An Introduction to the GPU Memory Model

S7700 – Session 2 of 4

Chris Mason
Product Manager, Acceleware
GPU Technology Conference
Date: May 8, 2017

About Acceleware

Programmer Training

- CUDA and other HPC training classes
- Over 100 courses taught
- http://acceleware.com/training

Consulting Services

- Projects for Oil & Gas, Medical, Finance, Security and Defence, CAD, Media & Entertainment
- Mentoring, code review and complete project implementation
- http://acceleware.com/services

GPU Accelerated Software

- Seismic imaging & modeling
- Electromagnetics

Seismic Imaging & Modeling

AxWAVE™

- Seismic forward modeling
- 2D, 3D, constant and variable density models
- High fidelity finite-difference modeling

AxRTM™

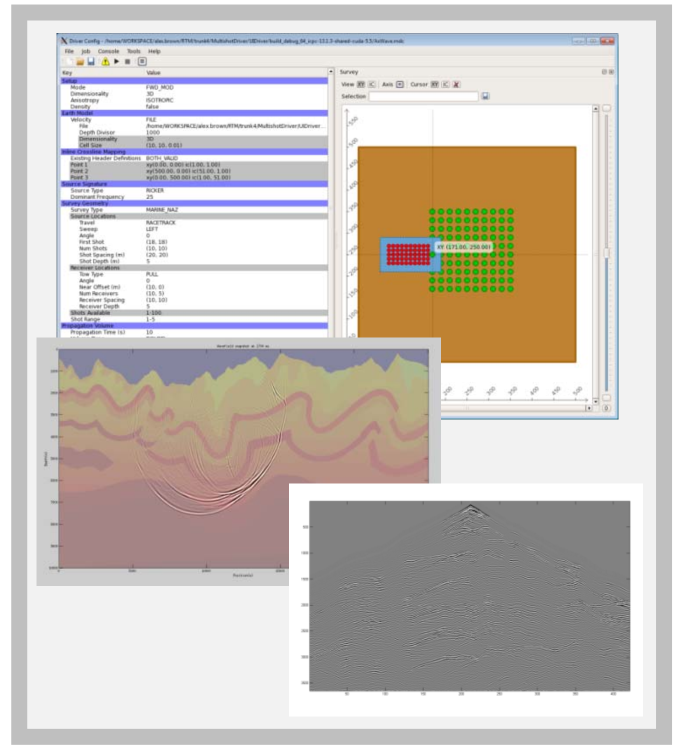
- High performance Reverse Time Migration application
- Isotropic, VTI and TTI media

AxFWI™

- Inversion of the full seismic data to provide an accurate subsurface velocity model
- Customizable for specific workflows

HPC Implementation

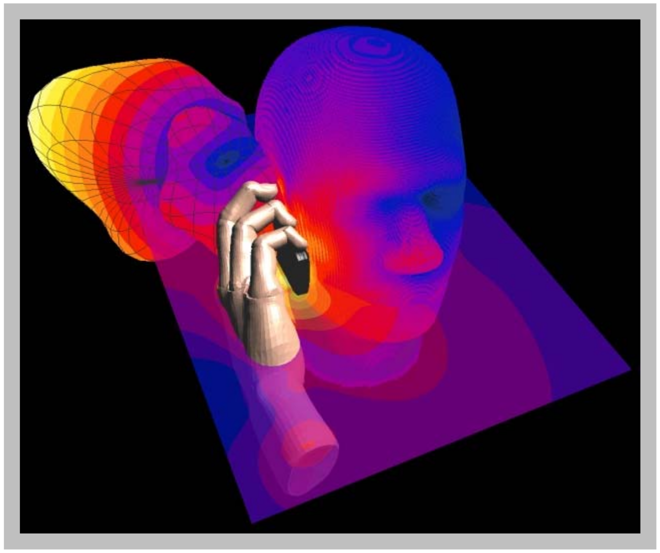
- Optimized for NVIDIA Tesla GPUs
- Efficient multi-GPU scaling



Electromagnetics

AxFDTD™

- Finite-Difference Time-Domain Electromagnetic Solver
- Optimized for NVIDIA GPUs
- Sub-gridding and large feature coverage
- Multi-GPU, GPU clusters, GPU targeting



Available from:

Consulting Services

Industry	Application	Work Completed	Results
Finance	Option Pricing	Debugged & optimized existing CUDA code Implemented the Leisen-Reimer version of the binomial model for stock option pricing	30-50x performance improvement compared to single-threaded CPU code
Security & Defense	Detection System	Replaced legacy Cell-based infrastructure with GPUs Implemented a GPU accelerated X-ray iterative image reconstruction and explosive detection algorithms	Surpassed the performance targets Reduced hardware cost by a factor of 10
CAE	SIMULIA Abaqus	Developed a GPU accelerated version Conducted a finite-element analysis and developed a library to offload LDLT factorization portion of the multi-frontal solver to GPUs	Delivered an accelerated (2-3x) solution that supports NVIDIA and AMD GPUs
Medical	CT Reconstruction Software	Developed a GPU accelerated application for image reconstruction on CT scanners and implemented advanced features including job batch manager, filtering and bad pixel corrections	Accelerated back projection by 31x
Oil & Gas	Seismic Application	Converted MATLAB research code into a standalone application & improved performance via algorithmic optimizations	20-30x speedup

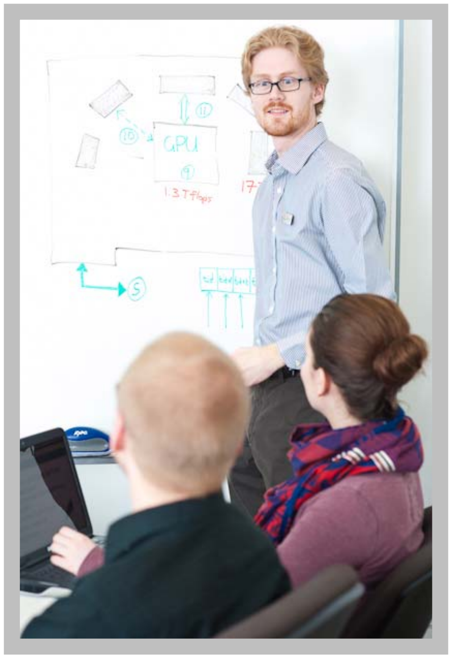
acceleware

Programmer Training

- CUDA and other HPC training classes
- Public, private onsite, and online courses
- Teachers with real world experience
- Hands-on lab exercises
- Progressive lectures
- Small class sizes to maximize learning
- 90 days post training support

"The level of detail is fantastic. The course did not focus on syntax but rather on how to expertly program for the GPU. I loved the course and I hope that we can get more of our team to take it."

Jason Gauci, Software Engineer
Lockheed Martin



acceleware

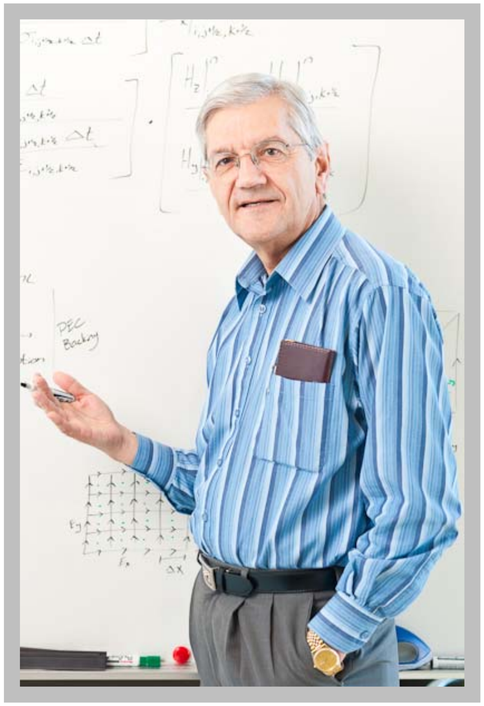
Outline

Task Parallelism

Thread Cooperation in GPU Computing

GPU Memory Model

- Shared Memory
- Constant Memory
- Global Memory



Data-Parallel Computing

Review: Data-parallelism

1. Performs operations on a data set organized into a common structure (eg. an array)

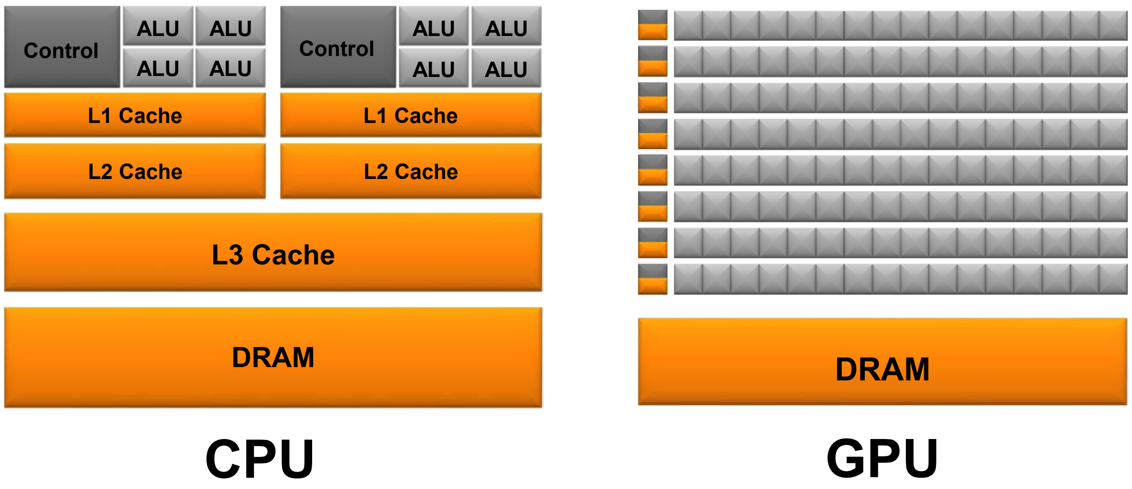
2. **Tasks** work collectively on the same structure with each task operating on its own portion of the structure

3. Tasks perform identical operations on their portions of the structure. Operations on each portion must not be **data dependent!**

Data-Parallel Computing on GPUs

Data-parallel computing maps well to GPUs:

- Identical operations executed on many data elements in parallel
- Simplified flow control allows increased ratio of compute logic (ALUs) to control logic



acceleware

The CUDA Programming Model

Until now we've considered CUDA as a strict data-parallel model

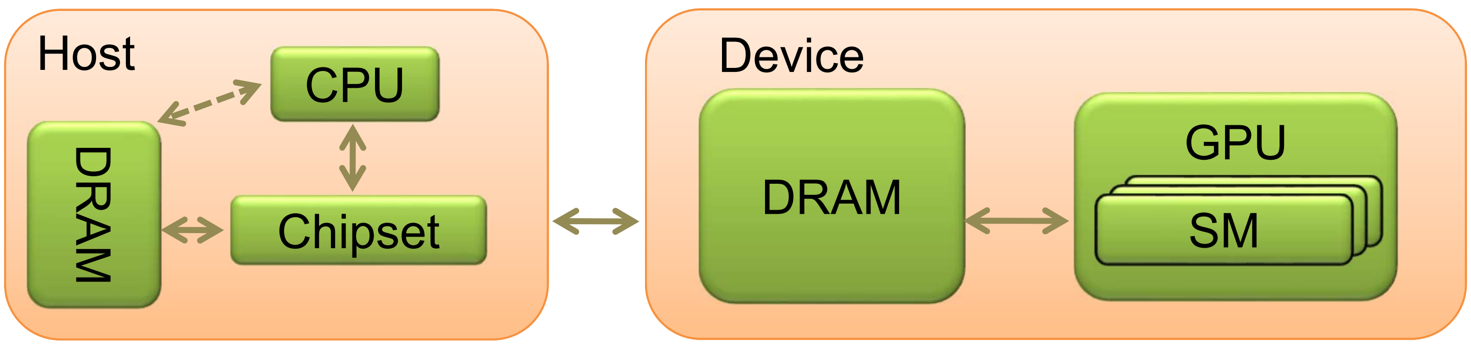
This isn't quite true!

We need to look at the hardware to understand when data-parallelism applies and when it doesn't

GPU Architecture Overview

Each GPU is comprised of one or more Streaming Multiprocessors (SMs)

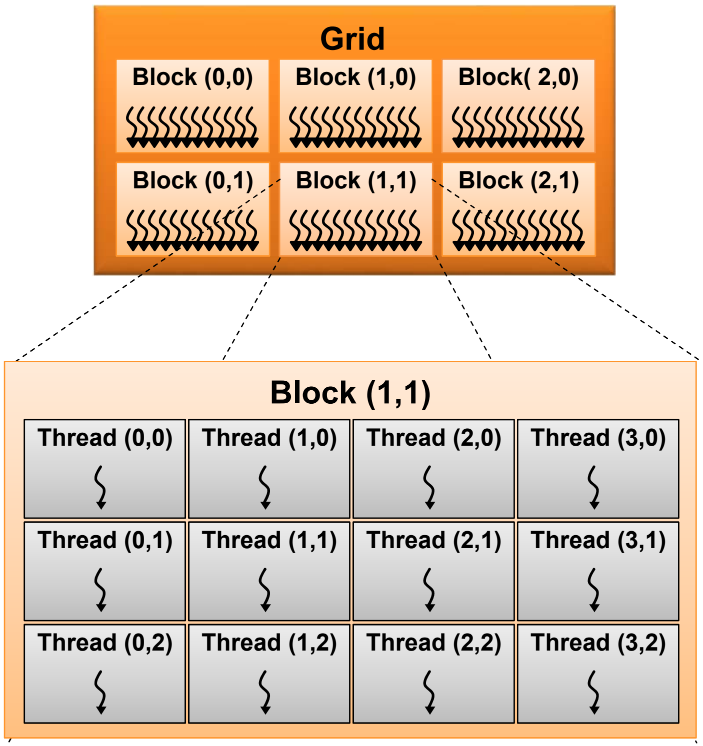
- Each SM has a collection of compute resources:
  - Processors (cores)
  - Registers
  - Specialized memory resources



Streaming Multiprocessors on GPUs

NVIDIA GPU	Number of SMs
Tesla K40	15
Tesla K80	2 x 13
Tesla P100	56
Quadro M3000M	8

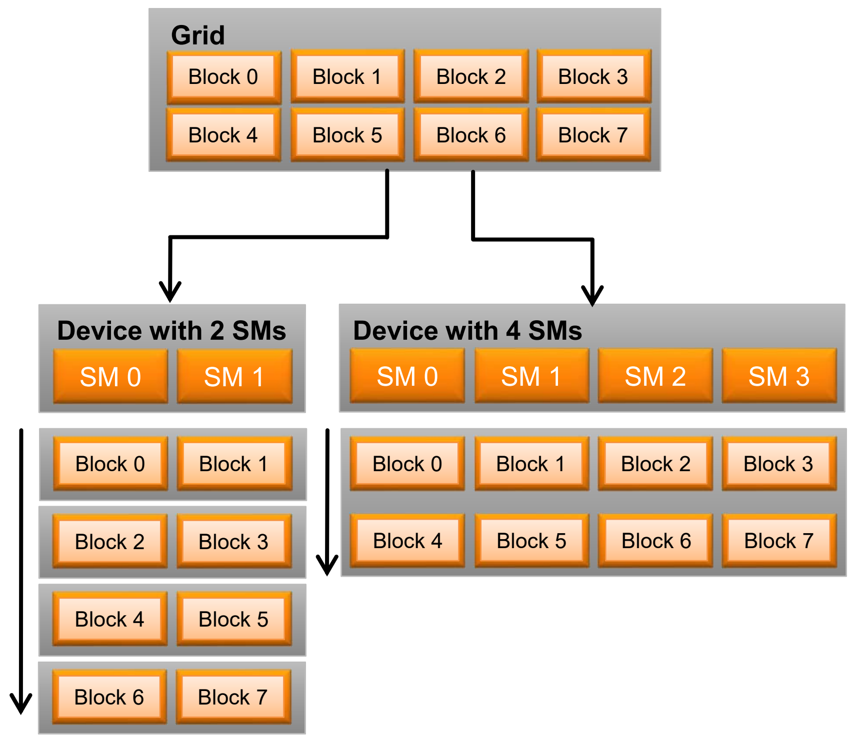
CUDA Thread Hierarchy



Recall: A kernel is executed over a thread hierarchy:

- Threads are grouped into **thread blocks**
- Thread blocks are grouped into a **grid**

The CUDA Programming Model



- Blocks from the grid are distributed across streaming multiprocessors (SMs)
- You (the programmer) have no control over this distribution
- A block will execute on one (and only one) multiprocessor
  - However, a multiprocessor can execute multiple blocks

Blocks Must Be Independent!

- Any possible distribution of blocks could be valid
  - Blocks are presumed to run to completion without pre-emption
  - Can run in any order
  - Can run concurrently ***or*** sequentially
- Blocks can [explicitly] coordinate
  - e.g. Blocks taking work from a queue
- Blocks may not synchronize
  - e.g. barrier synchronization
- Independence requirement gives scalability

The CUDA Programming Model

- Problems must be partitioned into sub-problems, with each sub-problem mapped to blocks
  - Blocks must be independent
  - There is no reliable mechanism to communicate between blocks (because of the order independence)

The CUDA Programming Model

- However, within a block, CUDA permits non data-parallel approaches
  - Implemented via control-flow statements in a kernel
  - Threads are free to execute unique paths through a kernel

- Since all threads within a block are active at the same time they can communicate between each other

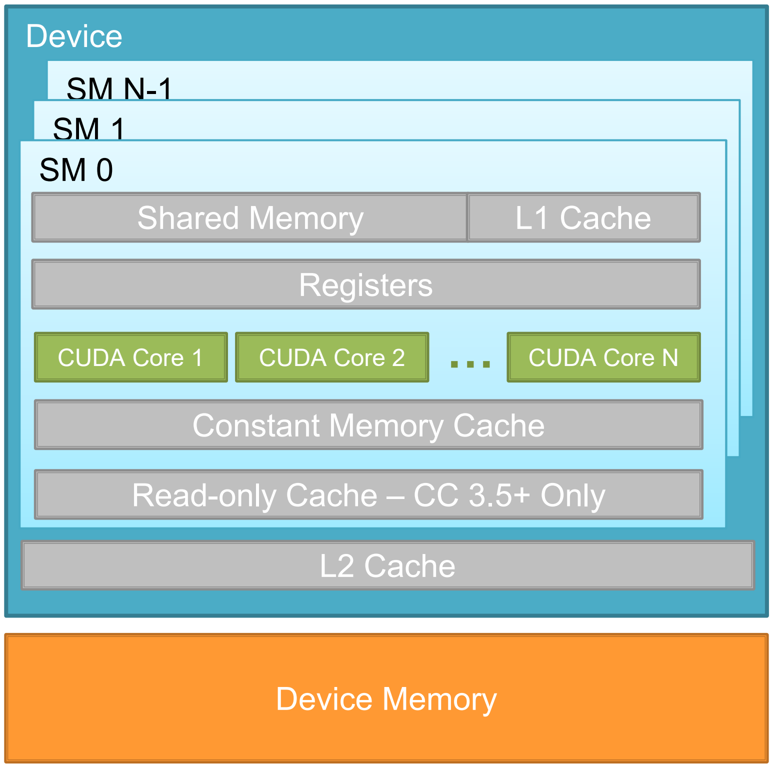
Not Strictly Data-Parallel Kernel

```
__global__ void kernel(int* var)
{
    if(threadIdx.x == 0)
    {
          var[blockIdx.x] = foo();
    }

   // Have all threads wait for thread 0

    // Every thread can now access var
   int temp = var[blockIdx.x] +
                  threadIdx.x;
   …
}
```

This kernel is not strictly data-parallel

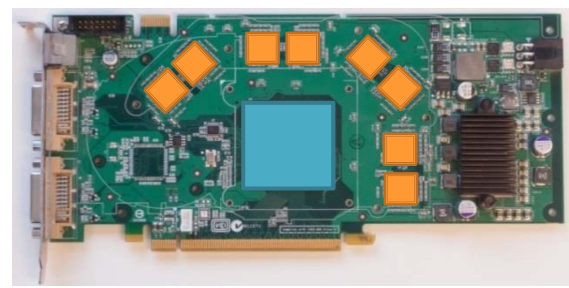
- Thread 0 of each block performs a different task than all other threads

Streaming Multiprocessor Architecture



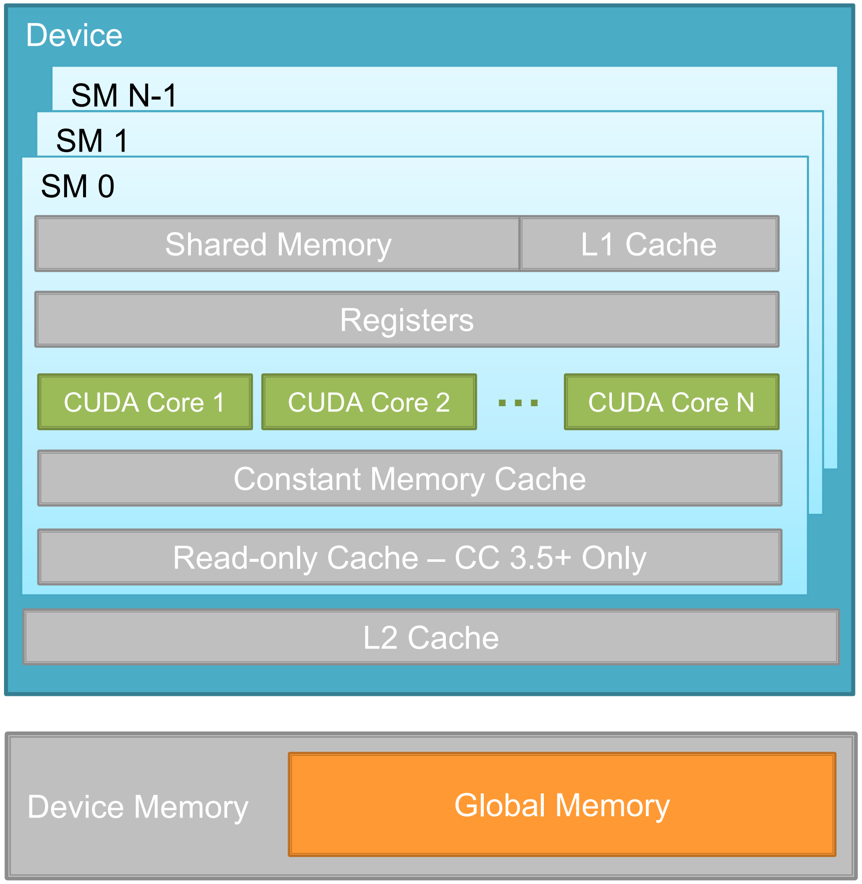
Many memory paths available each with different performance characteristics

- Must map data sets to right memory type
  - Shared memory
  - Registers
  - Constant caches
  - Device memory
  - Read-only Cache



Global Memory

- **Scope:** Visible to all threads and the CPU
- **Lifetime:** Persists between kernel calls within the same application
  - Programmer explicitly manages allocation and deallocation with `cudaMalloc` and `cudaFree`
- **Physical Implementation:** Device memory (HBM2, GDDR5)



Per Thread Memory

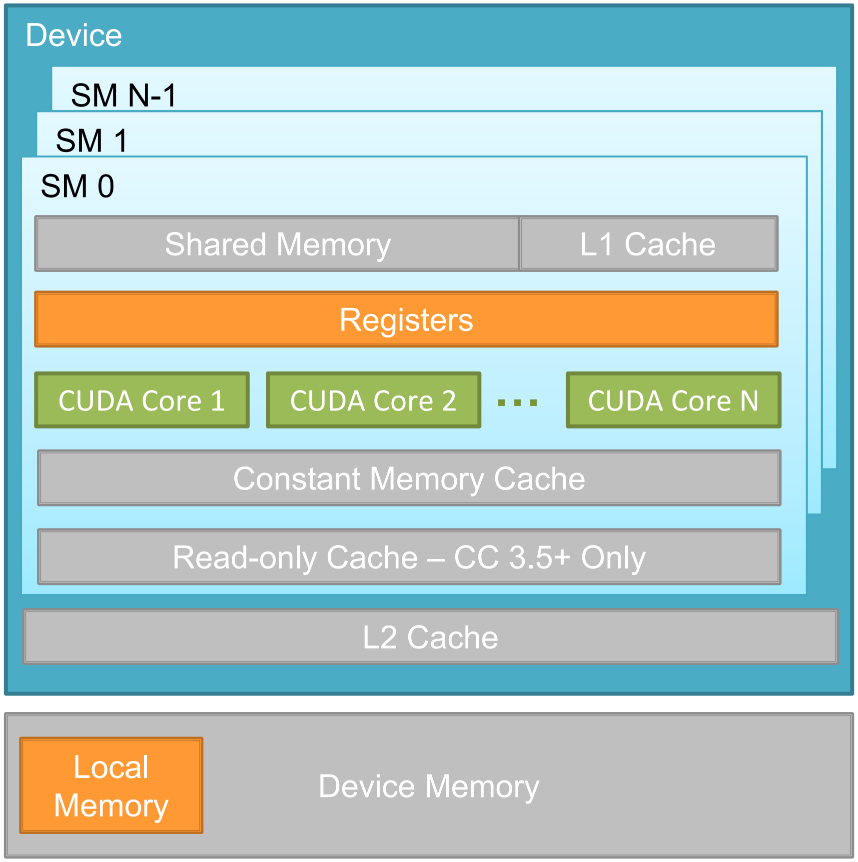
Variables declared within a kernel are allocated per thread

- Is only accessible by the thread
- Has lifetime of the thread

```
__global__ void kernel()
{
   // Each thread has its own copy of idx and array
   int idx = threadIdx.x + blockIdx.x * blockDim.x;
   float array[16];
   ...
}
```

Per Thread Memory

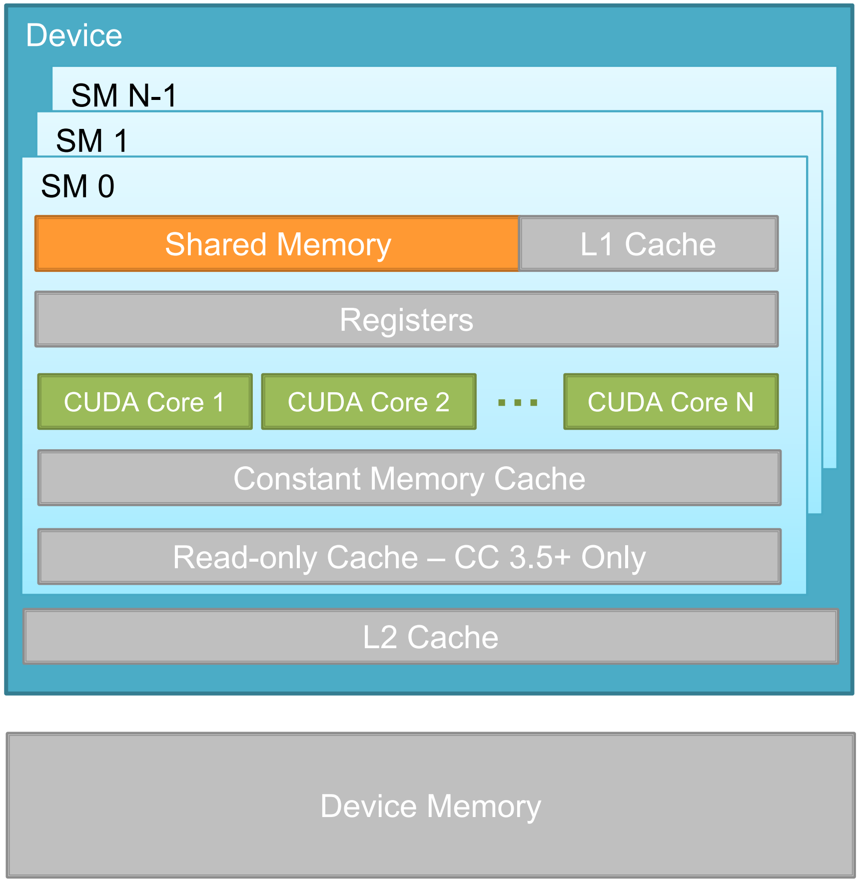
- Compiler controls where these variables are stored in physical memory
  - **Registers** (On-chip):
    - Fastest form of memory on the SM
  - **Local Memory** (Off-chip):
    - Compiler controlled region of device memory for storage of local variables when registers are insufficient or not suitable



Shared Memory

High performance memory

- 2 orders of magnitude lower latency than global memory
- Order of magnitude higher bandwidth than global memory
- Up to 112KB per multiprocessor, but a maximum of 48KB per thread block



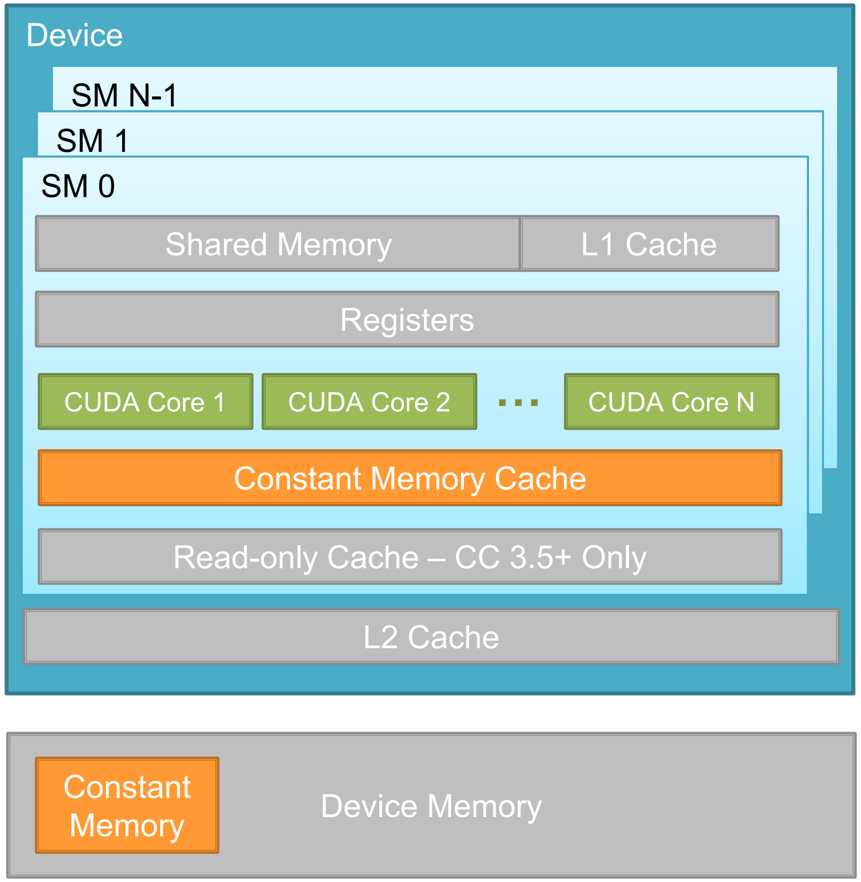
Shared Memory

Shared memory has block scope

- Only visible to threads in the same block
  - Threads can share results
    - Avoids redundant computation
  - Threads can share memory accesses
    - Reduces global memory bandwidth
- Similar benefits as CPU cache, however, must be explicitly managed by programmer

Constant Memory

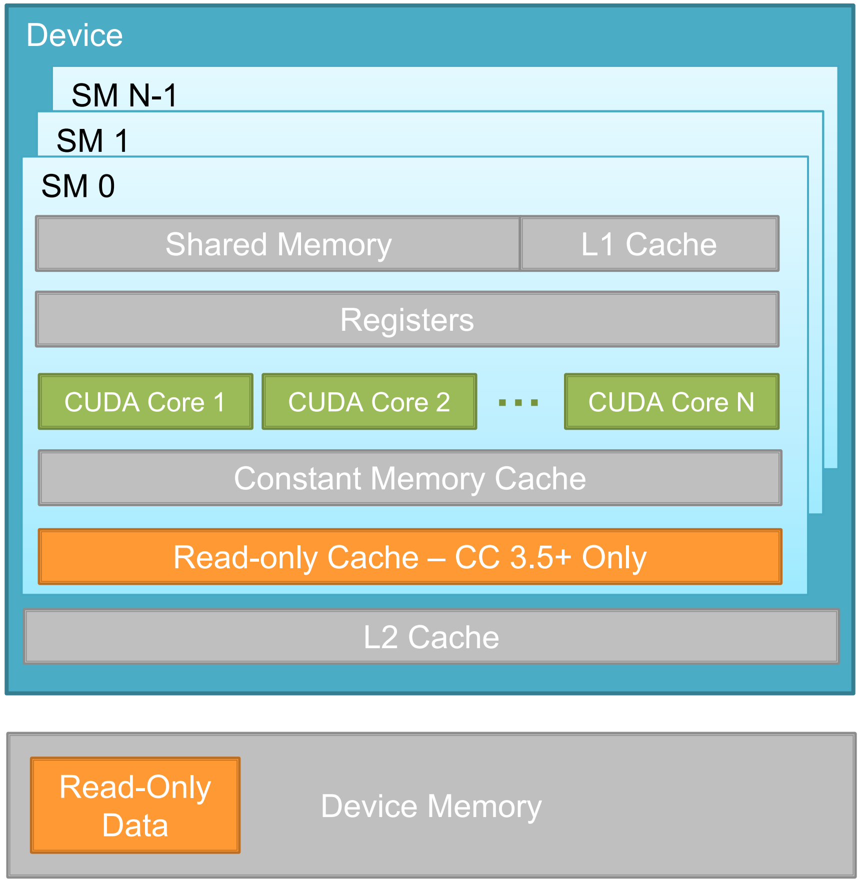
- Special region of device memory
- 64KB
- Read-only from kernel
  - Cached (8KB per multiprocessor)
- Constants are declared at file scope
- Constant values are set from host code
  - cudaMemcpyToSymbol()



Read-Only Cache

- 12KB to 48KB per SM
  - Traditionally the texture cache
- How to Access:
  - Allocate and manage global memory
  - Qualify kernel pointer argument as

    `const __restrict__`



Communication Between Threads?

Is thread to thread communication possible within CUDA?

- No mechanism for reliable communication between threads in different thread blocks
- Threads within a block can communicate via global memory and/or shared memory!
  - Need some sort of synchronization to avoid concurrency hazards

Parallel Computing Concurrency Hazards

- Concurrent processing introduces potential for flaws due to the order in which tasks are executed
  - Eg. race conditions, deadlocks
  - Recent notable examples:
    - Mars Pathfinder mission (priority inversion)
    - 2003 power blackouts in North America (race condition)

Synchronization

- Concurrency hazards are eliminated or avoided through synchronization
- Process synchronization – tasks coordinate execution order to prevent sequence dependent problems
  - Mutual exclusion, barriers, locks, semaphores

Concurrency Hazards in CUDA

- Attempts to communicate between blocks result in undefined behavior
- Communication between threads in the same thread block via shared memory or global memory at risk for concurrency hazards
- `void __syncthreads();`
  - Synchronizes all threads in a block
    - Barrier-type synchronization primitive
    - No thread proceeds until all threads in a block reach the barrier
    - Used to avoid read-after-write (RAW)/WAR/WAW hazards in shared memory
    - Allowed in conditional code *only* if the condition is *uniform* across the thread block (undefined behavior otherwise)

CUDA Memory Model Summary

Memory Space	Managed by	Physical Implementation	Scope on GPU	Scope on CPU	Lifetime
Registers	Compiler	On-chip	Per Thread	Not visible	Lifetime of a thread
Local	Compiler	Device Memory	Per Thread	Not visible	
Shared	Programmer	On-chip	Block	Not visible	Block lifetime
Global	Programmer	Device Memory	All threads	Read/Write	Application or until explicitly freed
Constant	Programmer	Device Memory	All threads Read-only	Read/Write	

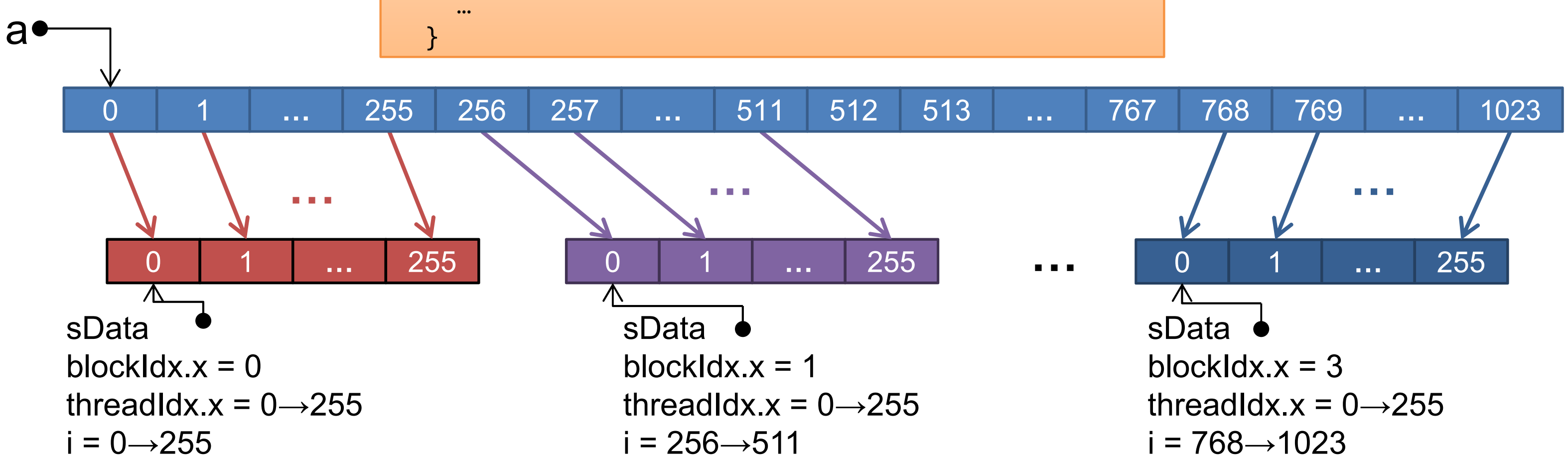
CUDA Syntax - Shared Memory

```
//  Static shared memory syntax

#define BLOCK_SIZE 256

__global__ void kernel(float* a)
{
  __shared__ float sData[BLOCK_SIZE];
  int i;
  i = blockIdx.x * blockDim.x + threadIdx.x;
  sData[threadIdx.x] = a[i];
  __syncthreads();
  …
  a[i] = sData[blockDim.x – 1 - threadIdx.x];
}

int main(void)
{
  …
  kernel<<<nBlocks, BLOCK_SIZE>>>(…);
  …
}
```

- `__shared__` qualifier used to declare variables/arrays in shared memory
- Shared memory is not visible from host
- Threads read/write to shared memory just like any other variable/array

CUDA Syntax – Shared Memory (2)

```
#define BLOCK_SIZE 256_
_global__ void kernel(float* a)
{
   __shared__ float sData[BLOCK_SIZE];
   int i;
   i = blockIdx.x * blockDim.x + threadIdx.x;
   sData[threadIdx.x] = a[i];
   __syncthreads();
   …
}
```



Shared Memory Syntax

```
// Deferred allocation of shared memory

__global__ void kernel(int sizeA, …)
{
   …
   extern __shared__ float sData[];
   float* a, float* b;

   a = sData;
   b = &a[sizeA];
   …
}

int main(void)
{
   int sizeA = 64;
   int sizeB = 16;
   int smBytes = (sizeA + sizeB) *sizeof(float);

   kernel<<<nBlocks, bSize, smBytes>>>(…);
   …
}
```

Deferred allocation is possible, however:

- Only one `extern __shared__` per kernel
  - Manual offsets if you want to logically subdivide dynamically allocated shared memory
- Specify size of extern allocation from host, as 3rd argument to kernel launch <<< >>> construct

CUDA Syntax – Constant Memory

- `__constant__` qualifier used to declare variables (including arrays) as constant memory-resident
- Constant variables may be written/read from host code, but are read-only from kernels

```
__constant__ float staticCoeff = 1.0f;
__constant__ float runtimeCoeff;
__constant__ float runtimeArray[5];

__global__ void kernel(float *array)
{
     array[threadIdx.x] += staticCoeff;
     array[threadIdx.x] *= runtimeCoeff;
     array[threadIdx.x]  = runtimeArray[0];
}

int main(void)
{
   float val = calculateCoefficient();
   cudaMemcpyToSymbol(runtimeCoeff, &val,
                      sizeof(val));
   …
   cudaMemcpyToSymbol(runtimeArray,
                      hostArray,
                      5*sizeof(float));

   kernel<<<gSize,bSize>>>(…);
   …
}
```

Acceleware CUDA Training

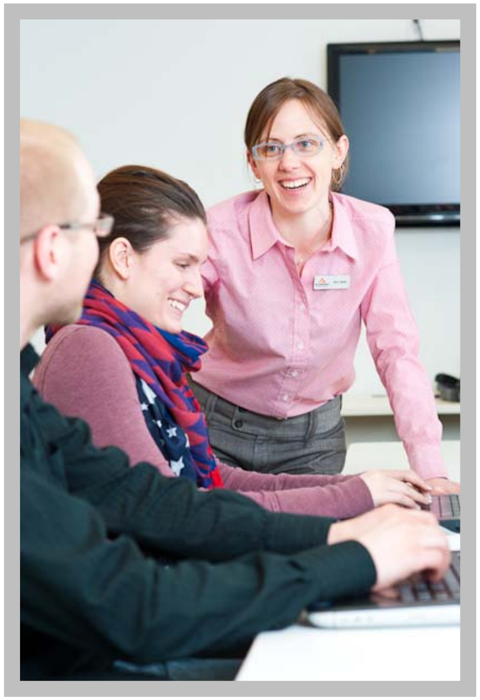
Scheduled CUDA Courses (also available online)

- June 13 – 16: Calgary, Alberta
  - 35% Discount using code: **AXECUDAGTC17**
- September 12 – 15: Calgary, Alberta
- December 5 – 8: Calgary, Alberta

Private training courses

- Courses held onsite at your company
- Delivered anywhere in the world

http://acceleware.com/cuda-training



Questions?

Visit us at booth #520

Acceleware Ltd.
Tel: +1 403.249.9099
Email: **services@acceleware.com**

CUDA Blog: **http://acceleware.com/blog**
Website: **http://acceleware.com**

Chris Mason
chris.mason@acceleware.com