Scale to 1000s of cores & 100,000s of threads
  - GPU threads are lightweight — create / switch is free
  - GPU needs 1000s of threads for full utilization

Parallel Abstractions in CUDA

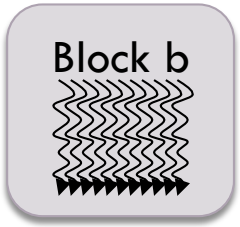
- Hierarchy of concurrent threads
  - Concurrent thread blocks
- Lightweight synchronization primitives
- Shared memory model for cooperating threads

Hierarchy of concurrent threads

- Parallel kernels composed of many threads
  - All threads execute the same **kernel** = sequential program

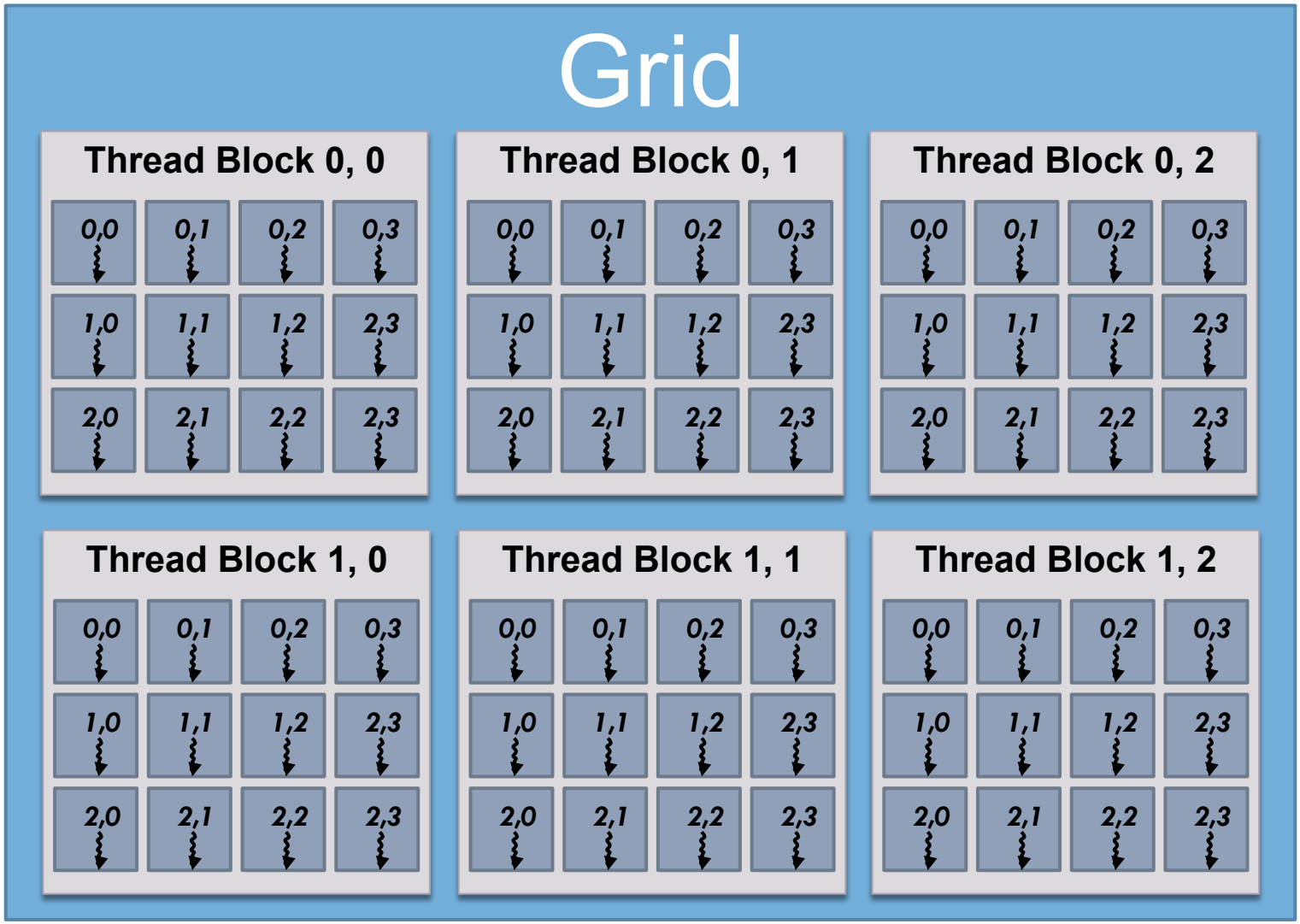


- Threads are grouped into thread blocks
  - Threads in the same block **can** cooperate
  - Threads in different blocks **cannot** cooperate

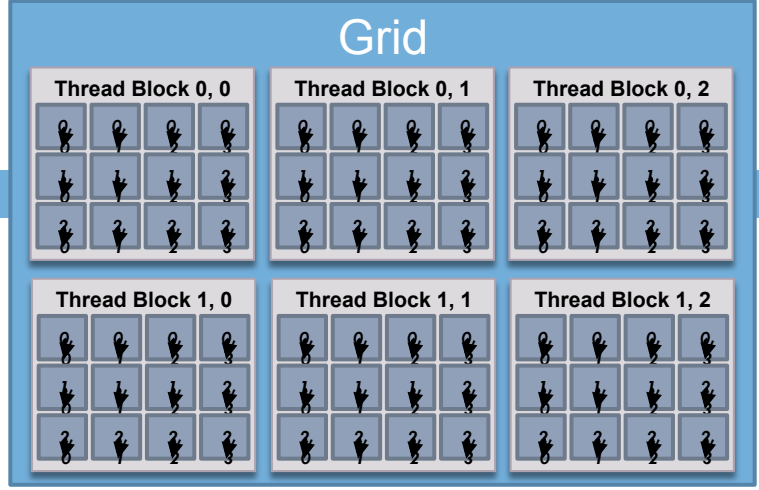


- All thread blocks are organized in a Grid
  - 1D or 2D or 3D
- Threads and blocks have unique IDs

Grids, Thread Blocks and Threads



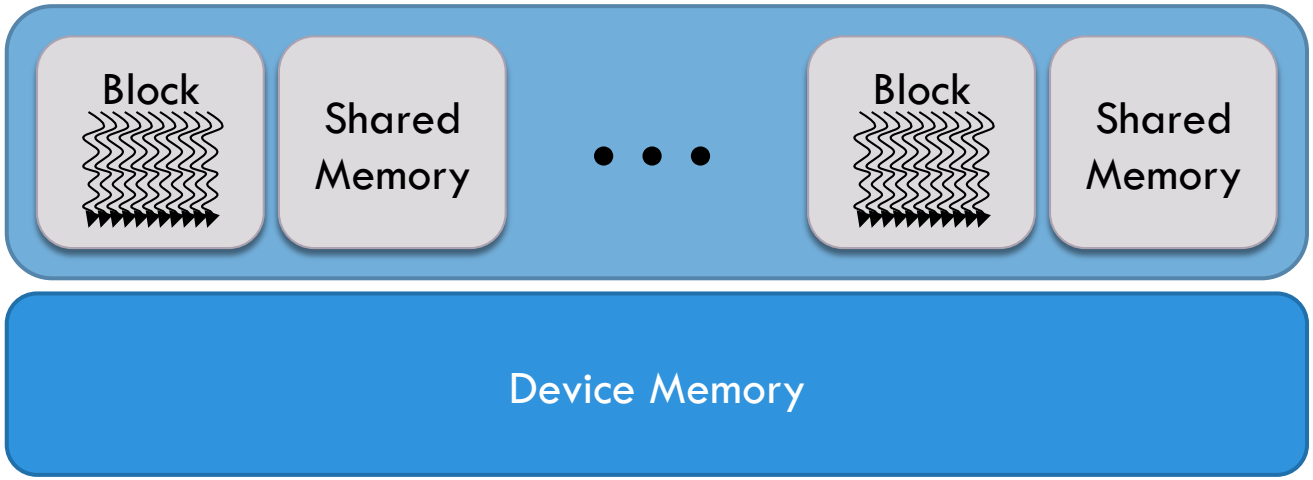
Indexing



- dim3 threadsPerBlock(3, 4);
  - threadsPerBlock.x = 3
  - threadsPerBlock.y = 4
  - threadID = (threadIdx.x, threadIdx.y)
- dim3 numBlocks(2, 3);
  - blockDim.x = 2
  - blockDim.y=3
  - blockID = (blockIdx.x, blockIdx.y)
- Launch kernel:

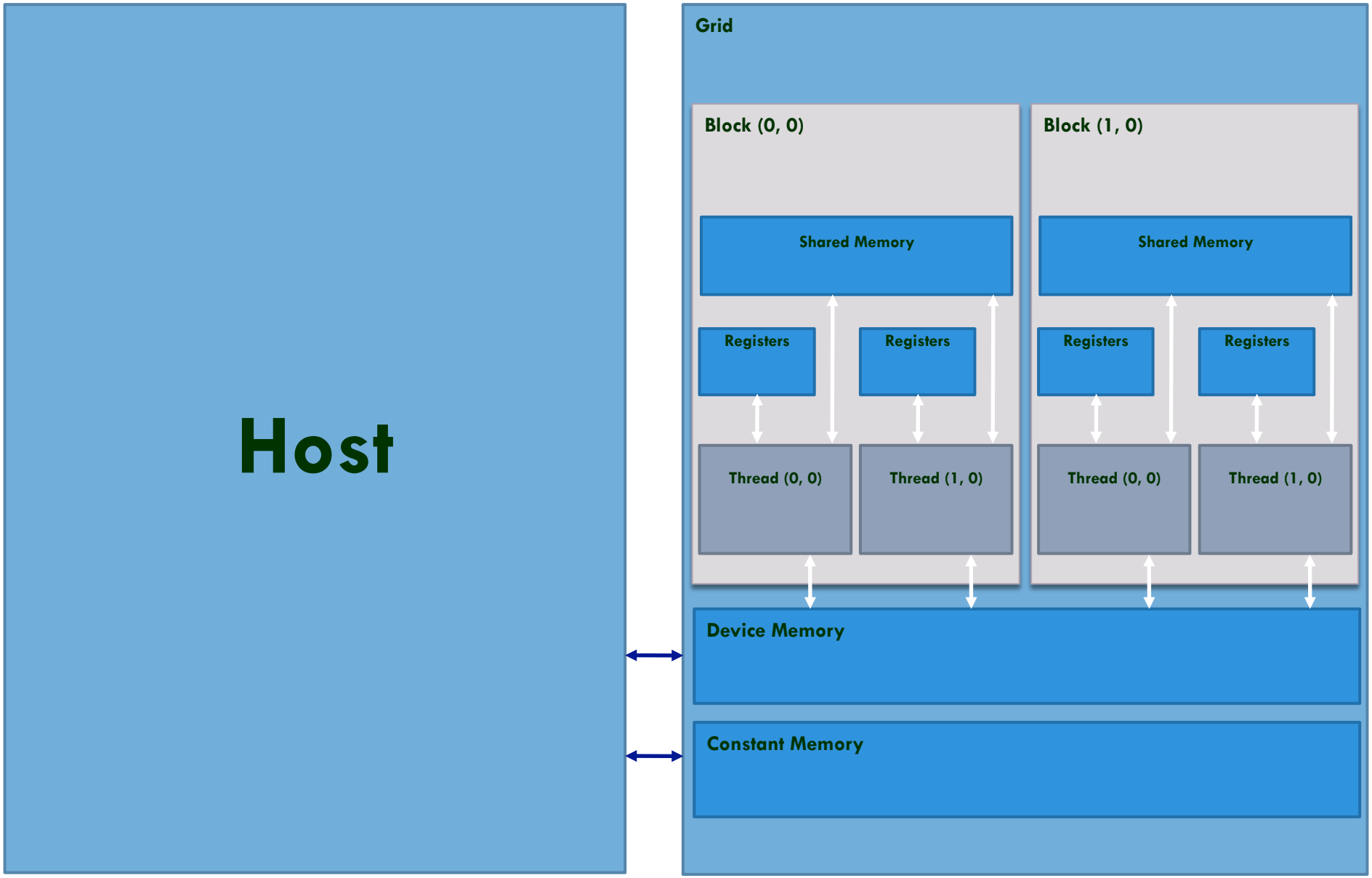
myKernel<<<numBlocks, threadsPerBlock>>>(…);

CUDA Model of Parallelism



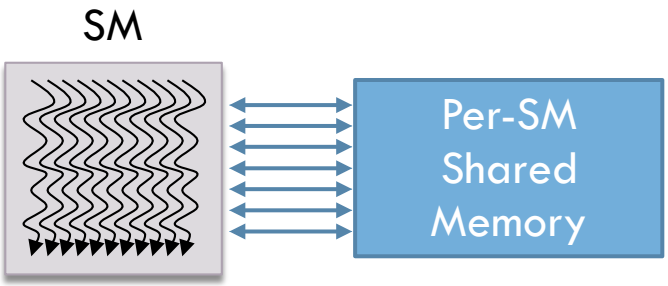
- CUDA virtualizes the physical hardware
  - Devices have
    - Different numbers of SMs
    - Different compute capabilities (Fermi = 2.0, before: 1.0, 1.1, 1.2)
  - block is a virtualized streaming multiprocessor (threads, shared memory)
  - thread is a virtualized scalar processor (registers, PC, state)
- Scheduled onto physical hardware without pre-emption
  - threads/blocks launch & run to completion
  - blocks have to be independent

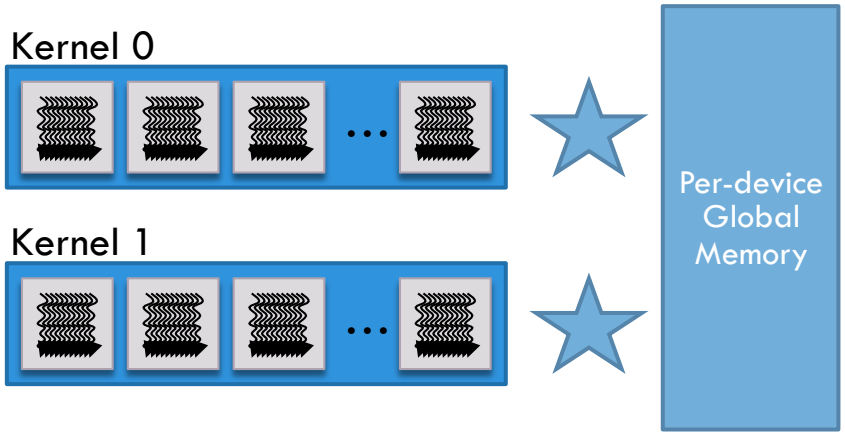
Memory Spaces in CUDA



Multiple Memory Scopes

- Per-thread private memory
  - Each thread has its own local memory
  - Stacks, other private data, **registers**
- Per-SM shared memory
  - Small memory close to the processor, low latency
- Device memory
  - GPU frame buffer
  - Can be accessed by any thread in any SM





Device 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

GPU Memory Allocation / Release

- Host (CPU) manages device (GPU) memory:
  - `cudaMalloc(void **pointer, size_t nbytes)`
  - `cudaMemset(void *pointer, int val, size_t count)`
  - `cudaFree(void* pointer)`

```
int n = 1024;
int nbytes = n * sizeof(int);
int* data = 0;
cudaMalloc(&data, nbytes);
cudaMemset(data, 0, nbytes);
cudaFree(data);
```

Data Copies

- `cudaMemcpy(void *dst, void *src,`
  `size_t nbytes,`
  `enum cudaMemcpyKind direction);`
  - returns after the copy is complete
  - blocks CPU thread until all bytes have been copied
  - doesn't start copying until previous CUDA calls complete

- `enum cudaMemcpyKind`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

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

CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	device	grid	application
`__constant__ int constant_var;`	constant	grid	application

C for CUDA

- Philosophy: provide minimal set of extensions necessary

- Function qualifiers:

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

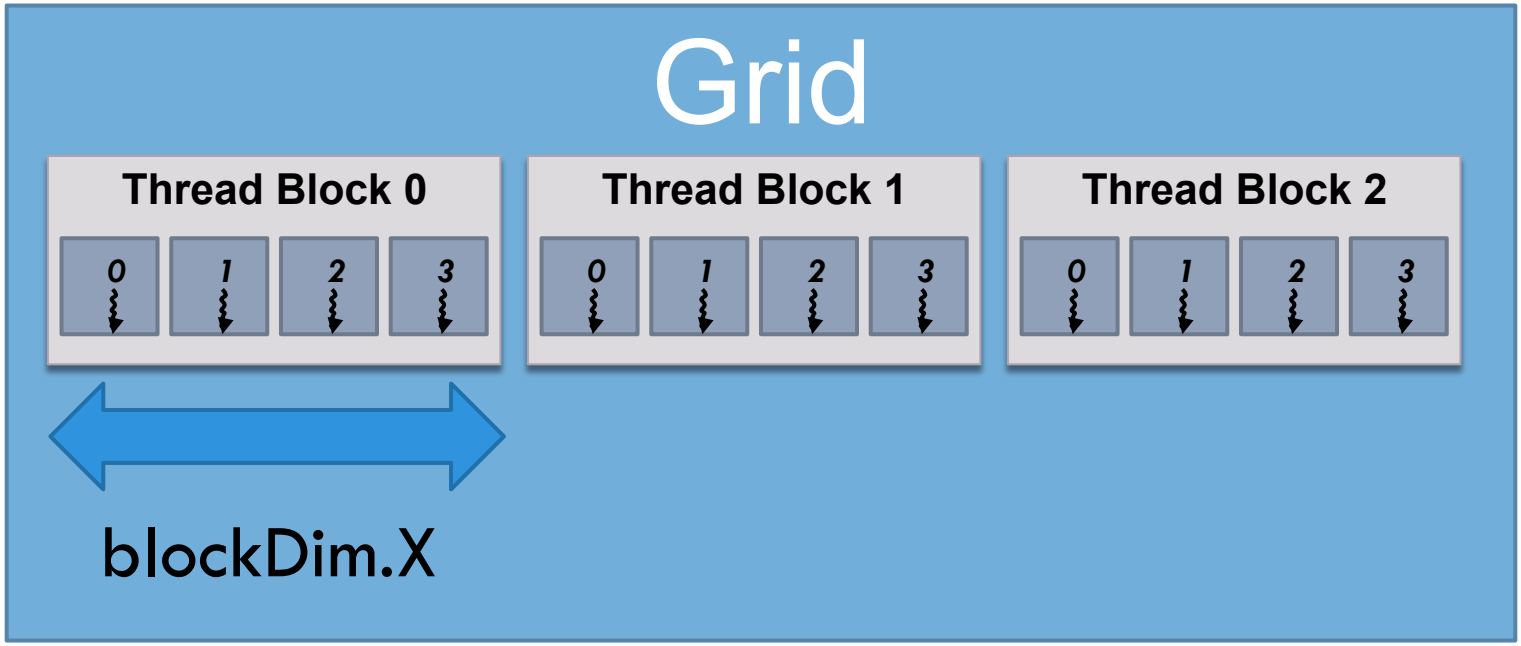
- Execution configuration:

```
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
```

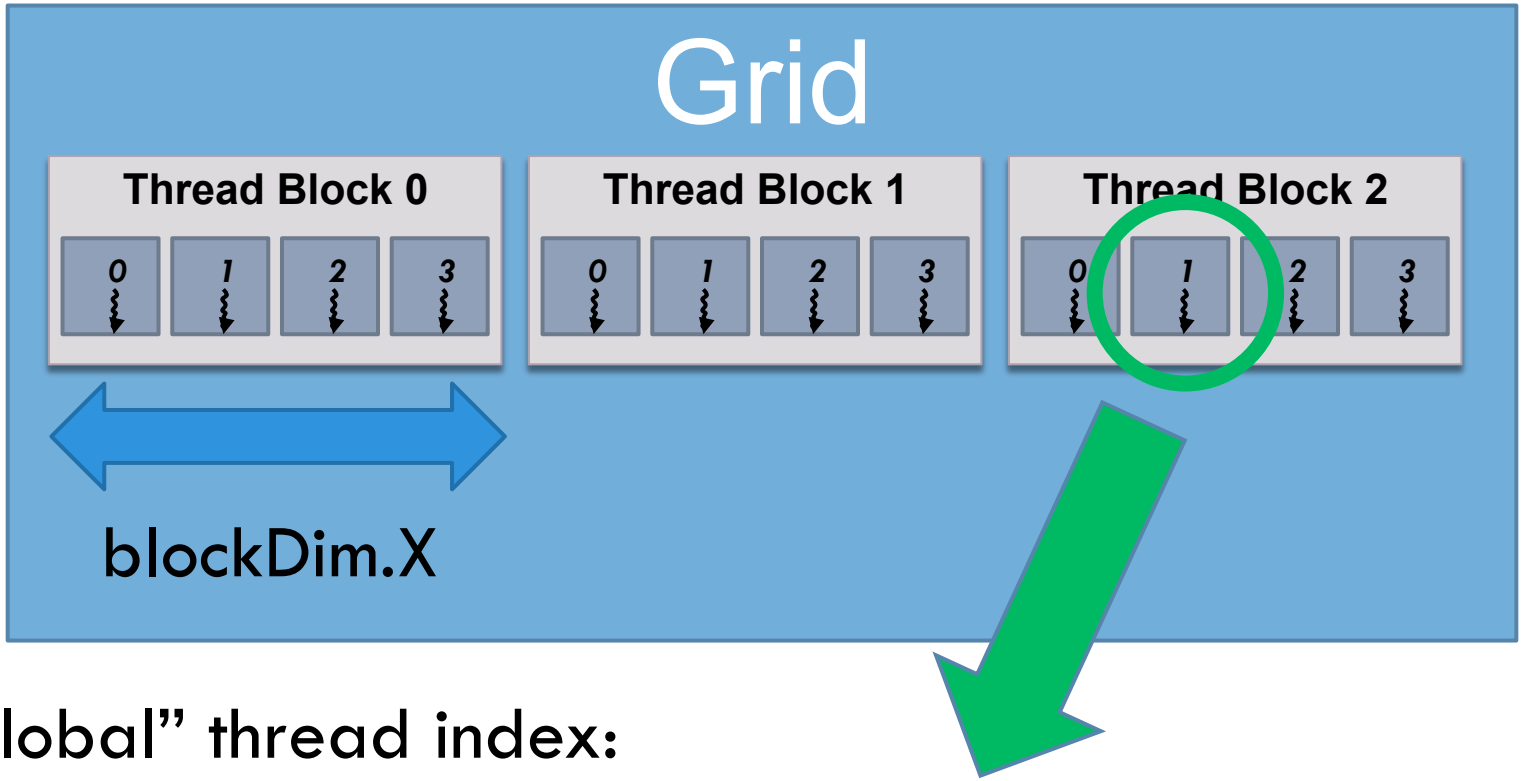
Calculating the global thread index



- "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


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

Vector add kernel: GPU & Host


```
// 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 kernel: GPU & Host


```
// 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?

```
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 ...
}
```

Host code

(can be in the same file)

Vector add kernel: GPU & Host


```
// 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);
}
```

Summary

- 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

Advanced CUDA: Scheduling, Synchronization, Atomics

Thread Scheduling

- Order in which thread blocks are scheduled is undefined!
  - any possible interleaving of blocks should be valid
  - presumed to run to completion without preemption
  - can run in any order
  - can run concurrently OR sequentially

- Order of threads within a block is also undefined!

Global synchronization

- Q: How do we do global synchronization with these scheduling semantics?

Global synchronization

- Q: How do we do global synchronization with these scheduling semantics?
- A1: Not possible!

Global synchronization

- Q: How do we do global synchronization with these scheduling semantics?
- A1: Not possible!
- A2: Finish a grid, and start a new one!

Global synchronization

- Q: How do we do global synchronization with these scheduling semantics?
- A1: Not possible!
- A2: Finish a grid, and start a new one!

```
step1<<<grid1,blk1>>>(...);
// CUDA ensures that all writes from step1 are complete.
step2<<<grid2,blk2>>>(...);
```

- We don't have to copy the data back and forth!

Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
  - No loss of data
  - Ordering is still **arbitrary**
- Different types of atomic instructions
  - Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
  - On device memory and/or shared memory
- Much more expensive than load + operation + store

Example: 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;
}
```

Example: 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;
}
```

Why is this incorrect?

Example: Histogram

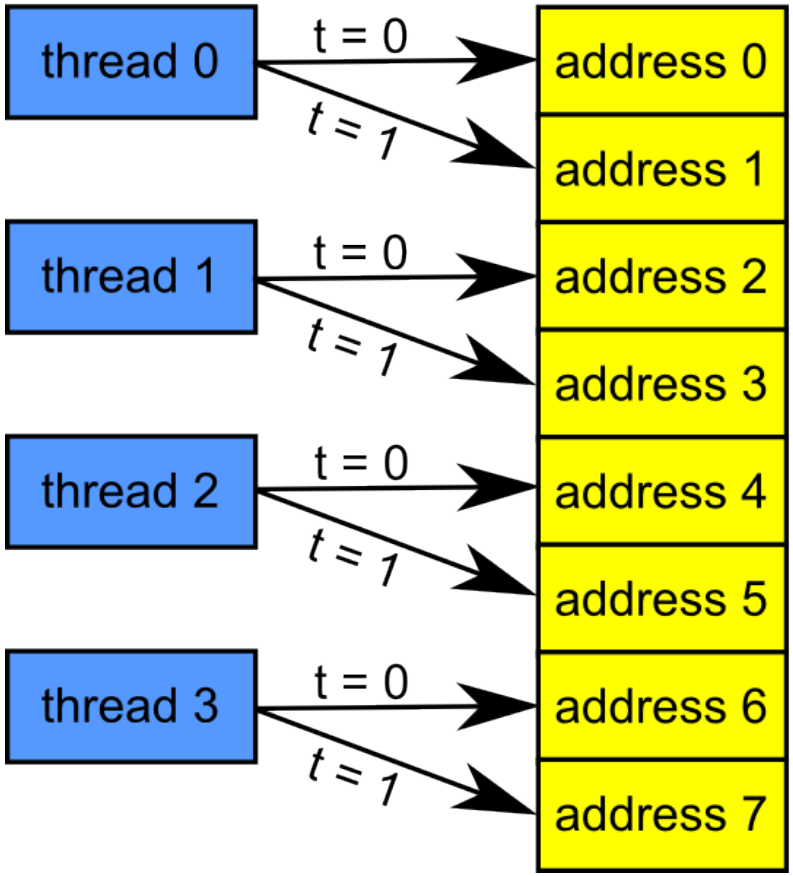

```
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter atomically

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

CUDA: optimizing your application

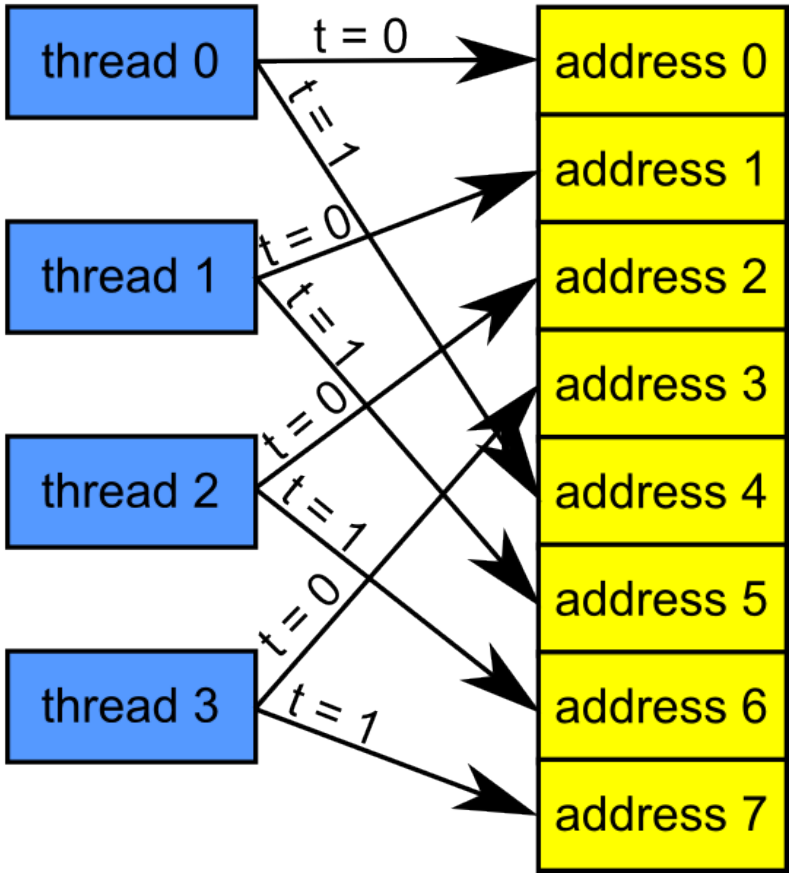
1. Coalescing
2. Shared Memory
3. Occupancy
4. Shared Memory Bank Conflicts

Coalescing

traditional multi-core
optimal memory access pattern



many-core GPU
optimal memory access pattern



Consider the stride of your accesses


```
__global__ void foo(int* input, float3* input2) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;

  // Stride 1, OK!
  int a = input[i];

  // Stride 2, half the bandwidth is wasted
  int b = input[2*i];

  // Stride 3, 2/3 of the bandwidth wasted
  float c = input2[i].x;
}
```

Example: Array of Structures (AoS)


```
struct record {
    int key;
    int value;
    int flag;
};

record *d_records;
cudaMalloc((void**)&d_records, ...);
```

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, ...);
```

Example: SoA vs AoS


```
__global__ void kernel(record* AoS_data,
                       SoA  SoA_data) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;

  // AoS wastes bandwidth
  int key1 = AoS_data[i].key;

  // SoA efficient use of bandwidth
  int key2 = SoA_data.keys[i];
}
```

Memory Coalescing

- Structure of arrays is often better than array of structures
- Very clear win on regular, stride 1 access patterns
- Unpredictable or irregular access patterns are case-by-case
- Can lose a factor of 10x – 30x!

4

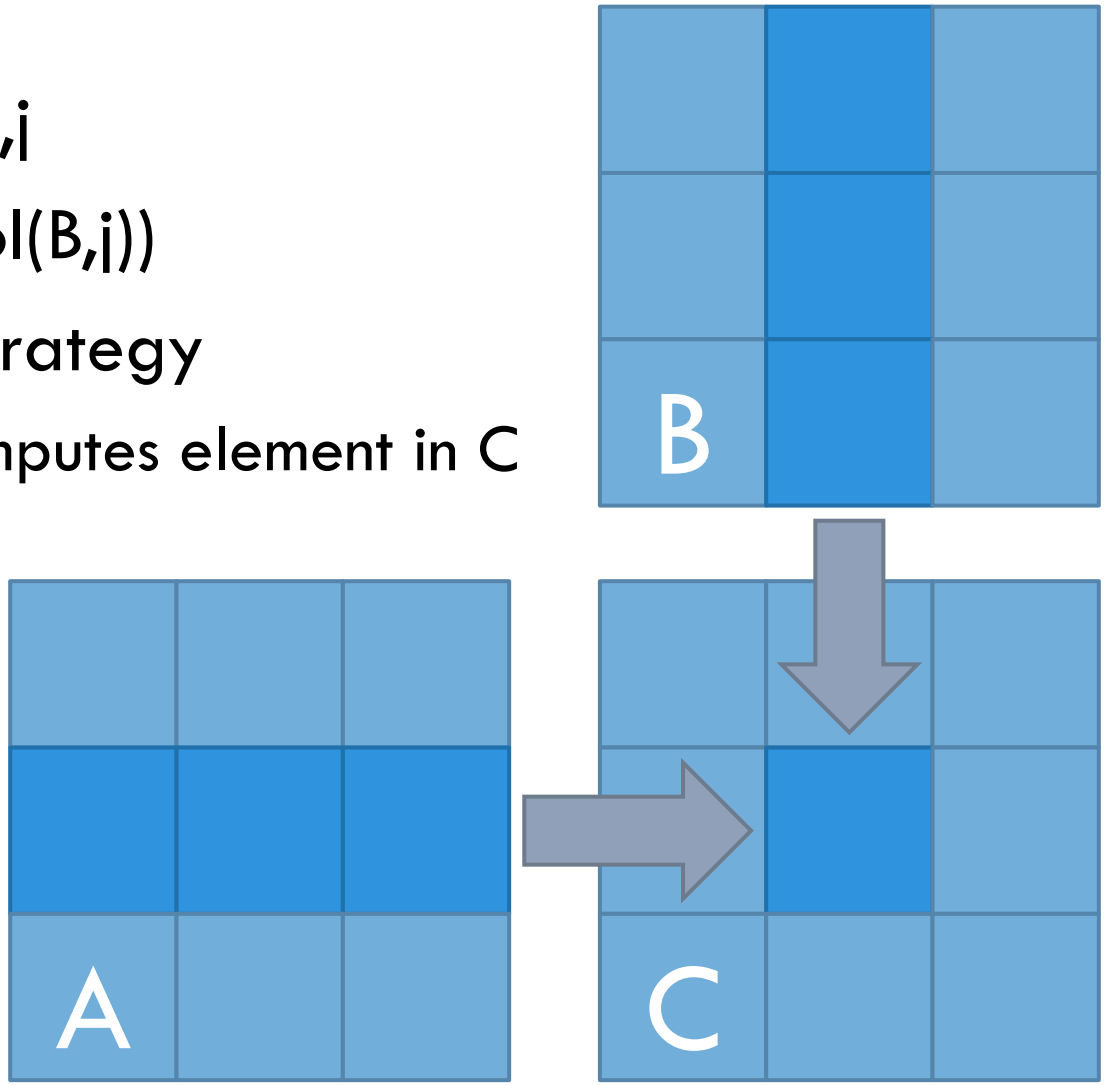
CUDA: optimizing your application

1. Coalescing
2. Shared Memory
3. Occupancy
4. Shared Memory Bank Conflicts

Matrix multiplication example

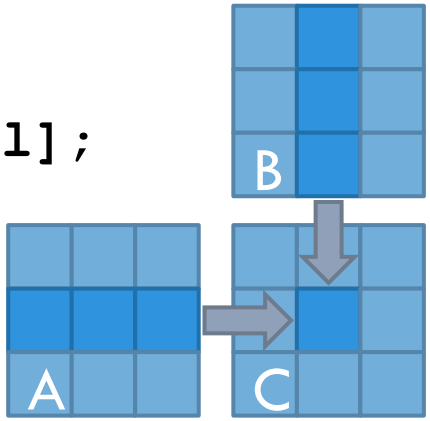
- C = A * B
- Each element C,i,j

  = dot(row(A,i),col(B,j))
- Parallelization strategy
  - Each thread computes element in C
  - 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 b
  for(int k = 0; k < width; k++) {
      result += a[row*width+k] * b[k*width+col];
  }
  c[row*width+col] = result;
}
```

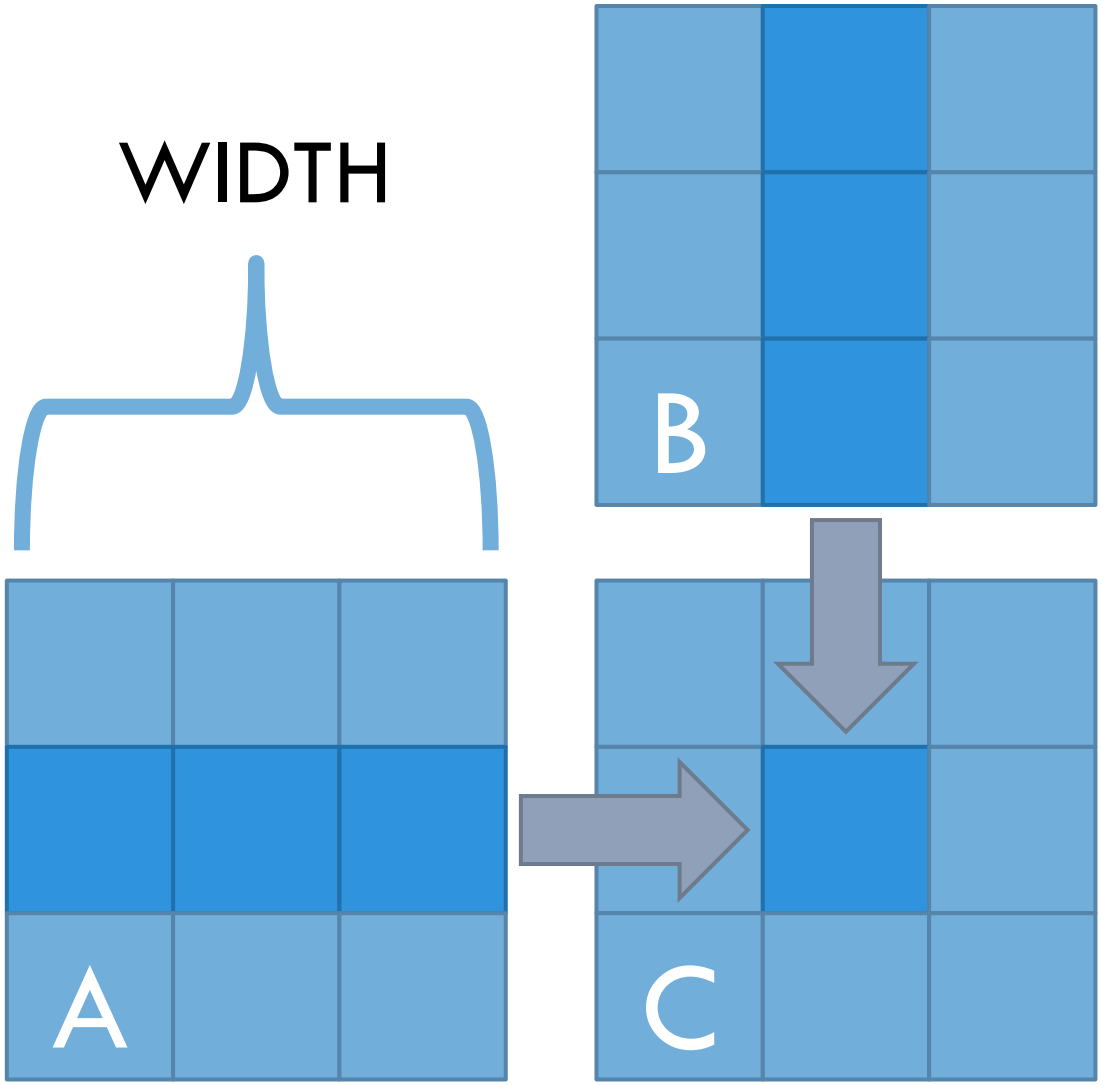


Matrix multiplication performance

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

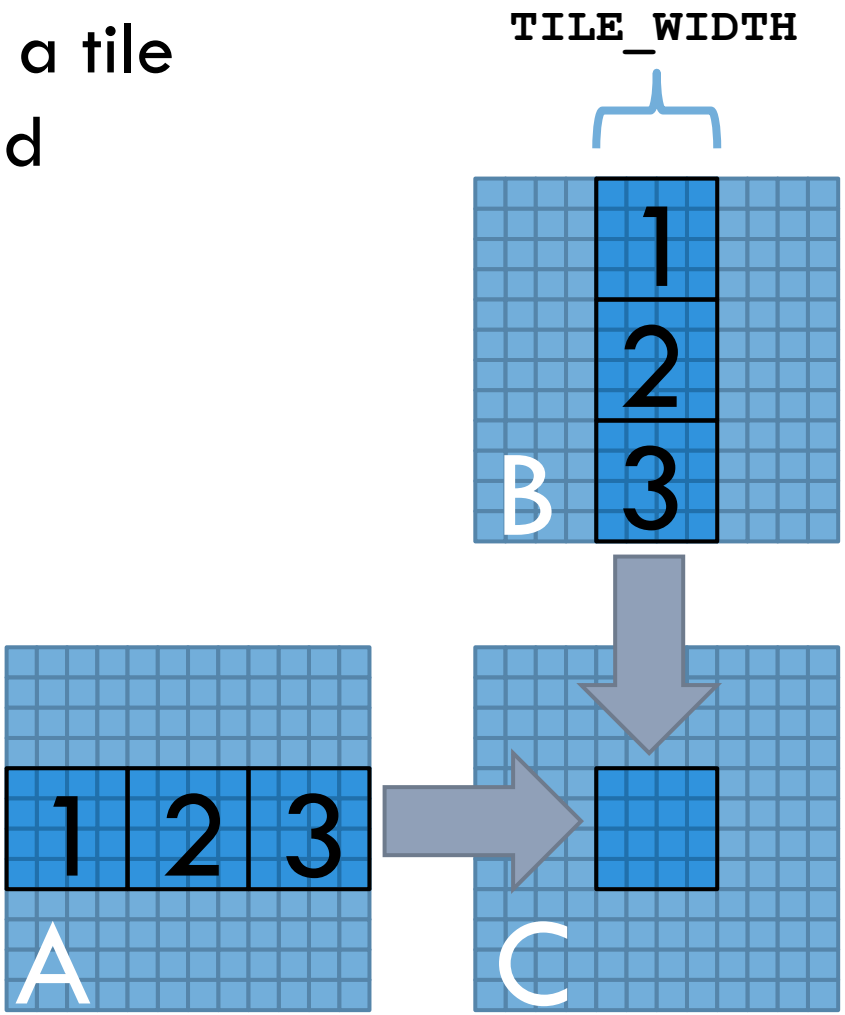
Data reuse

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

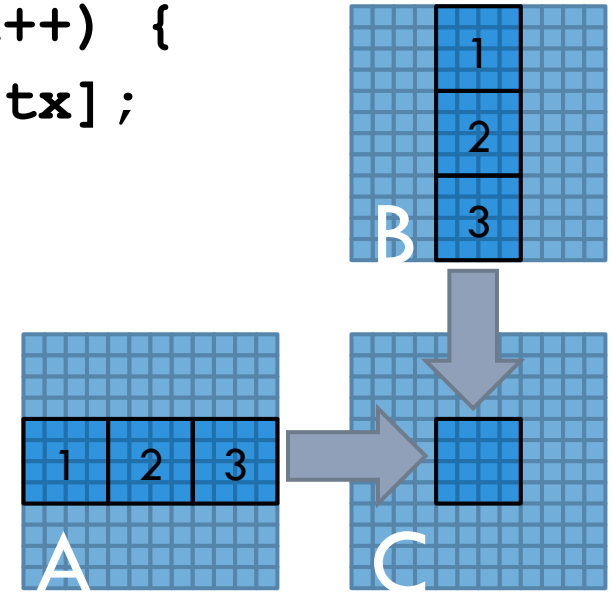


Matrix multiply with shared memory


```
__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
  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
  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];
    __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];
    }
    __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

- Use barriers to guard data
  - Guard against using uninitialized data
  - Guard against corrupting live data

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

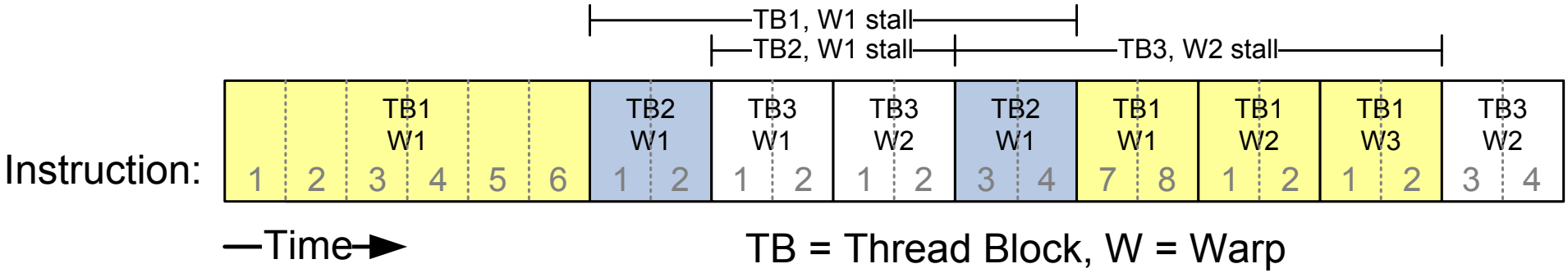
Performance GTX 580	1581 GFLOPs
Memory bandwidth GTX 580	192 GB/s
AI needed for peak	1581 / 192 = **8.23**
TILE_WIDTH required to achieve peak	0.25 * TILE_WIDTH = **8.23,** TILE_WIDTH = 32.9

CUDA: optimizing your application

1. Coalescing
2. Shared Memory
3. Occupancy
4. Shared Memory Bank Conflicts

Thread Scheduling

- SM implements zero-overhead warp scheduling
  - A warp is a group of 32 threads that runs concurrently on an SM
  - At any time, only one of the warps is executed by an SM
  - 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



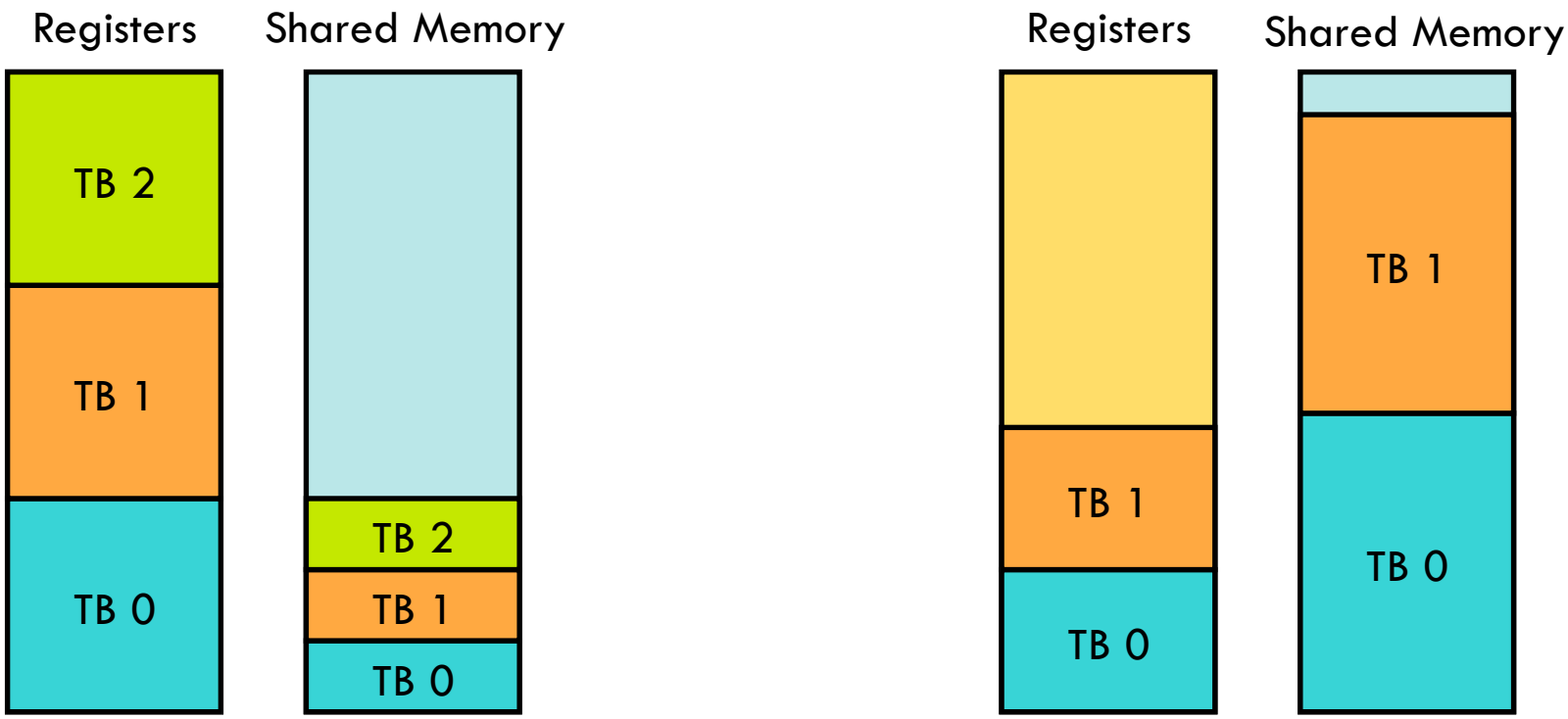
Stalling warps

- 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

- What determines occupancy?
  - Number of threads and blocks
  - Memory consumption
- Limited hardware resources
  - Register usage per thread => may limit number of threads
  - Shared memory per thread block => may limit number of blocks

Resource Limits (1)



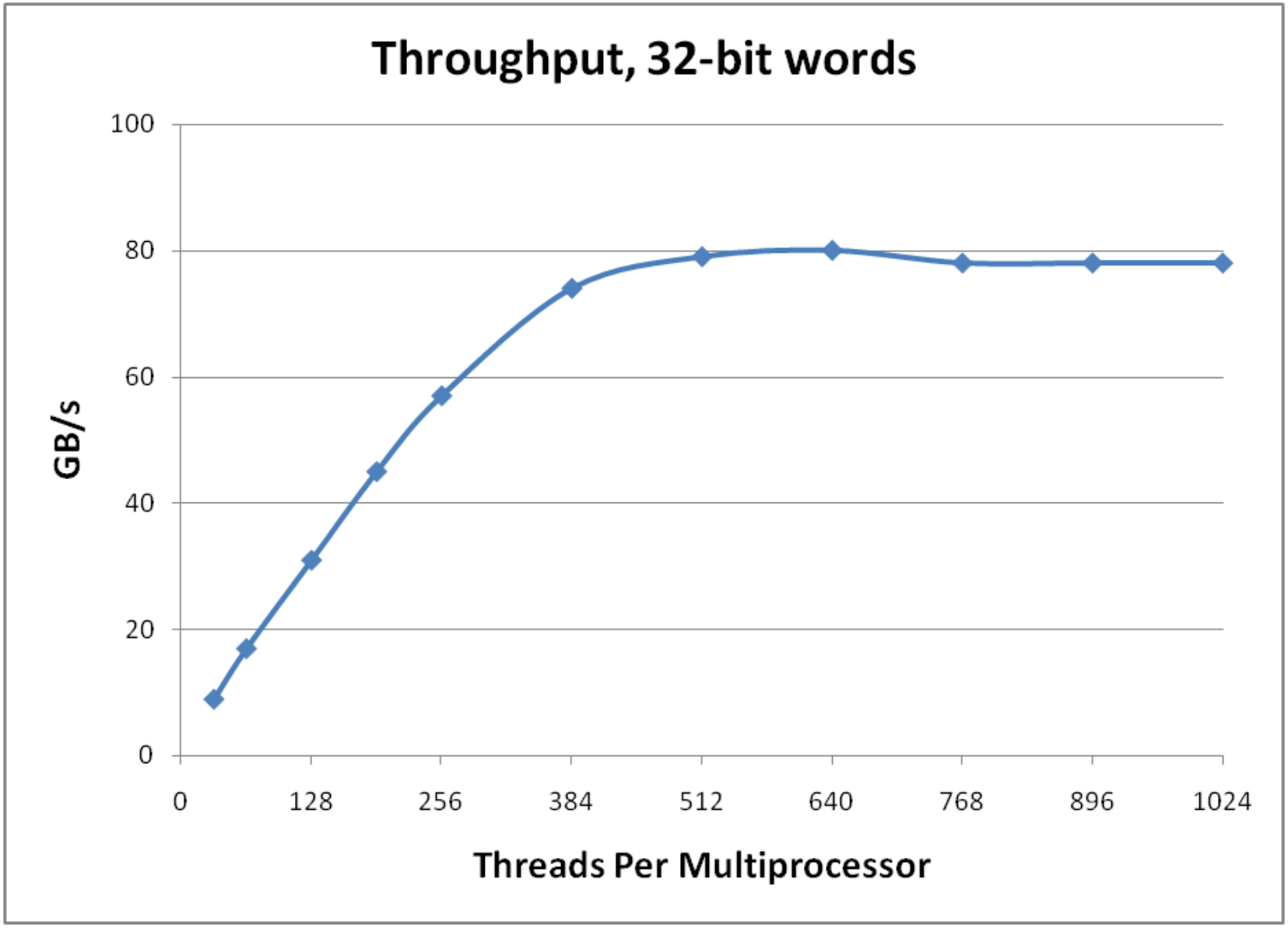
- Pool of registers and shared memory per SM
  - Each thread block grabs registers & shared memory
  - If one or the other is fully utilized ➡ no more thread blocks

Resource Limits (2)

- Can only have 8 thread blocks per SM
  - If they're too small, can't fill up the SM
  - Need 128 threads / block on gt200 (4 cycles/instruction)
  - Need 192 threads / block on Fermi (6 cycles/instruction)

- Higher occupancy has diminishing returns for hiding latency

Hiding Latency with more threads



How do you know what you're using?

- Use "**`nvcc -Xptxas -v`**" to get register and shared memory usage

- Plug those numbers into CUDA Occupancy Calculator

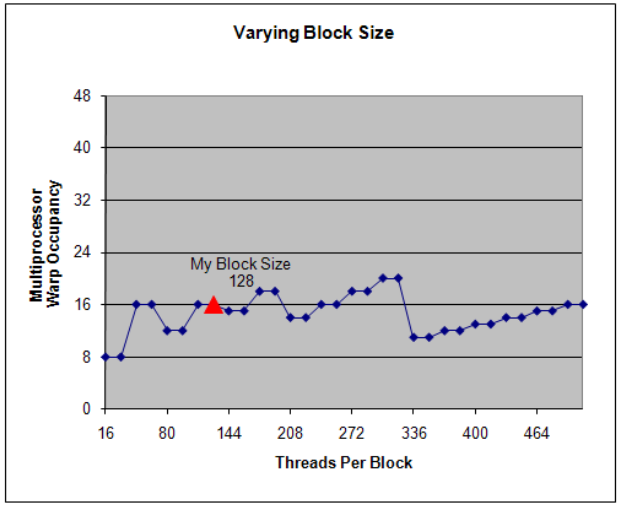
CUDA_Occupancy_calculator.xlsm - Microsoft Excel

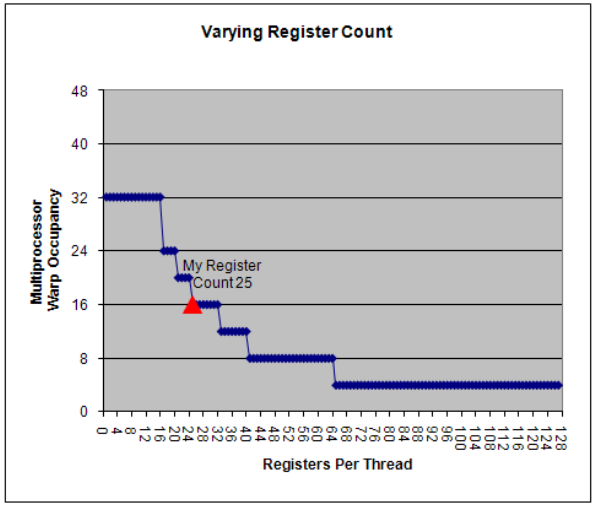
Security Warning Macros have been disabled. Options...

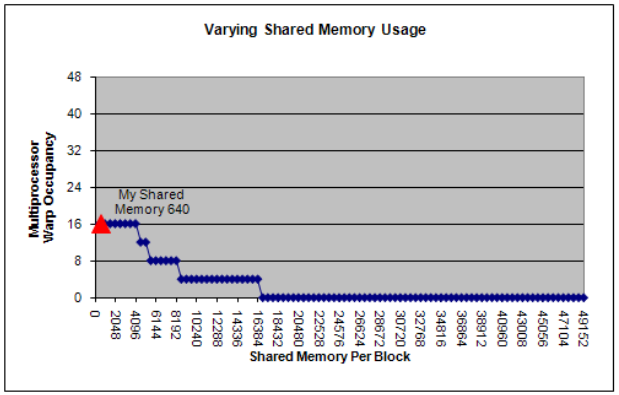
MyRegCount 25

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

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		







Calculator | Help | GPU Data | Copyright & License

Ready 100%

CUDA: optimizing your application

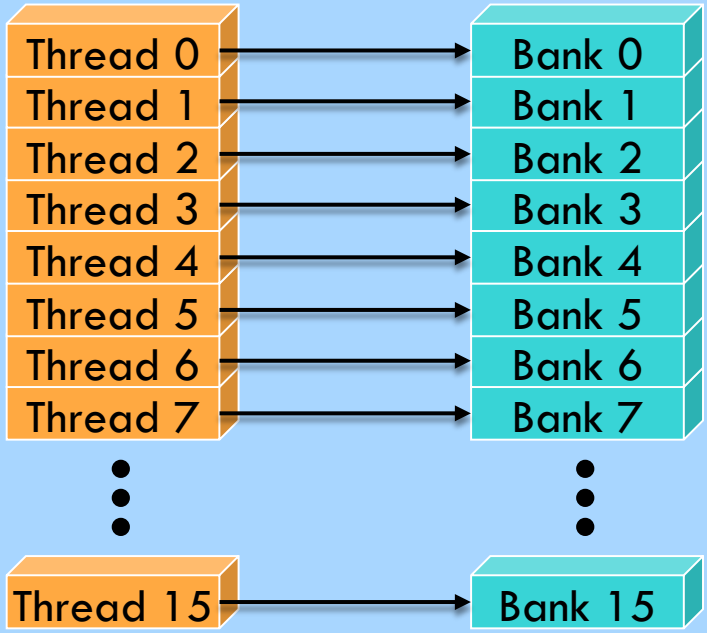
1. Coalescing
2. Shared Memory
3. Occupancy
4. Shared Memory Bank Conflicts

Shared Memory Banks

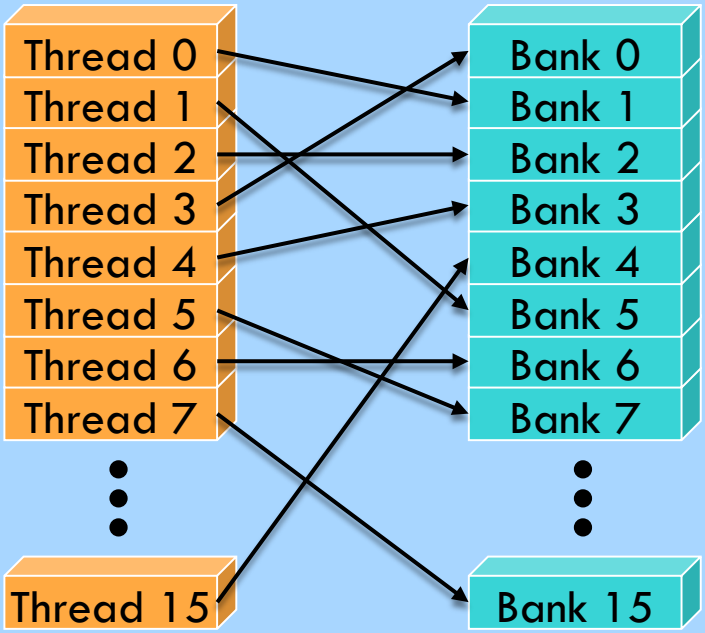
- Shared memory is banked
  - Only matters for threads within a warp
  - Full performance with some restrictions
    - Threads can each access different banks
    - Or can all access the same value
- Consecutive words are in different banks
- If two or more threads access the same bank but different value, we get bank conflicts

Bank Addressing Examples: OK

- **No Bank Conflicts**

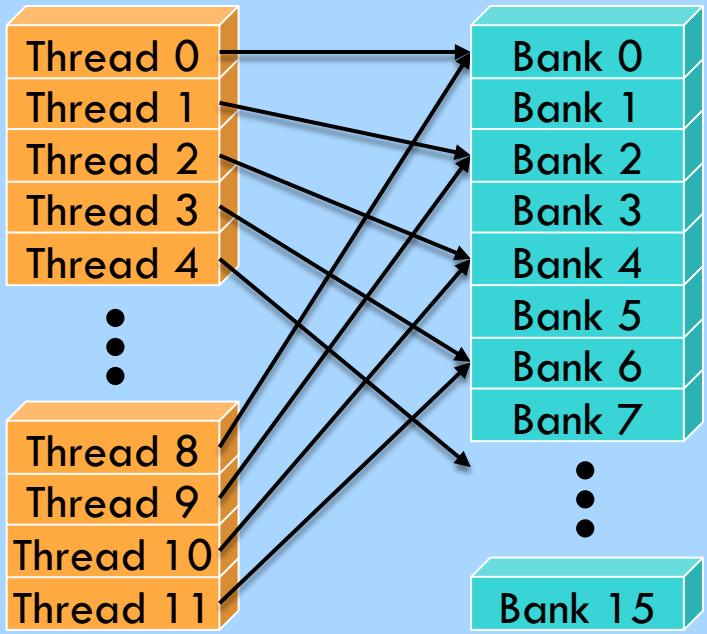


- **No Bank Conflicts**

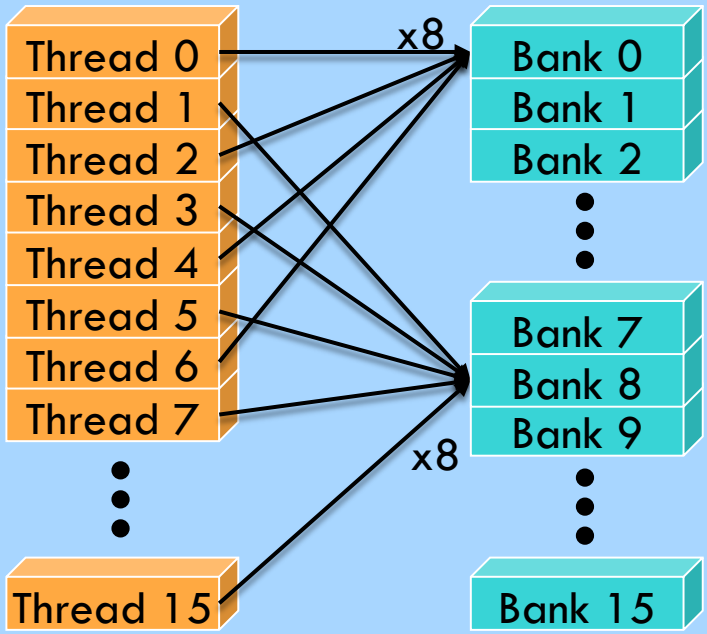


Bank Addressing Examples: BAD

- **2-way Bank Conflicts**



- **8-way Bank Conflicts**



Trick to Assess Performance Impact

- Change all shared memory reads to the same value
- All broadcasts = no conflicts
- Will show how much performance could be improved by eliminating bank conflicts

- The same doesn't work for shared memory writes
  - So, replace shared memory array indices with **`threadIdx.x`**
  - (Could also be done for the reads)

5

OpenCL: Programming GPUs, CPUs, APUs

Portability

- Inter-family vs inter-vendor
  - NVIDIA Cuda runs on all NVIDIA GPU families
  - OpenCL runs on all GPUs, Cell, CPUs
- Parallelism portability
  - Different architecture requires different granularity
  - Task vs data parallel
- Performance portability
  - Can we express platform-specific optimizations?

The Khronos group

OpenCL: Open Compute Language

- Architecture independent
- Explicit support for many-cores
- Low-level host API
  - Uses C library, no language extensions
- Separate high-level kernel language
  - Explicit support for vectorization
- Run-time compilation
- Architecture-dependent optimizations
  - Still needed
  - Possible

Cuda vs OpenCL Terminology

CUDA	OpenCL
Thread	Work item
Thread block	Work group
Device memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory

Cuda vs OpenCL Qualifiers

Functions

CUDA	OpenCL
__global__	__kernel
__device__	(no qualifier needed)

Variables

CUDA	OpenCL
__constant__	__constant
__device__	__global
__shared__	__local

Cuda vs OpenCL Indexing

CUDA	OpenCL
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
Calculate manually	get_global_id()
Calculate manually	get_global_size()

`__syncthreads()` → `barrier()`

Vector add: Cuda vs OpenCL kernel

CUDA

```
__global__ void
vectorAdd(float* a, float* b, float* c) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  c[index] = a[index] + b[index];
}
```

OpenCL

```
__kernel void
vectorAdd(__global float* a, __global float* b,
          __global float* c) {
    int index = get_global_id(0);
    c[index] = a[index] + b[index];
}
```

OpenCL VectorAdd host code (1)


```
const size_t workGroupSize = 256;
const size_t nrWorkGroups = 3;
const size_t totalSize = nrWorkGroups * workGroupSize;

cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

// create properties list of key/values, 0-terminated.
cl_context_properties props[] = {
  CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
  0
};

cl_context context = clCreateContextFromType(props,
  CL_DEVICE_TYPE_GPU, 0, 0, 0);
```

OpenCL VectorAdd host code (2)


```
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT, 1,
                &device, NULL);

// create command queue on 1st device the context reported
cl_command_queue commandQueue =
    clCreateCommandQueue(context, device, 0, 0);

// create & compile program
cl_program program = clCreateProgramWithSource(context, 1,
    &programSource, 0, 0);
clBuildProgram(program, 0, 0, 0, 0, 0);

// create kernel
cl_kernel kernel = clCreateKernel(program, "vectorAdd",0);
```

OpenCL VectorAdd host code (3)


```
float* A, B, C = new float[totalSize]; // alloc host vecs
// initialize host memory here...

// allocate device memory
cl_mem deviceA = clCreateBuffer(context,
  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  totalSize * sizeof(cl_float), A, 0);

cl_mem deviceB = clCreateBuffer(context,
  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
  totalSize * sizeof(cl_float), B, 0);

cl_mem deviceC = clCreateBuffer(context,
  CL_MEM_WRITE_ONLY, totalSize * sizeof(cl_float), 0, 0);
```

OpenCL VectorAdd host code (4)


```
// setup parameter values
clSetKernelArg(kernel, 0, sizeof(cl_mem), &deviceA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &deviceB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &deviceC);

clEnqueueNDRangeKernel(commandQueue, kernel, 1, 0,
  &totalSize, &workGroupSize, 0,0,0); // execute kernel

// copy results from device back to host, blocking
clEnqueueReadBuffer(commandQueue, deviceC, CL_TRUE, 0,
  totalSize * sizeof(cl_float), C, 0, 0, 0);

delete[] A, B, C; // cleanup
clReleaseMemObject(deviceA); clReleaseMemObject(deviceB);
clReleaseMemObject(deviceC);
```

Summary and Conclusions

Summary and conclusions

- Higher performance cannot be reached by increasing clock frequencies anymore
- Solution: introduction of large-scale parallelism
- Multiple cores on a chip
  - Today:
    - Up to 48 CPU cores in a node
    - Up to 3200 compute elements on a single GPU
  - Host system can contain multiple GPUs: 10,000+ cores
  - We can build clusters of these nodes!
- Future: 100,000s – millions of cores?

Summary and conclusions

- Many different types of many-core hardware
- Very different properties
  - Performance
  - Programmability
  - Portability
- It's all about the memory
- Choose the right platform for your application
  - Arithmetic intensity / Operational intensity
  - Roofline model

Open questions

- New application domains – e.g., signal processing, graph processing.
  - Perfomrance analysis
  - Peformance prediction
  - Modeling
- Memory patterns understanding, description, detection, automatic improvement
  - Local memory usage
- Heterogeneous computing
  - Using both the host and the device
- Application-device fitting

Questions?

- Slides are/will be available
- If you are interested in working with us on using GPUs for new applications, let us know!

A.L.Varbanescu@uva.nl