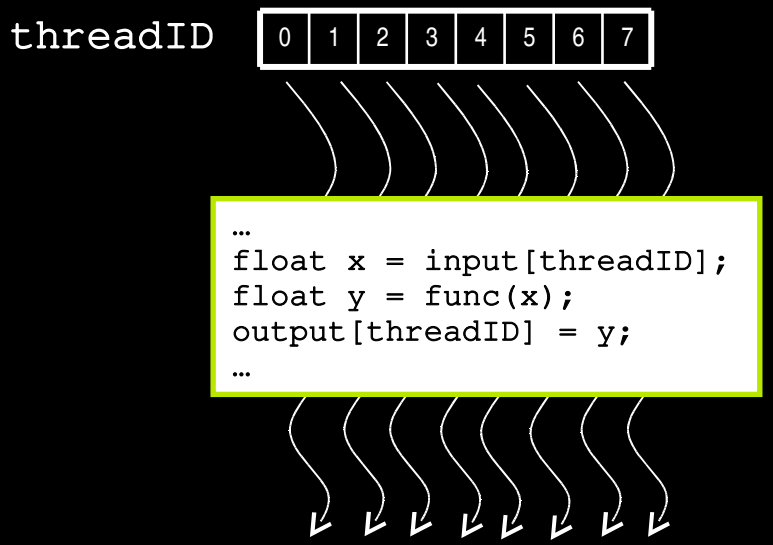
CUDA Programming Model: Massively Multithreaded Processor

- The GPU (aka device) is a highly multithreaded coprocessor to the CPU (aka host)
  - Has its own DRAM (device memory)
  - Executes many threads in parallel on several multiprocessor cores

- CUDA threads are extremely lightweight
  - Very little creation overhead
  - Context switching is essentially free
  - GPU needs 1000's of threads for full utilization

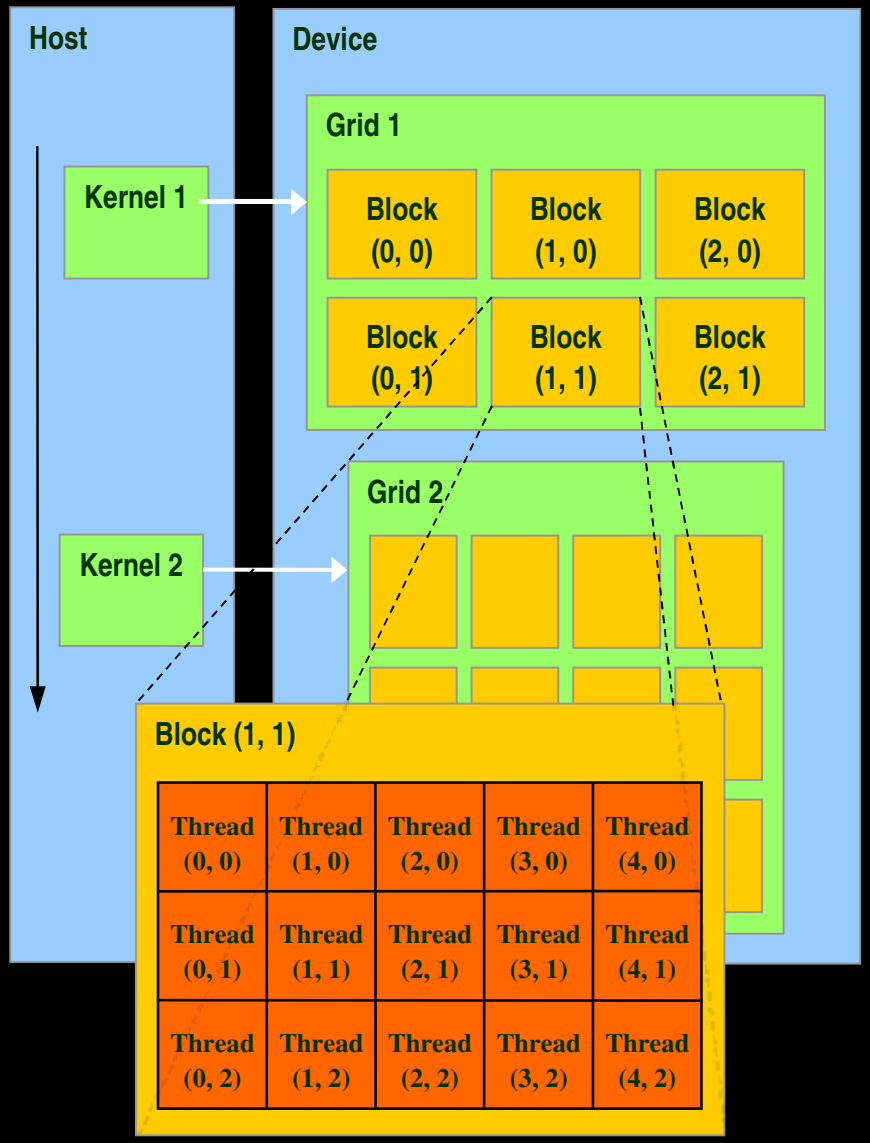
CUDA Programming Model: Kernel

- Parallel portions of an application are executed on the device as kernels
- One kernel is executed at a time on the device
- Many threads execute each kernel
  - Each thread executes the same code…
  - … on different data based on its threadID



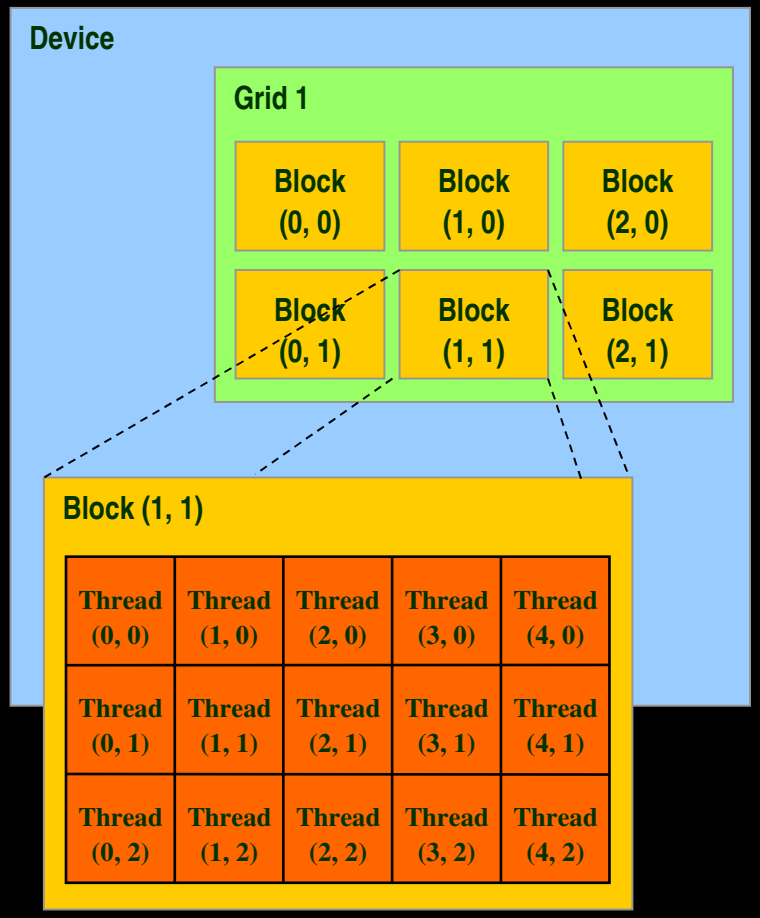
CUDA Programming Model: Grid of Thread Blocks

- A kernel is executed as a 1D or 2D grid of 1D, 2D,or 3D thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data via shared memory
  - Synchronizing their execution
    - For hazard-free shared memory accesses
- Threads from different blocks cannot synchronize their execution



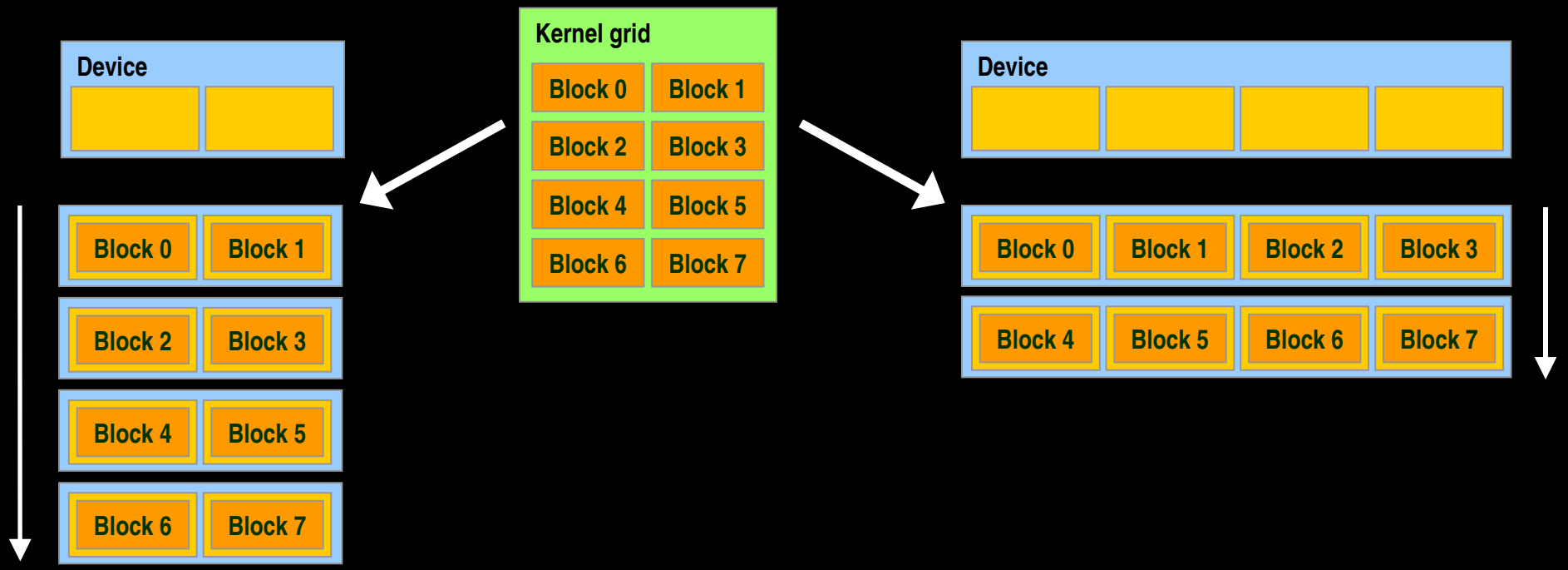
CUDA Programming Model: Thread and Block IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
- Block ID: 1D or 2D
- Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes



Transparent Scalability

- Since thread blocks cannot synchronize, the **hardware** is free to schedule the execution of a thread block on any multiprocessor at any time
  - A kernel scales across any number of parallel multiprocessors

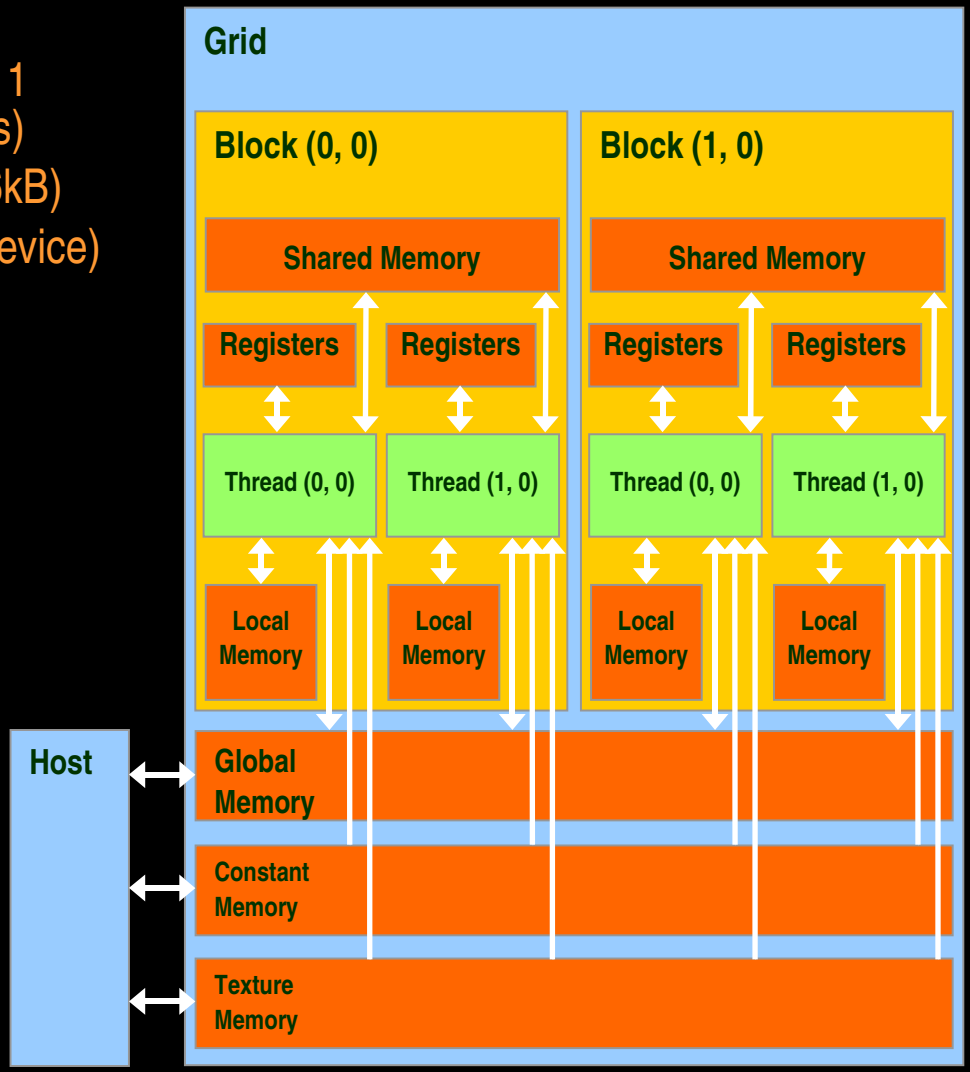


- Kernel launch serves as a synchronization point for blocks
  - Kernel launch has negligible HW overhead, low SW overhead

Programming Model: Memory Spaces

- Each thread can:
  - Read/write per-thread registers (8192 on 1 multiprocessor, shared among all threads)
  - Read/write per-block shared memory (16kB)
  - Read/write per-grid global memory (on device)
  - Most important, commonly used
- Each thread can also:
  - Read/write per-thread local memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
  - Used for convenience/performance
    - More details later

- The host can read/write global, constant, and texture memory (stored in DRAM)



CUDA = C with Language Extensions

- Function qualifiers for functions that execute on the device:

```
__global__ void MyKernel() { }
__device__ float MyDeviceFunc() { }
```

- Variable qualifiers for variables that resides on the device:

```
__shared__   float MySharedArray[32];
__constant__ float MyConstantArray[32];
```

- Execution configuration to launch kernels:

```
dim3 dimGrid(100, 50);  // 5000 thread blocks
dim3 dimBlock(4, 8, 8); // 256 threads per block
MyKernel <<< dimGrid, dimBlock >>> (...); // Launch kernel
```

- Built-in variables and functions accessible in __global__ and __device__ functions:

```
dim3 gridDim;   // Grid dimension
dim3 blockDim;  // Block dimension
dim3 blockIdx;  // Block index within the grid
dim3 threadIdx; // Thread index within the block
void __syncthreads(); // Thread synchronization
```

CUDA = C with Runtime Extensions

- Device management:
  `cudaGetDeviceCount()`, `cudaGetDeviceProperties()`
- Device memory management:
  `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- Texture management:
  `cudaBindTexture()`, `cudaBindTextureToArray()`
- Graphics interoperability:
  `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`

Example: Increment Array Elements

CPU program

```
void increment_cpu(float *a, float b, int N)
{
    for (int idx = 0; idx<N; idx++)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    increment_cpu(a, b, N);
}
```

CUDA program

```
__global__ void increment_gpu(float *a, float b, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + b;
}


void main()
{
  .....
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize)  );
    increment_gpu<<<dimGrid, dimBlock>>>(a, b, N);
}
```

Example: Increment Array Elements

Increment N-element vector a by scalar b

Let's assume N=16, blockDim=4 -> 4 blocks

blockIdx.x=0
blockDim.x=4
threadIdx.x=0,1,2,3
idx=0,1,2,3

blockIdx.x=1
blockDim.x=4
threadIdx.x=0,1,2,3
idx=4,5,6,7

blockIdx.x=2
blockDim.x=4
threadIdx.x=0,1,2,3
idx=8,9,10,11

blockIdx.x=3
blockDim.x=4
threadIdx.x=0,1,2,3
idx=12,13,14,15

int idx = blockDim.x * blockId.x + threadIdx.x;
will map from local index threadIdx to global index

Common Pattern!

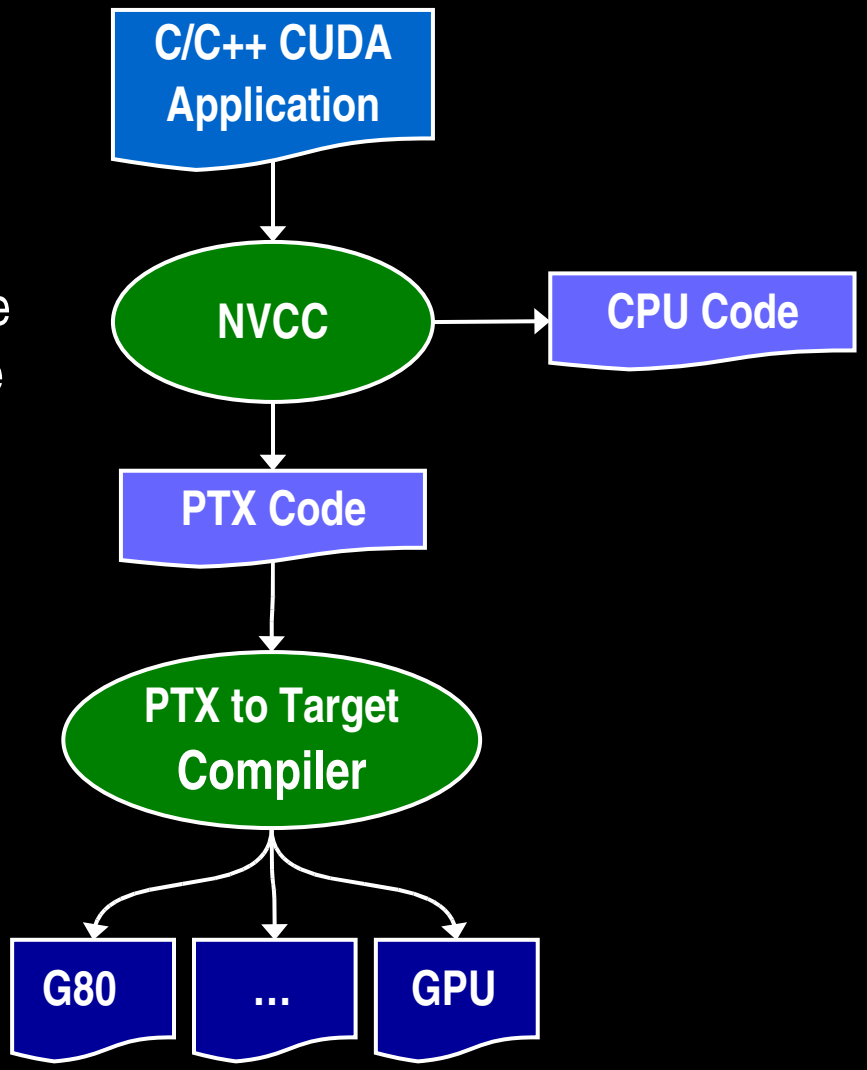
NB: blockDim should be >= 32 in real code, this is just an example

Example: Host Code

```
// allocate host memory
unsigned int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);


// execute the kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);


// copy data from device back to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// free device memory
cudaFree(d_A);
```

Compilation

- Any source file containing CUDA language extensions must be compiled with NVCC
  - NVCC separates code running on the host from code running on the device
- Two-stage compilation:
  - Virtual ISA
    - Parallel Thread eXecution
  - Device-specific binary object



Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - **No need of any device and CUDA driver**
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

Reduction Example

- Reduce N values to a single one:
  - Sum(v_0, v_1, … , v_{N-2}, v_{N-1})
  - Min(v_0, v_1, … , v_{N-2}, v_{N-1})
  - Max(v_0, v_1, … , v_{N-2}, v_{N-1})
- Common primitive in parallel programming
- Easy to implement in CUDA
  - Less so to get it right
- Divided into 4 exercises throughout this morning
  - Each exercise illustrates one particular optimization strategy

Reduction Exercise

- At the end of each exercise, the result of the reduction computed on the device is checked for correctness
  - “Test PASSED” or “Test FAILED” is printed out to the console
- The goal is to replace the “`TODO`“ words in the code by the right piece of code to get “test PASSED”

Reduction Exercise 1

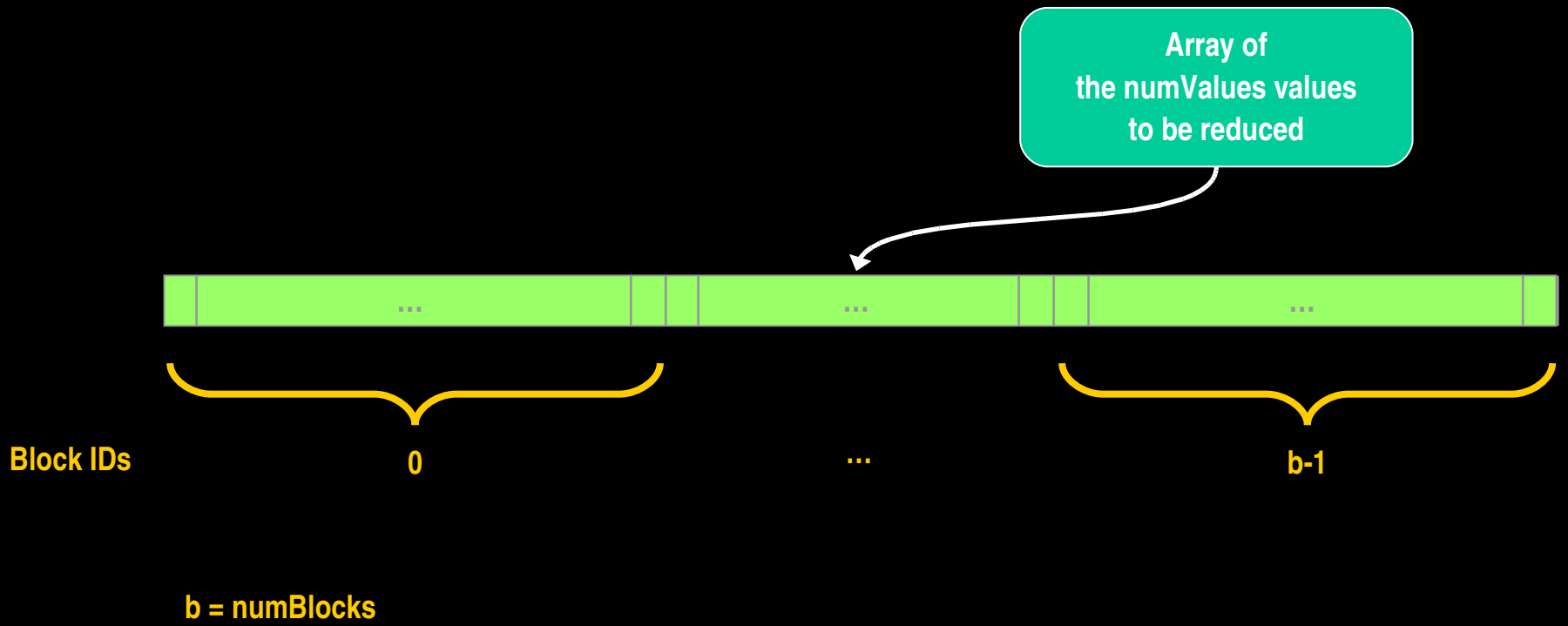
- Look up directory `reduce1`
- Code walkthrough:
  - `main.cpp`
    - Allocate host and device memory
    - Call `reduce()` defined in `reduce1.cu`
    - Profile and verify result
  - `reduce1.cu`
    - CUDA code to be compiled with `nvcc`
    - Contains `TODO`s
- Default : device emulation compilation configurations: `reduce1_e` (runs slowly)

Reduce 1: Multi-Pass Reduction

- Blocks cannot synchronize so `reduce_kernel` is called multiple times:
  - First call reduces from `numValues` to `numThreads`
  - Each subsequent call reduces by half the number of threads and thus the size of the output array (down to an array of 1 element). Each thread computes at most the sum of 2 elements
  - Need ping pong between input and output buffers (`d_Result[2]`): input for one iteration is the ouput of the previous, avoid conflicts

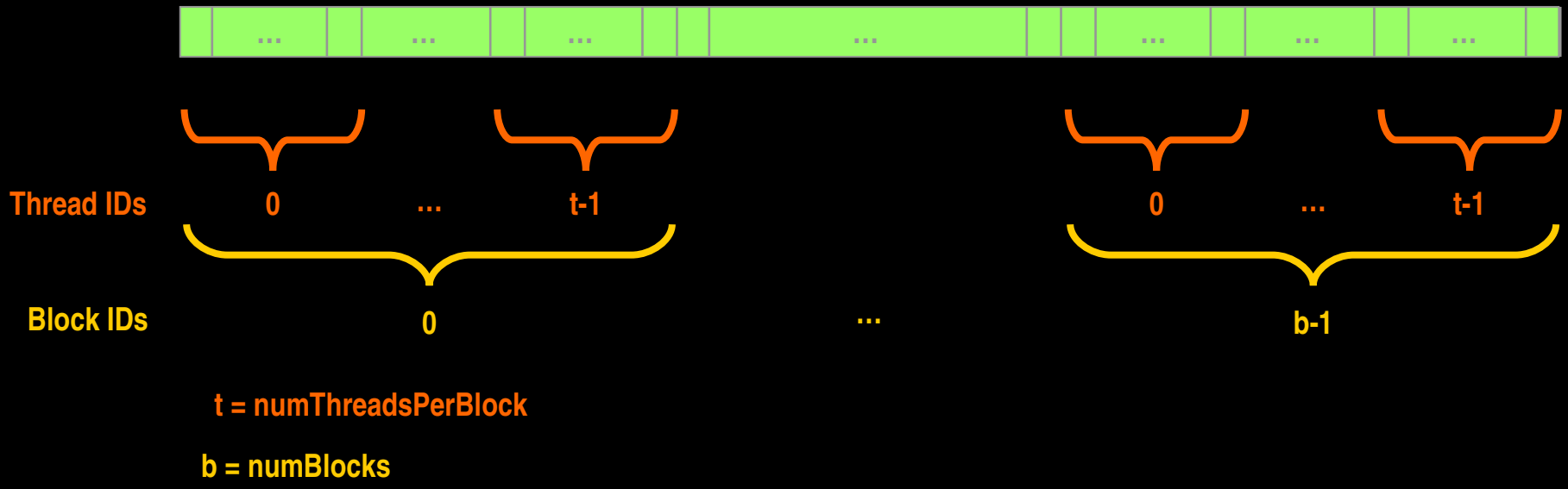
Reduce 1: Blocking the Data

- Split the work among the N multiprocessors by launching `numBlocks`=N thread blocks



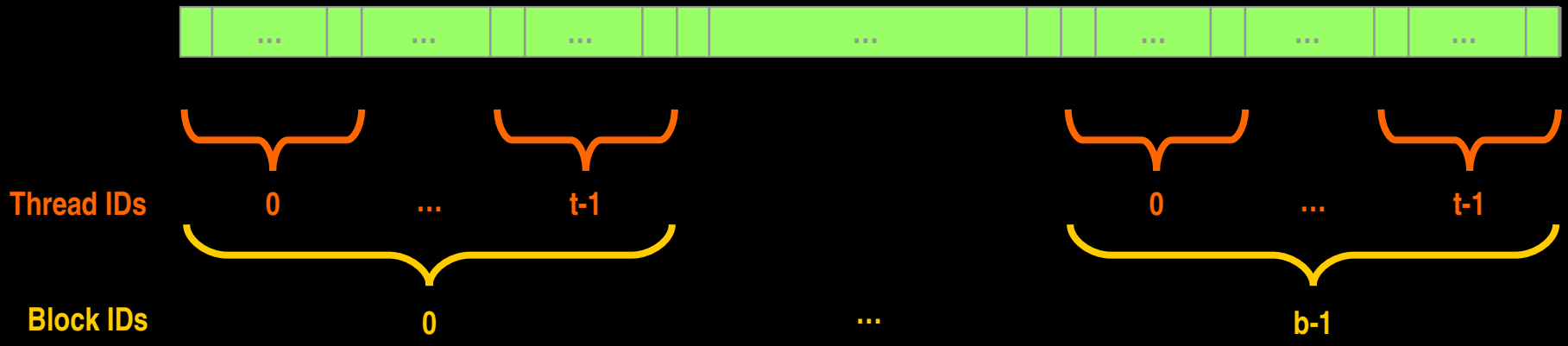
Reduce 1: Blocking the Data

- Within a block, split the work among the threads (max 512 tpb)



Reduce 1: hands-on !

- Replace the `TODO`s in `reduce1.cu` to get “test PASSED”



Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed
- cudaMemcpy is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed
- CudaThreadSynchronize() (cuda_runtime_api.h)
  - blocks until all previous CUDA calls complete

Thread Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
  - Generates barrier synchronization instruction
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Allowed in conditional code only if the conditional evaluates uniformly across the entire thread block

Device Management

- CPU can query and select GPU devices
  - cudaGetDeviceCount(int *count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *current_device)
  - cudaGetDeviceProperties(cudaDeviceProp* prop,
    int device)
  - cudaChooseDevice(int *device, cudaDeviceProp* prop)
- Multi-GPU setup:
  - device 0 is used by default
  - one CPU thread can control only one GPU
    - multiple CPU threads can control the same GPU
      - calls are serialized by the driver

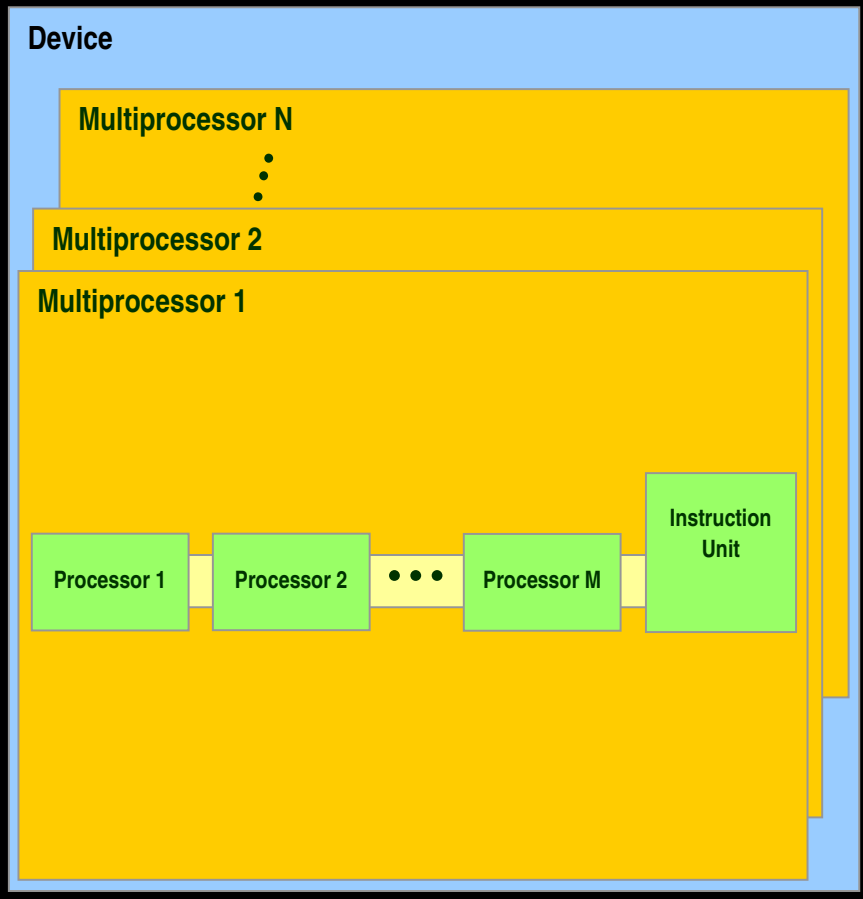
Multiple CPU Threads and CUDA = cuda not thread-safe (yet ?)

- CUDA resources allocated by a CPU thread can be consumed only by CUDA calls from the same CPU thread
- Violation Example:
  - CPU thread 2 allocates GPU memory, stores address in *p*
  - thread 3 issues a CUDA call that accesses memory via *p*

Optimization techniques

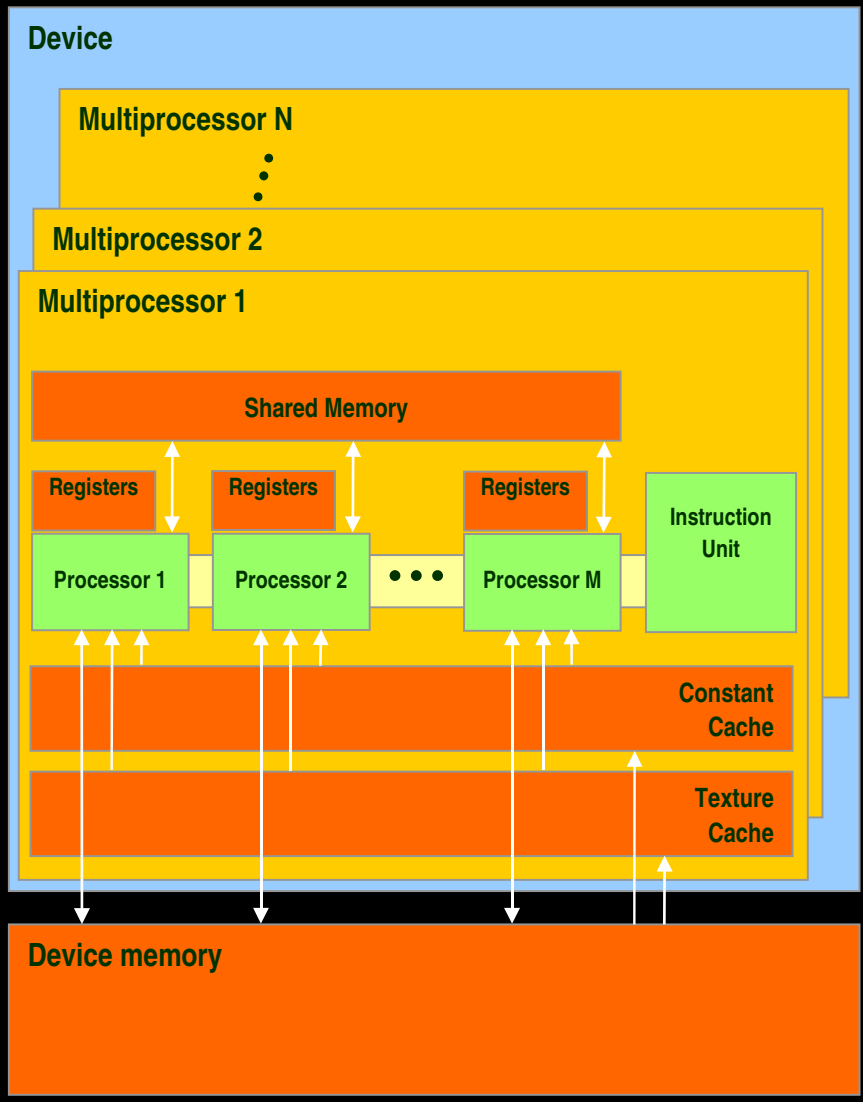
Hardware Implementation: A Set of SIMD Multiprocessors

- Each multiprocessor is a set of 32-bit processors with a Single Instruction Multiple Data architecture
  - 16 multiprocessors on G80
  - 8 processors per multiprocessor
- At each clock cycle, a multiprocessor executes the same instruction on a group of threads called a warp
  - The number of threads in a warp is the warp size (= 32 threads on G80)
  - A half-warp is the first or second half of a warp



Hardware Implementation: Memory Architecture

- The global, constant, and texture spaces are regions of device memory
- Each multiprocessor has:
  - A set of 32-bit registers per processor (8192 per multiprocessor)
  - On-chip shared memory (16K on G80)
    - Where the shared memory space resides
  - A read-only constant cache
    - To speed up access to the constant memory space
  - A read-only texture cache
    - To speed up access to the texture memory space

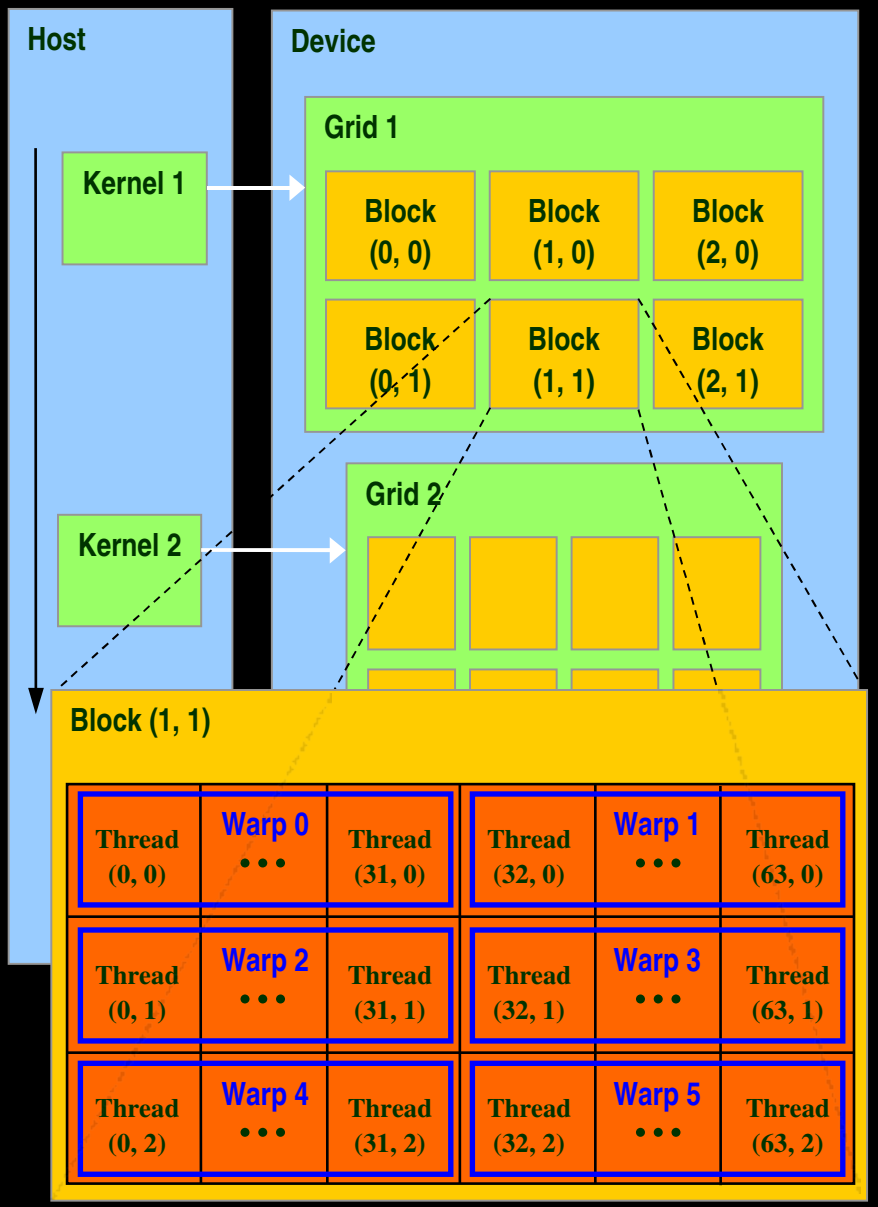


Hardware Implementation: Execution Model

- Each multiprocessor processes batches of blocks one batch after the other
  - Active blocks = the blocks processed by one multiprocessor in one batch
  - Active threads = all the threads from the active blocks
- The multiprocessor's registers and shared memory are split among the active threads
- Therefore, for a given kernel, the number of active blocks depends on:
  - The number of registers the kernel compiles to
  - How much shared memory the kernel requires
- If there cannot be at least one active block, the kernel fails to launch

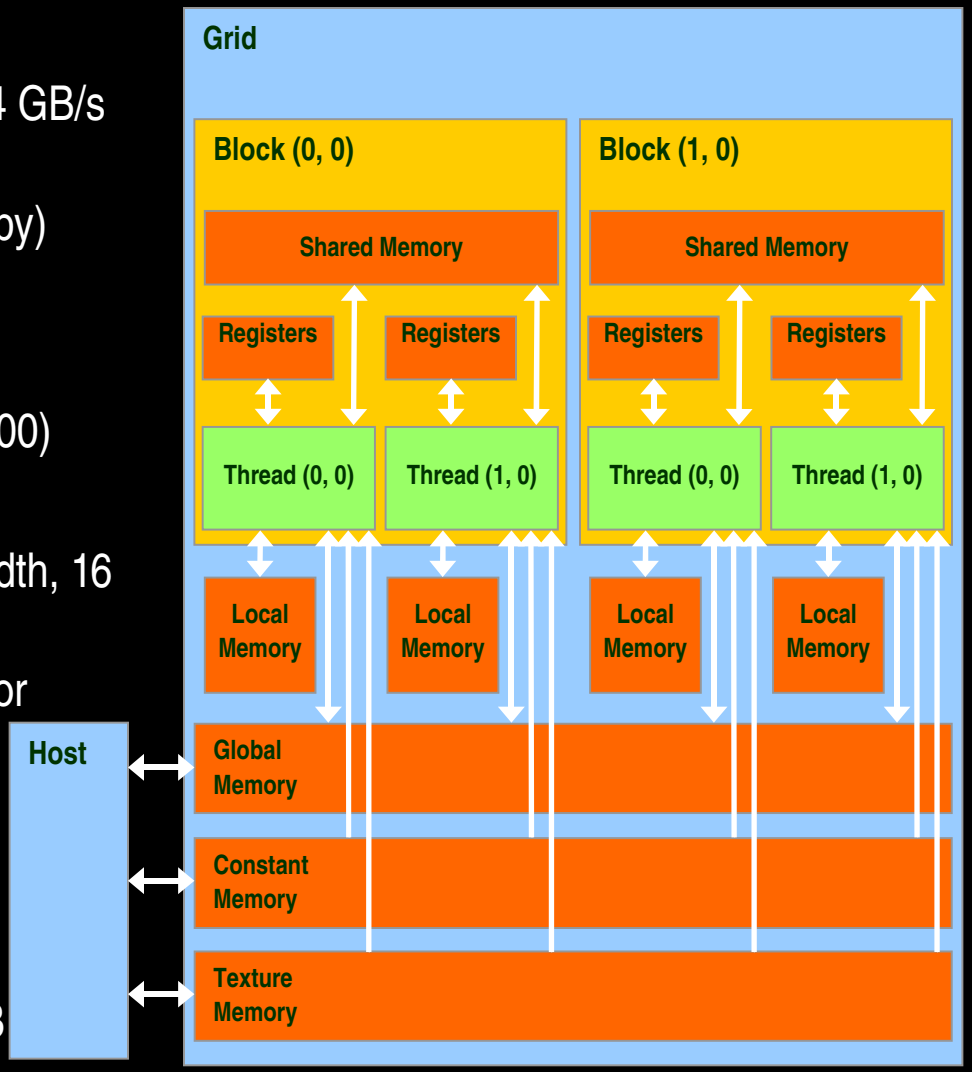
Hardware Implementation: Execution Model

- Each active block is split into warps in a well-defined way
- Warps are time-sliced
- In other words:
  - Threads within a warp are executed *physically* in parallel
  - Warps and blocks are executed *logically* in parallel



Memory Latency and Bandwidth

- Host memory
  - Device ↔ host memory bandwidth is 4 GB/s peak (PCI-express x16)
  - Test with SDK's bandwidthTest (memcpy)
- Global/local device memory
  - High latency, **not cached**
  - 80 GB/s peak, 1.5 GB (Quadro FX 5600)
- Shared memory
  - On-chip, low latency, very high bandwidth, 16 KB
  - Like a user-managed per-multiprocessor cache
- Texture memory
  - Read-only, high latency, cached
- Constant memory
  - Read-only, low latency, cached, 64 KB



Performance Optimization

- Expose as much parallelism as possible
- Optimize memory usage for maximum bandwidth
- Maximize occupancy to hide latency (while fetching memory)
- Optimize instruction usage for maximum throughput (choose instructions using few cycles)

Expose Parallelism: GPU Thread Parallelism

- Structure algorithm to maximize independent parallelism
- If threads of same block need to communicate, use shared memory and __syncthreads()
- If threads of different blocks need to communicate, use global memory and split computation into multiple kernels
  - No synchronization mechanism between blocks
- High parallelism is especially important to hide memory latency by overlapping memory accesses with computation

Optimize Memory Usage: Basic Strategies

- Processing data is cheaper than moving it around
  - Especially for GPUs as they devote many more transistors to ALUs than memory
- And will be increasingly so
  - The less memory bound a kernel is, the better it will scale with future GPUs
- So you want to:
  - Maximize use of low-latency, high-bandwidth memory
  - Optimize memory access patterns to maximize bandwidth
  - Leverage parallelism to hide memory latency by overlapping memory accesses with computation as much as possible
    - Kernels with high arithmetic intensity (ratio of math to memory transactions)
  - Sometimes recompute data rather than cache it

Minimize CPU ↔ GPU Data Transfers

- CPU ↔ GPU memory bandwidth much lower than GPU memory bandwidth
  - Use page-locked host memory (`cudaMallocHost()`) for maximum CPU ↔ GPU bandwidth
    - 3.2 GB/s common on PCI-e x16
    - ~4 GB/s measured on nForce 680i motherboards (8GB/s for PCI-e 2.0)
    - Be cautious however since allocating too much page-locked memory can reduce overall system performance
- Minimize CPU ↔ GPU data transfers by moving more code from CPU to GPU
  - Even if that means running kernels with low parallelism computations
  - Intermediate data structures can be allocated, operated on, and deallocated without ever copying them to CPU memory
- Group data transfers
  - One large transfer much better than many small ones

Optimize Memory Access Patterns

- Effective bandwidth can vary by an order of magnitude depending on access pattern
- Optimize access patterns to get:
  - *Coalesced* **global memory** accesses
  - Shared memory accesses with *no or few bank conflicts*
  - *Cache-efficient* texture memory accesses
  - *Same-address* constant memory accesses

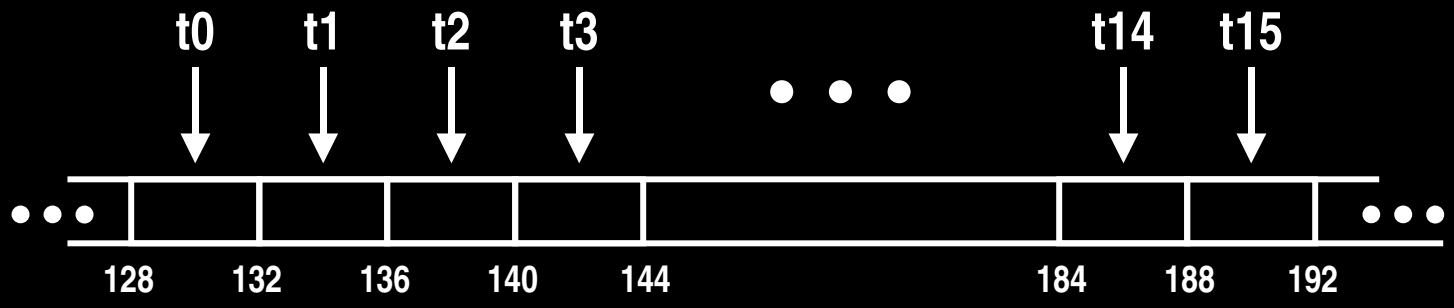
Global Memory Reads/Writes

- Global memory is not cached on G8x
- Highest latency instructions: 400-600 clock cycles
- Likely to be performance bottleneck
- Optimizations can greatly increase performance

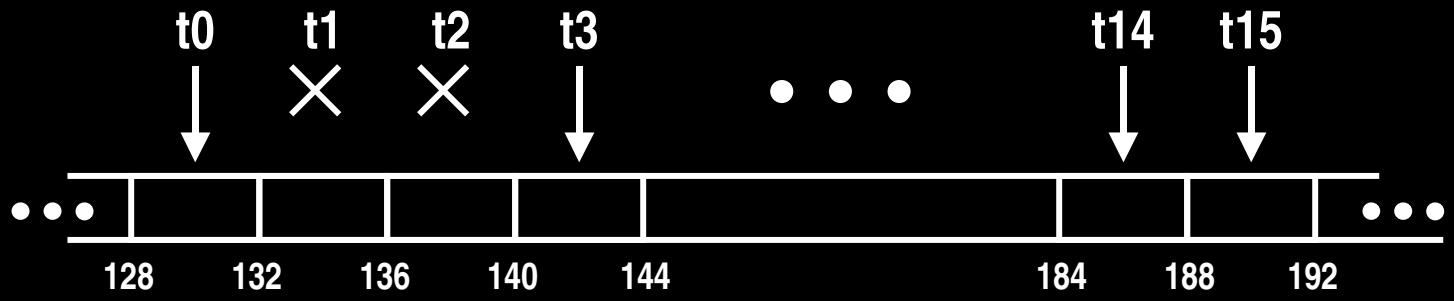
Coalesced Global Memory Accesses

- The simultaneous global memory accesses by each thread of a half-warp (16 threads on G80) during the execution of a single read or write instruction will be *coalesced* into a single access if:
  - The size of the memory element accessed by each thread is either 4, 8, or 16 bytes
  - The elements form a contiguous block of memory
  - The N^{th} element is accessed by the N^{th} thread in the half-warp
  - The address of the first element is aligned to 16 times the element's size
- Coalescing happens even if some threads do not access memory (divergent warp)

Coalesced Global Memory Accesses



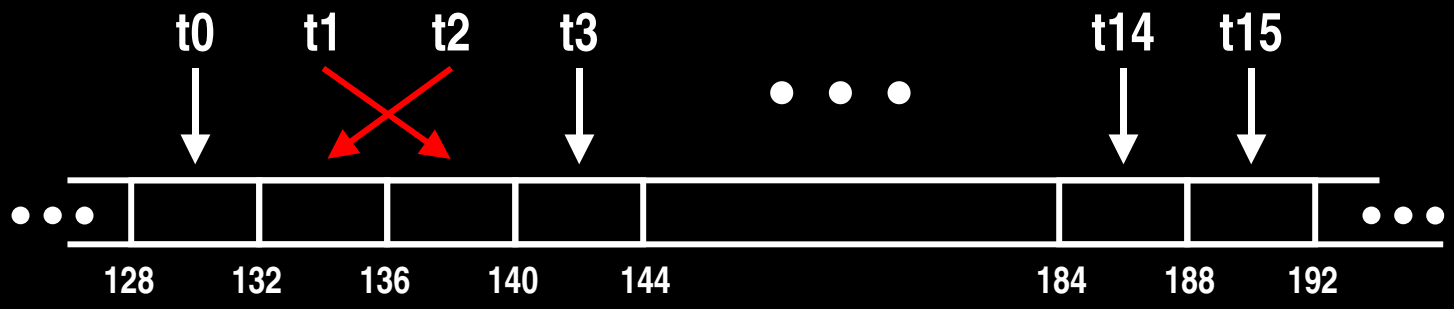
Coalesced `float` memory access



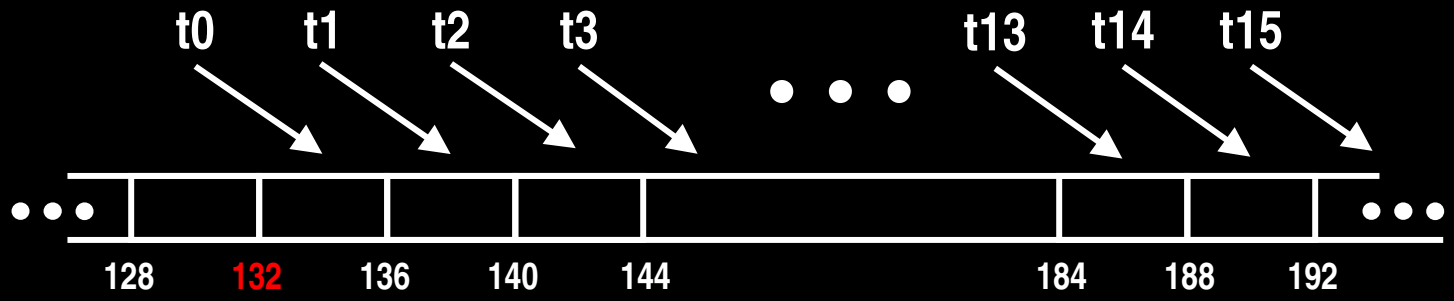
Coalesced `float` memory access

(divergent warp)

Non-Coalesced Global Memory Accesses

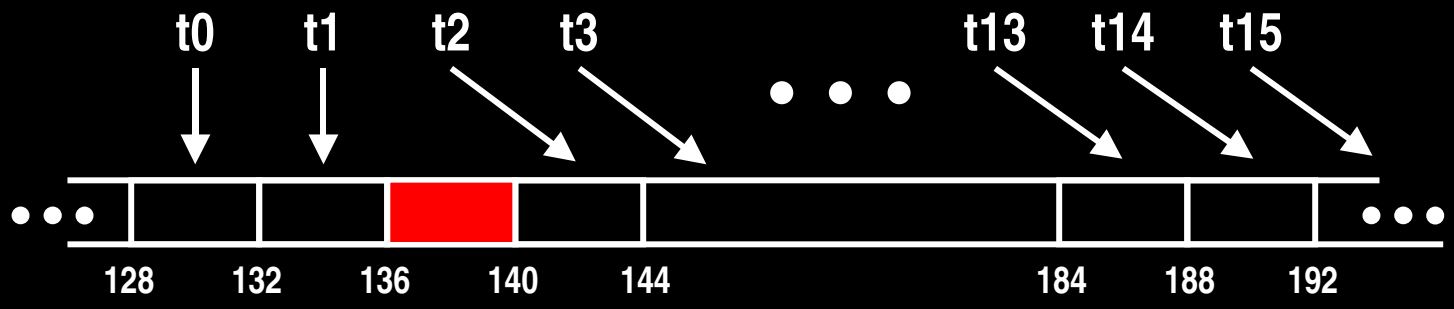


Non-sequential `float` memory access

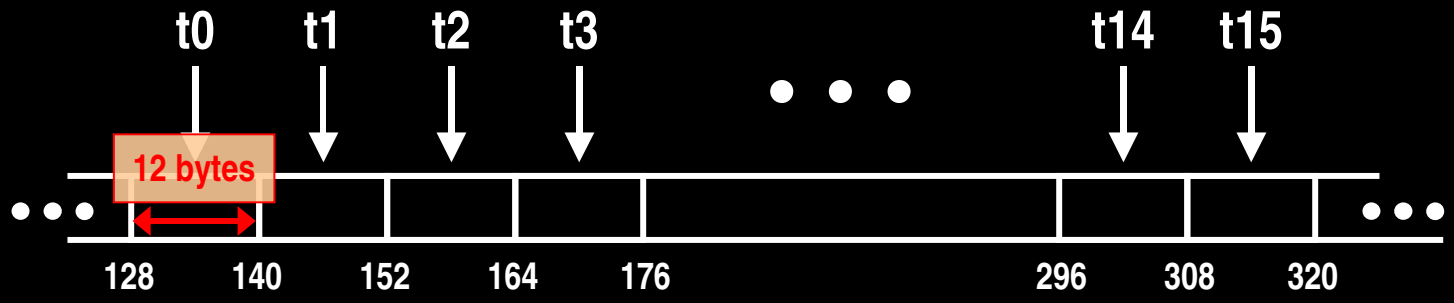


Misaligned starting address

Non-Coalesced Global Memory Accesses



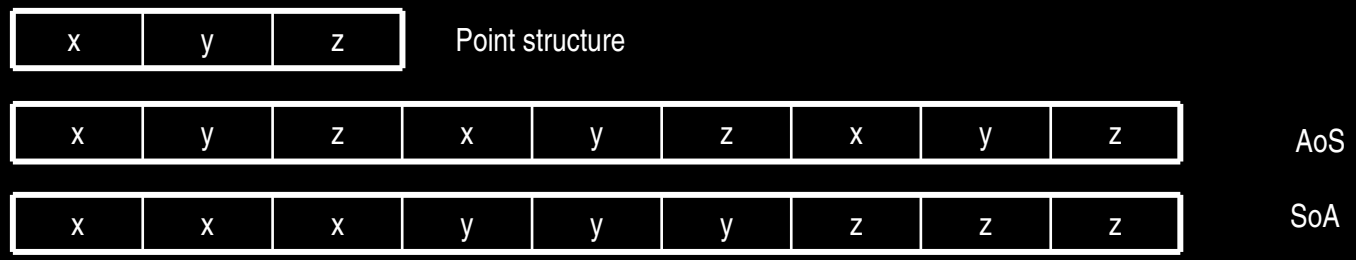
Non-contiguous `float` memory access



Non-coalesced `float3` memory access
(element size $\neq 2^n$)

Avoiding Non-Coalesced Accesses

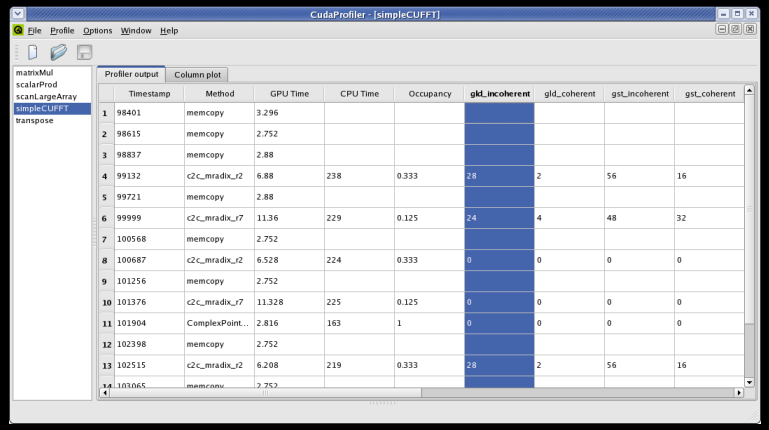
- For irregular read patterns, texture fetches can be a better alternative to global memory reads
- If all threads read the same location, use constant memory
- For sequential access patterns, but a structure of size ≠ 4, 8, or 16 bytes:
  - Use a Structure of Arrays (SoA) instead of Array of Structures (AoS)

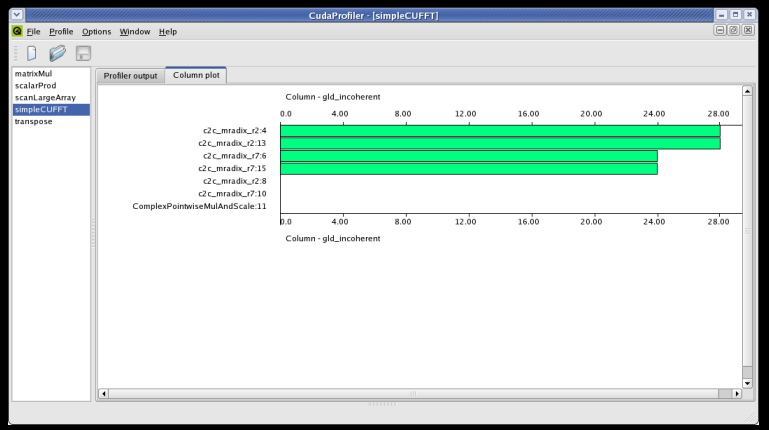


  - Or force structure alignment
    - Using __align(X), where X = 4, 8, or 16
  - Or use shared memory to achieve coalescing
    - More on this later

CUDA Visual Profiler

- Helps measure and find potential performance problem
  - GPU and CPU timing for all kernel invocations and memcpys
  - Time stamps
- Access to hardware performance counters





Profiler Signals

- Events are tracked with hardware counters on signals in the chip:
  - timestamp
  - gld_incoherent
  - gld_coherent
  - gst_incoherent
  - gst_coherent

  Global memory loads/stores are coalesced (coherent) or non-coalesced (incoherent)
  - local_load
  - local_store

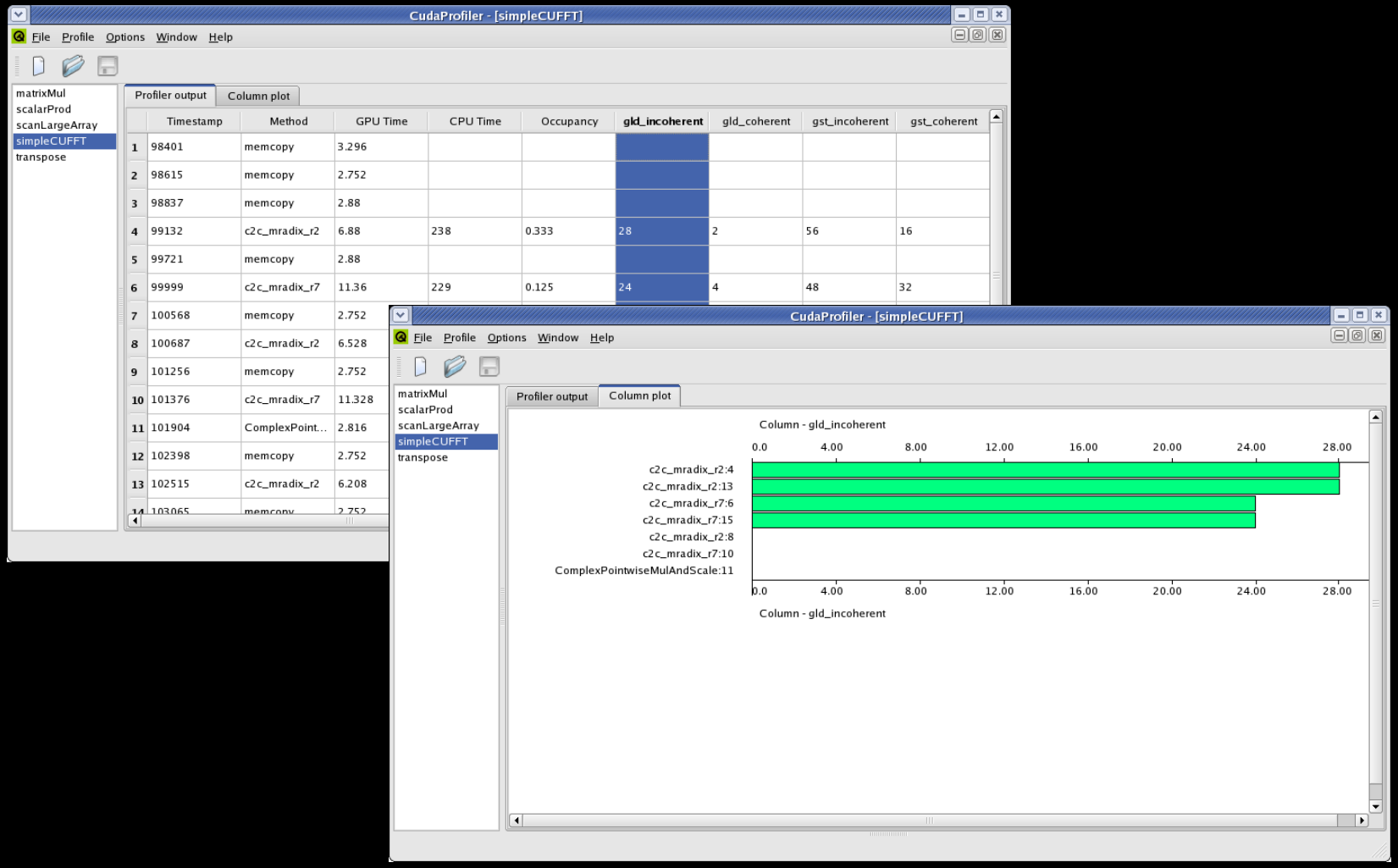
  Local loads/stores
  - branch
  - divergent_branch

  Total branches and divergent branches taken by threads
  - instructions – instruction count
  - warp_serialize – thread warps that serialize on address conflicts to shared or constant memory
  - cta_launched – executed thread blocks

Interpreting profiler counters

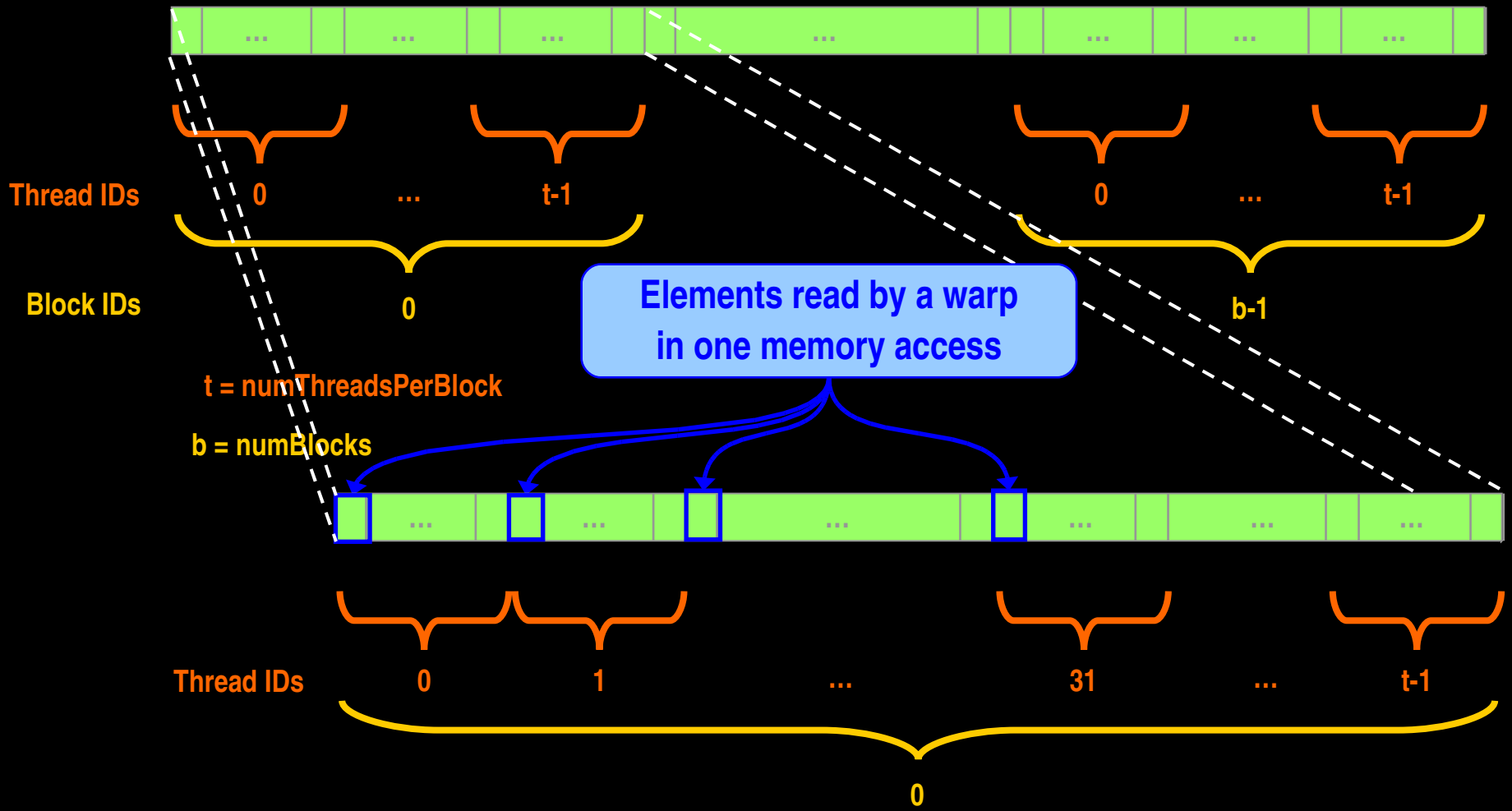
- Values represent events within a thread warp
- Only targets one multiprocessor
  - Values will not correspond to the total number of warps launched for a particular kernel.
  - Launch enough thread blocks to ensure that the target multiprocessor is given a consistent percentage of the total work.
- Values are best used to identify relative performance differences between unoptimized and optimized code
  - In other words, try to reduce the magnitudes of gld/gst_incoherent, divergent_branch, and warp_serialize

Back to Reduce Exercise: Profile with the Visual Profiler



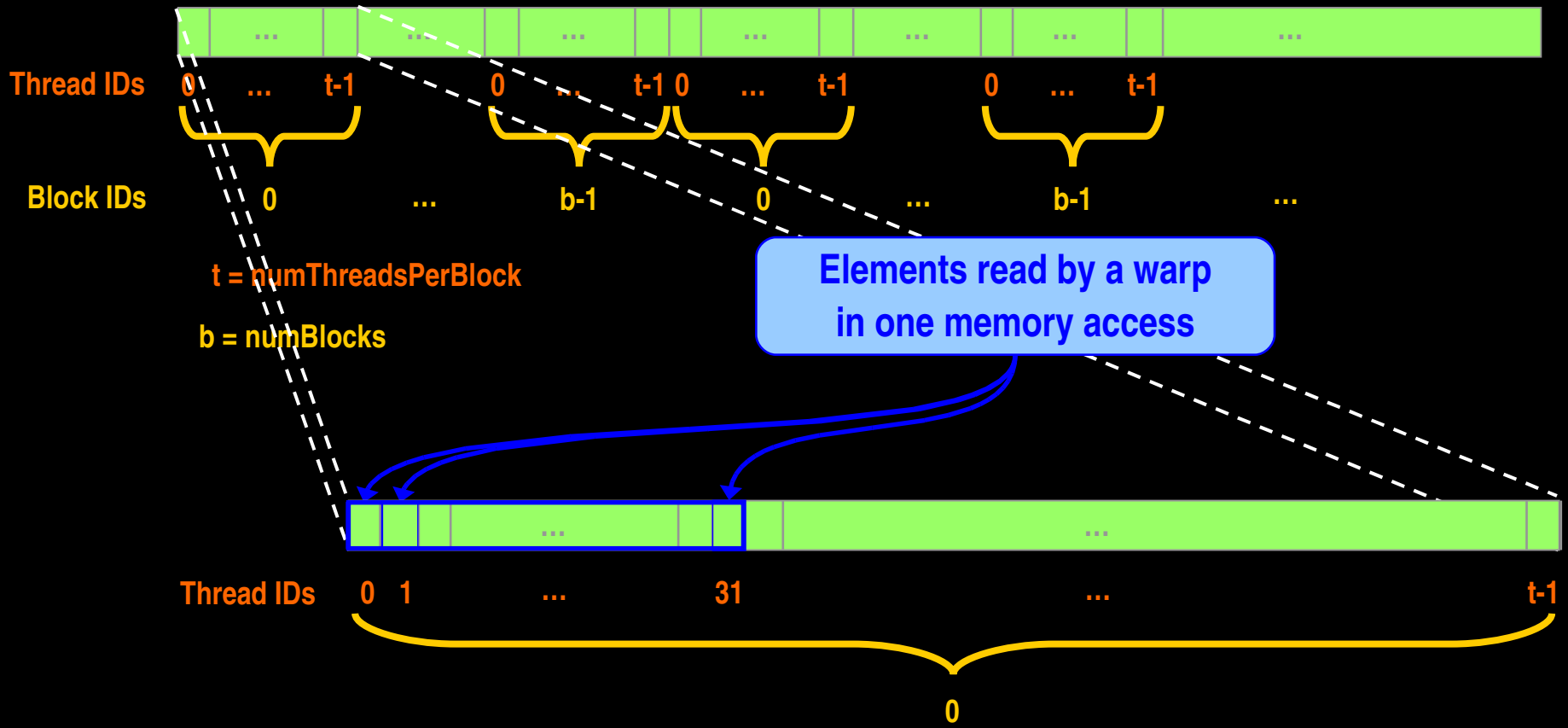
Back to Reduce Exercise: Problem with Reduce 1

- Non-coalesced memory reads!



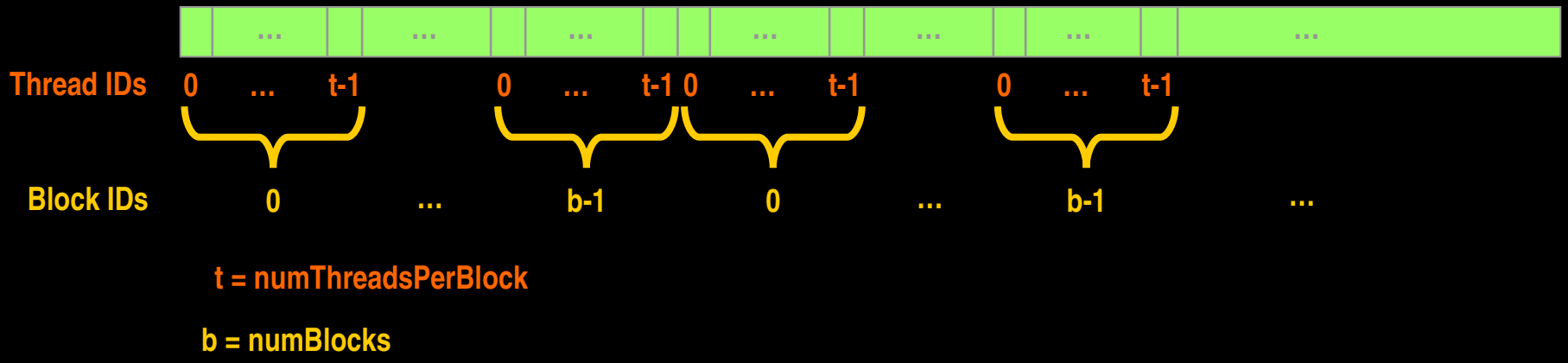
Reduce 2

- Distribute threads differently to achieve coalesced memory reads
  - Bonus: No need to ping pong anymore : no thread writes where another thread reads



Reduce 2: Go Ahead!

- Have a look in `reduce2` directory
- Goal: Replace the `TODO`s in `reduce2.cu` to get "test PASSED"



Maximize Use of Shared Memory

- Shared memory is hundreds of times faster than global memory
- Threads can cooperate via shared memory
  - Not so via global memory
- A common way of scheduling some computation on the device is to block it up to take advantage of shared memory:
  - Partition the data set into data subsets that fit into shared memory
  - Handle each data subset with one thread block:
    - Load the subset from global memory to shared memory
    - __syncthreads()
    - Perform the computation on the subset from shared memory
      - each thread can efficiently multi-pass over any data
    - __syncthreads() (if needed)
    - Copy results from shared memory to global memory

Maximize Occupancy to Hide Latency

- Sources of latency:
  - Global memory access: 400-600 cycle latency
  - Read-after-write register dependency
    - Instruction's result can only be read 11 cycles later
- Latency blocks dependent instructions in the same thread
- But instructions in other threads are not blocked
- Hide latency by running as many threads per multiprocessor as possible!
- Choose execution configuration to maximize

  occupancy = (# of active warps) / (maximum # of active warps)
  - Maximum # of active warps is 24 on G8x

Execution Configuration: Constraints

- Maximum # of threads per block: 512
- # of active threads limited by resources:
  - # of registers per multiprocessor (register pressure)
  - Amount of shared memory per multiprocessor
- Use –maxrregcount=N flag to NVCC
  - N = desired maximum registers / kernel
  - At some point “spilling” into LMEM may occur
    - Reduces performance – LMEM is slow
    - Check .cubin file for LMEM usage

Determining Resource Usage

- Compile the kernel code with the -cubin flag to determine register usage.
- Open the .cubin file with a text editor and look for the “code” section.

```
architecture {sm_10}
abiversion {0}
modname {cubin}
code  {
        name = BlackScholesGPU
        lmem = 0
        smem = 68
        reg = 20
        bar = 0
        bincode  {
                0xa0004205 0x04200780 0x40024c09 0x00200780
                ...
```

per thread local memory
(used by compiler to spill registers to device memory)

per thread block shared memory

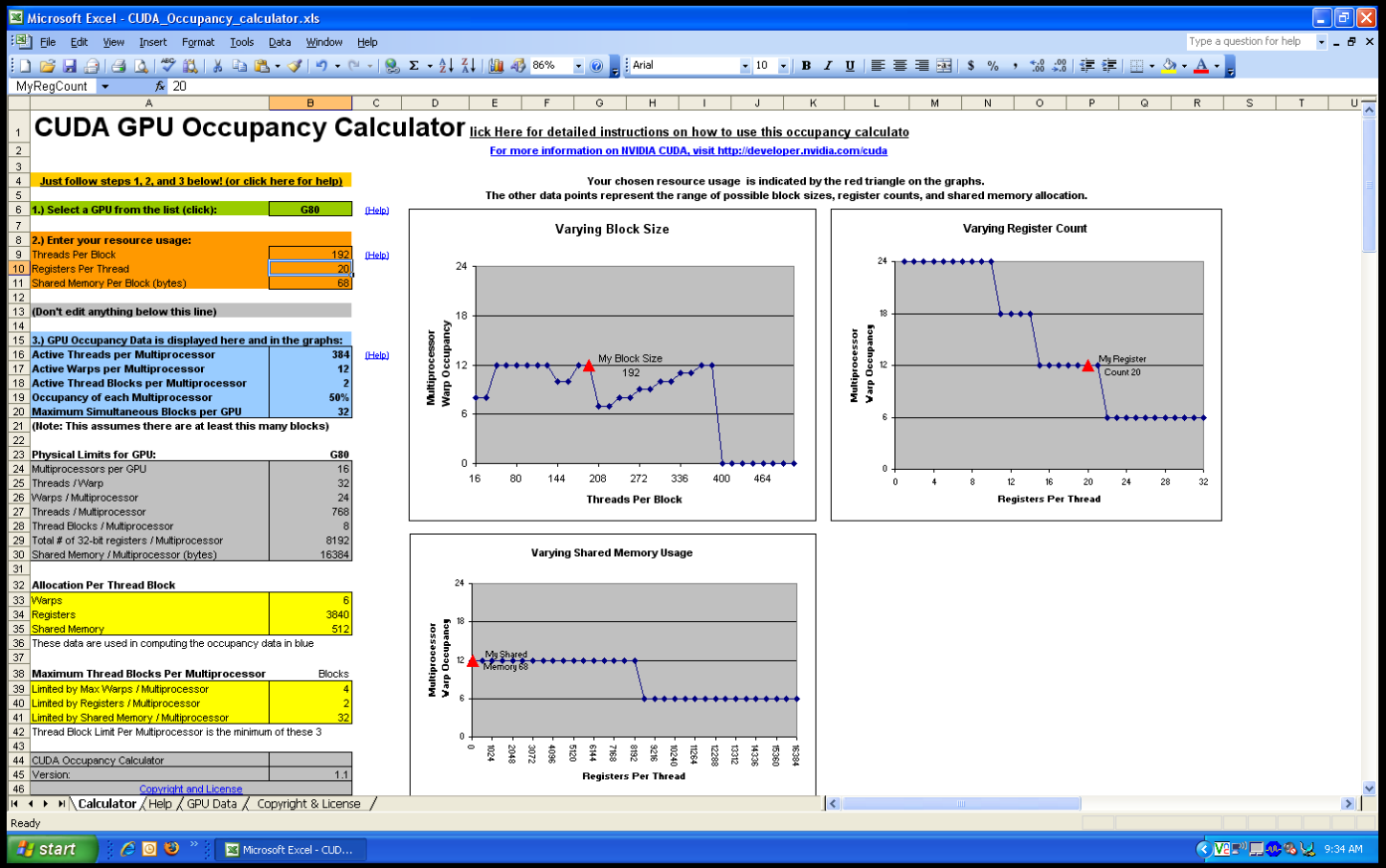
per thread registers

Execution Configuration: Heuristics

- (# of threads per block) = multiple of warp size (32)
  - To avoid wasting computation on under-populated warps
- (# of blocks) / (# of multiprocessors) > 1
  - So all multiprocessors have at least a block to execute
- Per-block resources (shared memory and registers) at most half of total available
- And: (# of blocks) / (# of multiprocessors) > 2
  - To get more than 1 active block per multiprocessor
  - With multiple active blocks that aren't all waiting at a __syncthreads(), the multiprocessor can stay busy
- (# of blocks) > 100 to scale to future devices
  - Blocks stream through machine in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations
- Very application-dependent: experiment!

Occupancy Calculator

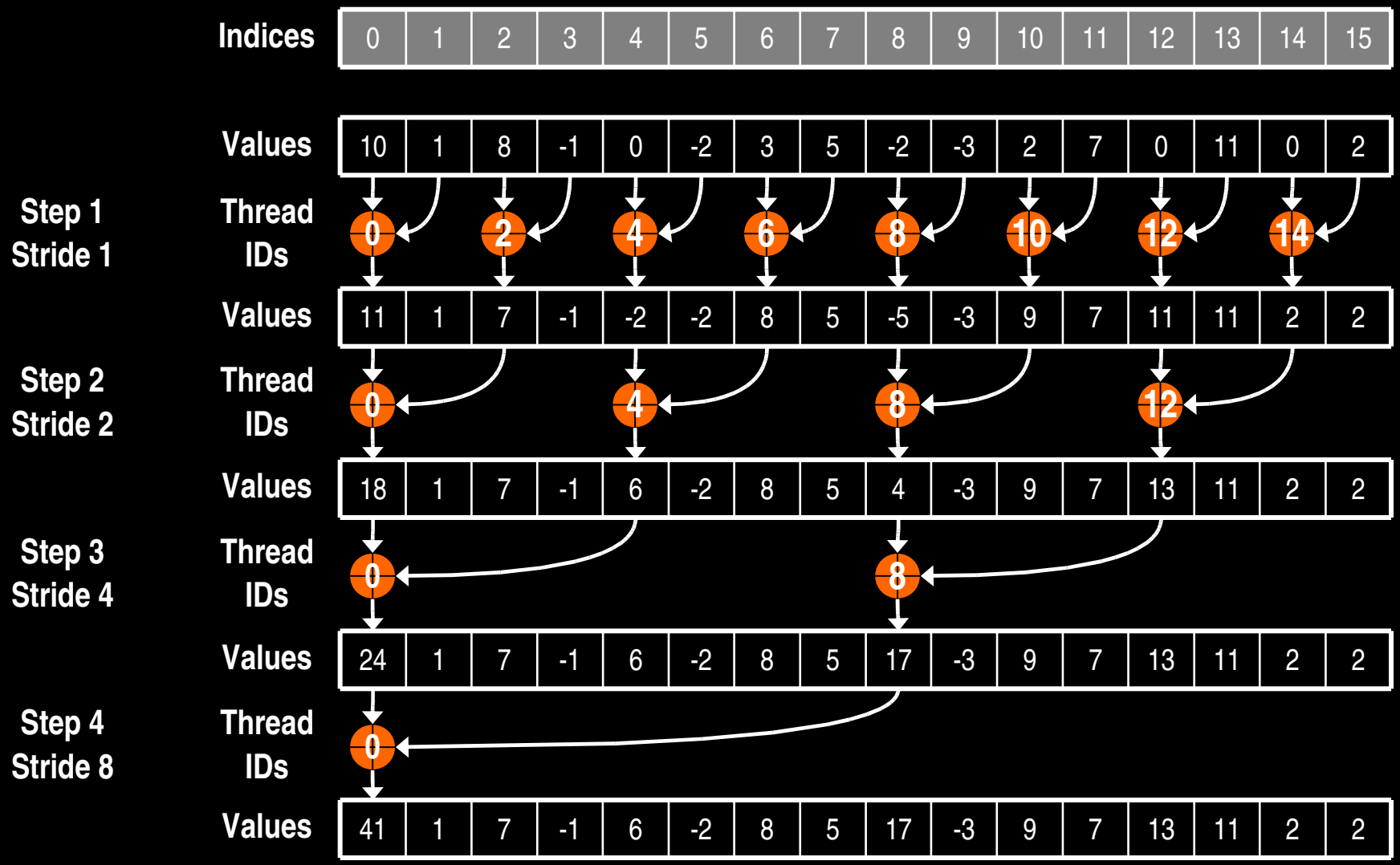
- To help you: the CUDA occupancy calculator
  http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls



Back to Reduce Exercise: Problem with Reduce 2

- Reduce 2 does not take advantage of shared memory!
- Reduce 3 fixes this by implementing parallel reduction

Parallel Reduction Implementation



Parallel Reduction Complexity

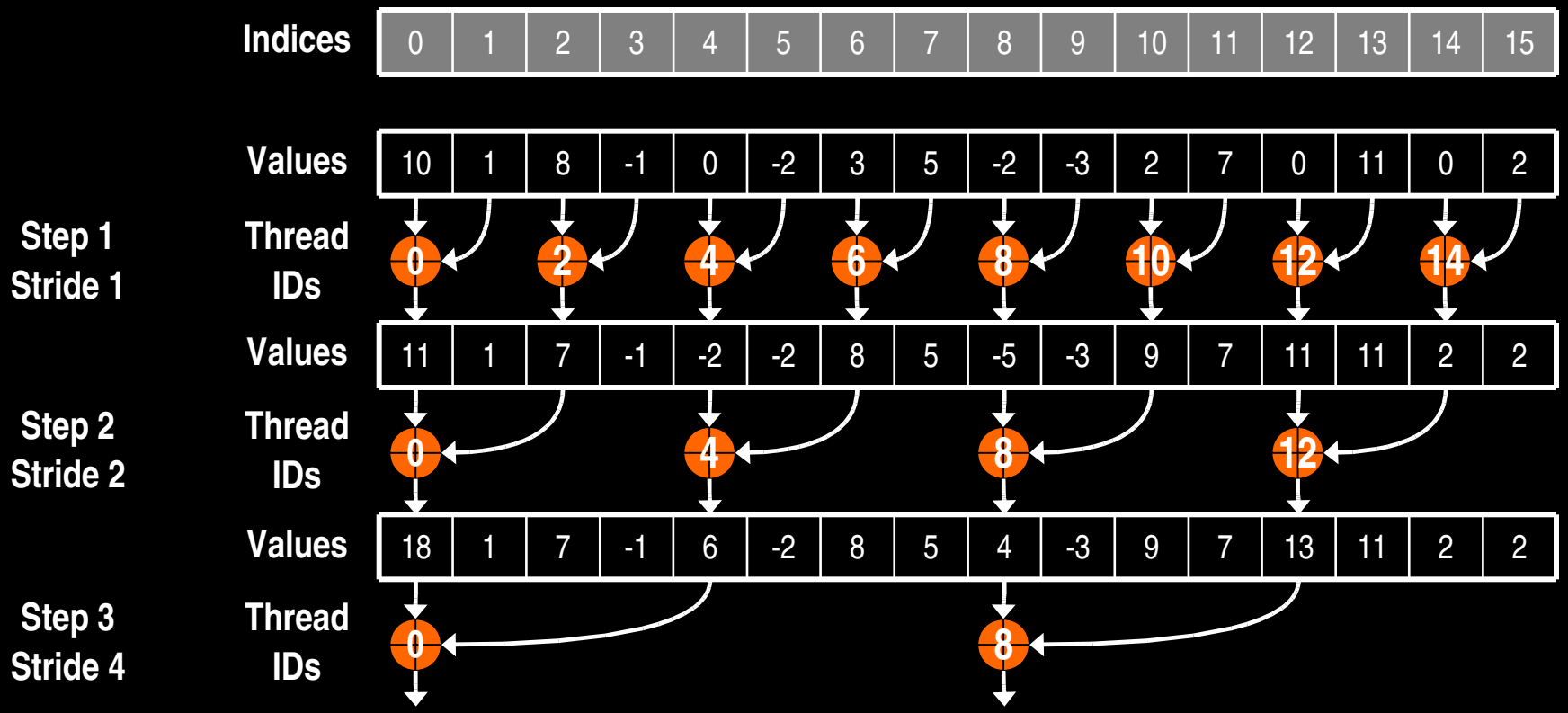
- Takes log(N) steps and each step S performs $N/2^S$ independent operations
  - Step complexity is O(log(N))
- For $N=2^D$, performs $\sum_{S \in [1..D]} 2^{D-S} = N-1$ operations
  - Work complexity is O(N)
  - Is work-efficient (i.e. does not perform more operations than a sequential reduction)
- With P threads physically in parallel (P processors), performs $\sum_{S \in [1..D]} \text{ceil}(2^{D-S}/P)$ operations
  - $\sum_{S \in [1..D]} \text{ceil}(2^{D-S}/P) < \sum_{S \in [1..D]} (\text{floor}(2^{D-S}/P) + 1) < N/P + \log(N)$
  - Time complexity is O(N/P + log(N))
  - Compare to O(N) for sequential reduction

Reduce 3

- Each thread stores its result in an array of `numThreadsPerBlock` elements in shared memory
- Each block performs a parallel reduction on this array
- `reduce_kernel` is called only 2 times:
  - First call reduces from `numValues` to `numBlocks`
  - Second call performs final reduction using one thread block

Reduce 3: Go Ahead!

- Have a look in `reduce3 directory`
- Goal: Replace the `TODO`s in `reduce3.cu` to get "test PASSED"



Optimize Instruction Usage: Basic Strategies

- Minimize use of low-throughput instructions
- Use high precision only where necessary
- Minimize divergent warps

Arithmetic Instruction Throughput

- `float` add/mul/mad, `int` add, shift, min, max: 4 cycles per warp
  - `int` multiply (*) is by default 32-bit
    - Requires multiple cycles per warp
    - Use `__[u]mul24()` intrinsics for 4-cycle 24-bit `int` multiply
- Integer divide and modulo are more expensive
  - Compiler will convert literal power-of-2 divides to shifts
    - But we have seen it miss some cases
  - Be explicit in cases where compiler can't tell that divisor is a power of 2!
  - Useful trick: `foo%n == foo&(n-1)` if `n` is a power of 2

Arithmetic Instruction Throughput

- Reciprocal, reciprocal square root, sin/cos, log, exp: 16 cycles per warp
  - These are the versions prefixed with "__"
  - Examples: `__rcp()`, `__sin()`, `__exp()`
- Other functions are combinations of the above:
  - `y/x==rcp(x)*y` takes 20 cycles per warp
  - `sqrt(x)==rcp(rsqrt(x))` takes 32 cycles per warp

Runtime Math Library

- There are two types of runtime math operations:
  - `__func()`: direct mapping to hardware ISA
    - Fast but lower accuracy (see prog. guide for details)
    - Examples: `__sin(x), __exp(x), __pow(x,y)`
  - `func()`: compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: `sin(x), exp(x), pow(x,y)`

- The -use_fast_math compiler option forces every `func()` to compile to `__func()`

Make Your Program Float-Safe!

- Future hardware will have double precision support
  - G80 is single-precision only
  - Double precision will likely have additional cost
- Important to be float-safe to avoid using double precision where it is not needed
  - Add ‘f’ specifier on float literals:
    - `foo = bar * 0.123;  // double assumed`
    - `foo = bar * 0.123f;  // float explicit`
  - Use float version of standard library functions
    - `foo = sin(bar);  // double assumed`
    - `foo = sinf(bar);  // float explicit`

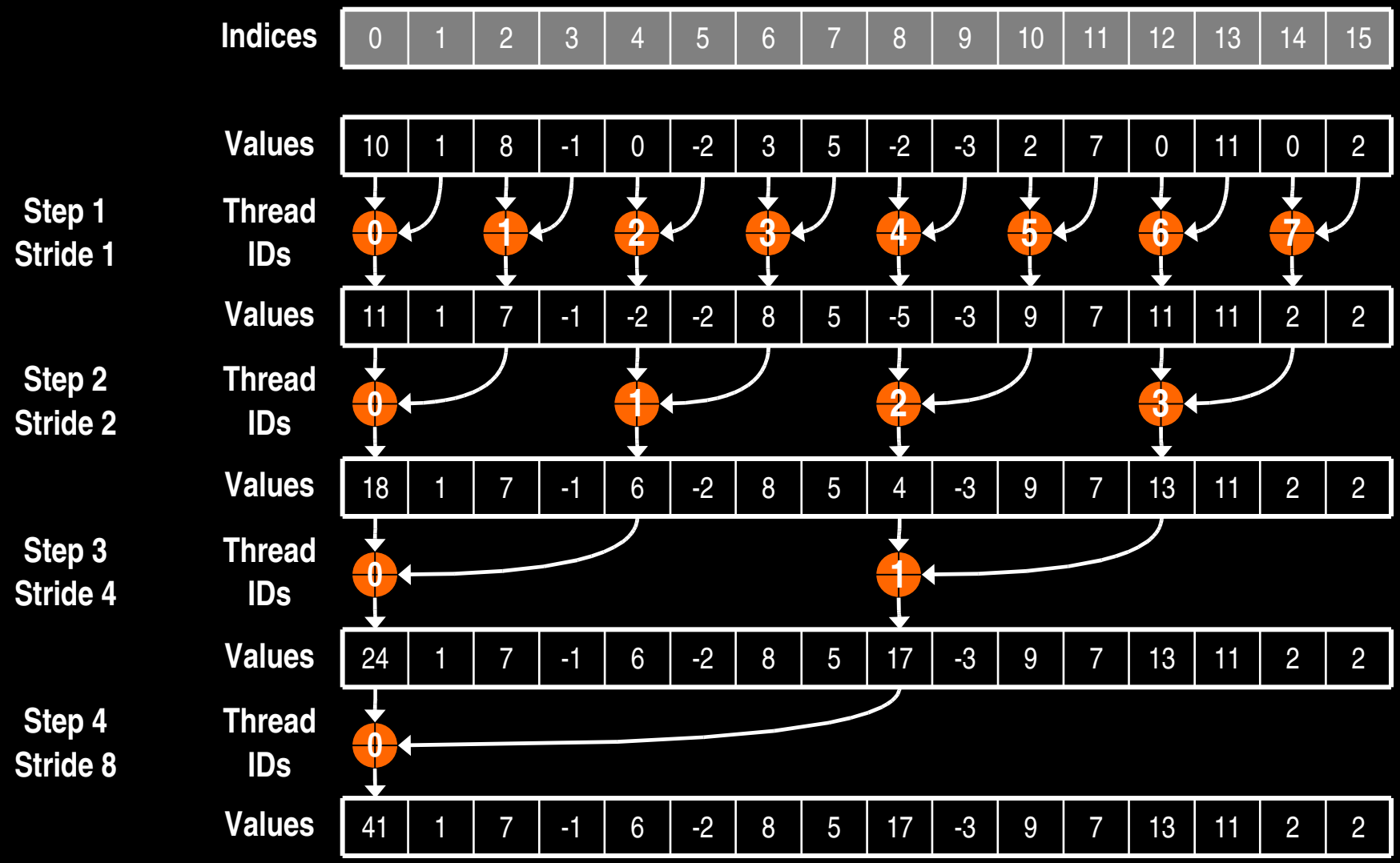
Deviations from IEEE-754

- Addition and multiplication are IEEE compliant
  - Maximum 0.5 ulp (Unit in the Last Place) error
- However, often combined into multiply-add (FMAD)
  - Intermediate result is truncated
- Division is non-compliant (2 ulp)
- Not all rounding modes are supported
- Denormalized numbers are not supported
- No mechanism to detect floating-point exceptions

Back to Reduce Exercise: Problem with Reduce 3

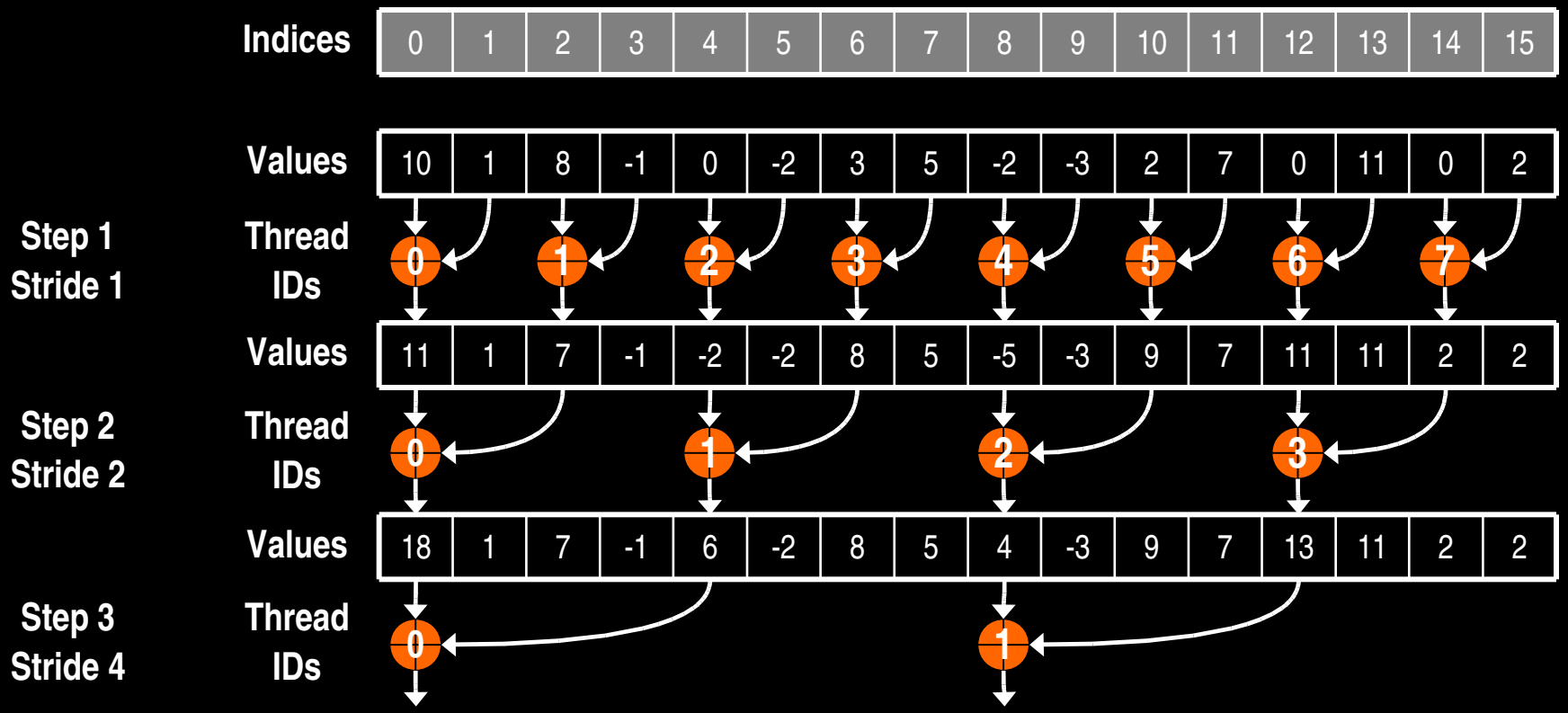
- Reduce 3 has high-cost test on thread-numbers!
- Reduce 4 fixes this by modifying the mapping between threads and data during parallel reduction

Reduce 4:
Parallel Reduction Implementation



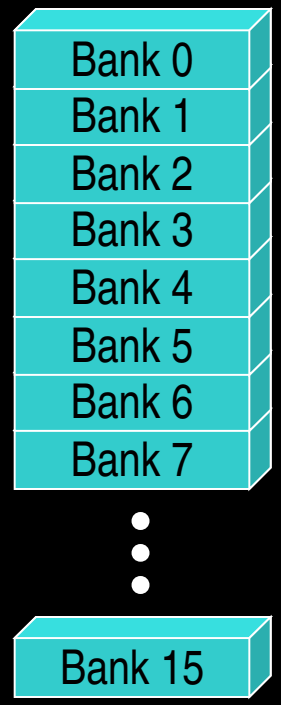
Reduce 4: Go Ahead!

- Open up `reduce4` directory
- Goal: Replace the `TODO`s in `reduce4.cu` to get "test PASSED"



Shared Memory Implementation: Banked Memory

- In a parallel machine, many threads access memory
  - Therefore, memory is divided into *banks*
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a *bank conflict*
  - Conflicting accesses are serialized

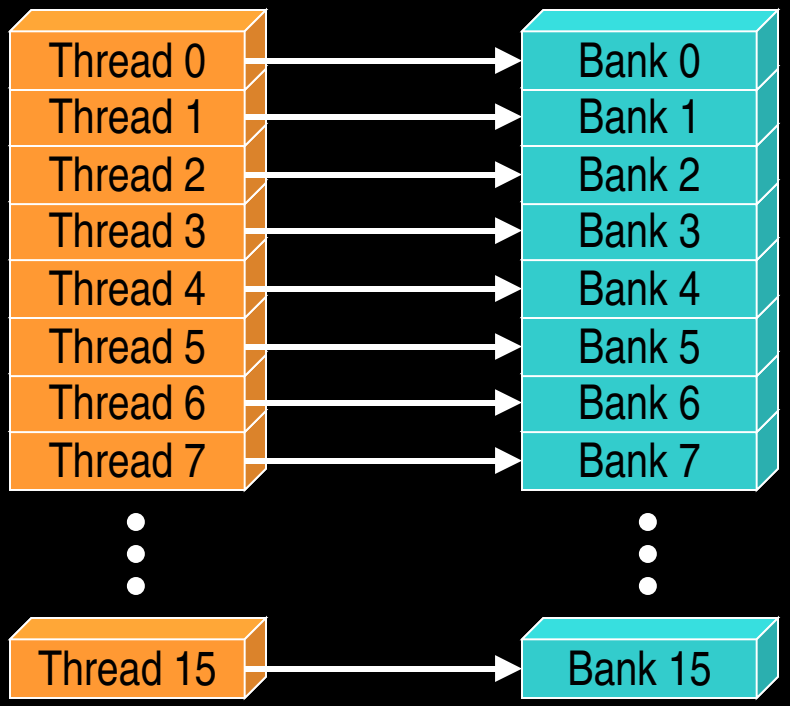


Shared Memory Is Banked

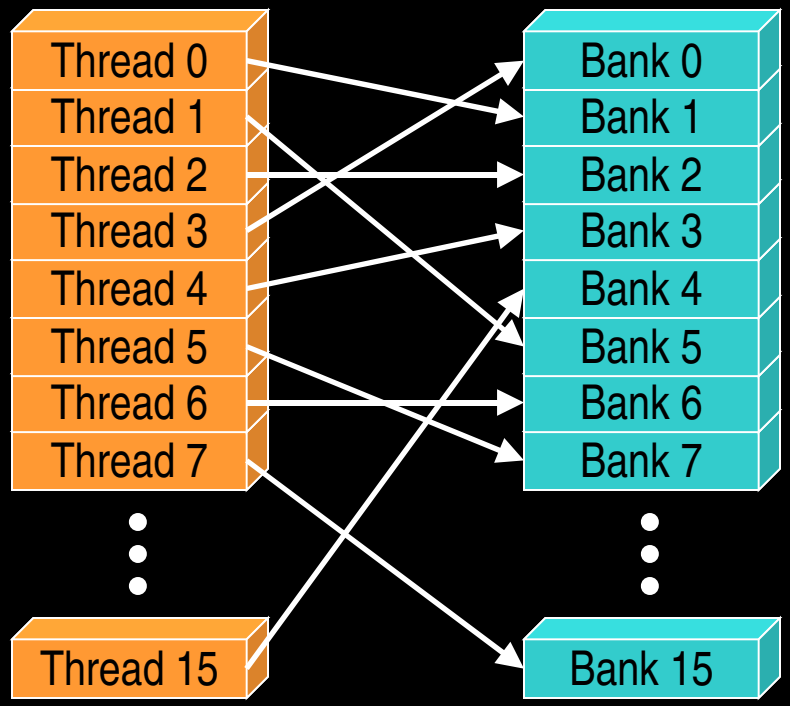
- Bandwidth of each bank is 32 bits per 2 clock cycles
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp

Bank Addressing Examples

- No bank conflicts
  - Linear addressing
    stride == 1

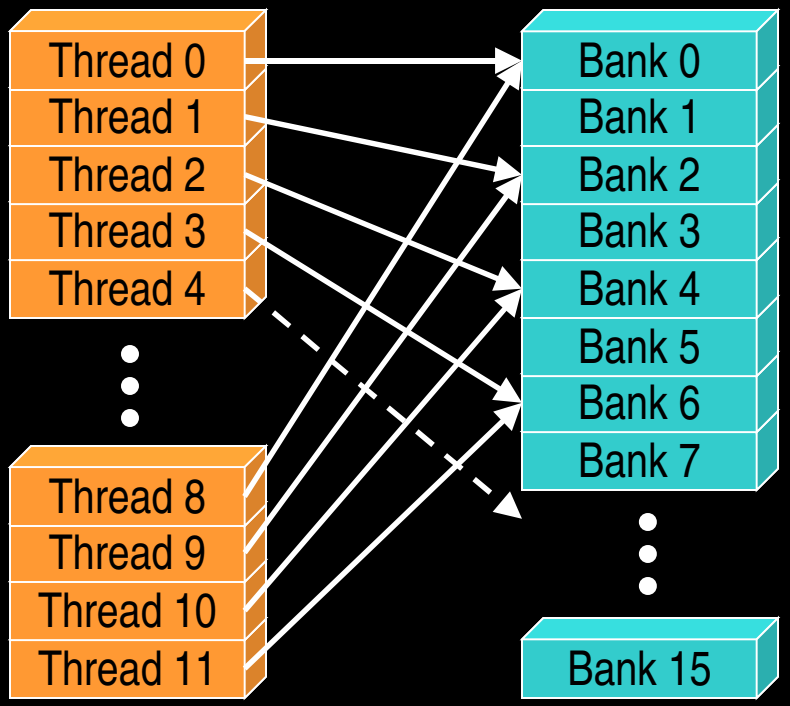


- No bank conflicts
  - Random 1:1 permutation

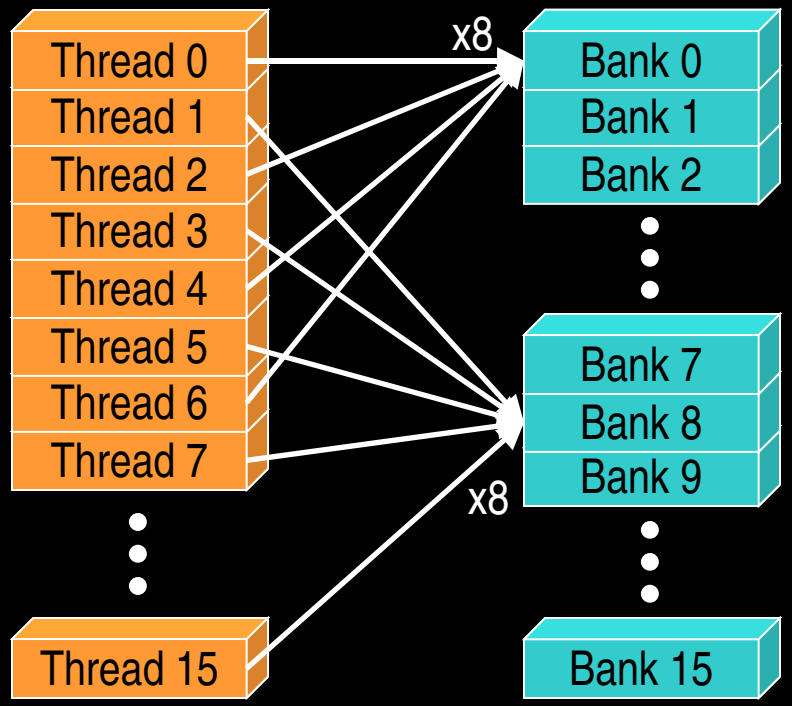


Bank Addressing Examples

- 2-way bank conflicts
  - Linear addressing
    stride == 2



- 8-way bank conflicts
  - Linear addressing
    stride == 8



Shared Memory Bank Conflicts

- Shared memory is as fast as registers if there are no bank conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp read the same word, there is no bank conflict (*broadcast*)
- The slow case:
  - Bank conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

Conclusion

- New architecture and programming model for GPU Computing
  - Fast on-chip shared memory for thread cooperation
  - General global memory access and storage model
  - Program the GPU in standard C with small extensions
- CUDA can achieve great results on data-parallel computations with a few simple performance optimization strategies:
  - Structure your application and select execution configurations to maximize exploitation of the GPU's parallel capabilities
  - Minimize CPU $\leftrightarrow$ GPU data transfers
  - Coalesce global memory accesses
  - Take advantage of shared memory
  - Minimize divergent warps
  - Minimize use of low-throughput instructions
  - Avoid shared memory accesses with high degree of bank conflicts

Where to go from here

- Get CUDA & SDK http://developer.nvidia.com/CUDA
  - CUDA works on all NVIDIA 8-Series GPUs (and later)
    - GeForce, Quadro, and Tesla

- Talk about CUDA http://forums.nvidia.com

NVIDIA®

Extra Slides

CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
  - Thread can access any memory location
- Unlimited access to memory
  - Thread can read/write as many locations as needed
- Shared memory (per block) and thread synchronization
  - Threads can cooperatively load data into shared memory
  - Any thread can then access any shared memory location
- Low learning curve
  - Just a few extensions to C
  - No knowledge of graphics is required
- No graphics API overhead

A quick review

- device = GPU = set of multiprocessors
- Multiprocessor = set of processors & shared memory
- Kernel = GPU program
- Grid = array of thread blocks that execute a kernel
- Thread block = group of SIMD threads that execute a kernel and can communicate via shared memory

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

Application Programming Interface

- The API is an extension to the C programming language
- It consists of:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library split into:
    - A common component providing built-in vector types and a subset of the C runtime library supported in both host and device codes
    - A host component to control and access one or more devices from the host
    - A device component providing device-specific functions

Language Extensions: Function Type Qualifiers

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together
- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Language Extensions: Variable Type Qualifiers

	Memory	Scope	Lifetime
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__shared__` or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except for large structures or arrays that reside in local memory
- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:

    ```
    __global__ void KernelFunc(float* ptr)
    ```

  - Obtained as the address of a global variable:

    ```
    float* ptr = &GlobalVar;
    ```

Language Extensions: Execution Configuration

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3    DimGrid(100, 50);    // 5000 thread blocks
dim3    DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

  - The optional `SharedMemBytes` bytes are:
    - Allocated in addition to the compiler allocated shared memory
    - Mapped to any variable declared as:

```
extern __shared__ float DynamicSharedMem[];
```

- Any call to a kernel function is asynchronous
  - Control returns to CPU immediately

Language Extensions: Built-in Variables

- `dim3 gridDim;`
  - Dimensions of the grid in blocks (`gridDim.z` unused)
- `dim3 blockDim;`
  - Dimensions of the block in threads
- `dim3 blockIdx;`
  - Block index within the grid
- `dim3 threadIdx;`
  - Thread index within the block

Common Runtime Component

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes

Common Runtime Component: Built-in Vector Types

- `[u]char[1..4]`, `[u]short[1..4]`, `[u]int[1..4]`, `[u]long[1..4]`, `float[1..4]`
  - Structures accessed with `x`, `y`, `z`, `w` fields:

```
uint4 param;
int y = param.y;
```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions

Common Runtime Component: Mathematical Functions

- `powf`, `sqrtf`, `cbrtf`, `hypotf`
- `expf`, `exp2f`, `expm1f`
- `logf`, `log2f`, `log10f`, `log1pf`
- `sinf`, `cosf`, `tanf`
- `asinf`, `acosf`, `atanf`, `atan2f`
- `sinhf`, `coshf`, `tanhf`
- `asinhf`, `acoshf`, `atanhf`
- `ceil`, `floor`, `trunc`, `round`
- Etc.

- When executed in host code, a given function uses the C runtime implementation if available
- These functions are only supported for scalar types, not vector types

Common Runtime Component: Texture Types

- Texture memory is accessed through texture references:

```
texture<float, 2> myTexRef; // 2D texture of float values

myTexRef.addressMode[0] = cudaAddressModeWrap;
myTexRef.addressMode[1] = cudaAddressModeWrap;
myTexRef.filterMode     = cudaFilterModeLinear;
```

- Texture fetching in device code:

```
float4 value = tex2D(myTexRef, u, v);
```

Host Runtime Component

- Provides functions to deal with:
  - Device management (including multi-device systems)
  - Memory management
  - Texture management
  - Interoperability with OpenGL and Direct3D9
  - Error handling
- Initializes the first time a runtime function is called
- A host thread can execute device code on only one device
  - Multiple host threads required to run on multiple devices

Host Runtime Component: Device Management

- Device enumeration
  - `cudaGetDeviceCount(),cudaGetDeviceProperties()`
- Device selection
  - `cudaChooseDevice(),cudaSetDevice()`

Host Runtime Component: Memory Management

- Two kinds of memory:
  - Linear memory: accessed through 32-bit pointers
  - CUDA arrays: opaque layouts with dimensionality, only readable through texture fetching
- Device memory allocation
  - `cudaMalloc()`, `cudaMallocPitch()`, `cudaFree()`, `cudaMallocArray()`, `cudaFreeArray()`
- Memory copy from host to device, device to host, device to device
  - `cudaMemcpy()`, `cudaMemcpy2D()`, `cudaMemcpyToArray()`, `cudaMemcpyFromArray()`, etc. `cudaMemcpyToSymbol()`, `cudaMemcpyFromSymbol()`
- Memory addressing
  - `cudaGetSymbolAddress()`

Host Runtime Component: Texture Management

- Texture references can be bound to:
  - CUDA arrays
  - Linear memory
    - 1D texture only, no filtering, integer texture coordinate
  - `cudaBindTexture(), cudaUnbindTexture()`

Host Runtime Component: Interoperability with Graphics APIs

- OpenGL buffer objects and Direct3D9 vertex buffers can be mapped into the address space of CUDA:
  - To read data written by OpenGL
  - To write data for consumption by OpenGL
  - `cudaGLMapBufferObject()`, `cudaGLUnmapBufferObject()` `cudaD3D9MapVertexBuffer()`, `cudaD3D9UnmapVertexBuffer()`

Host Runtime Component: Events

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed
  - asyncAPI sample in CUDA SDK

```
cudaEvent_t start, stop;
cudaEventCreate(&start);          cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start);                  cudaEventDestroy(stop);
```

Host Runtime Component: Error Handling

- All CUDA calls return error code:
  - except for kernel launches
  - cudaError_t type
- cudaError_t cudaGetLastError(void)
  - returns the code for the last error (no error has a code)
- char* cudaGetErrorString(cudaError_t code)
  - returns a null-terminted character string describing the error

```
printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
```

Device Runtime Component

- Provides device-specific functions

Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. `sin(x)`) have a less accurate, but faster device-only version (e.g. `__sin(x)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`

Device Runtime Component: GPU Atomic Integer Operations

- Atomic operations on integers in global memory:
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap
- Requires hardware with compute capability 1.1

Device Runtime Component: Texture Functions

- For texture references bound to CUDA arrays:

```
float u, v;
float4 value = tex2D(myTexRef, u, v);
```

- For texture references bound to linear memory:

```
int i;
float4 value = tex2D(myTexRef, i);
```

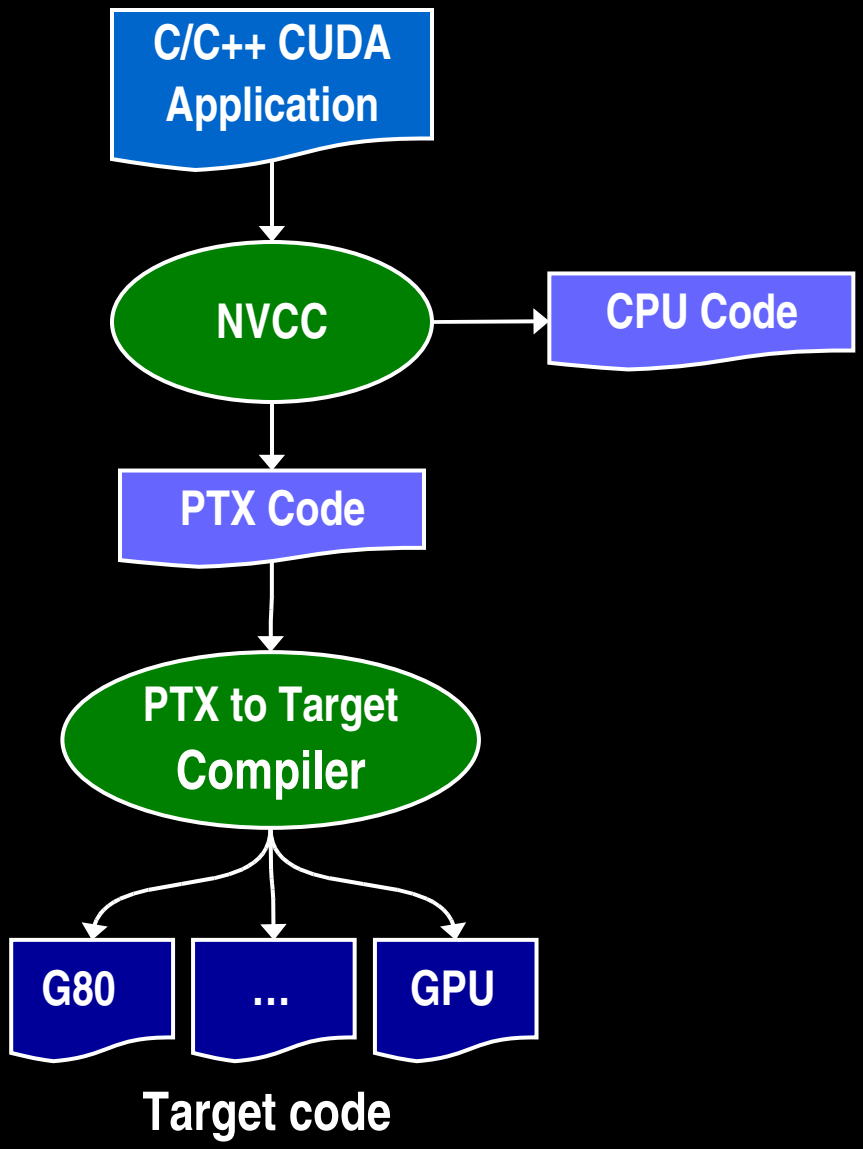
Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory
- Allowed in conditional code only if the conditional is uniform across the entire thread block

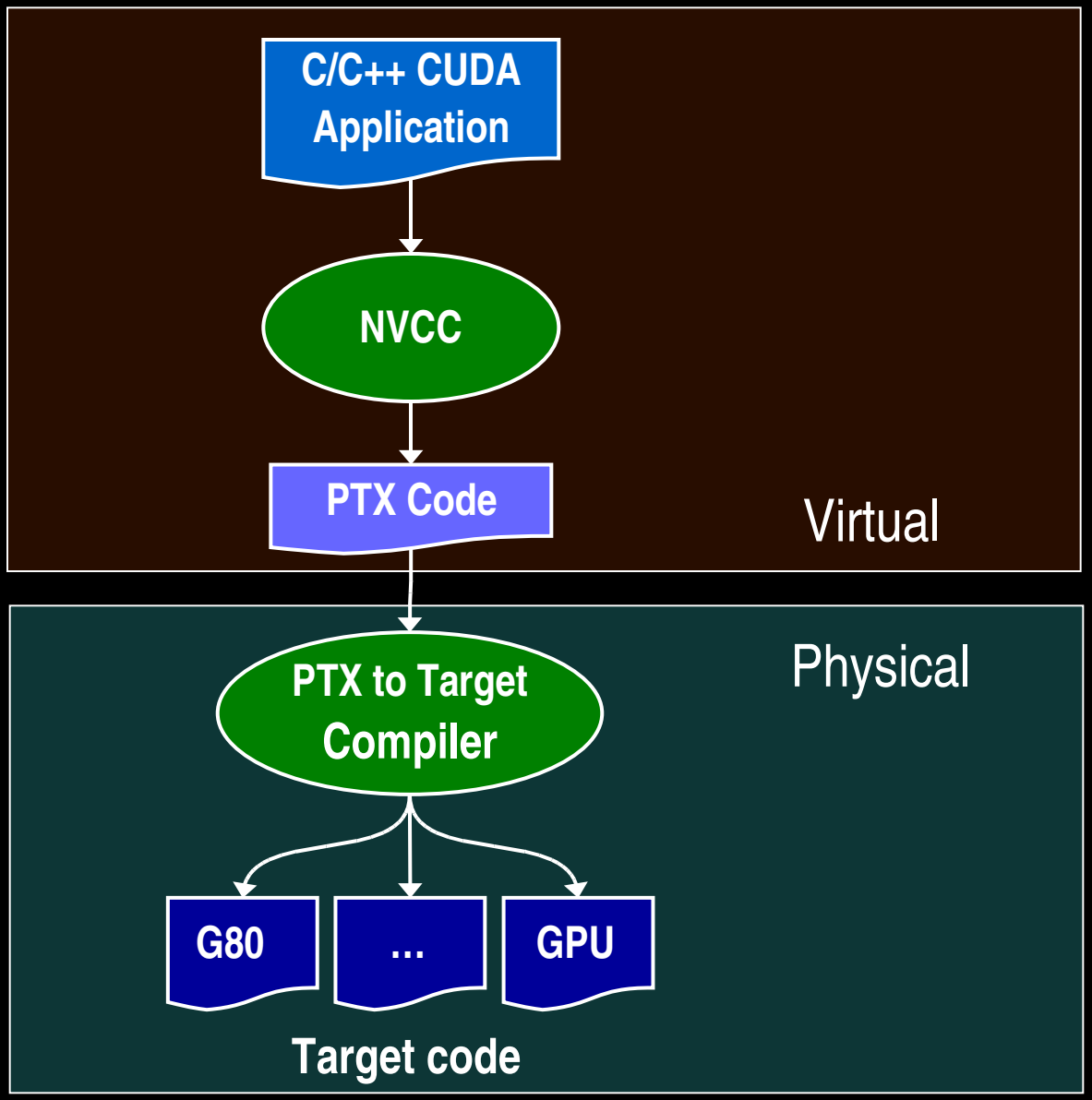
Compilation

- Any source file containing CUDA language extensions must be compiled with nvcc
- NVCC is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- NVCC can output:
  - Either C code (CPU Code)
    - That must then be compiled with the rest of the application using another tool
  - Or PTX object code directly
- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)

Compiling CUDA

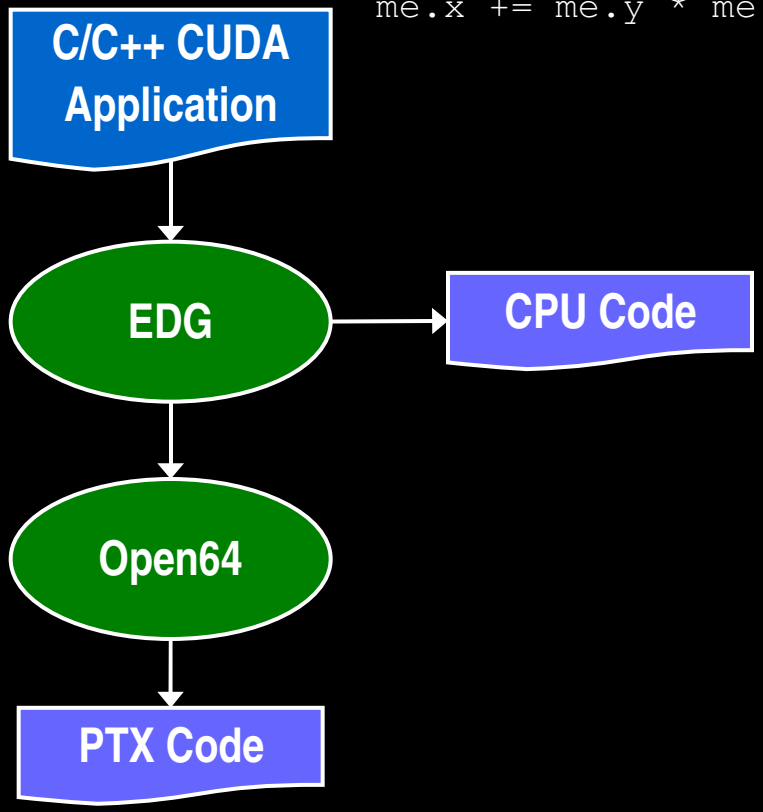


Compiling CUDA



NVCC & PTX Virtual Machine

```
float4 me = gx[gtid];
me.x += me.y * me.z;
```



- EDG
  - Separate GPU vs. CPU code
- Open64
  - Generates GPU PTX assembly
- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```

Role of Open64

Open64 compiler gives us

- A complete C/C++ compiler framework. Forward looking. We do not need to add infrastructure framework as our hardware arch advances over time.
- A good collection of high level architecture independent optimizations. All GPU code is in the inner loop.
- Compiler infrastructure that interacts well with other related standardized tools.

GeForce 8800 Series and Quadro FX 5600/4600 Technical Specifications

	Number of multiprocessors	Clock frequency (GHz)	Amount of device memory (MB)
GeForce 8800 GTX	16	1.35	768
GeForce 8800 GTS	12	1.2	640
Quadro FX 5600	16	1.35	1500
Quadro FX 4600	12	1.2	768

- Maximum number of threads per block: 512
- Maximum size of each dimension of a grid: 65535
- Warp size: 32 threads
- Number of registers per multiprocessor: 8192
- Shared memory per multiprocessor: 16 KB divided in 16 banks
- Constant memory: 64 KB

CUDA Libraries

- CUBLAS
  - CUDA “Basic Linear Algebra Subprograms”
  - Implementation of BLAS standard on CUDA
  - For details see cublas_library.pdf and cublas.h
- CUFFT
  - CUDA Fast Fourier Transform (FFT)
  - FFT one of the most important and widely used numerical algorithms
  - For details see cufft_library.pdf and cufft.h

CUBLAS Library

- Self-contained at API level
  - Application needs no direct interaction with CUDA driver
- Currently only a subset of CUBLAS core functions are implemented

- Simple to use:
  - Create matrix and vector objects in GPU memory
  - Fill them with data
  - Call sequence of CUBLAS functions
  - Upload results back from GPU to host
- Column-major storage and 1-based indexing
  - For maximum compatibility with existing Fortran apps

CUFFT Library

- Efficient implementation of FFT on CUDA
- Features
  - 1D, 2D, and 3D FFTs of complex-valued signal data
  - Batch execution for multiple 1D transforms in parallel
  - Transform sizes (in any dimension) in the range [2, 16384]

Tesla Architecture Family

	Number of Multiprocessors	Compute Capability
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 8800 GT	14	1.1
GeForce 8800M GTX	12	1.1
GeForce 8800 GTS	12	1.0
GeForce 8800M GTS	8	1.1
GeForce 8600 GTS, 8600 GT, 8700M GT, 8600M GT, 8600M GS	4	1.1
GeForce 8500 GT, 8400 GS, 8400M GT, 8400M GS	2	1.1
GeForce 8400M G	1	1.1
Tesla S870	4x16	1.0
Tesla D870	2x16	1.0
Tesla C870	16	1.0
Quadro Plex 1000 Model S4	4x16	1.0
Quadro Plex 1000 Model IV	2x16	1.0
Quadro FX 5600	16	1.0
Quadro FX 4600	12	1.0
Quadro FX 1700, FX 570, NVS 320M, FX 1600M, FX 570M	4	1.1
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M	2	1.1
Quadro NVS 130M	1	1.1