



MULTIMOORE SUMMERSCHOOL 2012 MANY-CORE ACCELERATORS

Who am I

- 10 years of Grid computing
- 5 years of many-core computing, radio astronomy
- Netherlands eScience center
 - ▣ Lead many-core solutions, astronomy, green computing
- ASTRON: Netherlands institute for radio astronomy
 - ▣ LOFAR
 - ▣ SKA
 - ▣ Many-core
- VU Amsterdam
 - ▣ CUDA teaching center

Today's schedule



- Morning:
 - Introduction
 - GPU hardware
 - GPU programming with Cuda
- Afternoon:
 - Practical: programming Cuda

4

What are many-cores?

What are many-cores

5

- From Wikipedia: “A many-core processor is a multi-core processor in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient — largely because of issues with **congestion** in supplying **instructions** and **data** to the many processors.”

What are many-cores

6

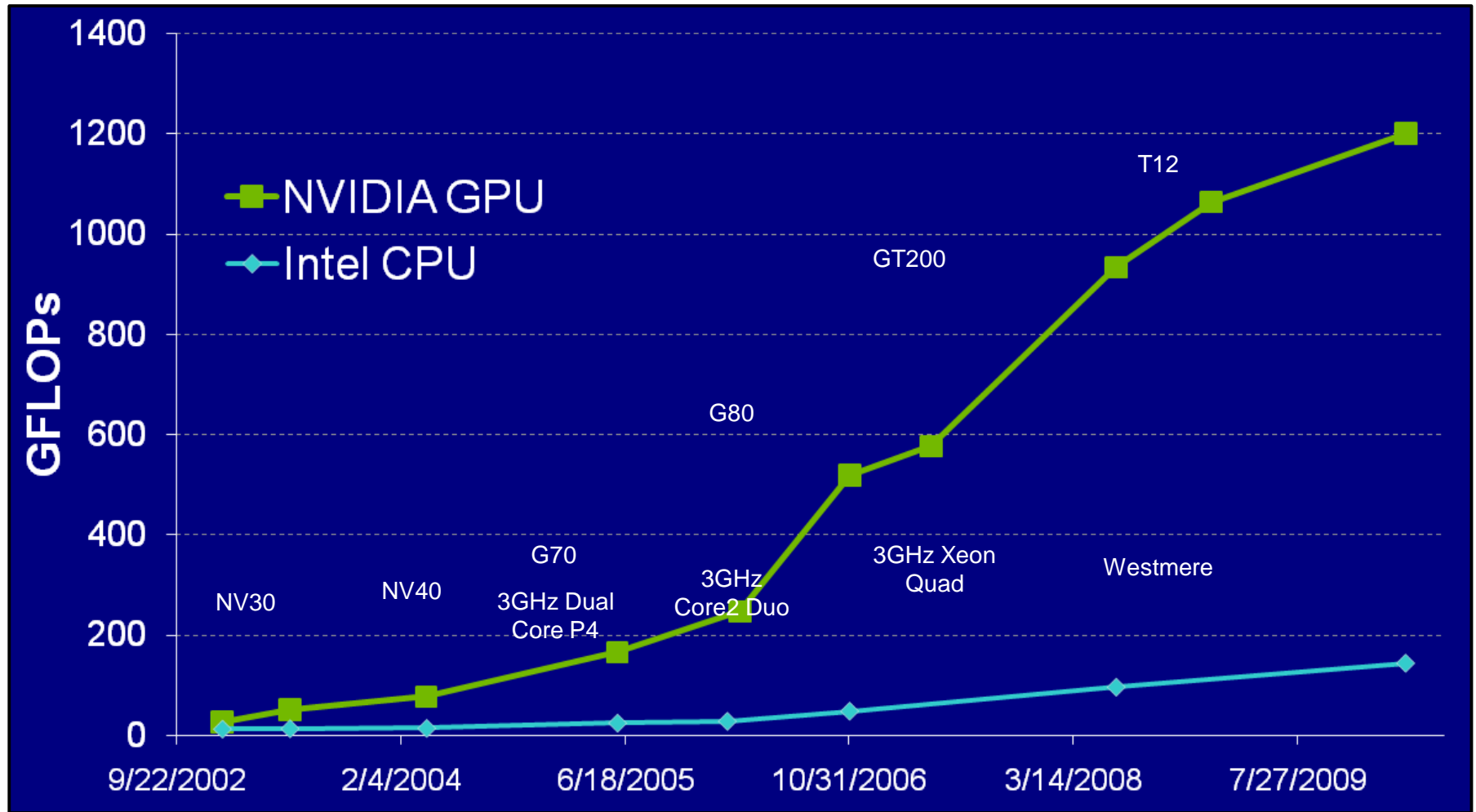
- How many is many?
 - ▣ Several tens of cores
- How are they different from multi-core CPUs?
 - ▣ Non-uniform memory access (NUMA)
 - ▣ Private memories
 - ▣ Network-on-chip
- Examples
 - ▣ Multi-core CPUs (48-core AMD magny-cours)
 - ▣ Graphics Processing Units (GPUs)
 - ▣ Cell processor (PlayStation 3)
 - ▣ Server processors (Sun Niagara)

7

Why do we need many-cores?

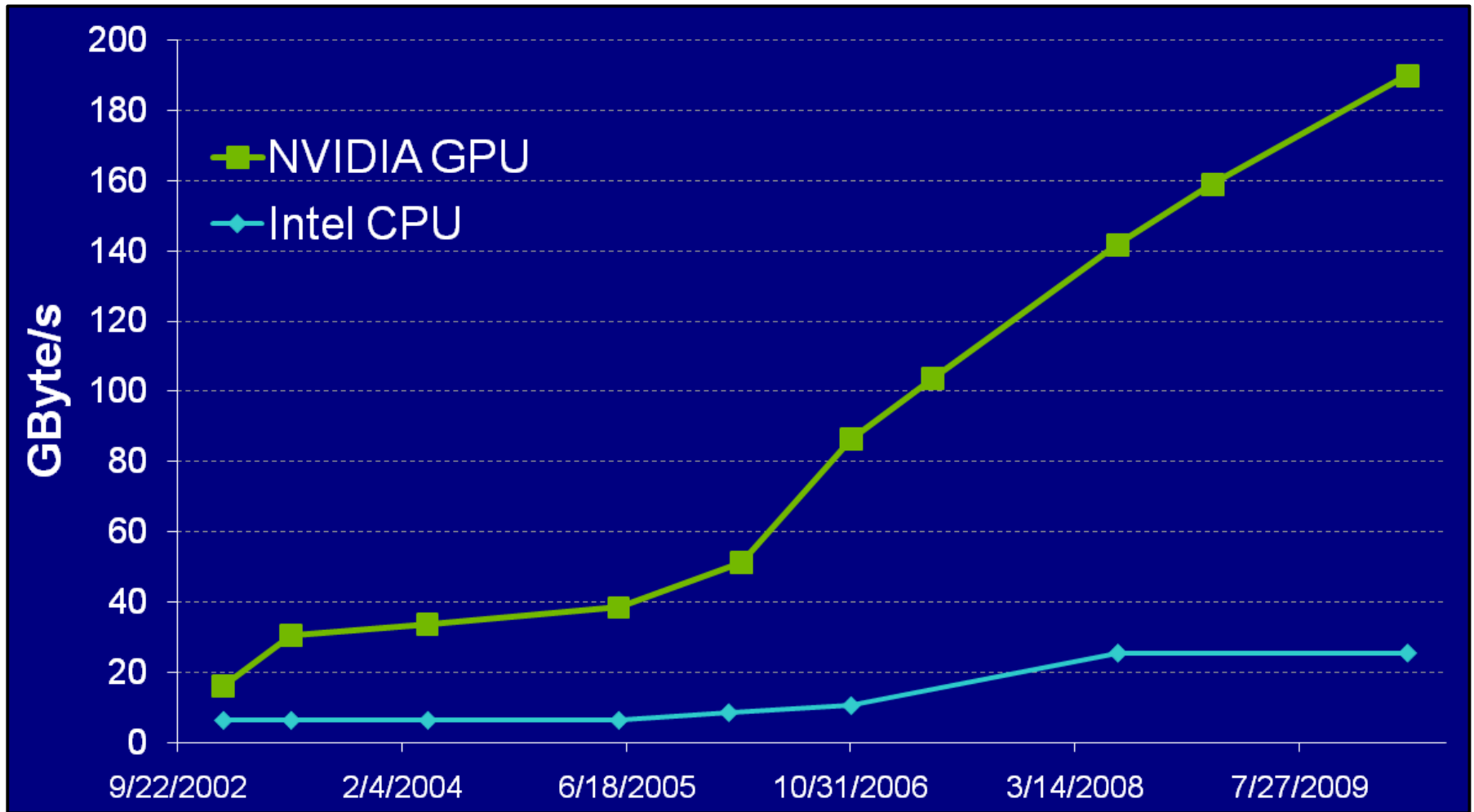
Why do we need many-cores?

8



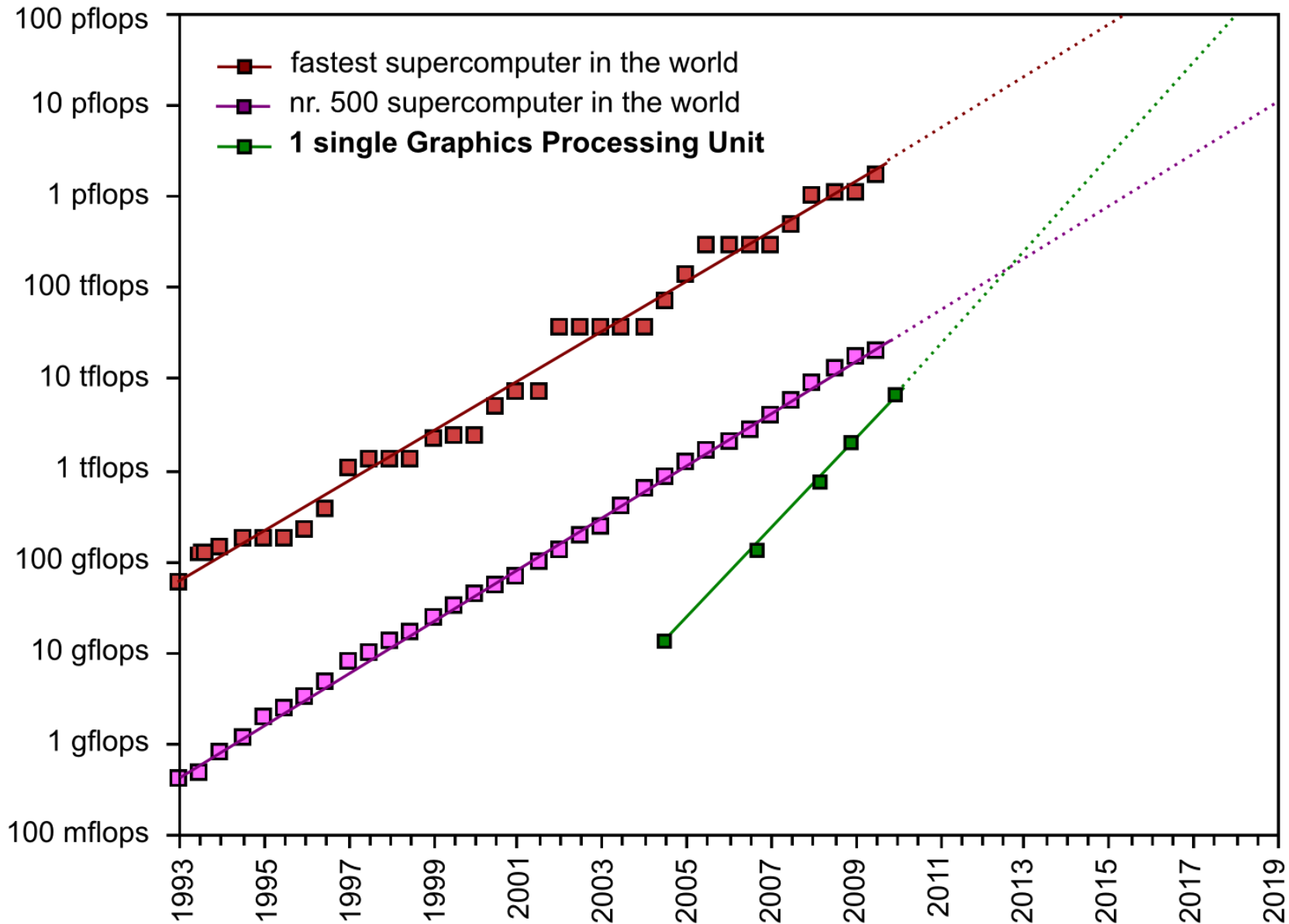
Why do we need many-cores?

9



Why do we need many-cores?

10



China's Tianhe-1 A

11

#5 in top500 list

4.701 pflops peak

2.566 pflops max



www.china-defense-mashup.com

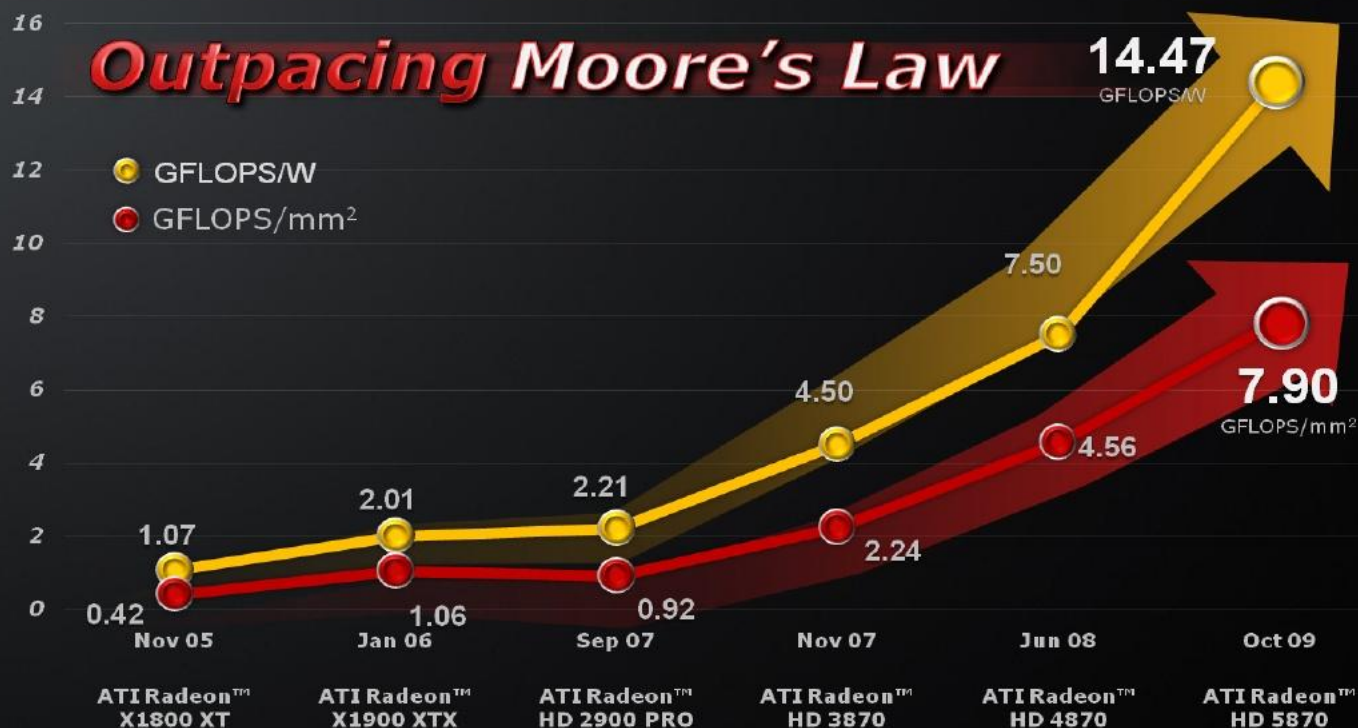
14,336 Xeon X5670 processors

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

Power efficiency

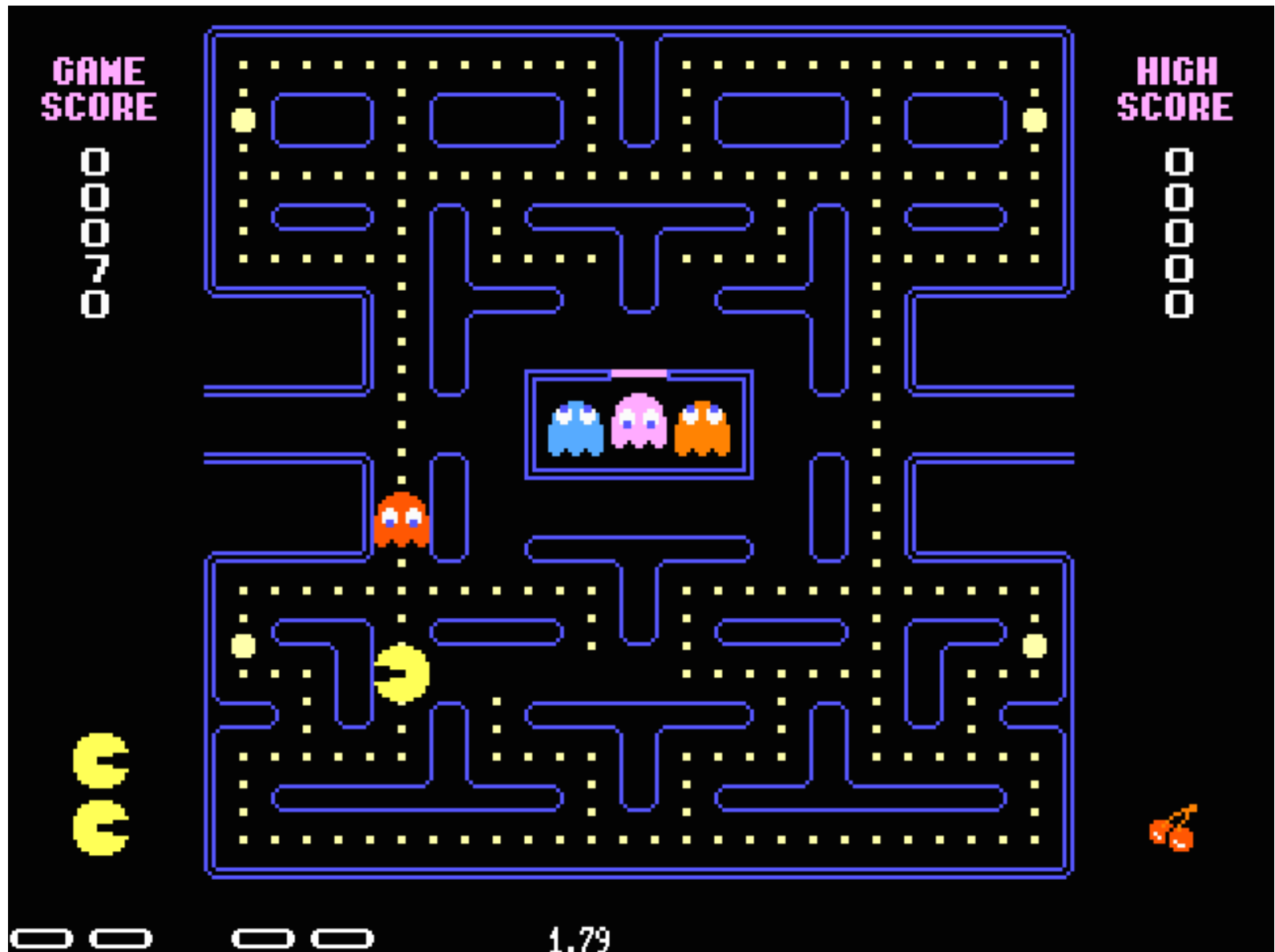
12

The World's Most Efficient GPU



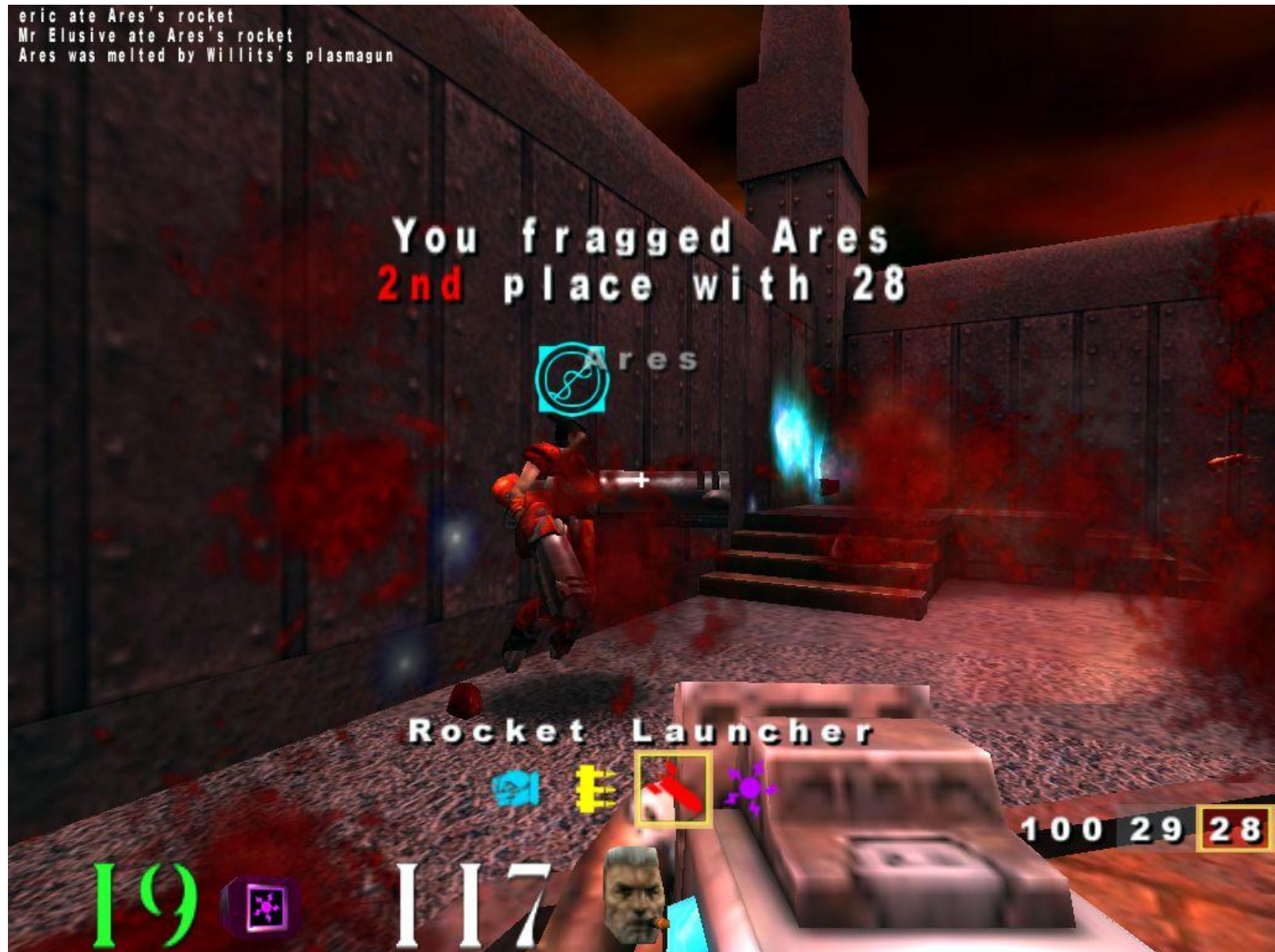
Graphics in 1980

13



Graphics in 2000

14



Graphics now: GPU movie

Why do we need many-cores?

16

- Performance
 - ▣ Large scale parallelism
- Power Efficiency
 - ▣ Use transistors more efficiently
- Price (GPUs)
 - ▣ Huge market, bigger than Hollywood
 - ▣ Mass production, economy of scale
 - ▣ “spotty teenagers” pay for our HPC needs!

17

GPU hardware introduction



Lessons from Graphics Pipeline

18

- Throughput is paramount
 - ▣ must paint every pixel within frame time
 - ▣ scalability

- Create, run, & 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

Why is this different from a CPU?

19

- 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 => skip the big caches
 - ▣ share control logic across many threads

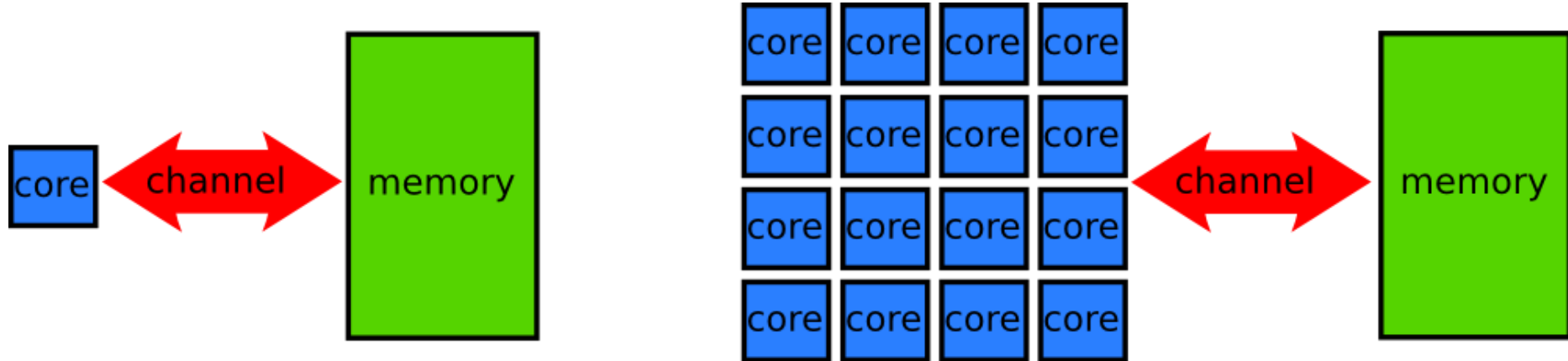
Key architectural Ideas

20

- SIMT (Single Instruction Multiple Thread) execution
 - ▣ HW automatically handles divergence
- Hardware multithreading
 - ▣ HW resource allocation & thread scheduling
 - ▣ HW relies on threads to hide latency
 - ▣ Context switching is (basically) free

It's all about the memory

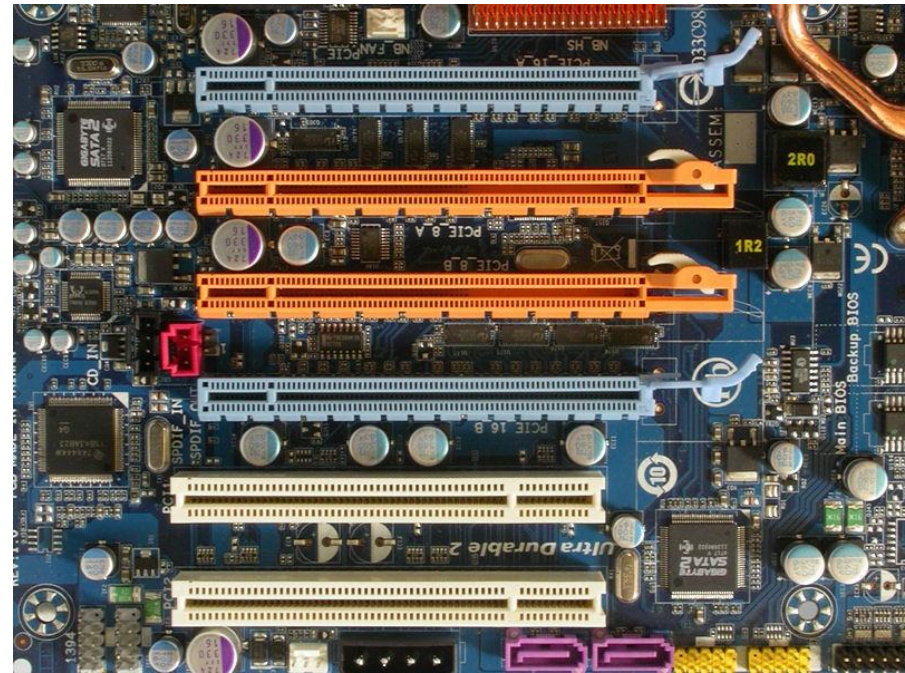
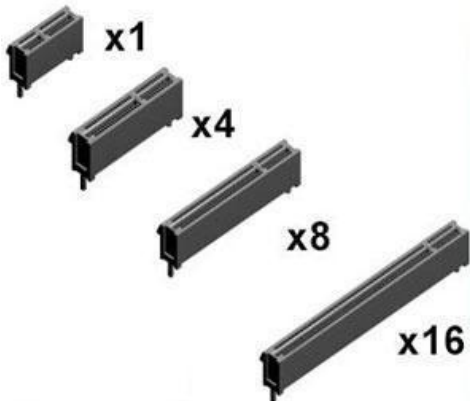
21



Integration into host system

22

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



23

GPU Hardware: NVIDIA



Fermi

24

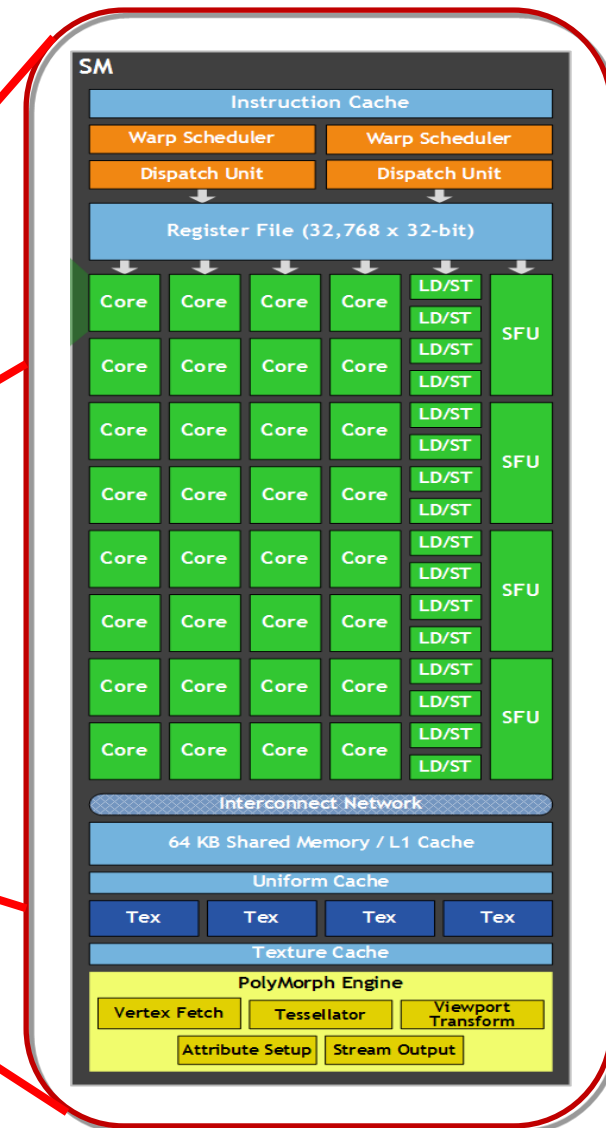
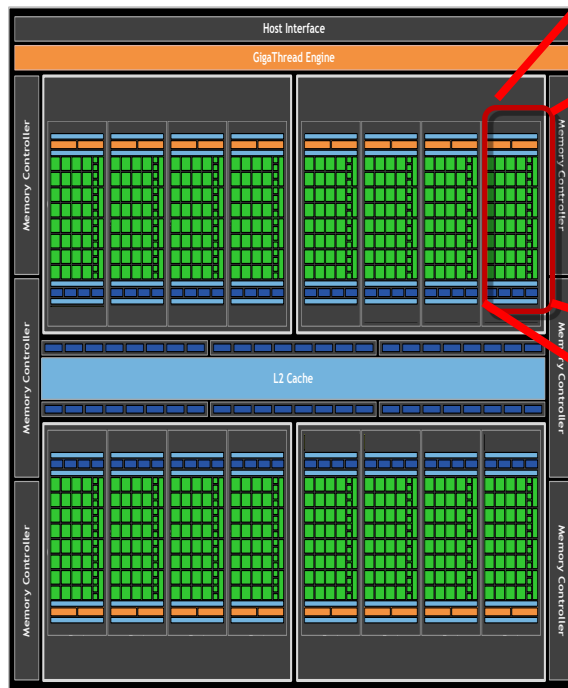
- Consumer: GTX 480, 580
- GPGPU: Tesla C2050
 - More memory, ECC
 - 1.0 teraflop single
 - 515 megaflop double
- Streaming multiprocessors (SM)
 - GTX 580: 16
 - GTX 480: 15
 - C2050: 14
- SMs are independent



Fermi Streaming Multiprocessor (SM)

25

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



Memory Hierarchy

26

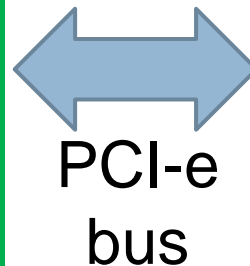
- Configurable L1 cache per SM
 - ▣ 16KB L1 cache / 48KB Shared
 - ▣ 48KB L1 cache / 16KB Shared
- Shared 768KB L2 cache

registers

L1 cache / shared memory

L2 cache

Host memory



Device memory

Multiple Memory Scopes

27

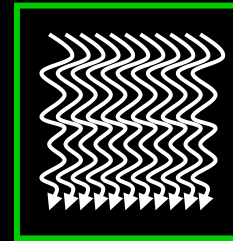
- Per-thread private memory
 - ▣ Each thread has its own local memory
 - ▣ Stacks, other private data
- Per-SM shared memory
 - ▣ Small memory close to the processor, low latency
- Device memory
 - ▣ GPU frame buffer
 - ▣ Can be accessed by any thread in any SM

Thread



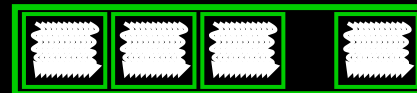
Per-thread
Local Memory

SM

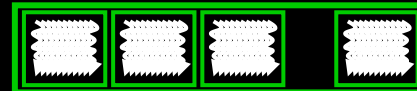


Per-SM
Shared
Memory

Kernel 0



Kernel 1



Per-device
Global
Memory

Atomic Operations

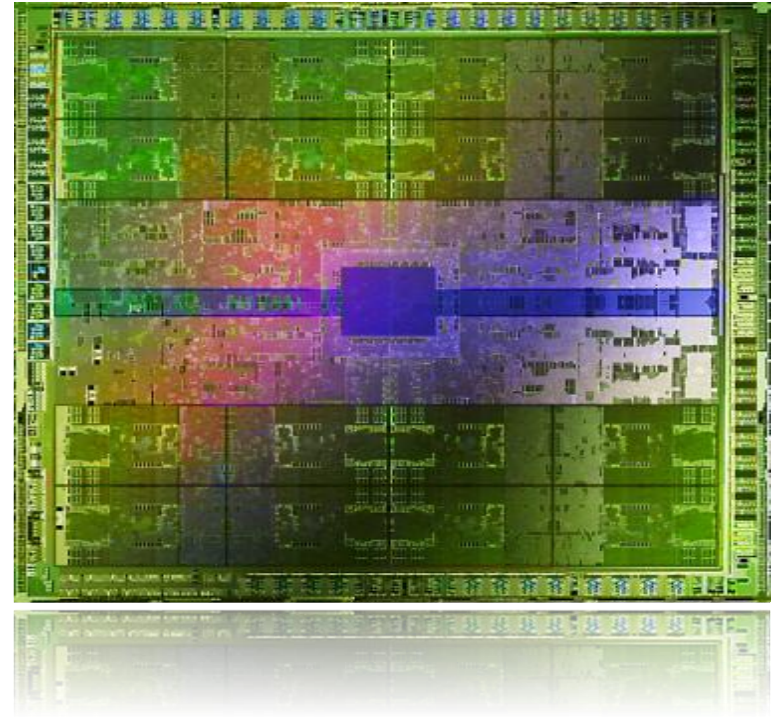
28

- Device memory is not coherent!
- Share data between streaming multiprocessors
- Read / Modify / Write

NVIDIA GPUs become more generic

29

- Expand performance sweet spot of the GPU
 - ▣ Caching
 - ▣ Concurrent kernels
 - ▣ Double precision floating point
 - ▣ C++
- Full integration in modern software development environment
 - ▣ Debugging
 - ▣ Profiling
- Bring more users, more applications to the GPU



30

Generic programming models

OpenCL

31



OpenCL: Open Compute Language

32

- 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

Portability

33

- 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
 - ▣ Architecture-dependent optimizations
 - ▣ Still needed
 - ▣ Possible

34

Programming NVIDIA GPUs

- CUDA: Scalable parallel programming
 - ▣ C/C++ extensions
- 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

36

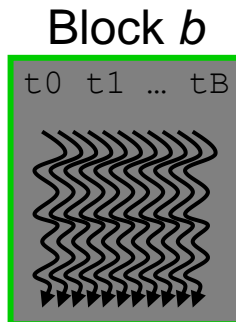

- Hierarchy of concurrent threads
- Lightweight synchronization primitives
- Shared memory model for cooperating threads

Hierarchy of concurrent threads

37

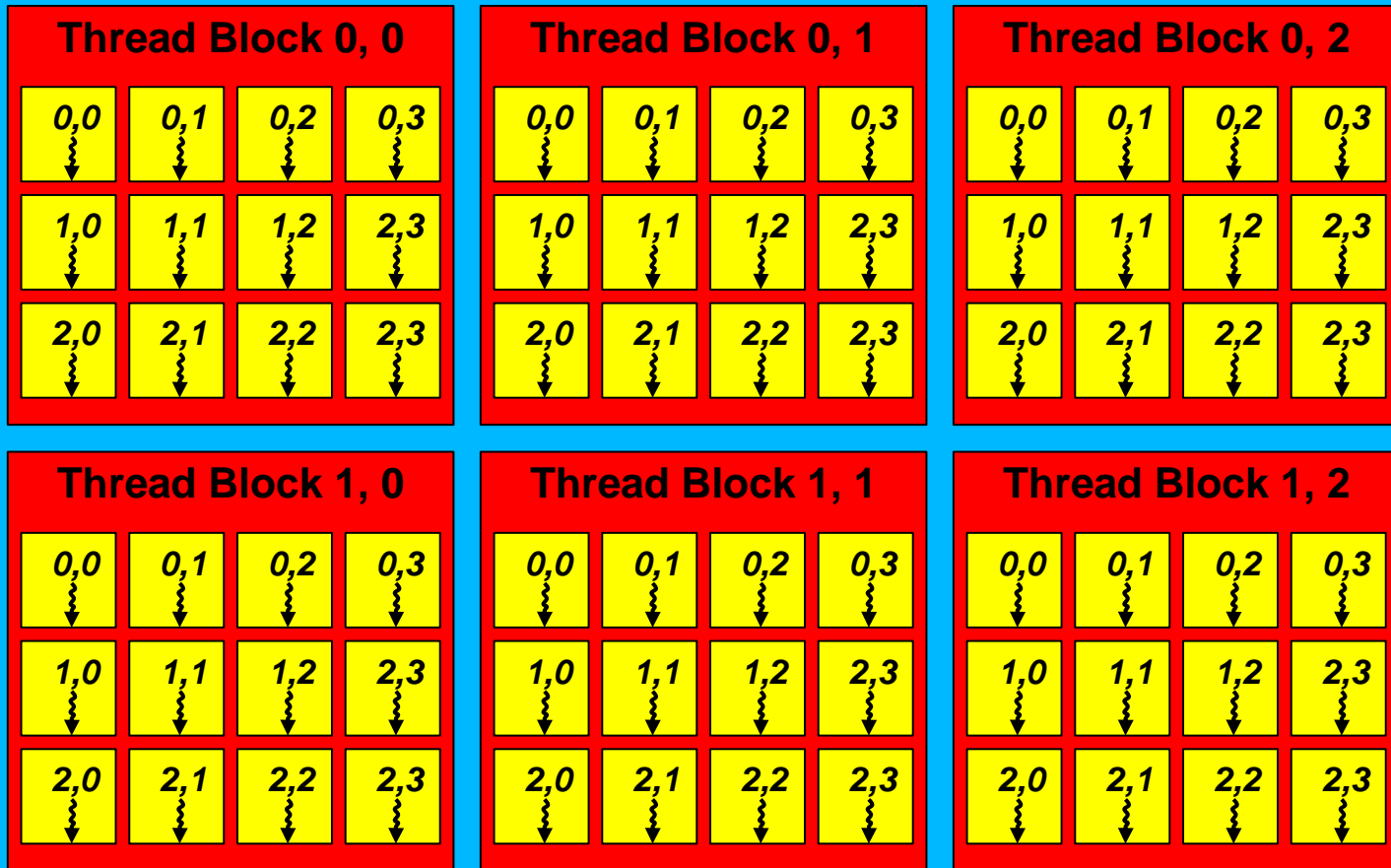
- Parallel kernels composed of many threads
 - ▣ All threads execute the same sequential program
 - ▣ Called the **Kernel**
- Threads are grouped into thread blocks
 - ▣ Threads in the same block can cooperate
 - ▣ Threads in different blocks cannot!
- All thread blocks are organized in a Grid
- Threads/blocks have unique IDs

Thread t



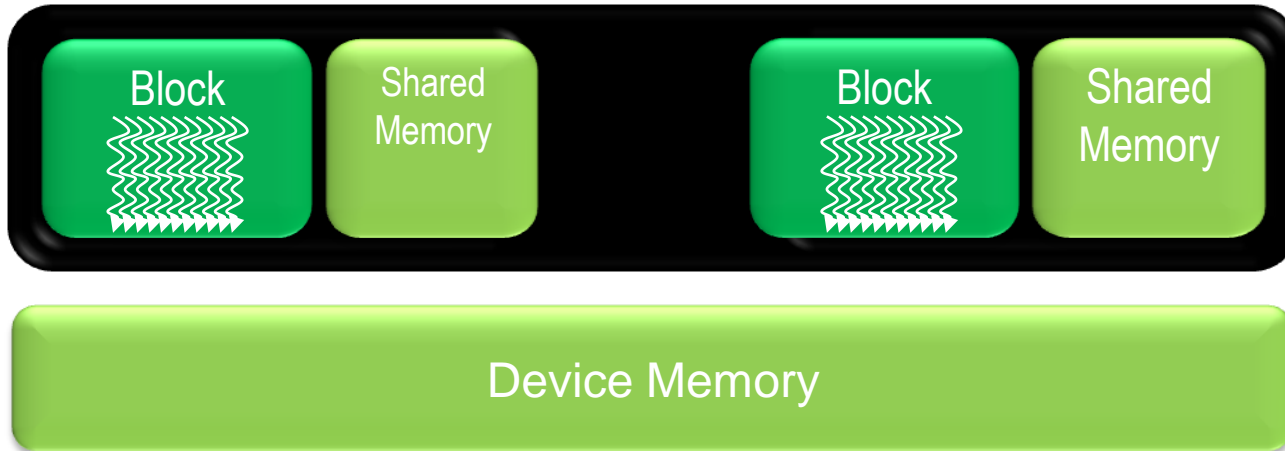
Grids, Thread Blocks and Threads

Grid



CUDA Model of Parallelism

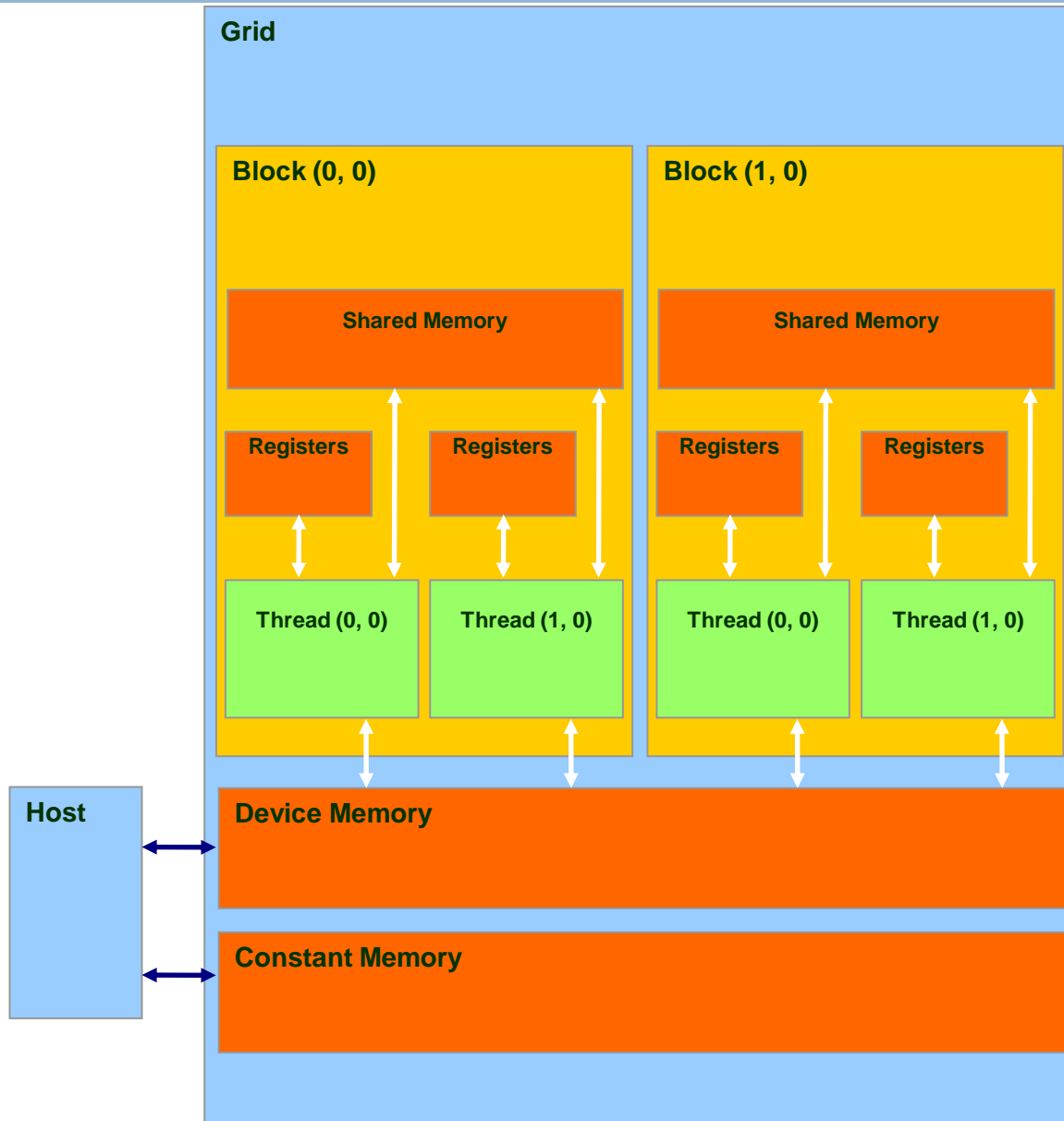
39



- CUDA virtualizes the physical hardware
 - ▣ Devices have
 - Different numbers of SMs
 - Different compute capabilities (Fermi = 2.0)
 - ▣ 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 should be independent

Hardware Memory Spaces in CUDA

40



Device Memory

41

- 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)
 - ▣ Use when all threads in a block read the same address
 - Serializes otherwise

GPU Memory Allocation / Release

43

- 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

44

- `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

45

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

CUDA language

46

- 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
```

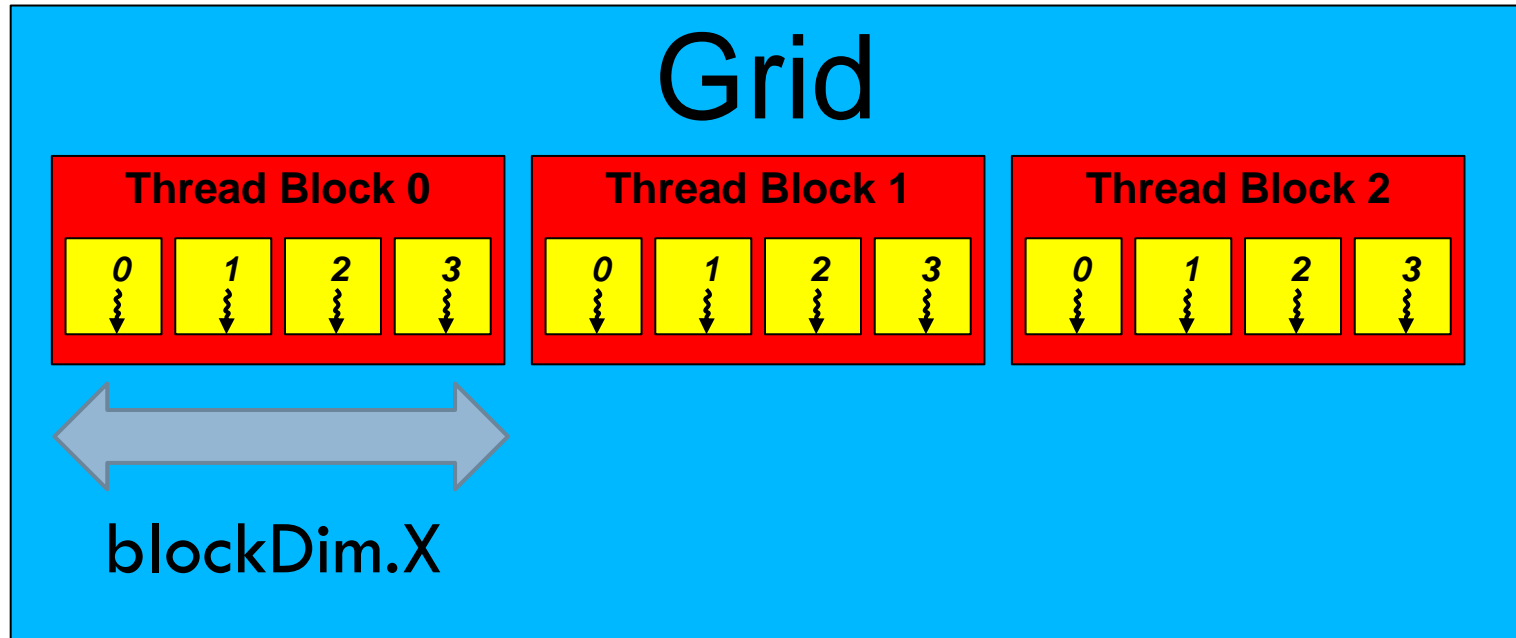
- Built-in variables and functions valid in device code:

```
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

47

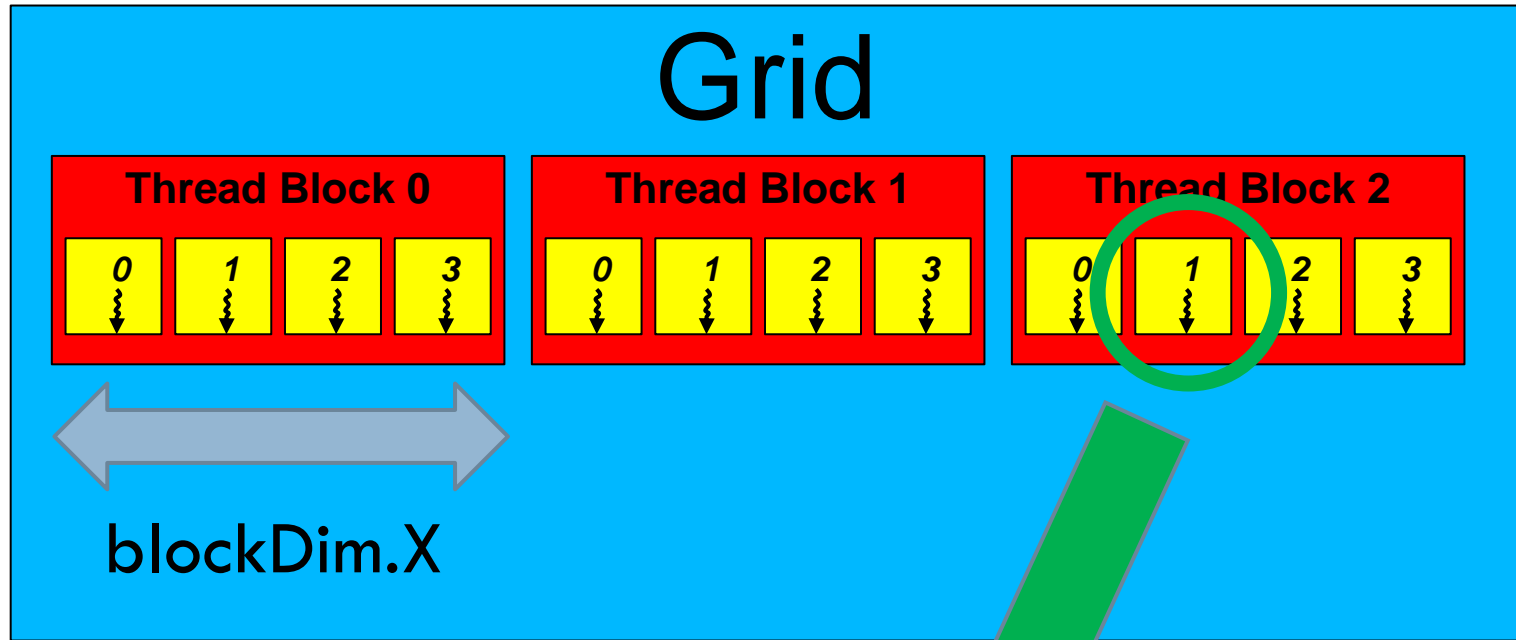


- “global” thread index:

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

Calculating the global thread index

48



- “global” thread index:

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

4 * 2 + 1 = 9

Vector add

49

```
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 addition GPU code

50

```
// 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 ...

    // 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 addition host code

```
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 addition host code

```
// transfer the data from the host to the device
cudaMemcpy(devA, hostA, size, cudaMemcpyHostToDevice);
cudaMemcpy(devB, 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);
}
```

53

CUDA: Scheduling, Synchronization and Atomics

Thread Scheduling

54

- 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

55

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

Global synchronization

56

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

Global synchronization

57

- 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

58

- 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

59

- Guarantee that only a single thread has access to a piece of memory during an operation
- No dropped data, but ordering is still arbitrary
- Different types of atomic instructions
- Atomic Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Can be done on device memory and shared memory
- Much more expensive than load + operation + store

Example: Histogram

60

```
// 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

61

```
// 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

62

```
// 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);  
}
```

Example: Work queue

63

```
// For algorithms where the amount of work per item  
// is highly non-uniform, it often makes sense to  
// continuously grab work from a queue.
```

__global__

```
void workq(int* work_q, int* q_counter,  
          int queue_max, int* output)  
{  
    int i = threadIdx.x + blockDim.x * blockIdx.x;  
    int q_index = atomicInc(q_counter, queue_max);  
    int result = do_work(work_q[q_index]);  
    output[q_index] = result;  
}
```

64

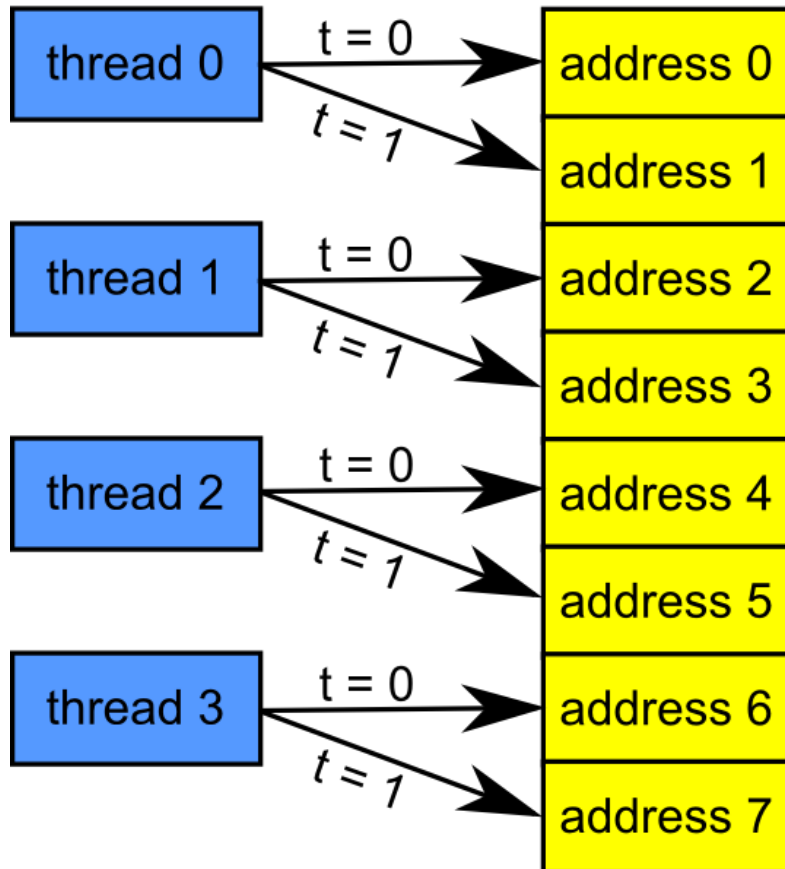
CUDA: optimizing your application

Coalescing

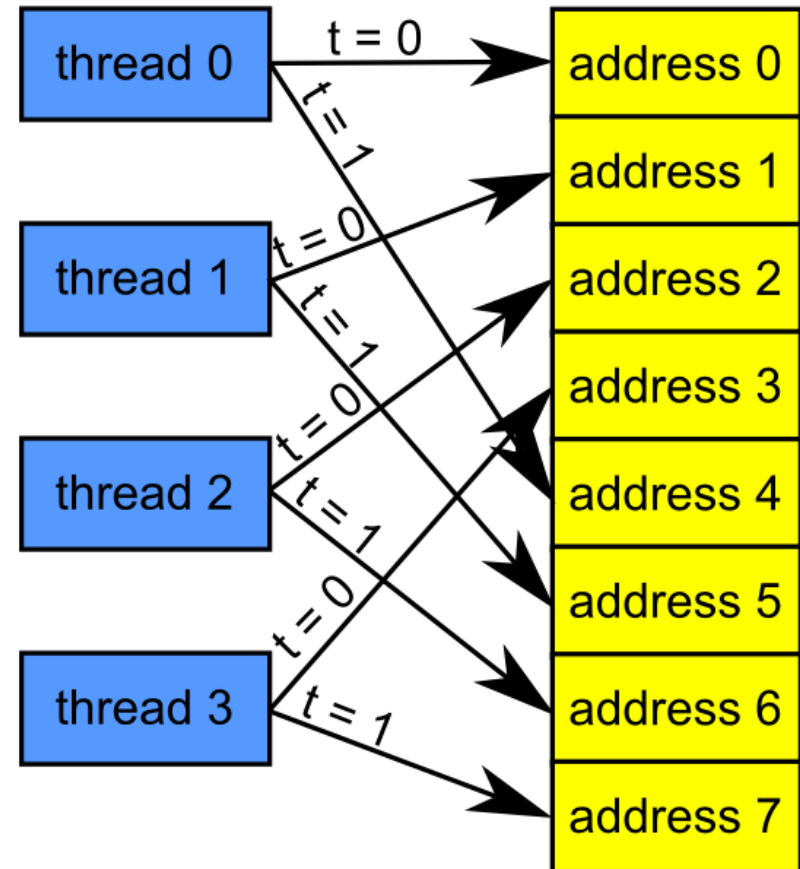
Coalescing

65

traditional multi-core
optimal memory access pattern



many-core GPU
optimal memory access pattern



Consider the stride of your accesses

66

```
__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)

67

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

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

Example: Structure of Arrays (SoA)

68

```
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

69

```
__global__ void bar(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

70

- 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 10 – 30!

71 CUDA: optimizing your application

Shared Memory

Using shared memory

72

```
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
```


Using shared memory

73

```
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
```

Using shared memory

74

```
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
```

How do we use device memory?

The next thread also reads input[i]

Using shared memory

75

```
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i];

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}
```

Using shared memory: coalescing

76

```
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

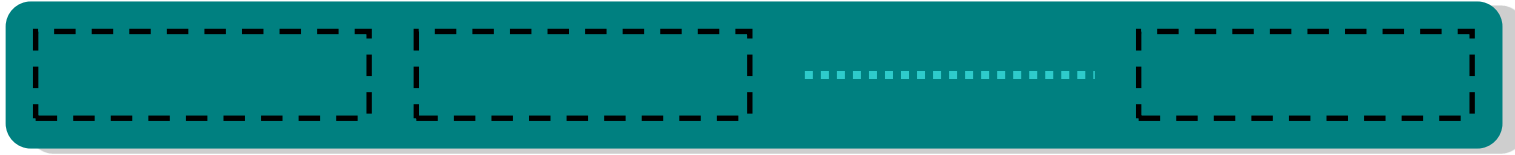
    __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i];    // COALESCED ACCESS!

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}
```

A Common Programming Strategy

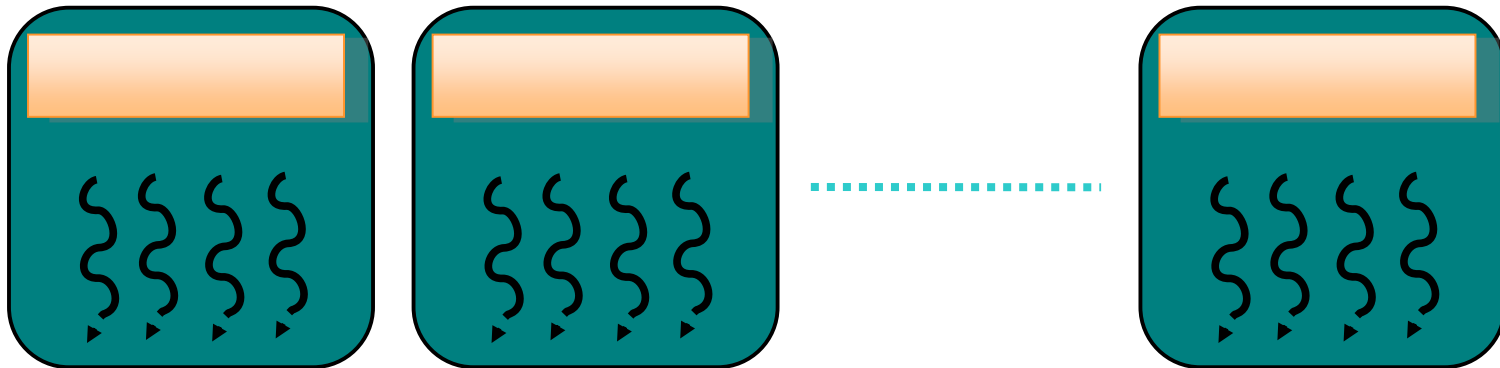
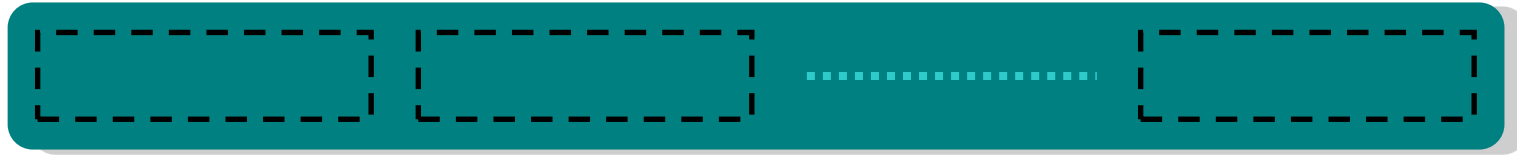
77



- Partition data into subsets that fit into shared memory

A Common Programming Strategy

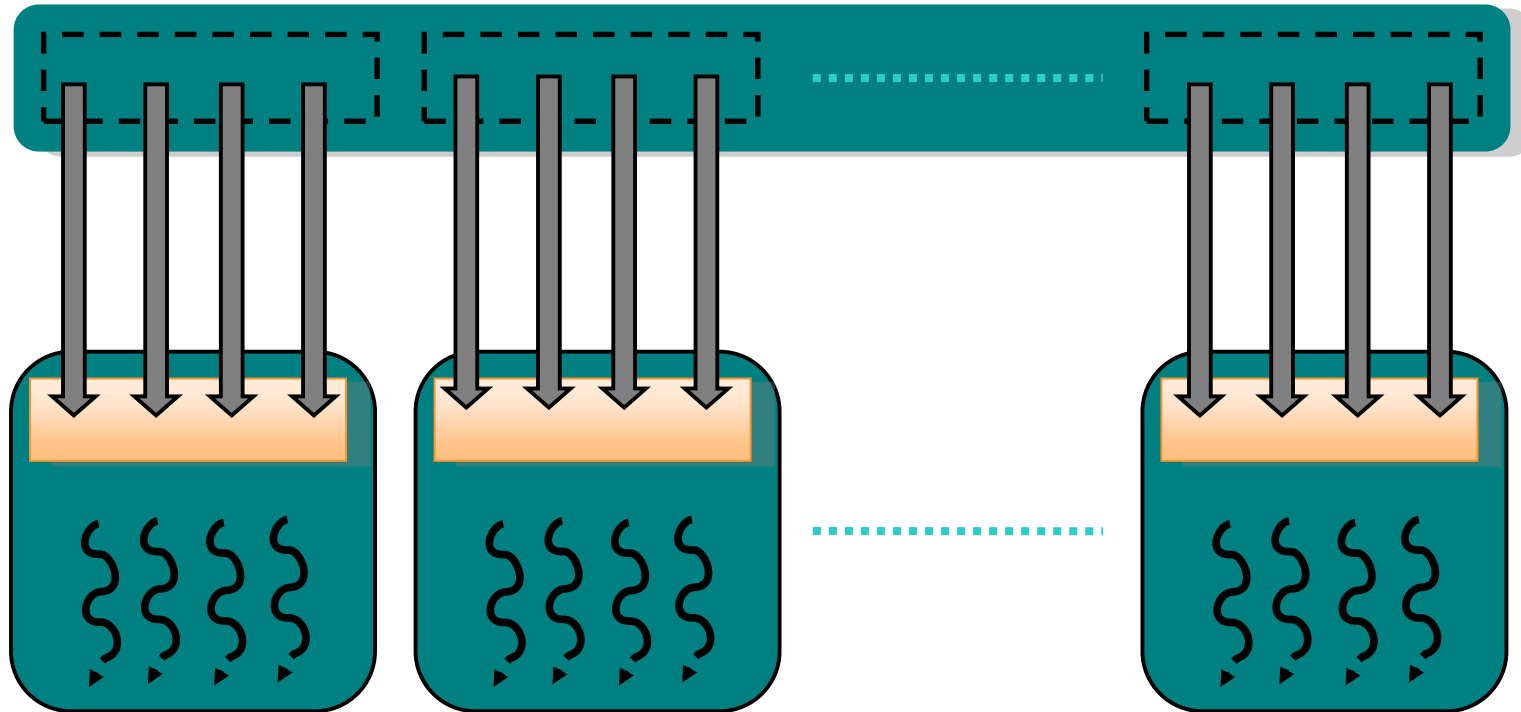
78



- Handle each data subset with one thread block

A Common Programming Strategy

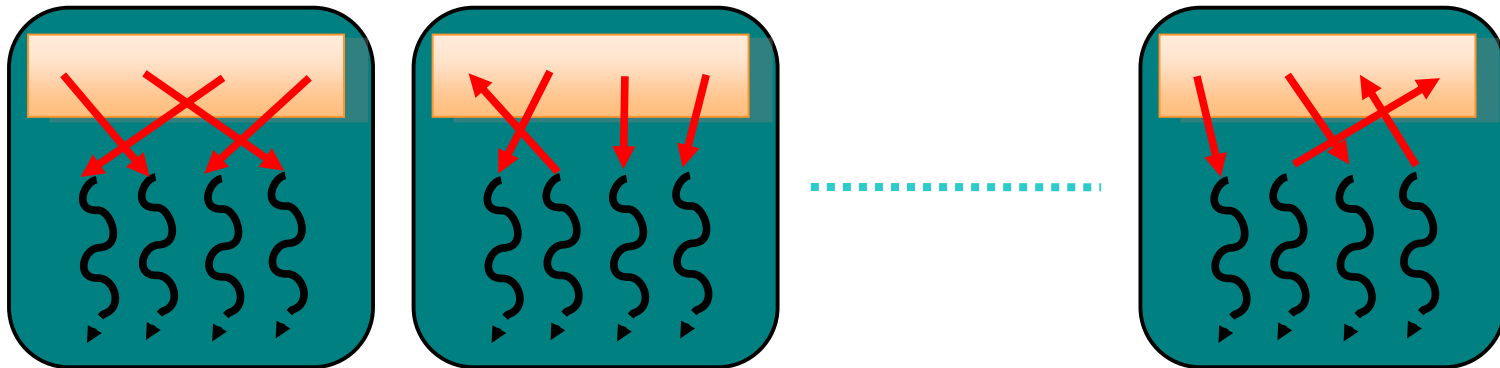
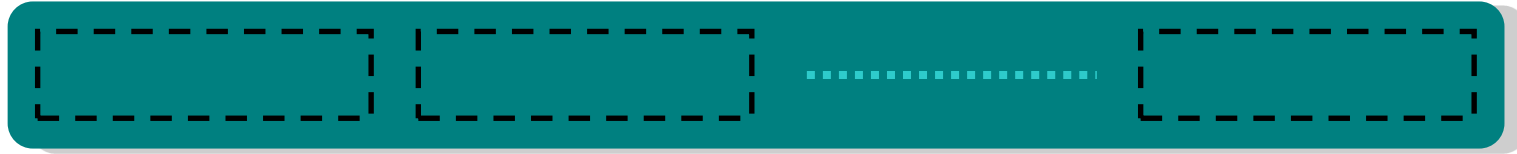
79



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

A Common Programming Strategy

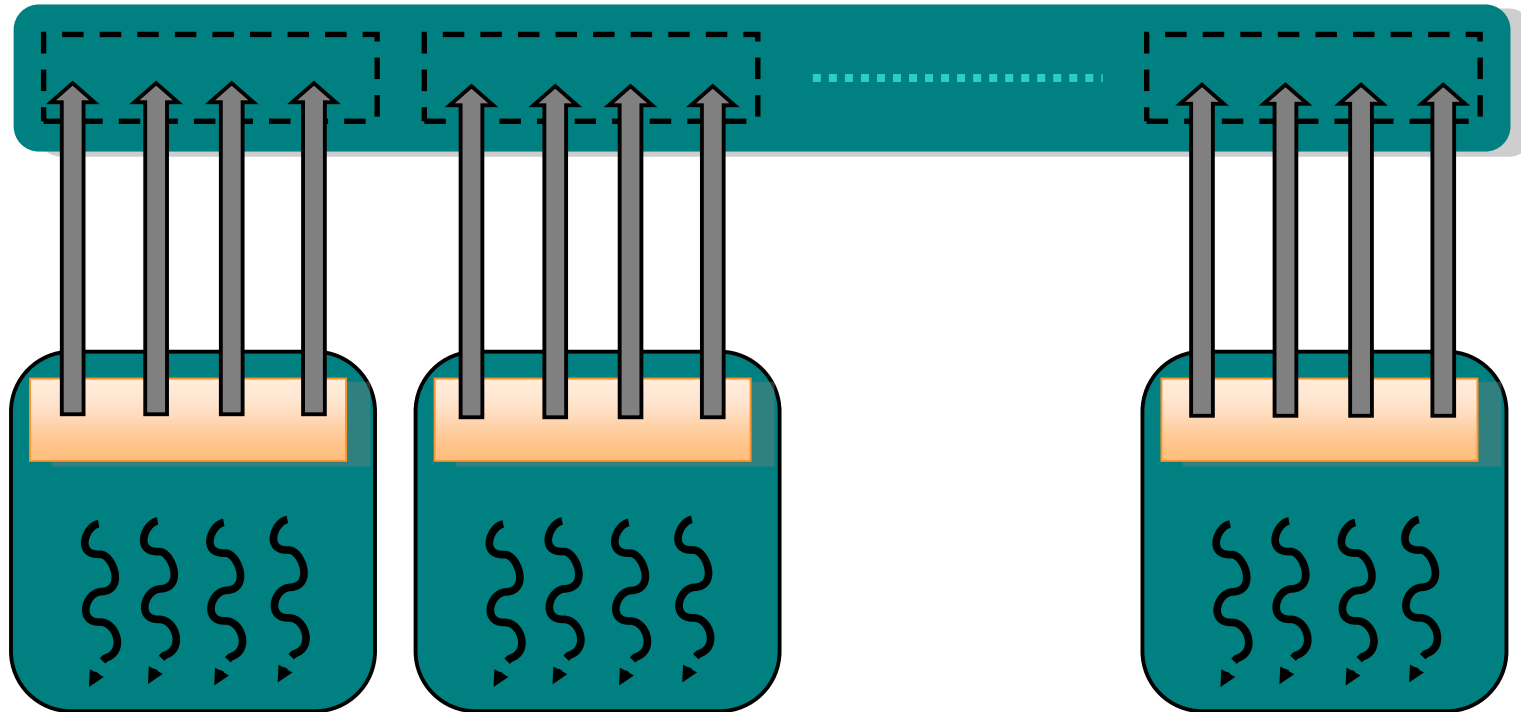
80



- Perform the computation on the subset from shared memory

A Common Programming Strategy

81



- Copy the result from shared memory back to device memory

82

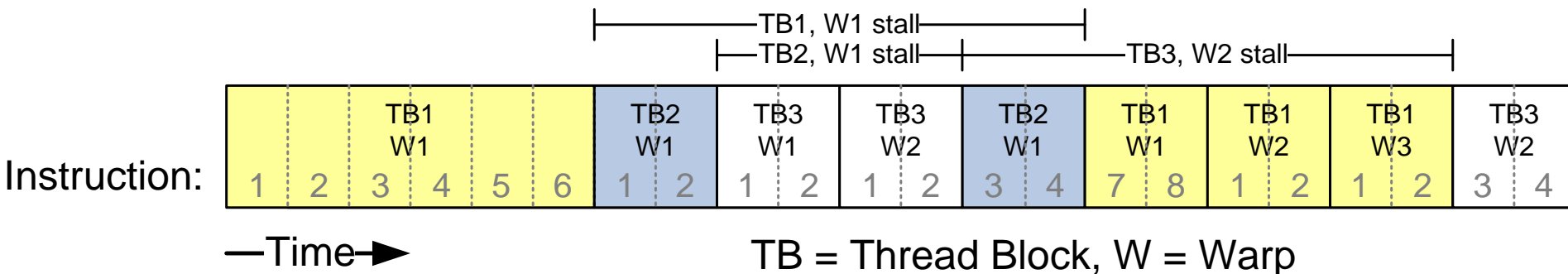
CUDA: optimizing your application

Optimizing Occupancy

Thread Scheduling

83

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

84

- 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

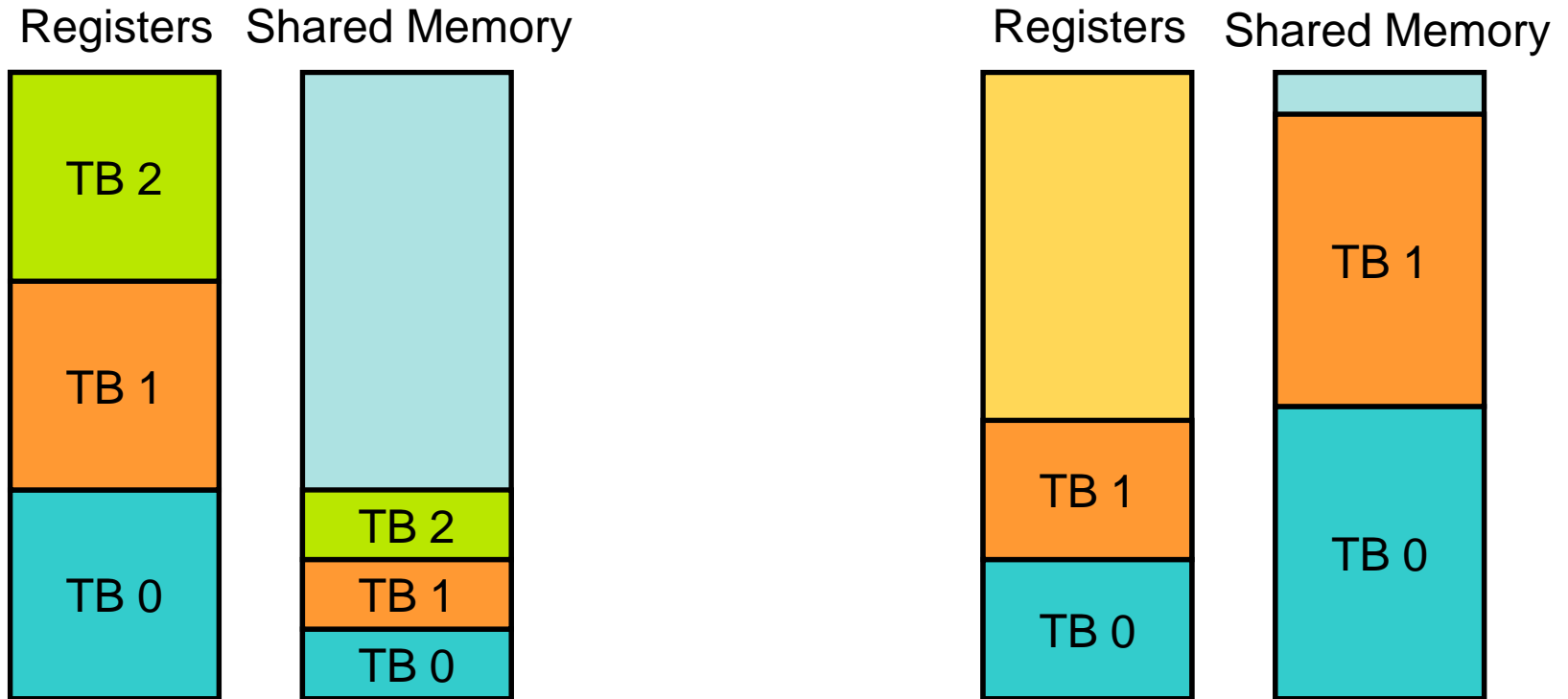
Occupancy

85

- What determines occupancy?
- Limited resources!
 - ▣ Register usage per thread
 - ▣ Shared memory per thread block

Resource Limits (1)

86



- 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)

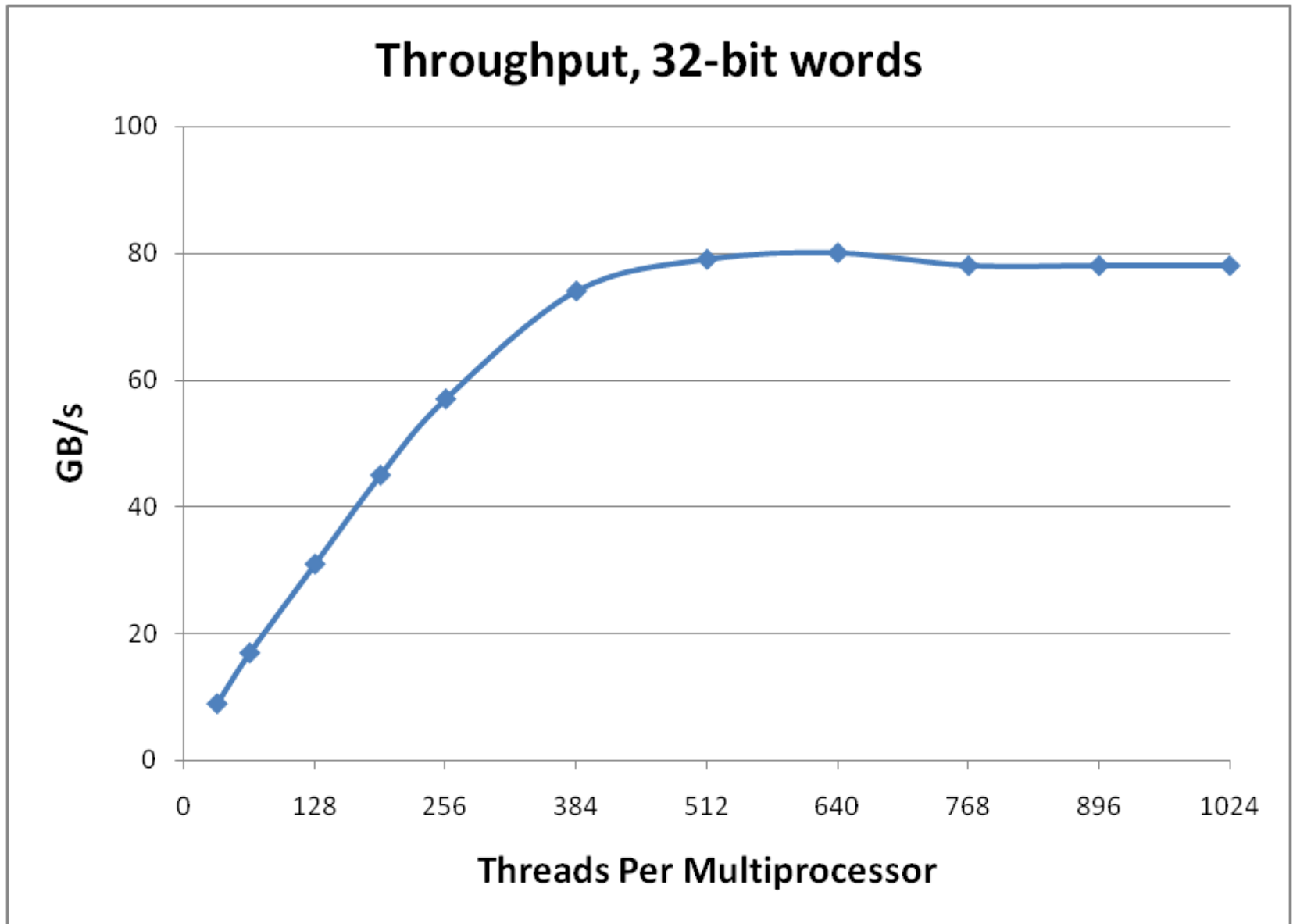
87

- 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

88



How do you know what you're using?

89

- Use “**nvcc -Xptxas -v**” to get register and shared memory usage
- Plug those numbers into CUDA Occupancy Calculator

CUDA_Occupancy_calculator.xlsm - Microsoft Excel

Home Insert Page Layout Formulas Data Review View

Paste Cut Copy Format Painter Clipboard Font Alignment Number Styles Cells Editing

General Conditional Formatting as Table Normal Bad Good Neutral Calculation Check Cell

Security Warning Macros have been disabled. Options...

MyRegCount 25

A B C D E F G H I J K L M N O P Q R

5
6 1.) Select Compute Capability (click): 1.3 (Help)

7
8 2.) Enter your resource usage:

9 Threads Per Block 128 (Help)

10 Registers Per Thread 25

11 Shared Memory Per Block (bytes) 640

12 (Don't edit anything below this line)

13 3.) GPU Occupancy Data is displayed here and in the graphs: (Help)

16 Active Threads per Multiprocessor 512

17 Active Warps per Multiprocessor 16

18 Active Thread Blocks per Multiprocessor 4

19 Occupancy of each Multiprocessor 50%

20

21 Physical Limits for GPU Compute Capability: 1.3

23 Threads per Warp 32

24 Warps per Multiprocessor 32

25 Threads per Multiprocessor 1024

26 Thread Blocks per Multiprocessor 8

27 Total # of 32-bit registers per Multiprocessor 16384

28 Register allocation unit size 512

29 Register allocation granularity block

30 Shared Memory per Multiprocessor (bytes) 16384

31 Shared Memory Allocation unit size 512

32 Warp allocation granularity (for register allocation) 2

33

34 Allocation Per Thread Block

35 Warps 4

36 Registers 3584

37 Shared Memory 1024

38 These data are used in computing the occupancy data in blue

39

40 Maximum Thread Blocks Per Multiprocessor Blocks

41 Limited by Max Warps / Blocks per Multiprocessor 8

42 Limited by Registers per Multiprocessor 4

43 Limited by Shared Memory per Multiprocessor 16

44 Thread Block Limit Per Multiprocessor highlighted RED

45

46 CUDA Occupancy Calculator

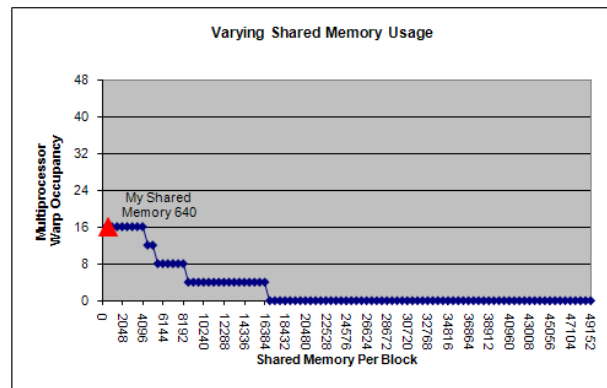
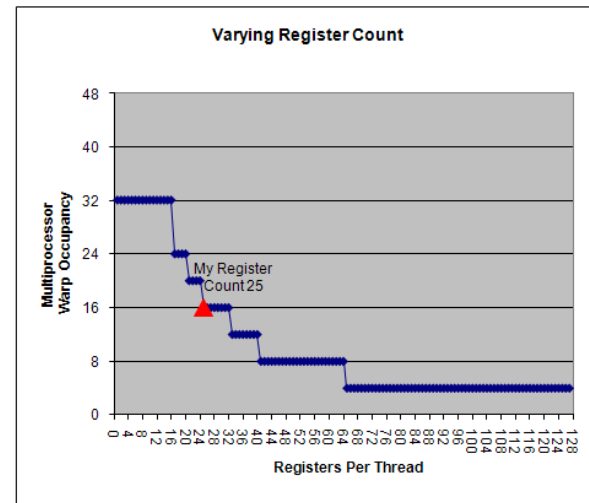
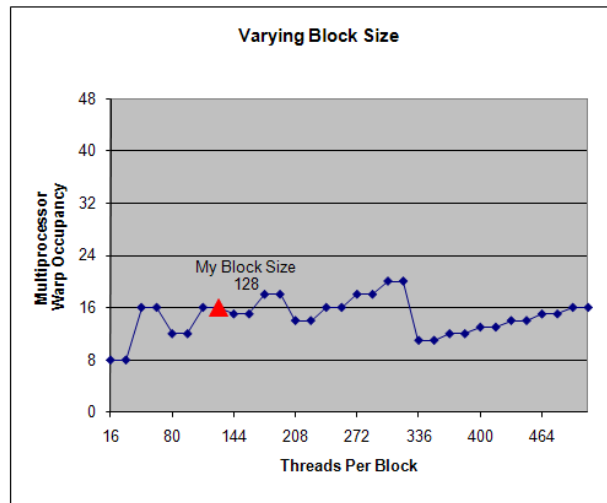
47 Version: 2.0

48 Copyright and License

49

50

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



Summary and Conclusions

Summary and conclusions

92

- 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 cores 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

93

- Many different types of many-core hardware and software
- Very different properties
 - ▣ Performance
 - ▣ Programmability
 - ▣ Portability
- It's all about the memory
- Many-cores are here to stay

94

Hands-on session

GPU hands-on session (1)

- Hostname: `fs0.das4.cs.vu.nl`
- Add to `.bashrc`:
 - ▣ `module load cuda42/toolkit prun`
- Try `nvcc --version`
- Everything you need is in: `/home/rob/multimoore`
 - ▣ Slides, handouts
 - ▣ Cuda documentation
 - ▣ Example programs
 - ▣ Template for the assignment, input images
- Use `display` to look at images

GPU hands-on session (2)

- Run with:
- `prun -v -np 1 -native '-l gpu=GTX480' EXECUTABLE`
- Try, for instance:
- `$CUDA_SDK/C/bin/linux/release/deviceQuery`
- Check queue status:
- `preserve -long-list`

GPU hands-on session (3)

- Vector add:
 - `/home/rob/multimoore/day3/vector-add`
- Assignment:
 - `/home/rob/multimoore/day3/seq-filters`
- CUDA template is already there
 - `/home/rob/multimoore/day3/cuda-filters`