



An Introduction to CUDA Programming

S7699 – Session 1 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™

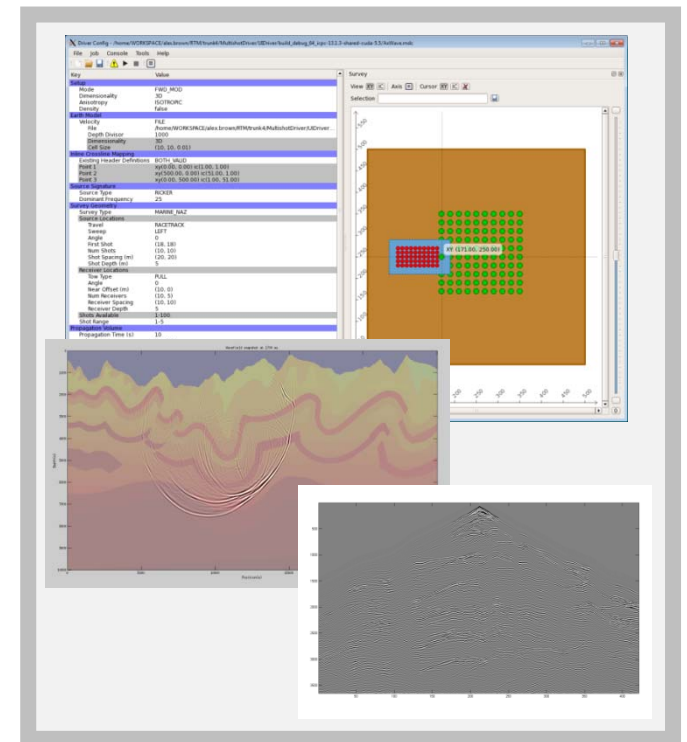
- High performance Reverse Time Migration application
- Isotropic, VTI and TTI media

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

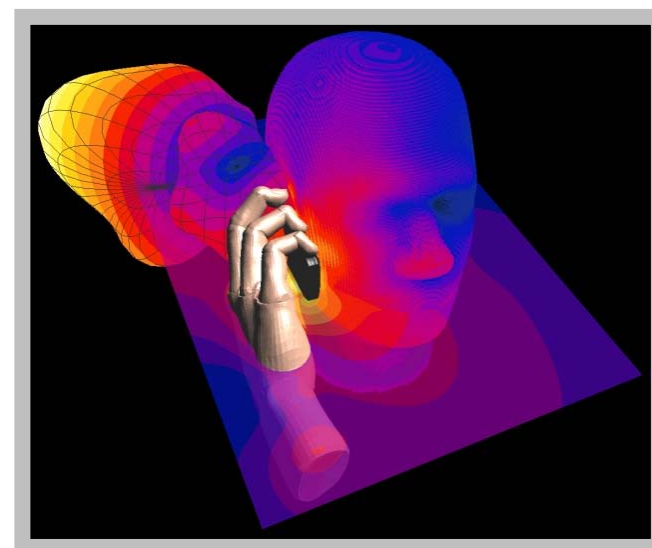
AxFDTD™

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

Available from:



Agilent Technologies



Consulting Services

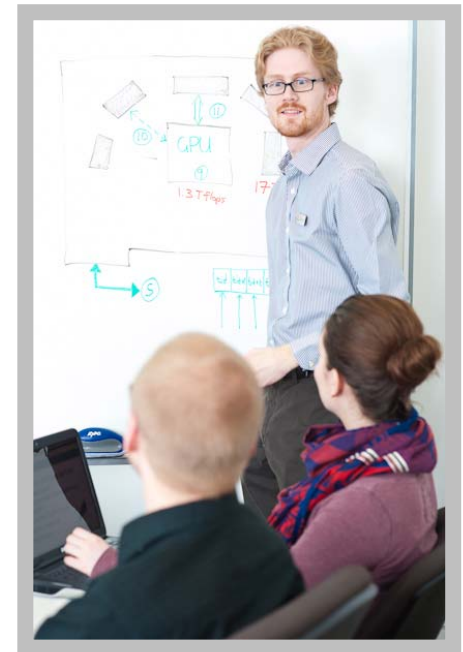
Industry	Application	Work Completed	Results
Finance	Option Pricing	Debugged & optimized existing CUDA code Implemented the Leisen-Reimer version of the binomial model for stock option pricing	30-50x performance improvement compared to single-threaded CPU code
Security & Defense	Detection System	Replaced 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

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



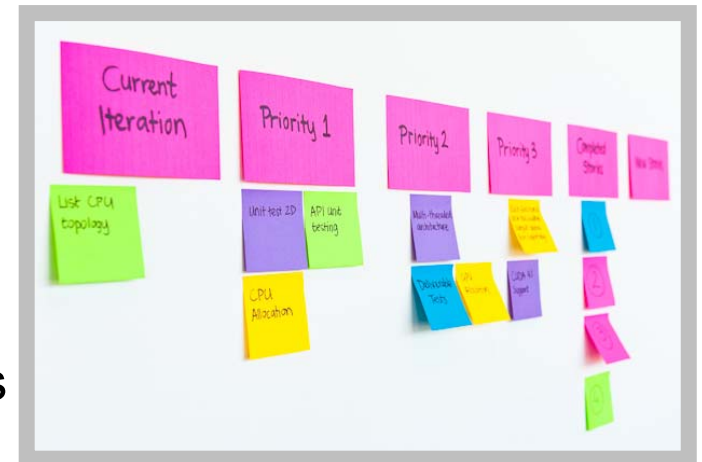
Outline

CUDA overview

Data-parallelism

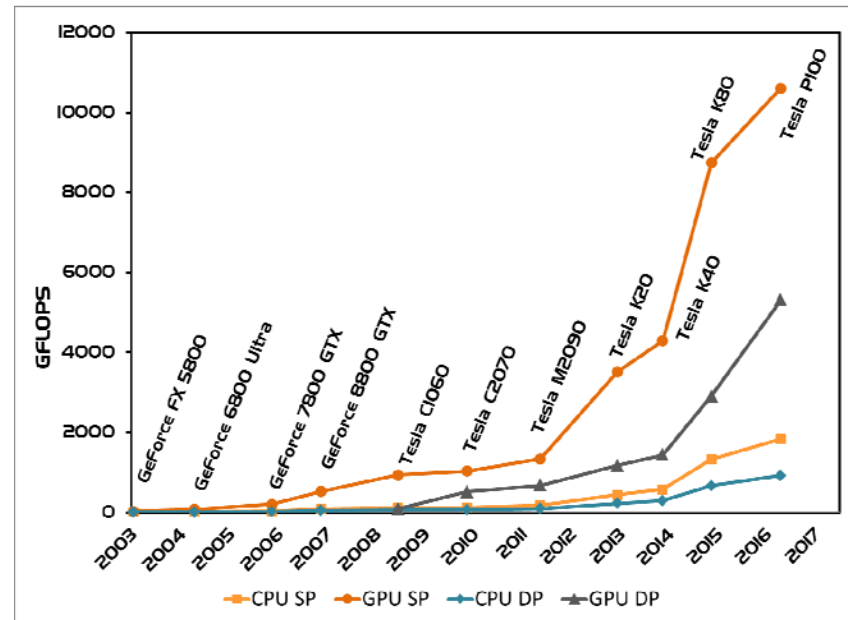
GPU programming model

- GPU kernels
- Host vs. device responsibilities
- CUDA syntax
- Thread hierarchy








Why use GPUs? Performance!

GPU advances are outpacing CPU advances
Continuing Moore's Law?



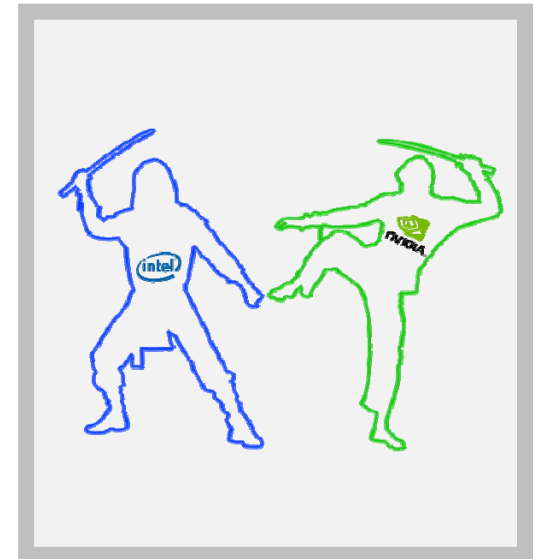
Why use GPUs? Performance!

	Intel Xeon E5-2699v4 (Broadwell-EP) 	NVIDIA Tesla K80 (Kepler) 	NVIDIA Tesla M60 (Kepler) 	NVIDIA Tesla P100 (Pascal) 	NVIDIA Jetson TX2 (Pascal) 
Processing Cores	22	4992	4096	3584	8 ARM + 256 Pascal
Clock Frequency	2.2-3.6GHz	0.562-0.875GHz	0.900-1.180GHz	1.328-1.48GHz	0.854 – 1.465GHz
Memory Bandwidth	76.8 GB/s / socket	480GB/s	320GB/s	720GB/s	58.4GB/s
Peak Tflops (single)	1.83 @ 2.6GHz	8.74 @ 0.875GHz	9.68 @ 1.180GHz	10.6 @ 1.48GHz	0.75@1.465GHz
Peak Tflops (double)	0.915 @ 2.6GHz	2.91 @ 0.875GHz	0.30 @ 1.180GHz	5.3 @ 1.48GHz	0.023@1.465GHz
Gflops/Watt (single)	12.62	29.1	32.2	35.3	50
Total Memory	>>24GB	24GB	16GB	16GB	8GB

GPU Potential Advantages

Tesla P100 vs. Xeon E5-2699 v4

- 5.8x more single-precision floating-point throughput
- 5.8x more double-precision floating-point throughput
- 9.4x higher memory bandwidth

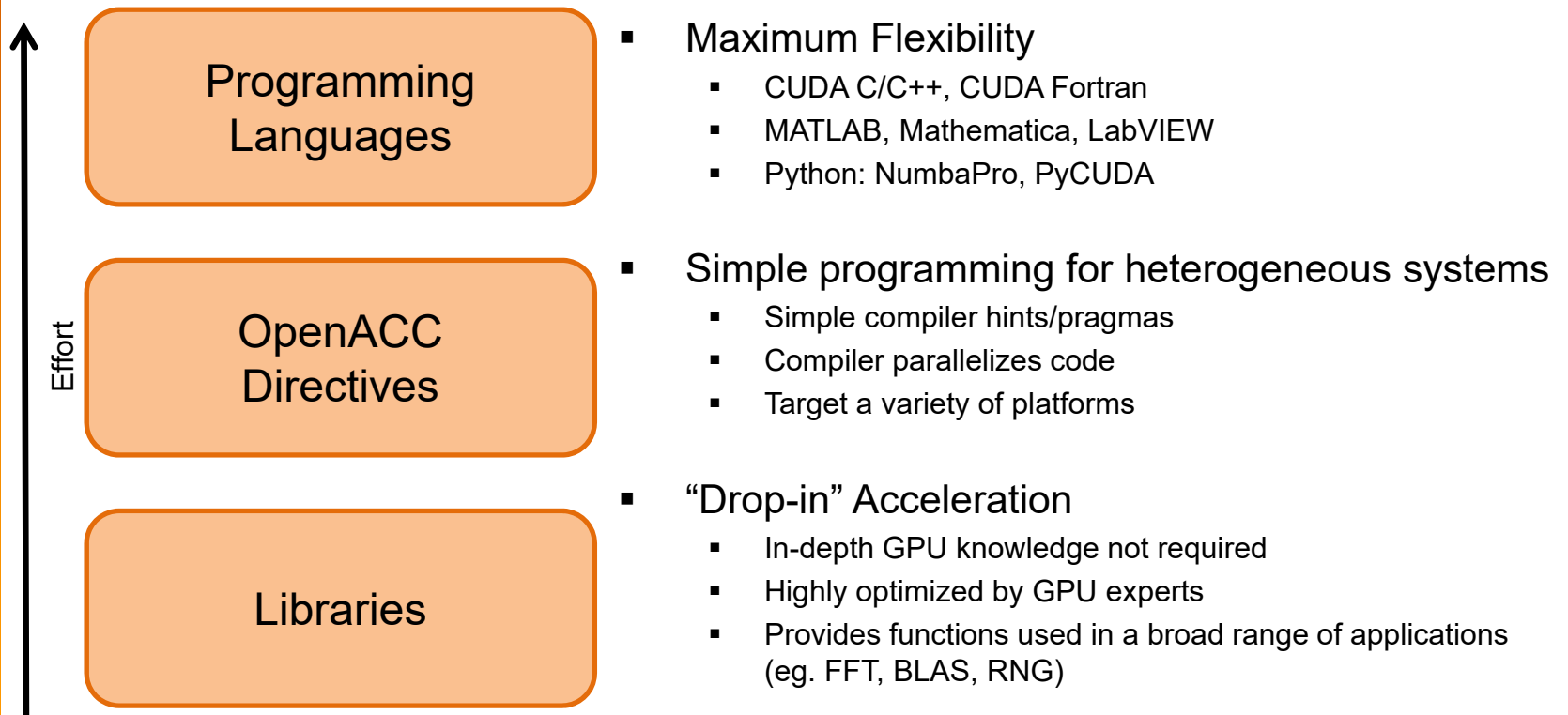


GPU Disadvantages

- Architecture not as flexible as CPU
- Must rewrite algorithms and maintain software in GPU languages
- Discrete GPUs attached to CPU via relatively slow PCIe
 - 32GB/s bi-directional for PCIe 3.0 16x
 - 40GB/s bi-directional for NVLink
- Limited memory (though 8-24GB is reasonable for many applications)



Software Approaches for Acceleration



Compute Capability

Hardware architecture version number

Defined by a major and minor version number

- Major version specifies core architecture type
- Minor version refers to incremental improvements and new features

Architecture	Compute Capability	GPUs	Example Features
Tesla	1.0	GeForce 8800, Tesla C870	Base Functionality
Fermi	2.0	GeForce GTX 480, Tesla C2050	Fast Double Precision, Memory Caches
Kepler	3.0	GeForce GTX 680, Tesla K10	Warp Shuffle Functions
	3.5	GeForce GTX Titan Black, Tesla K40	Dynamic Parallelism
	3.7	Tesla K80	More Registers / Shared Memory
Maxwell	5.0	GeForce GTX 750 Ti, Tegra X1	Power Efficient Architecture
	5.2	Tesla M40, Tesla M60	More Shared Memory
Pascal	6.0	Tesla P100	Half Precision (FP16)
	6.1	Titan Xp	Int8
	6.2	Tegra P1	Int8 + FP16

CUDA Overview

- Compute Unified Device Architecture (CUDA) – Parallel computing platform and programming model architecture developed by NVIDIA
 - Libraries, OpenACC, and programming languages built on top of CUDA
- CUDA C/C++ programming interface consists of:
 - C language extensions to target portions of source code for parallel execution on the device (GPU)
 - A library of C functions that execute on the host (CPU) to interact with the device



Data-Parallel Computing

1. Performs operations on a data set organized into a common structure (eg. an array)
2. A set of **tasks** work collectively and simultaneously 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 **be data independent!**

Data Dependence

- Data dependence occurs when a program statement refers to the data of a preceding statement

```
a = 2 * x;  
b = 2 * y;  
c = 3 * x;
```

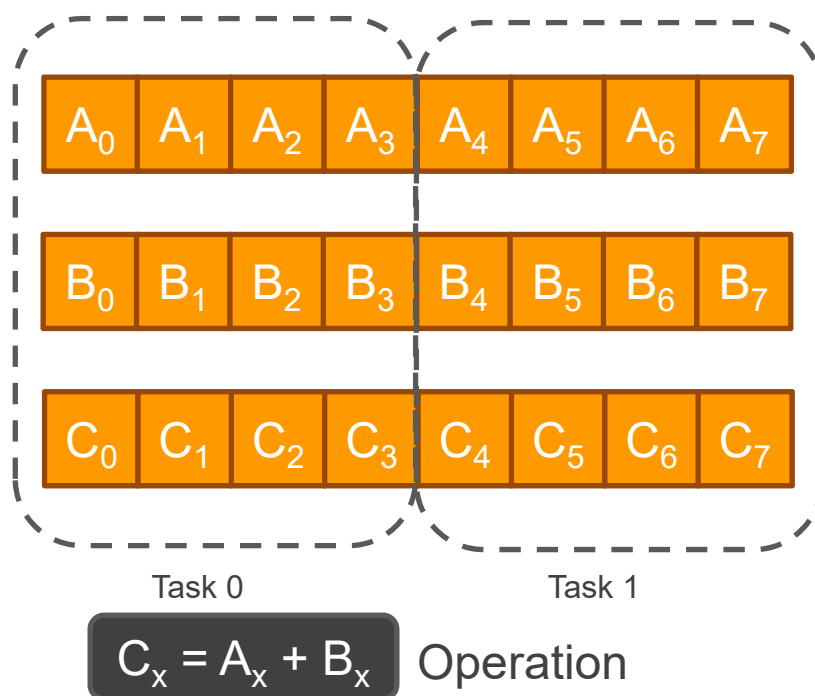
These 3 statements are
independent!

```
a = 2 * x;  
b = 2 * a * a;  
c = b * 9;
```

b depends on *a*, *c* depends
on *b* and *a*!

- Data dependence limits parallelism

Data-Parallel Computing Example

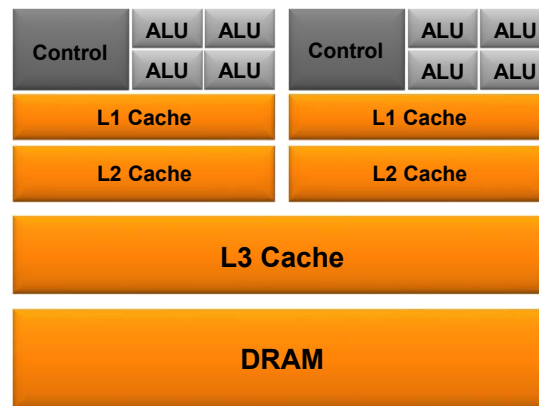


- Data set consisting of arrays A, B, and C
- Same operations performed on each element
 $C_x = A_x + B_x$
- Two tasks operating on a subset of the arrays. Tasks 0 and 1 are independent. Could have more tasks.

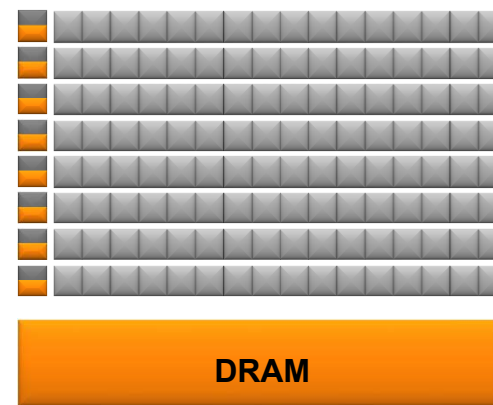
Data-Parallel Computing on GPUs

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



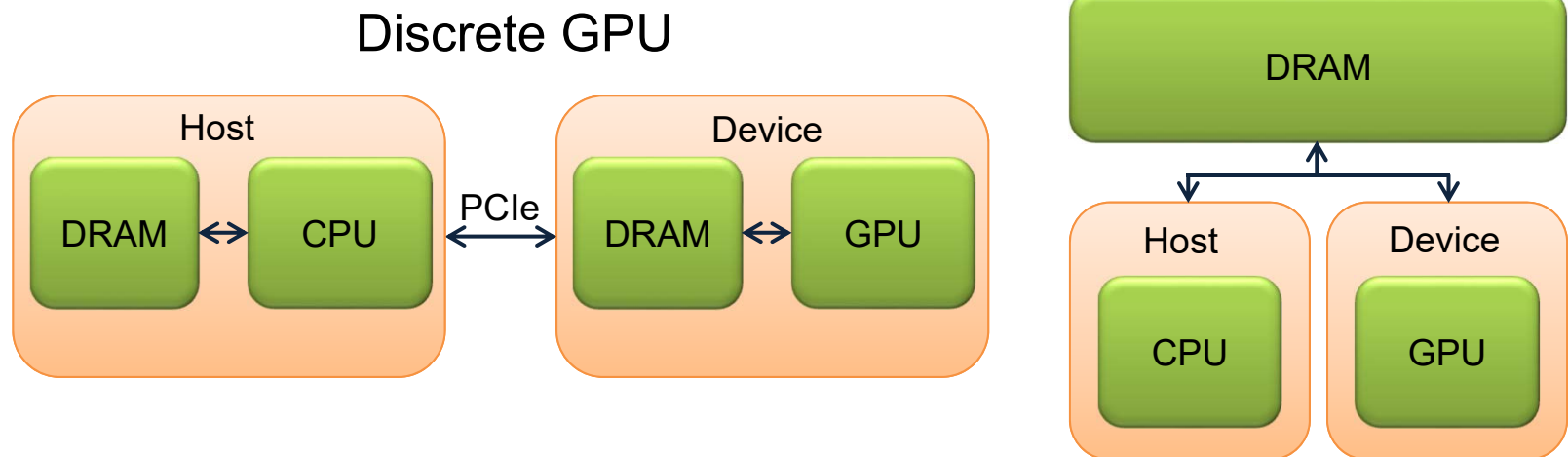
CPU



GPU

The CUDA Programming Model

CUDA is a heterogeneous model, including provisions for both **host** and **device**



The CUDA Programming Model

- Data-parallel portions of an algorithm are executed on the device as **kernels**
 - Kernels are C/C++ functions with some restrictions, and a few language extensions
- Only one kernel is executed at a time
 - Newer GPU architectures relax this restriction
- Each kernel is executed by many **threads**

CUDA Threads

- CUDA threads are conceptually similar to data-parallel tasks
 - Each thread performs the same operations on a subset of a data structure
 - Threads execute independently
- CUDA threads *are not* CPU threads
 - CUDA threads are extremely lightweight
 - Little creation overhead
 - Instant context-switching
- CUDA threads *must* execute the same kernel

CUDA Thread Hierarchy

- CUDA is designed to execute 1000s of threads
- Threads are grouped together into **thread blocks**
- Thread blocks are grouped together into a **grid**

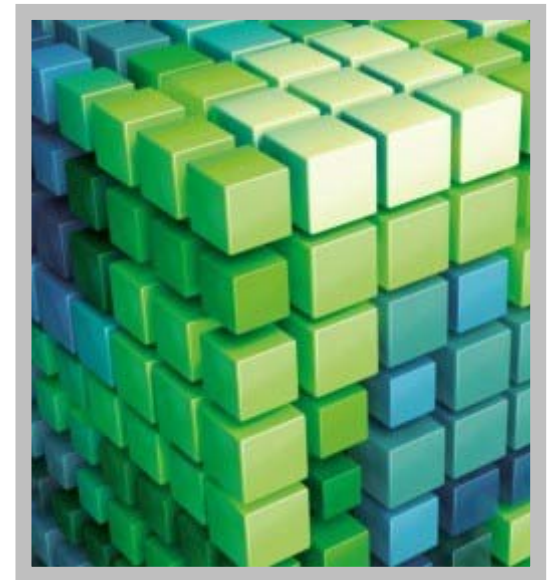
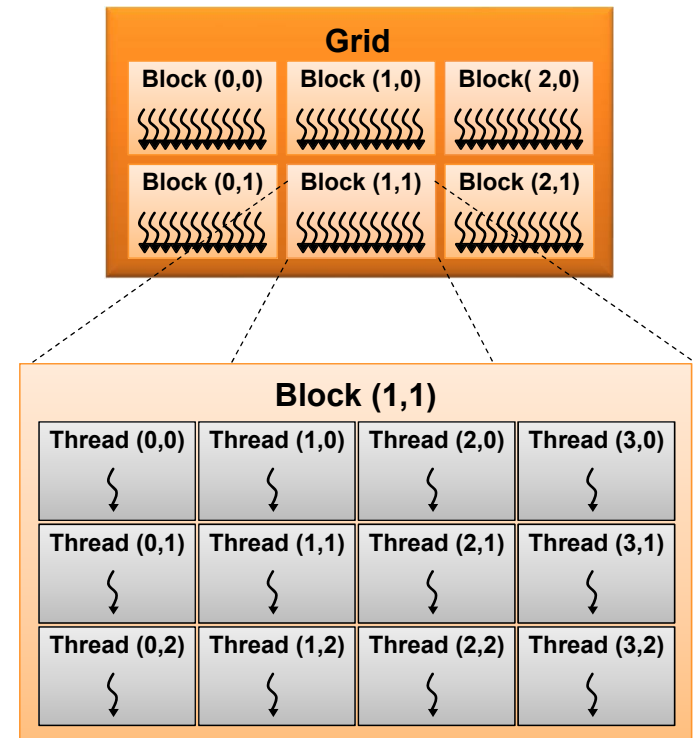


Image courtesy of NVIDIA Corp.

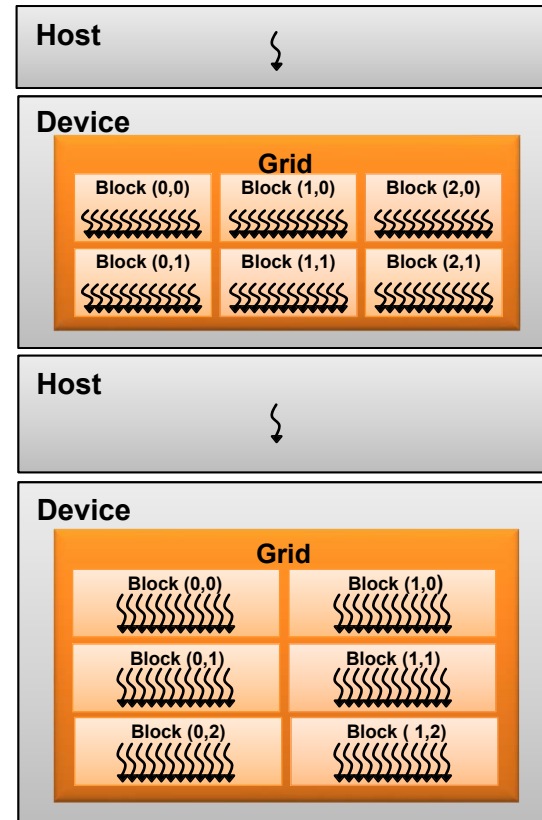
CUDA Thread Hierarchy

- Thread blocks and Grids can be 1D, 2D or 3D
- Dimensions set at launch time
- Thread blocks and grids do not need to have the same dimensionality
 - ie. 1D Grid of 2D Thread Blocks



The CUDA Programming Model

- The host launches kernels
- The host executes serial code between device kernel launches
 - Memory management
 - Data exchange to/from device
 - Error handling



CUDA APIs

Can use CUDA through CUDA C (Runtime API) and low-level Driver API

- **The tutorial presentations use CUDA C**
 - Uses host side C-extensions that greatly simplifies host code
- Driver API requires explicit resource management and more verbose syntax
- CUDA C is built on top of Driver API
 - Exposure of Driver API is for historical reasons
 - **No reason to start new development using Driver API**
- Don't confuse the two when referring to CUDA Documentation
 - `cuFunctionName()` – Driver API
 - `cudaFunctionName()` – Runtime API

CUDA Kernel Launch Syntax

- CUDA kernels are launched by the host using a modified C function call syntax:

```
myKernel<<<dim3 dGrid, dim3 dBlock>>>(...)
```

dim3 is vector type with x, y, and z components (eg. dG.x)

Maximum Values For Each Dimension

		Compute Capability		
		2.x (Fermi)	3.x (Kepler) 5.x (Maxwell)	6.x (Pascal)
Total Threads per Block		1024	1024	1024
Grid Size	dGrid.x	65535	$2^{31}-1$	$2^{31}-1$
	dGrid.y	65535	65535	65535
	dGrid.z	65535	65535	65535
Block Size	dBlock.x	1024	1024	1024
	dBlock.y	1024	1024	1024
	dBlock.z	64	64	64

CUDA Kernels

- Denoted by `__global__` function qualifier
 - Eg. `__global__ void myKernel(float* a)`
- Called from host, executed on device
- A few noteworthy restrictions:
 - No access to host memory (in general!)
 - Must return *void*
 - No static variables
 - No access to host functions

CUDA Syntax – Kernels (I)

Kernels can take arguments just like any C/C++ function

- Pointers to device memory
- Parameters passed by value

```
__global__ void SimpleKernel(float* a, float b)
{
    a[0] = b;
}
```

CUDA Syntax – Kernels (II)

Kernels must be declared (but not necessarily defined) in source/header files before they are called

```
// Kernel declaration
__global__ void kernel(float* a);

int main()
{
    dim3 gridSize, blockSize;
    ...
    kernel<<<gridSize,blockSize>>>(a);
}

__global__ void kernel(float* a)
{
    ...
}
```


CUDA Syntax - Kernels

Kernels have read-only built-in variables:

- `gridDim`: dimensions of the grid
 - Uniform for all threads
- `blockIdx`: unique index of a block within grid
- `blockDim`: dimensions of the block
 - Uniform for all threads
- `threadIdx`: unique index of the thread within the thread block
- Cannot vary the size of blocks or grids during a kernel call

CUDA Syntax - Kernels

Built-in variables are typically used to compute unique thread identifiers

- Map local thread ID to a global array index

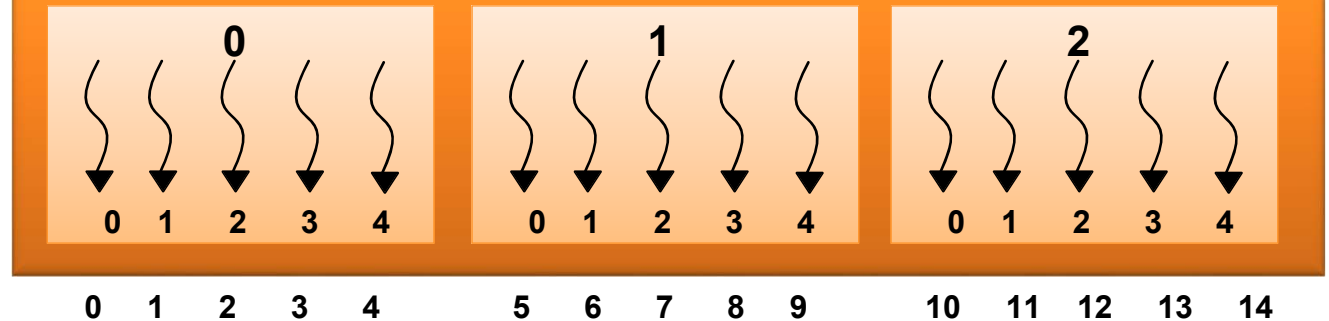
`myKernel<<<3,5>>>(...)`

`blockDim.x = 5`
`gridDim.x = 3`

`blockIdx.x`

`threadIdx.x`

Grid



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

CUDA Syntax – Index & Size Calculations

- Global index calculation
 - $idx = blockIdx.x * blockDim.x + threadIdx.x$
- Grid size calculation

$$GridSize = \frac{Size + BlkDim - 1}{BlkDim} \leftarrow \text{Integer Division}$$

- Where
 - Size: Total size of the array
 - BlkDim: Size of the block (max 1024)
 - GridSize: Number of blocks in the grid

CUDA Syntax – Thread Identifiers

Result for each kernel launched with the following execution configuration:

`MyKernel<<<3,4>>>(a);`

```
__global__ void MyKernel(int* a)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    a[idx] = 7;
}

__global__ void MyKernel(int* a)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    a[idx] = blockIdx.x;
}

__global__ void MyKernel(int* a)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    a[idx] = threadIdx.x;
}
```

a: 7 7 7 7 7 7 7 7 7 7 7 7

a: 0 0 0 0 1 1 1 1 2 2 2 2

a: 0 1 2 3 0 1 2 3 0 1 2 3

CUDA Syntax - Kernels

- All C operators are supported
 - eg. +, *, /, ^, >, >>
- All functions from the standard math library
 - eg. sinf(), cosf(), ceilf(), fabsf()
- Control flow statements too!
 - eg. if(), while(), for()

CUDA Kernel C++ Support

- Supported
 - Classes
 - Including inheritance and virtual functions
 - Need to add `__device__` qualifiers to member functions!
 - Templates
 - C++11 features including auto and lambda functions
- Not supported
 - C++ Standard Library
 - Run time type information (RTTI)
 - Exception handling
 - Classes with virtual functions are not binary compatible between host and device

User-defined Device Functions

- Can write/call your own device functions

- `__device__ float myDeviceFunction()`
- Device functions cannot be called by host

```
__device__ float myDeviceFunction(int i)
{
    ...
}

__global__ void myKernel(float* a)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    a[idx] = myDeviceFunction(idx);
}
```

- Functions declared with both `__device__` and `__host__` will be compiled for both the CPU and GPU

CUDA Syntax - Memory Management

- Typically, host code manages device memory:
 - `cudaMalloc(void** pointer, size_t nbytes)`
 - `cudaMemset(void* pointer, int value, size_t count)`
 - `cudaFree(void* pointer)`

```
// Memory allocation example  
  
int n = 1024;  
int nBytes = 1024*sizeof(int);  
int* a = 0;  
cudaMalloc((void**)&a, nbytes);  
cudaMemset( a, 0, nbytes);  
cudaFree(a);
```

CUDA Syntax – Memory Spaces

- Host and device have separate memory spaces
 - For discrete GPUs data is moved between them via PCIe/NVLink bus
- Pointers are just addresses
 - Can't tell from the pointer value whether the address is on device or host
 - Must exercise caution when dereferencing pointers
 - Dereferencing host pointers on device likely crashed, and vice versa



CUDA Syntax – Data Transfers

- Host code manages data transfers to and from the device:
 - `cudaMemcpy(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind direction);`
 - Direction is one of:
 - `cudaMemcpyHostToDevice`
 - `cudaMemcpyDeviceToHost`
 - `cudaMemcpyDeviceToDevice`
 - `cudaMemcpyHostToHost`
 - `cudaMemcpyDefault`
 - Blocking call - returns once copy is complete
 - Waits for all outstanding CUDA calls to complete before starting transfer
 - With `cudaMemcpyDefault`, runtime determines which way to copy data

CUDA Syntax - Synchronization

- Kernel launches are asynchronous
 - Control returns to CPU immediately
 - Kernel starts executing once all outstanding CUDA calls are complete
- `cudaMemcpy()` is synchronous
 - Blocks until copy is complete
 - Copy starts once all outstanding CUDA calls are complete
- `cudaDeviceSynchronize()`
 - Blocks until all outstanding CUDA calls are complete

```
cudaMemcpy(..., cudaMemcpyHostToDevice);  
  
// Data is on the GPU at this point  
  
MyKernel<<<...>>>(...);  
  
// Kernel is launched but  
// not necessarily complete  
  
cudaMemcpy(..., cudaMemcpyDeviceToHost);  
// CPU waits until kernel is complete  
// and then transfers data  
  
// Data is on the CPU at this point
```

CUDA Syntax – Error Management

- Host code manages errors
- Most CUDA function calls return `cudaError_t`
 - Enumeration type
 - `cudaSuccess` (value 0) indicates no errors
- `char* cudaGetErrorString(cudaError_t err)`
 - Returns a string describing the error condition

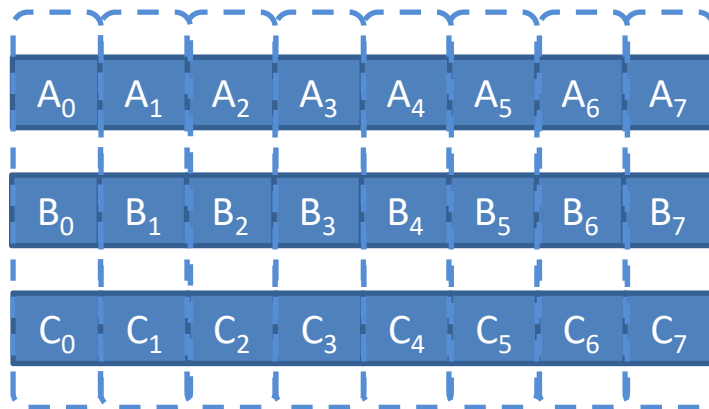
```
cudaError_t e;  
e = cudaMemcpy(...);  
if(e)  
    printf("Error: %s\n", cudaGetErrorString(err));
```

CUDA Syntax – Error Management

- Kernel launches have no return value!
- `cudaError_t cudaGetLastError()`
 - Returns error code for last CUDA runtime function (including kernel launches)
 - Resets global error state to `cudaSuccess`
 - In case of multiple errors, only the last one is reported
 - For kernels:
 - Asynchronous, must call `cudaDeviceSynchronize()` first, then `cudaGetLastError()`

```
MyKernel<<< ... >>> (...);  
  
cudaDeviceSynchronize();  
e = cudaGetLastError();
```

Putting It All Together



One Thread per Output

$$C_x = A_x + B_x \quad \text{Operation}$$

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

This kernel assumes that the size of the array fits evenly into the block size.

What happens if does not?

Vector Add – Host Code

```
void VectorAdd(float* aH, float* bH, float* cH, int N)
{
    float* aD, *bD, *cD;
    int N_BYTES = N * sizeof(float);
    dim3 blockSize, gridSize;

    cudaMalloc((void**)&aD, N_BYTES);
    cudaMalloc((void**)&bD, N_BYTES);
    cudaMalloc((void**)&cD, N_BYTES);

    cudaMemcpy(aD, aH, N_BYTES, cudaMemcpyHostToDevice);
    cudaMemcpy(bD, bH, N_BYTES, cudaMemcpyHostToDevice);

    blockSize.x = 512;
    gridSize.x = N / blockSize.x;
    VectorAddKernel<<<gridSize, blockSize>>>>(aD, bD, cD);

    cudaMemcpy(cH, cD, N_BYTES, cudaMemcpyDeviceToHost);
}
```

This code assumes N
is a multiple of 512

Allocate memory on
GPU

Transfer input
arrays to GPU

Launch kernel

Transfer output
array to CPU

CUDA Syntax – Unified Memory

- Instead of explicitly declaring memory for the host and the device, use managed memory

```
cudaMallocManaged(void **devPtr, size_t size)
```

```
// Memory allocation example  
int n = 1024;  
int nBytes = 1024*sizeof(int);  
int* a = 0;  
cudaMallocManaged((void**)&a, nbytes);  
cudaFree(a);
```

Putting It All Together... Again!

```
int* a, *b, *c;  
int N_BYTES = 2 * sizeof(int);
```

```
cudaMallocManaged((void**)&a, N_BYTES);  
cudaMallocManaged((void**)&b, N_BYTES);  
cudaMallocManaged((void**)&c, N_BYTES);
```



Allocate managed
memory

```
a[0] = 5; b[0] = 7;  
a[1] = 3; b[1] = 4;
```



Initializing memory from
host

```
VectorAddKernel<<<1,2>>>(a, b, c);  
cudaDeviceSynchronize();
```



Launch kernel and
synchronize device

```
printf("%d %d\n", c[0], c[1]);
```

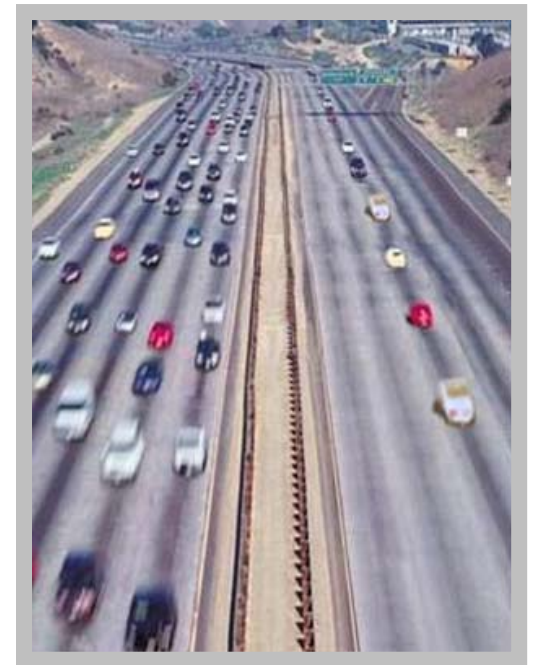
Summary

GPUs are data-parallel architectures

CUDA provides a heterogeneous compute model:

- Host:
 - Memory management (usually)
 - Data transfers
 - Data-parallel kernel launches on device as a grid of thread blocks
 - Error management
- Device (GPU):
 - Executes data-parallel kernels in threads
 - Implemented in C/C++ with a few important extensions, and a few restrictions

Unified memory simplifies transfers and memory management



Acceleware CUDA Training

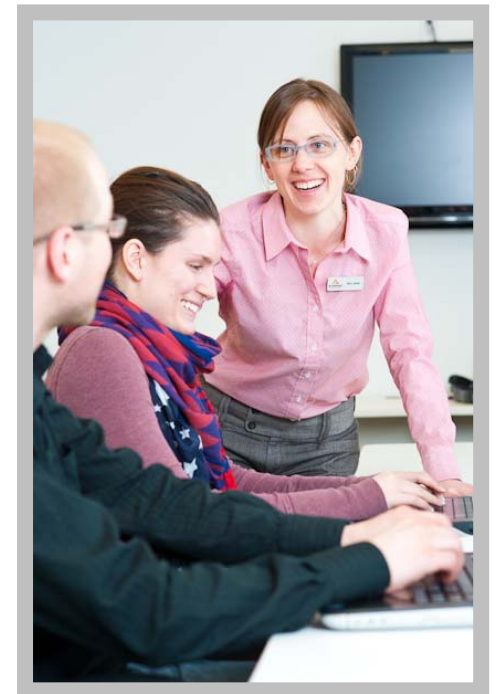
Scheduled CUDA Courses (also available online)

- 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



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™

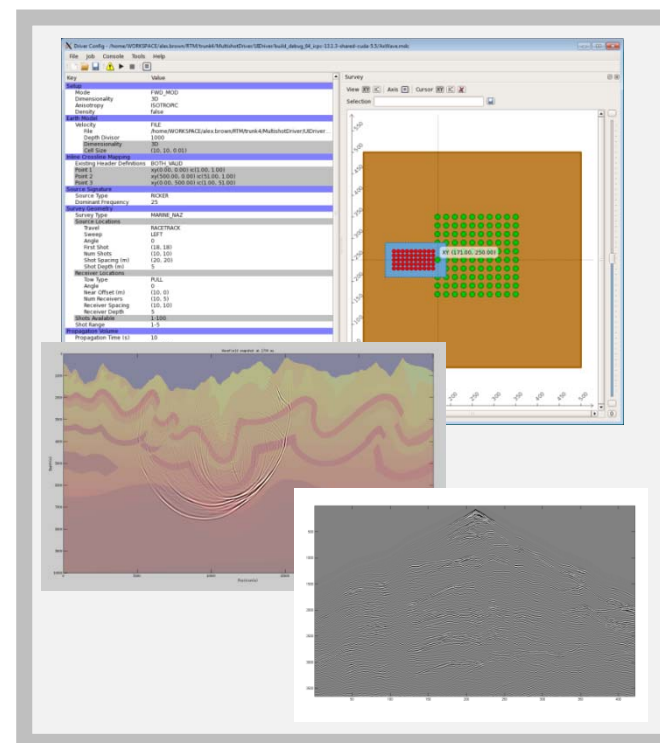
- High performance Reverse Time Migration application
- Isotropic, VTI and TTI media

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

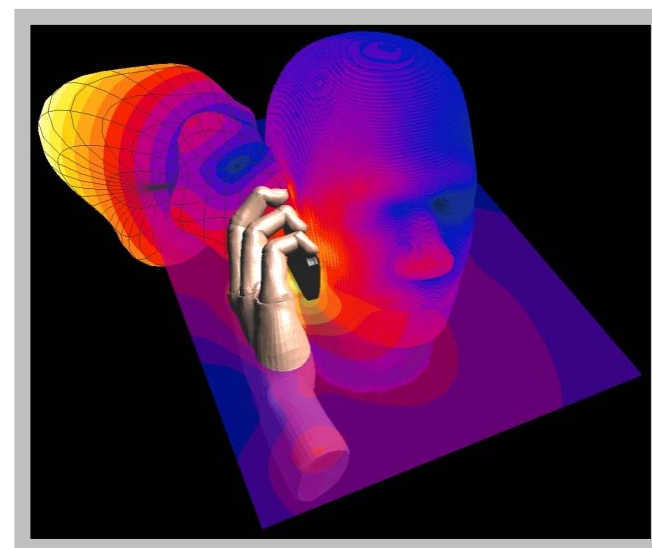
AxFDTD™

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

Available from:



Agilent Technologies



Consulting Services

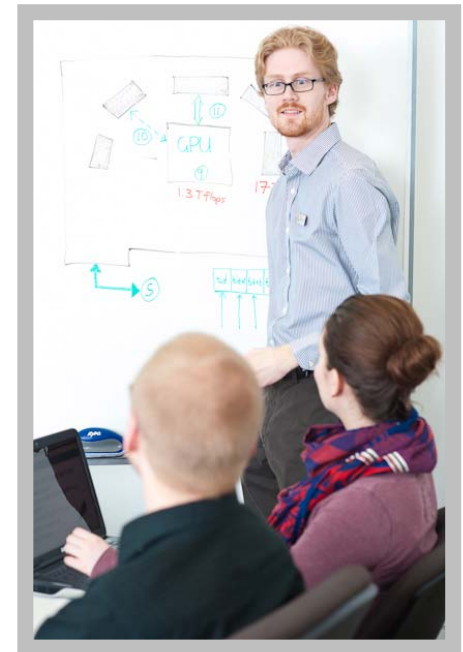
Industry	Application	Work Completed	Results
Finance	Option Pricing	Debugged & optimized existing CUDA code Implemented the Leisen-Reimer version of the binomial model for stock option pricing	30-50x performance improvement compared to single-threaded CPU code
Security & Defense	Detection System	Replaced 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

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



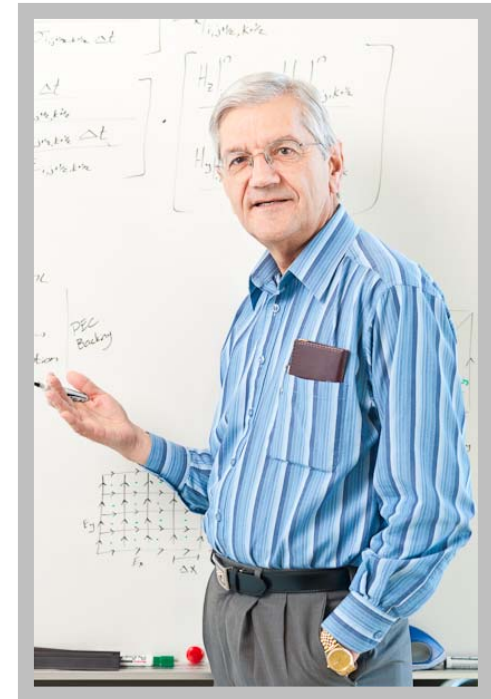
Outline

Task Parallelism

Thread Cooperation in GPU
Computing

GPU Memory Model

- Shared Memory
- Constant Memory
- Global Memory



Data-Parallel Computing

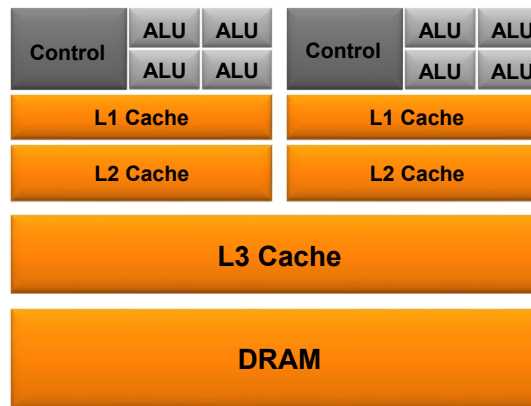
Review: Data-parallelism

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!**

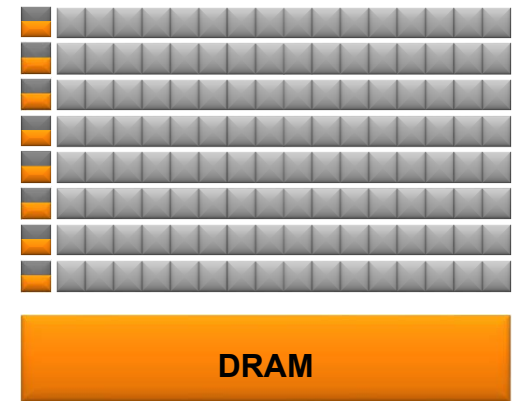
Data-Parallel Computing on GPUs

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



CPU



GPU

The CUDA Programming Model

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

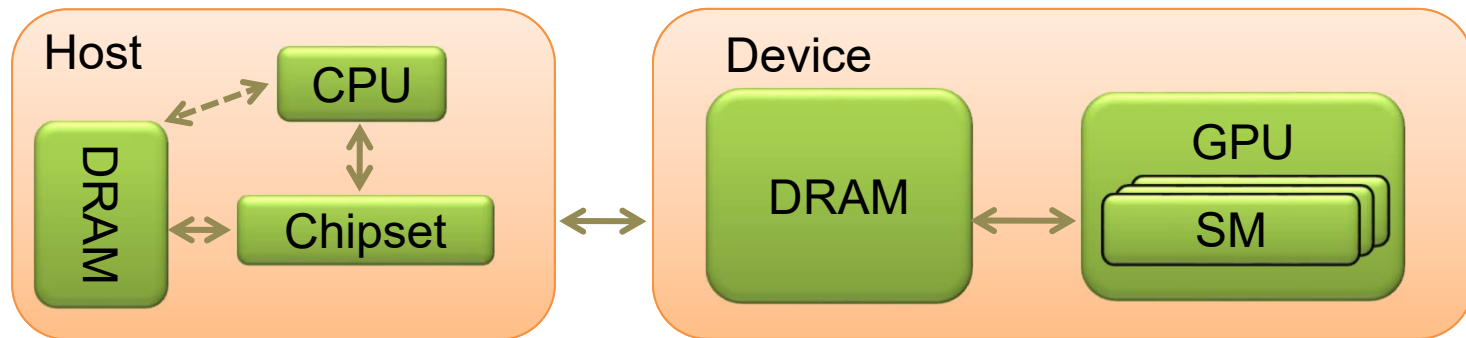
This isn't quite true!

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)

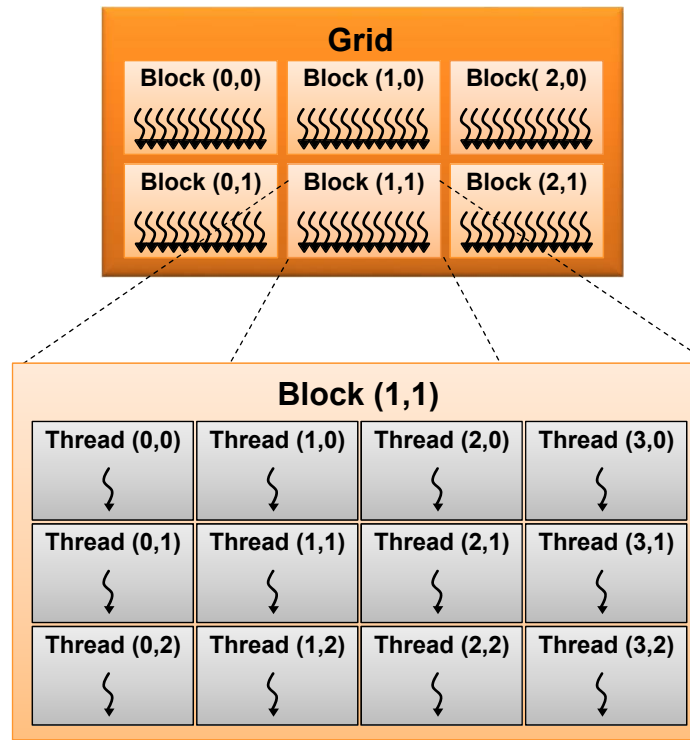
- Each SM has a collection of compute resources:
 - Processors (cores)
 - Registers
 - Specialized memory resources



Streaming Multiprocessors on GPUs

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

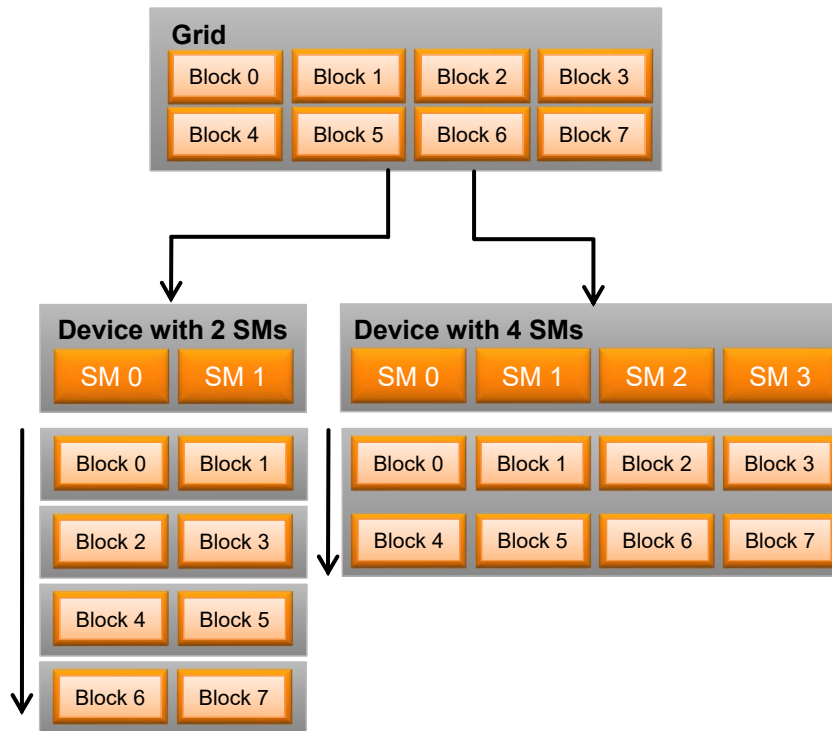
CUDA Thread Hierarchy



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

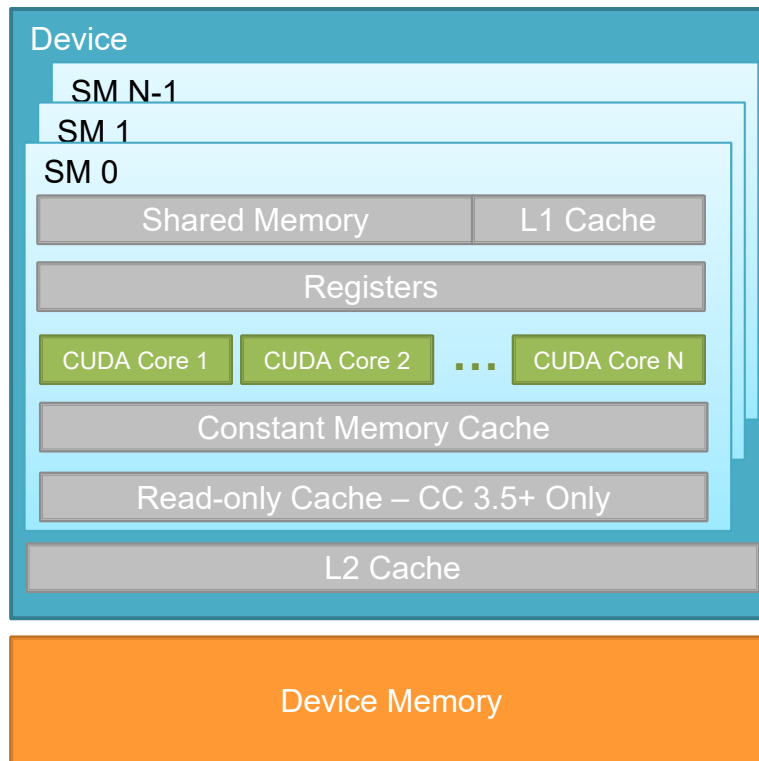
    // Every thread can now access var
    int temp = var[blockIdx.x] +
               threadIdx.x;

    ...
}
```

This kernel is not strictly data-parallel

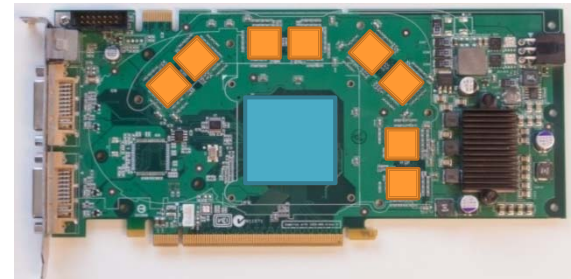
- Thread 0 of each block performs a different task than all other threads

Streaming Multiprocessor Architecture



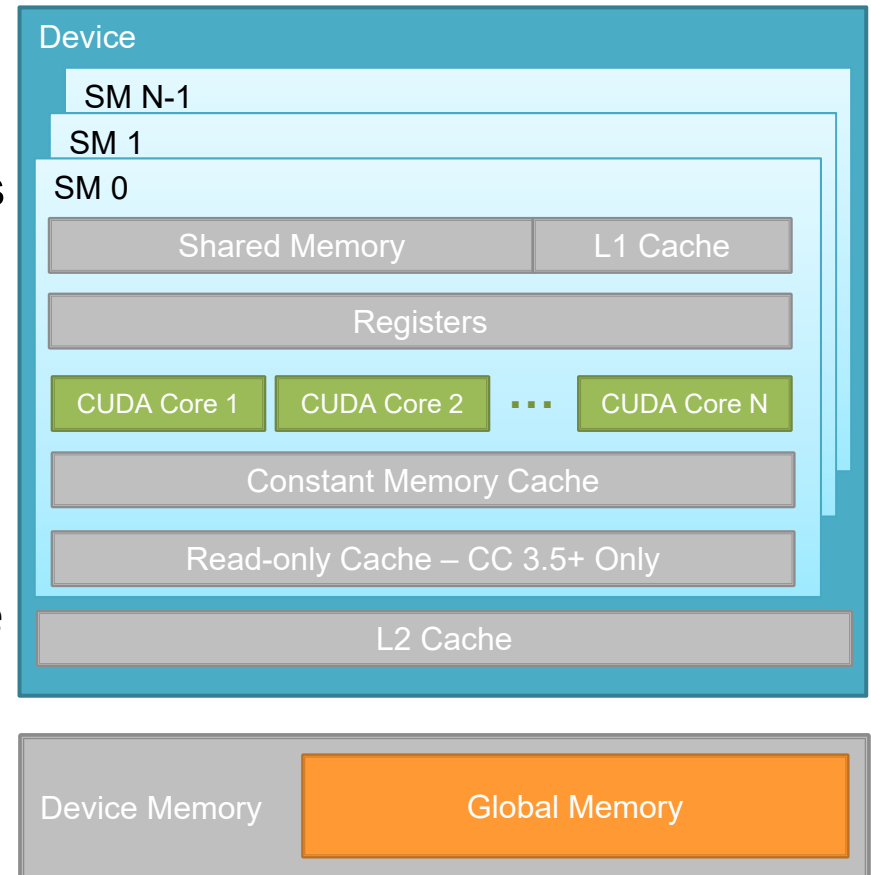
Many memory paths available each with different performance characteristics

- 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

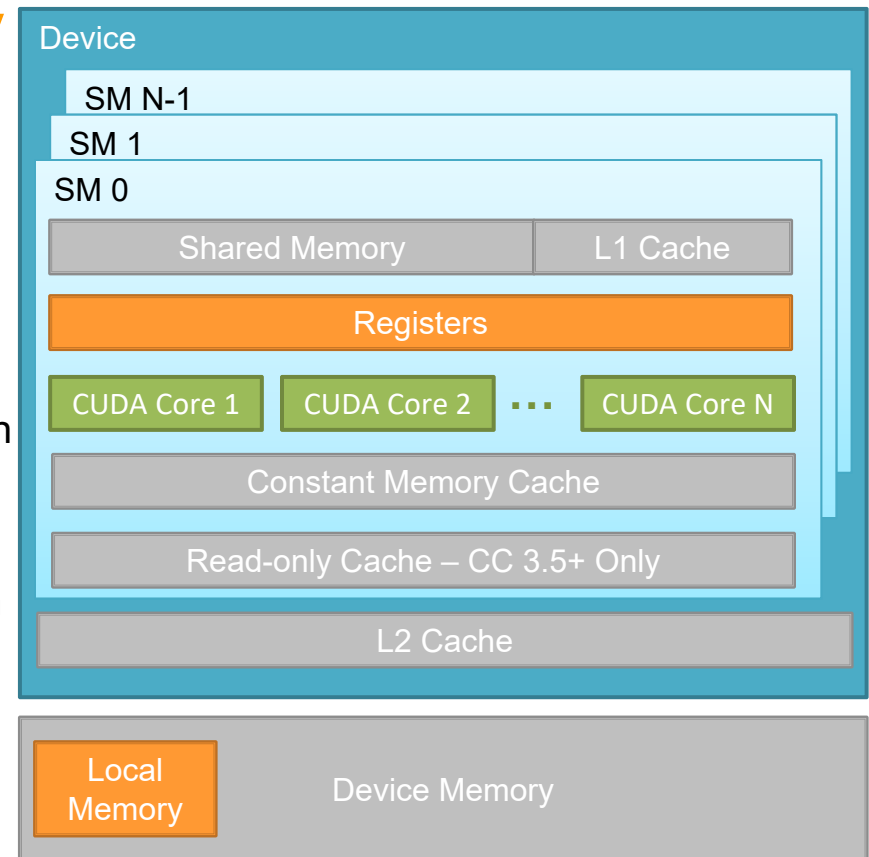
Variables declared within a kernel are allocated per thread

- 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

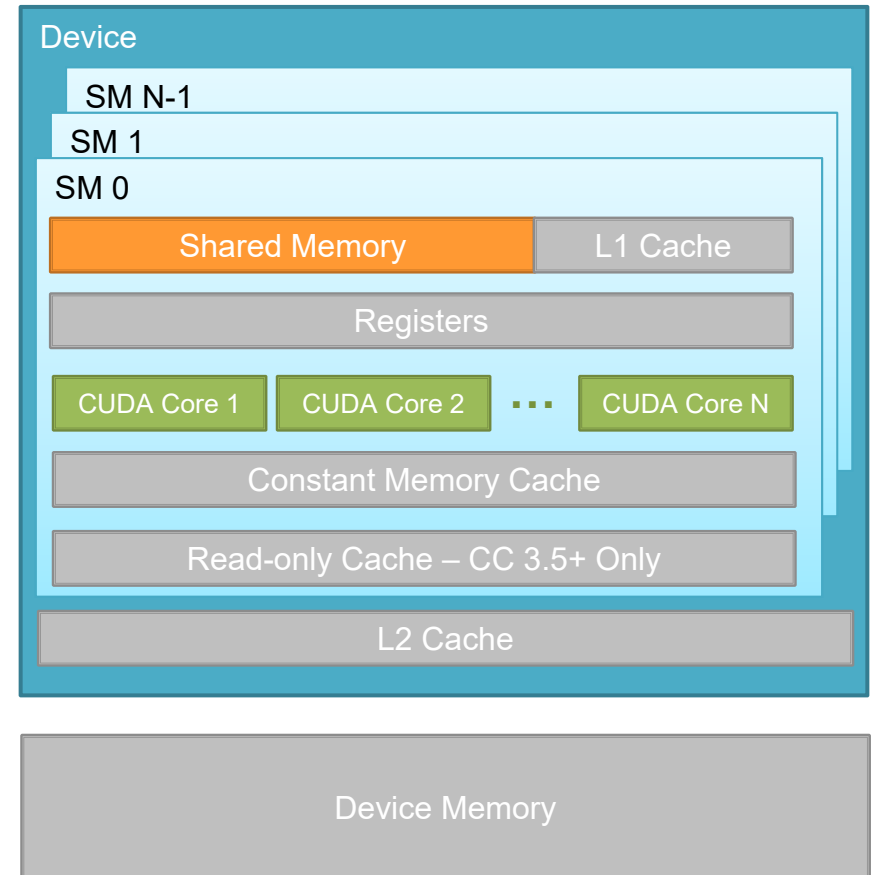
- Compiler controls where these variables are stored in physical memory
 - **Registers** (On-chip):
 - Fastest form of memory on the SM
 - **Local Memory** (Off-chip):
 - Compiler controlled region of device memory for storage of local variables when registers are insufficient or not suitable



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



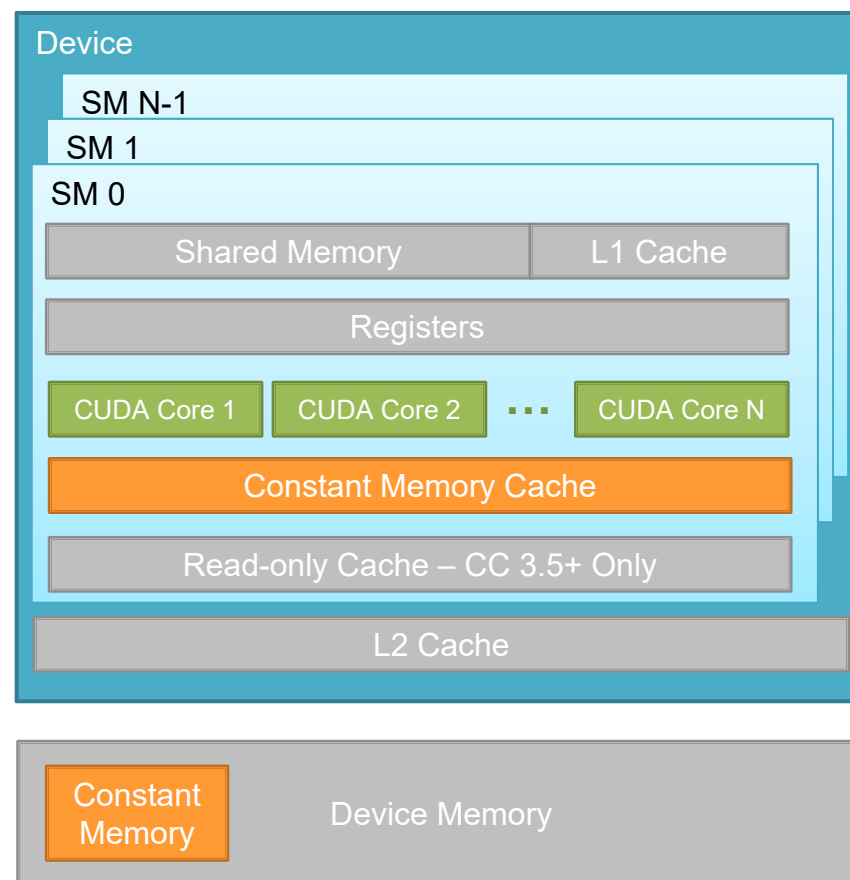
Shared Memory

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

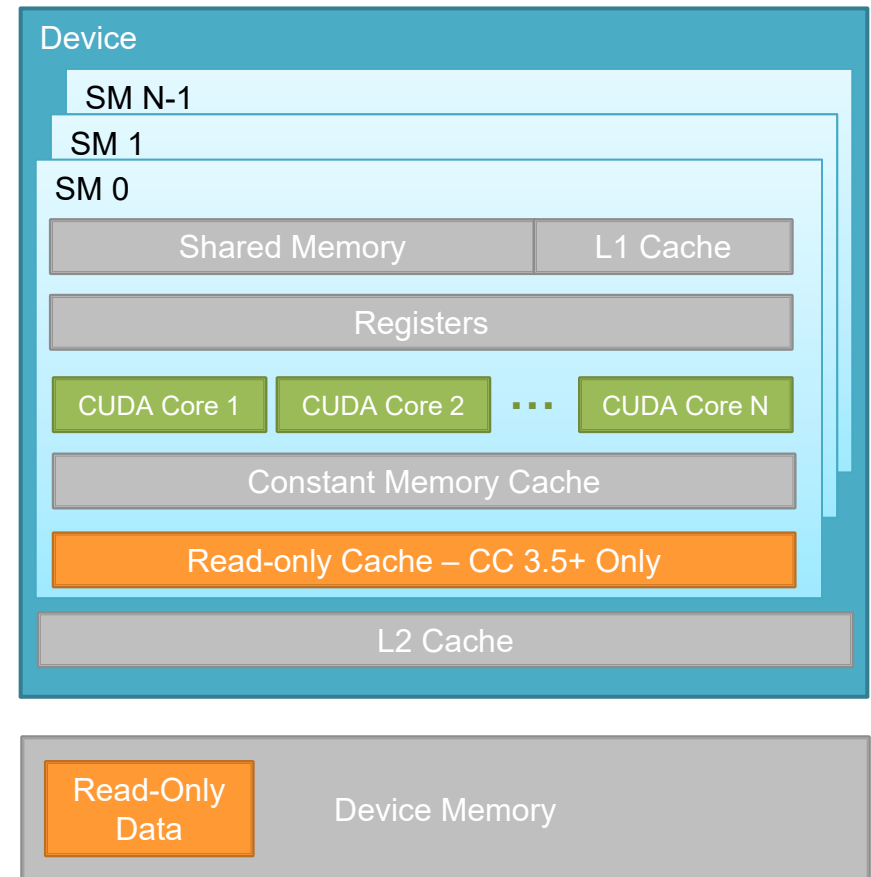
Constant Memory

- 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

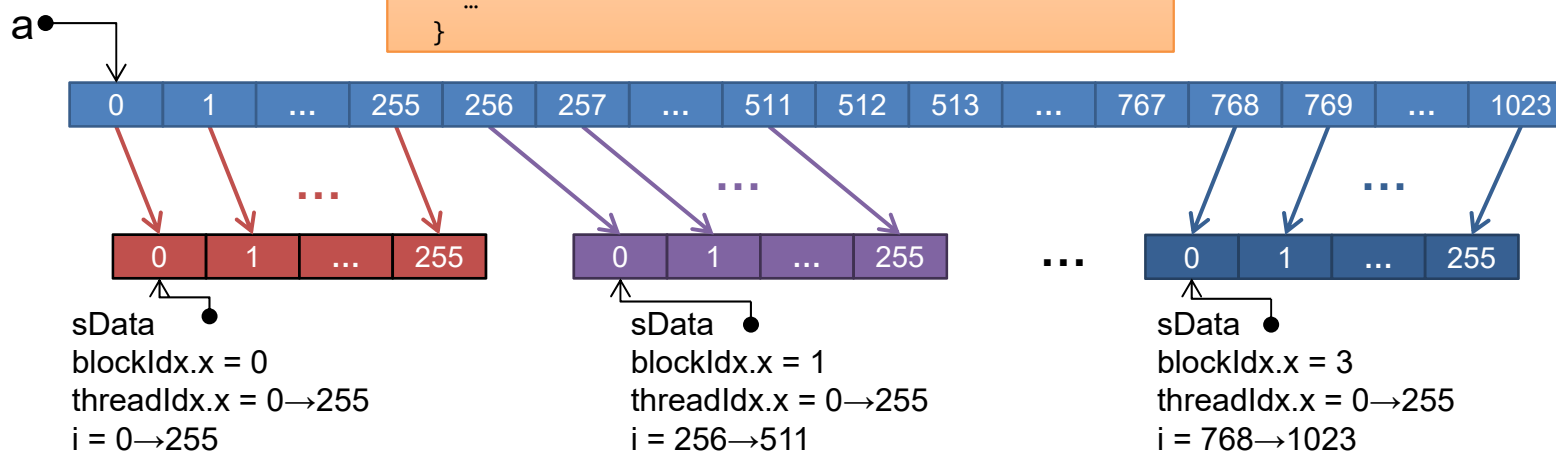
__global__ void kernel(float* a)
{
    __shared__ float sData[BLOCK_SIZE];
    int i;
    i = blockIdx.x * blockDim.x + threadIdx.x;
    sData[threadIdx.x] = a[i];
    __syncthreads();
    ...
    a[i] = sData[blockDim.x - 1 - threadIdx.x];
}

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

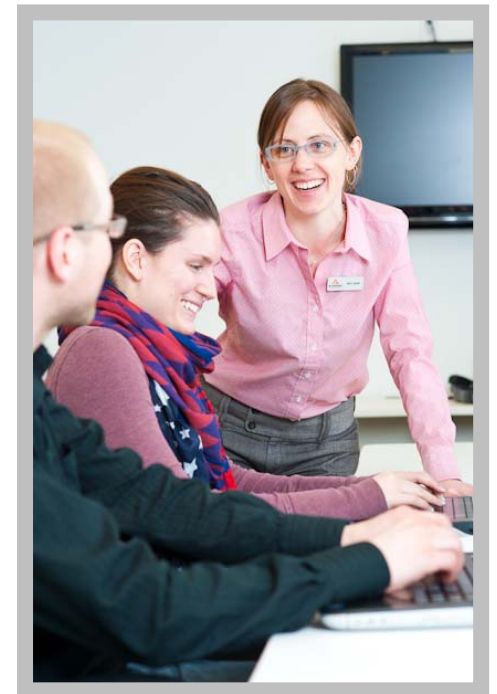
Scheduled CUDA Courses (also available online)

- 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



Asynchronous Operations & Dynamic Parallelism in CUDA

S7705 – Session 3 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™

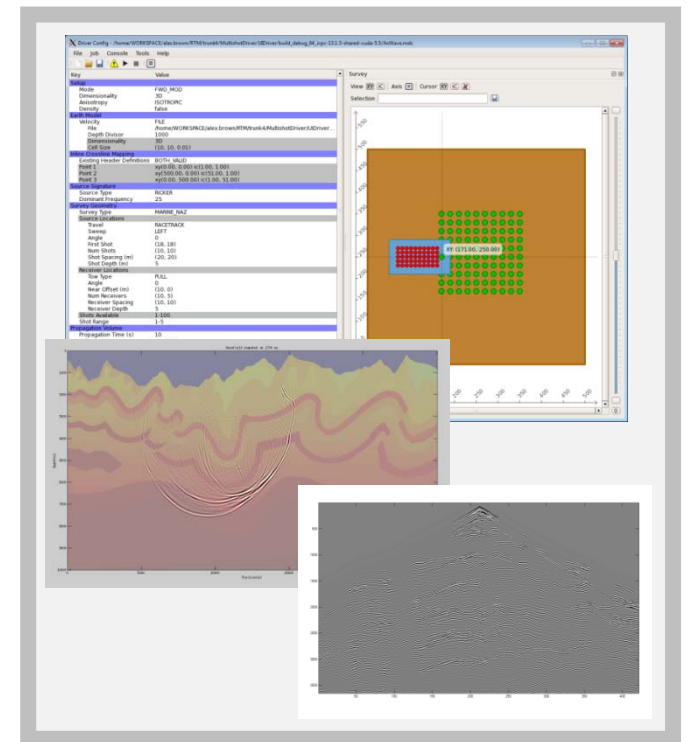
- High performance Reverse Time Migration application
- Isotropic, VTI and TTI media

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

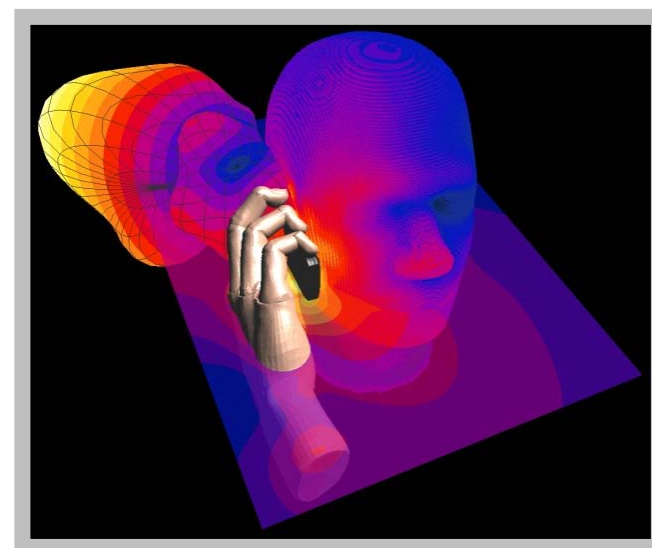
AxFDTD™

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

Available from:



Agilent Technologies



Consulting Services

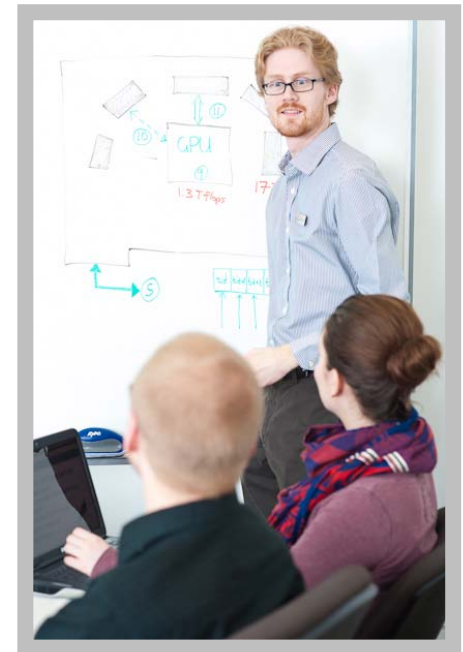
Industry	Application	Work Completed	Results
Finance	Option Pricing	Debugged & optimized existing CUDA code Implemented the Leisen-Reimer version of the binomial model for stock option pricing	30-50x performance improvement compared to single-threaded CPU code
Security & Defense	Detection System	Replaced 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

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

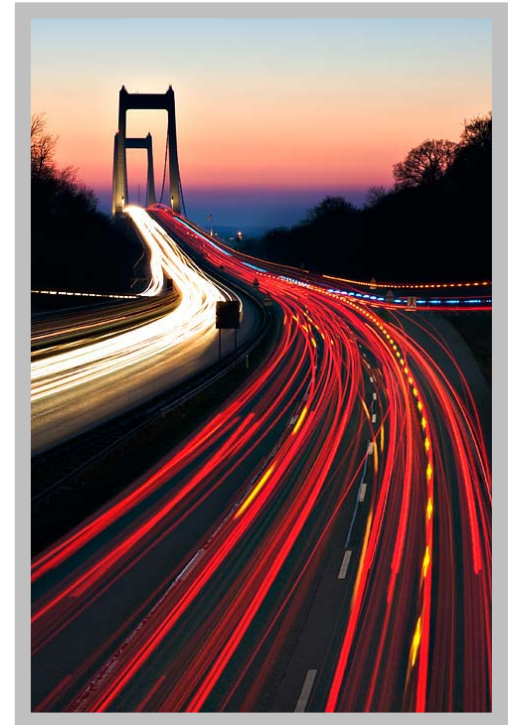
“The level of detail is fantastic. The course did not focus on syntax but rather on how to expertly program for the GPU. I loved the course and I hope that we can get more of our team to take it.”

Jason Gauci, Software Engineer
Lockheed Martin



Outline

- Concurrent CPU Computation and Kernel Execution
- Concurrent Transfers and Kernel Execution
- Streams/Events/Synchronization
- Pipelining
- Profilers and Asynchronous Operations
- Dynamic Parallelism



Motivation

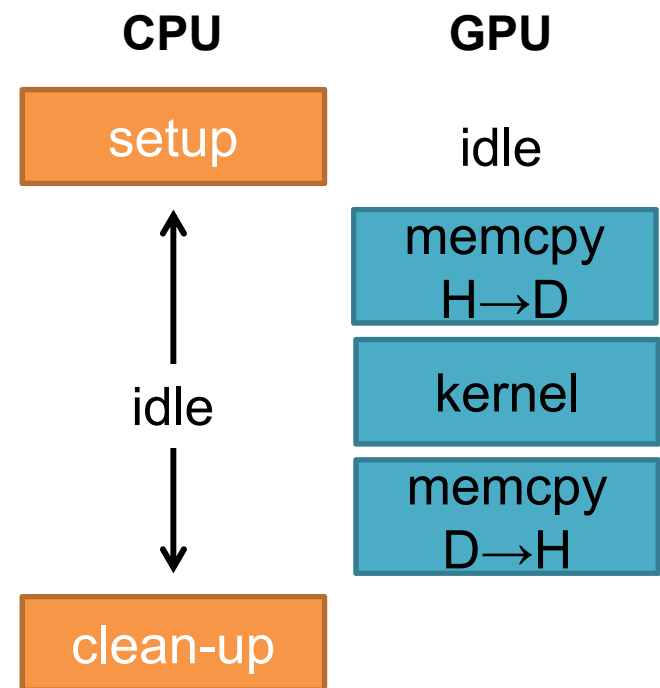
- Use asynchronous operations to introduce another level of parallelism
 - Better utilization of the system
 - Use all the available resources on your system concurrently
 - CPU cores
 - GPU(s)
 - PCIe bus
- Example:
 - Distribute a single large matrix multiply across multiple GPUs
 - Support matrices that are larger than available GPU memory by streaming data to/from host memory

Synchronicity in CUDA

- All CUDA calls are either synchronous or asynchronous relative to the host
 - Synchronous – enqueue work and wait for completion
 - Asynchronous – enqueue work and return immediately

What is the CPU doing during GPU kernels?

- Waiting for the GPU, typically...
- Use it for an independent task in parallel!
 - Eg. Reordering data, writing to a file



Concurrent CPU-GPU Computation

- Kernel invocations are asynchronous
- Need to synchronize CPU and GPU
 - Explicit: `cudaDeviceSynchronize()`
 - Implicit: `cudaMemcpy()`
- Order matters: launch kernels then call CPU functions

```
void foo()
{
    cpuCode1();

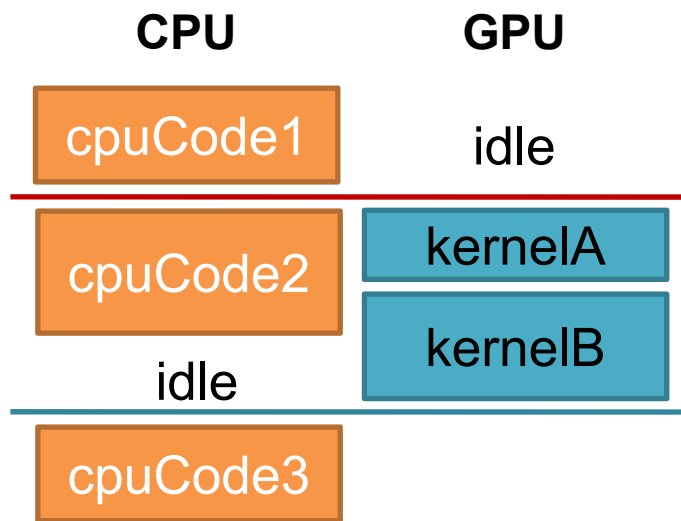
    KernelA<<<grid,block>>>(...);
    KernelB<<<grid,block>>>(...);

    cpuCode2();

    cudaDeviceSynchronize();

    // or cudaMemcpy for
    // implicit synchronization
    // cudaMemcpy(...,
    // cudaMemcpyDeviceToHost)
}
```

Task Timeline



```
void foo()
{
    cpuCode1();

    KernelA<<<grid,block>>>(…);
    KernelB<<<grid,block>>>(…);

    cpuCode2();

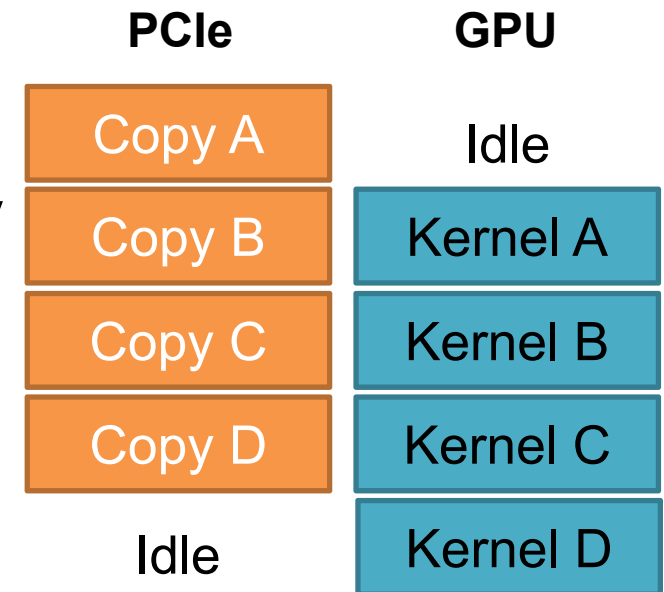
    // Explicit synchronization
    cudaDeviceSynchronize();

    // Or implicit synchronization
    // cudaMemcpy(...)

    cpuCode3();
}
```

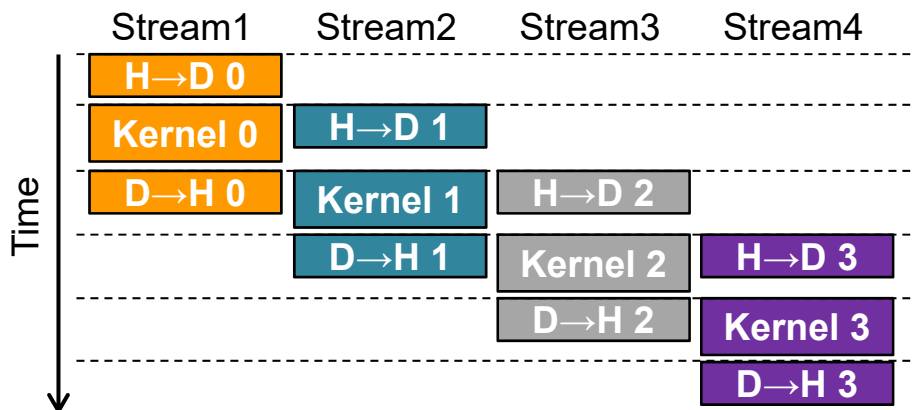
Concurrent Transfers

- Copy data over PCIe bus while kernels are executing on GPUS
- Check **asyncEngineCount** property in **cudaGetDeviceProperties()**
 - 0: No overlap is supported
 - 1: One data transfer can overlap with kernel execution
 - 2: Bidirectional data transfers can overlap with kernel execution
 - These devices have 2 DMA Engines!



Asynchronous Communications

- Implemented via asynchronous variations of `cudaMemcpy*()`
- Stream = sequence (queue) of operations that execute *in order* on GPU
 - No ordering constraints on operations between different streams
- Operations can execute asynchronously if they:
 - 1) Are from different streams
 - 2) Use different hardware resources
 - 3) The necessary hardware resources are available



Page-Locked Host Memory (1)

- Page-locked (“pinned”) host memory required for asynchronous transfers
- `cudaMemcpyAsync()` does not require page-locked host memory
 - HOWEVER then transfers are synchronous
- `cudaMallocHost()` or `cudaHostAlloc()` and `cudaFreeHost()`
 - Allocate/free page-locked memory
- `cudaHostRegister()/cudaHostUnregister()`
 - Make existing memory page-locked



Page-Locked Host Memory (2)

- Page-locked memory is typically faster with regular `cudaMemcpy()` too!
 - 2.7 vs 5.6GB/s on Tesla C2050, PCIe x16 Gen2
 - 3.3 vs 8.9GB/s on Tesla K40, PCIe x16 Gen 3
- Use with caution!
 - Allocating too much page-locked memory can reduce system performance
 - Calls to `cudaMallocHost()` will fail long before allocations by `malloc()`

Syntax – Asynchronous Communications

- Asynchronous Data Transfers
 - `cudaMemcpyAsync(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind direction, cudaStream_t stream = 0)`
- Handles created and destroyed using API
 - `cudaError_t cudaStreamCreate(cudaStream_t* stream)`
 - `cudaError_t cudaStreamDestroy(cudaStream_t stream)`
- Launch kernels to different streams using the 4th argument in the chevron syntax `<<<>>>`
 - `myKernel<<<gSize, bSize, smBytes, stream>>>(...)`

Default Stream

- Unless otherwise specified all operations are enqueued in the default stream or “Stream 0”
- Default stream has special synchronization properties
 - Synchronous with all streams
 - Operations in default stream cannot overlap with operations from any other stream
- Exception: Streams with non-blocking flag set
 - `cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking)`
 - Allows operations in current stream to run concurrently with default stream
 - Use to get concurrency with operations out of your control (eg. libraries)

Syntax Example

```
void foo(...)
{
    cudaStream_t stream1, stream2;

    cudaStreamCreate(&stream1);
    cudaStreamCreate(&stream2);

    cudaMemcpyAsync(d_bar, h_bar, sizeof(bar),
                   cudaMemcpyHostToDevice, stream1);

    kernelA<<<grid, block, 0, stream2>>>(...);
    kernelB<<<grid, block, 0, stream1>>>(...);

    ...

    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);
}
```

These can overlap because they:

- 1) are on different streams
- 2) use different resources

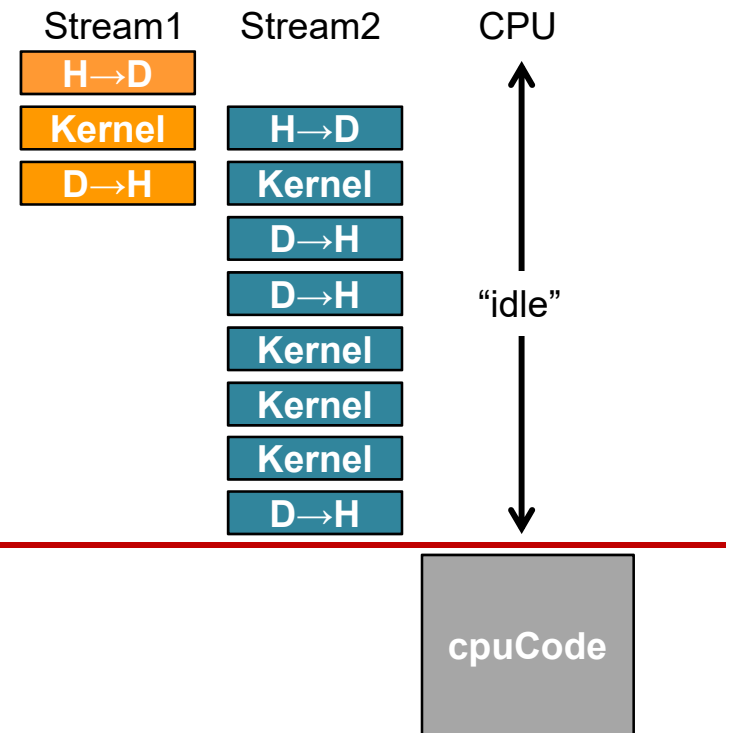
- These are asynchronous with CPU execution
- Synchronization is required!

Synchronization

1. cudaDeviceSynchronize()

CPU ↔ all operations on current device

```
cudaMemcpyAsync(...HostToDevice, stream1);  
  
cudaMemcpyAsync(...HostToDevice, stream2);  
Kernel<<<..., stream1>>>(…);  
  
cudaMemcpyAsync(...DeviceToHost, stream1);  
Kernel<<<..., stream2>>>(…);  
  
cudaMemcpyAsync(...DeviceToHost, stream2);  
cudaMemcpyAsync(...DeviceToHost, stream2);  
Kernel<<<..., stream2>>>(…);  
Kernel<<<..., stream2>>>(…);  
Kernel<<<..., stream2>>>(…);  
cudaMemcpyAsync(...DeviceToHost, stream2);  
  
cudaDeviceSynchronize();  
cpuCode();
```

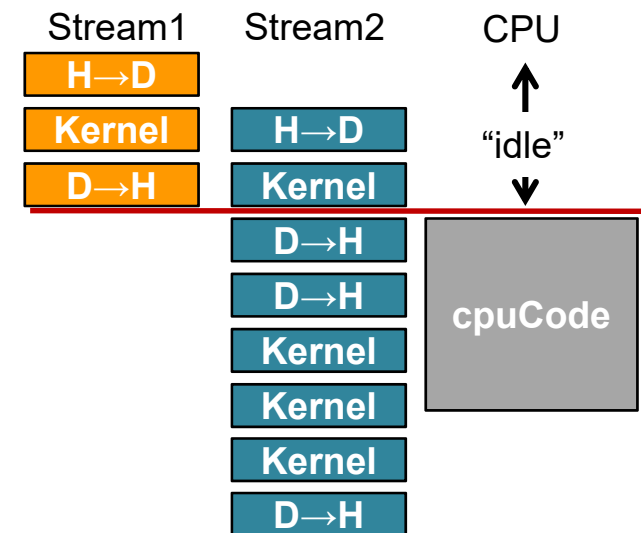


Synchronization

2. cudaStreamSynchronize()

- CPU ↔ all operations in a stream

```
cudaMemcpyAsync(...HostToDevice, stream1);  
  
cudaMemcpyAsync(...HostToDevice, stream2);  
Kernel<<<..., stream1>>>(…);  
  
cudaMemcpyAsync(...DeviceToHost, stream1);  
Kernel<<<..., stream2>>>(…);  
  
cudaMemcpyAsync(...DeviceToHost, stream2);  
cudaMemcpyAsync(...DeviceToHost, stream2);  
Kernel<<<..., stream2>>>(…);  
Kernel<<<..., stream2>>>(…);  
Kernel<<<..., stream2>>>(…);  
cudaMemcpyAsync(...DeviceToHost, stream2);  
  
cudaStreamSynchronize(stream1);  
cpuCode();
```



Events (1)

- Events provide a mechanism to determine when operations in a stream are complete
 - 'Dummy' operations that do no work BUT
 - Useful for profiling and synchronization
- Events have a boolean state
 - Occurred (cudaSuccess)
 - Not Occurred
 - **Default state = Occurred**

Events (2)

- Create and destroy events
 - `cudaEventCreate(cudaEvent_t* e)`
 - `cudaEventDestroy(cudaEvent_t e)`
- Record an event
 - `cudaEventRecord(cudaEvent_t e, cudaStream_t s = 0)`
 - Sets the event state to not occurred
 - Enqueues the event into a stream
 - Event state is set to occurred when all previous operations in the stream are complete
 - If stream is default stream, all prior operations in ALL other streams must be complete too!

Event Synchronization

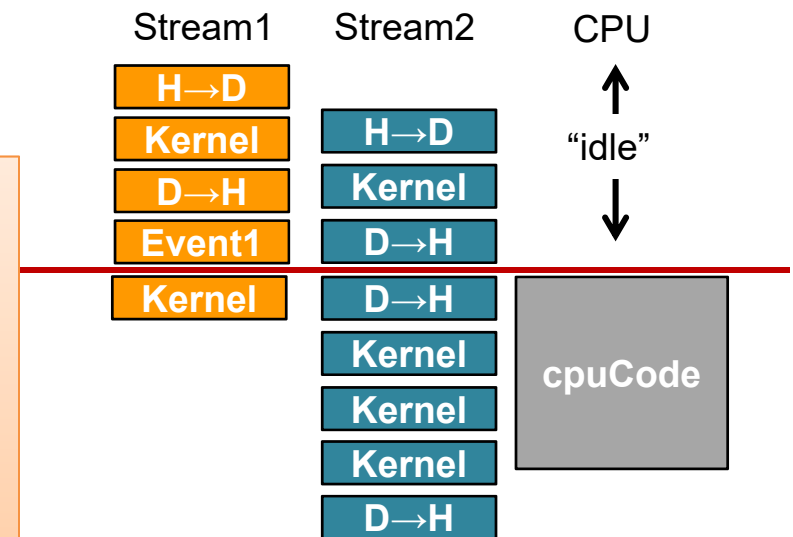
3. cudaEventSynchronize()

- CPU ↔ specific point in a stream's queue of operations

```
cudaMemcpyAsync(...HostToDevice, stream1);  
Kernel<<<..., stream1>>>(…);  
cudaMemcpyAsync(...DeviceToHost, stream1);  
cudaEventRecord(event1, stream1);  
Kernel<<<..., stream1>>>(…);
```

```
cudaMemcpyAsync(...HostToDevice, stream2);  
Kernel<<<..., stream2>>>(…);  
cudaMemcpyAsync(...DeviceToHost, stream2);  
cudaMemcpyAsync(...DeviceToHost, stream2);  
Kernel<<<..., stream2>>>(…);  
Kernel<<<..., stream2>>>(…);  
Kernel<<<..., stream2>>>(…);  
cudaMemcpyAsync(...DeviceToHost, stream2);
```

```
cudaEventSynchronize(event1);  
cpuCode();
```



Example: Putting it all Together

```
void foo()
{
    cpuCode1(...);

    cudaMemcpyAsync(..., stream1);

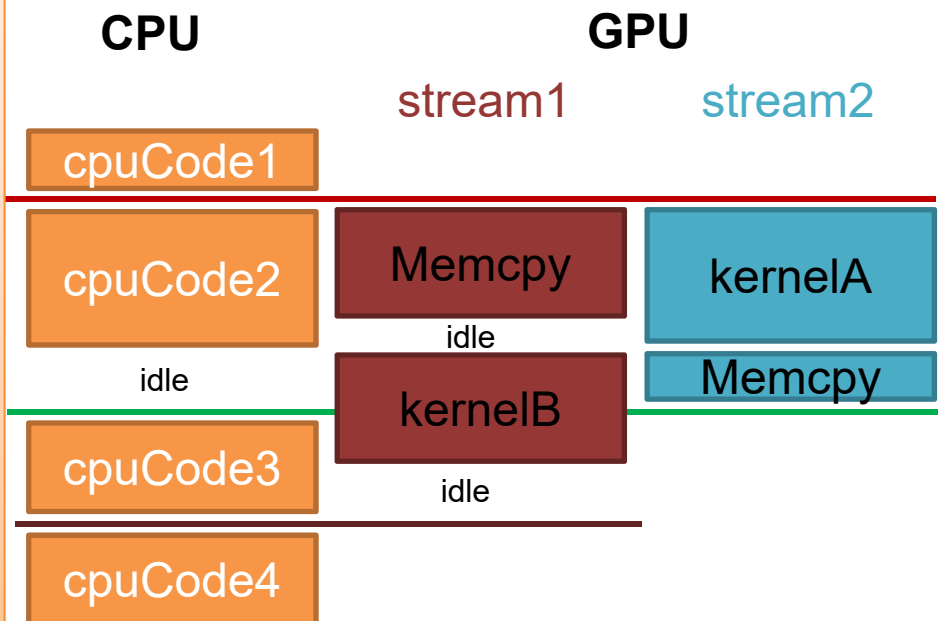
    kernelA<<<grid,block,0,stream2>>>(...);
    cudaMemcpyAsync(..., stream2);// Memcpy 2
    cudaEventRecord(event2, stream2);

    cpuCode2(...);

    kernelB<<<grid,block,0,stream1>>>(...);
    cudaEventRecord(event1, stream1);

    cudaEventSynchronize(event2);
    cpuCode3(...);

    cudaEventSynchronize(event1);
    cpuCode4(...);
}
```

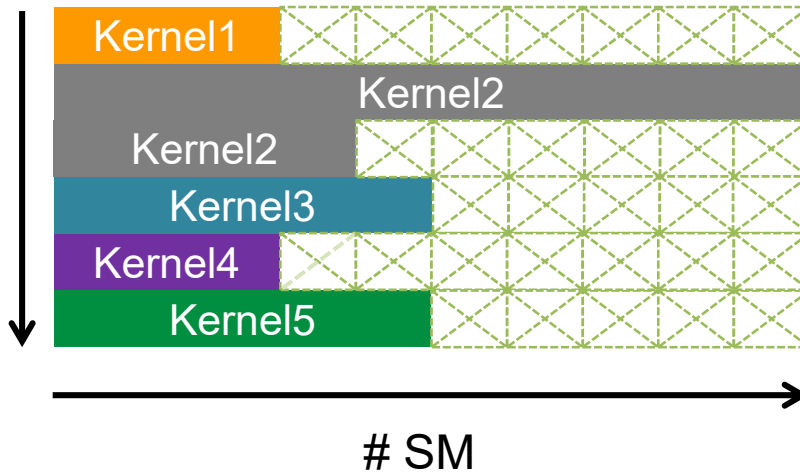


Concurrent Kernel Execution

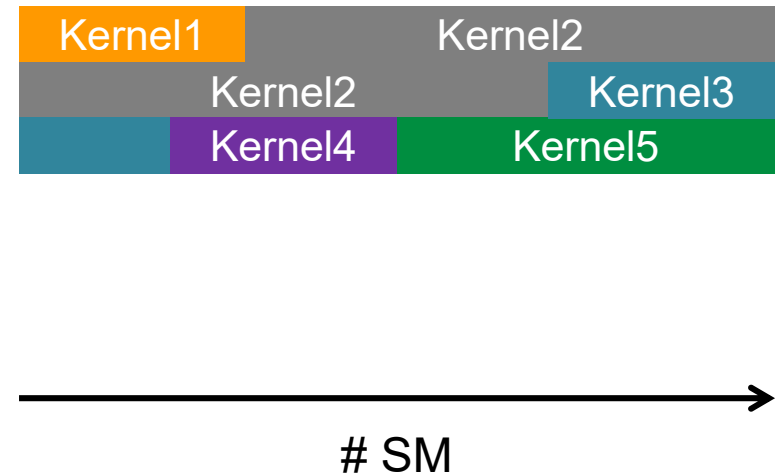
- Some Compute 2.0+ devices can execute multiple kernels concurrently
 - Check **concurrentKernels** property in **cudaGetDeviceProperties**
 - Maximum number of concurrent kernels
 - 32 kernels on compute capability 3.5 devices
 - 16 kernels on compute capability 2.0 – 3.0 devices
 - Kernels must be from the same process
 - Kernels must be from different, **non-default**, streams

Concurrent Kernel Execution

Serial Kernels

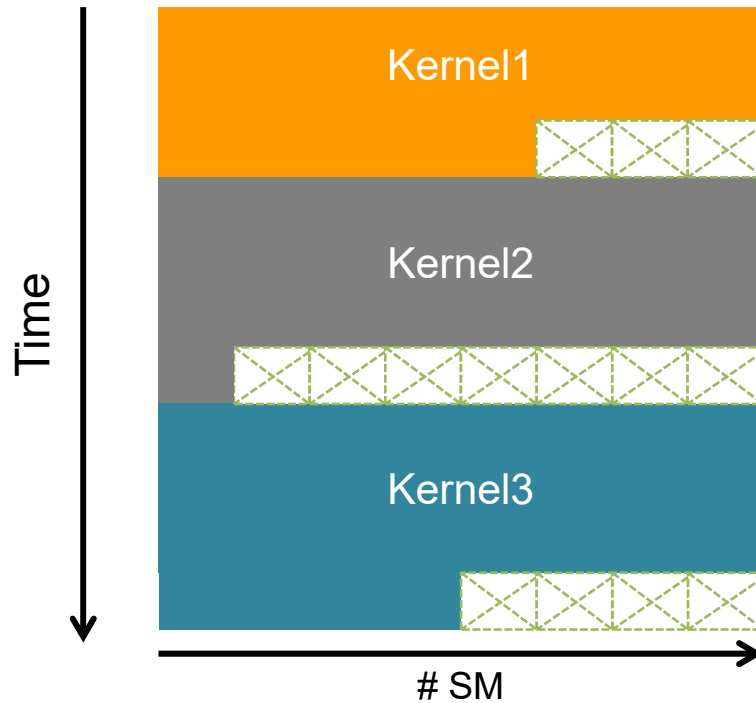


Concurrent Kernels

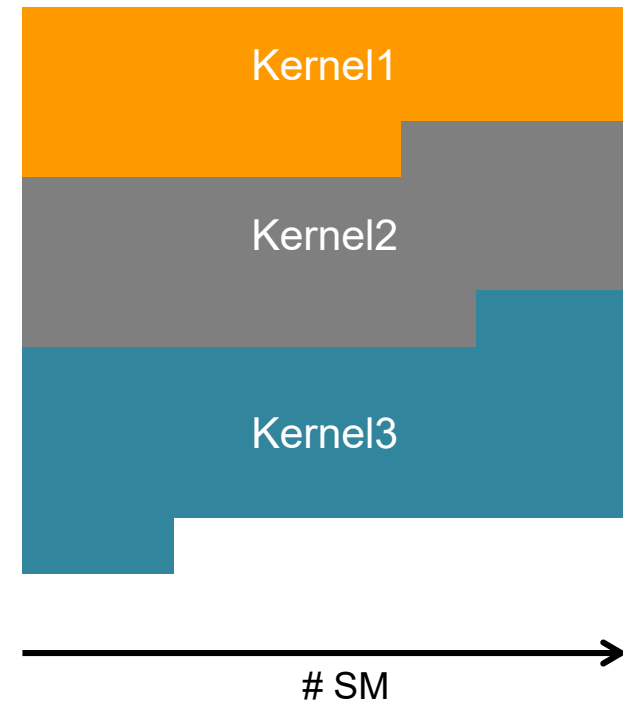


Concurrent Kernel Execution

Serial Kernels

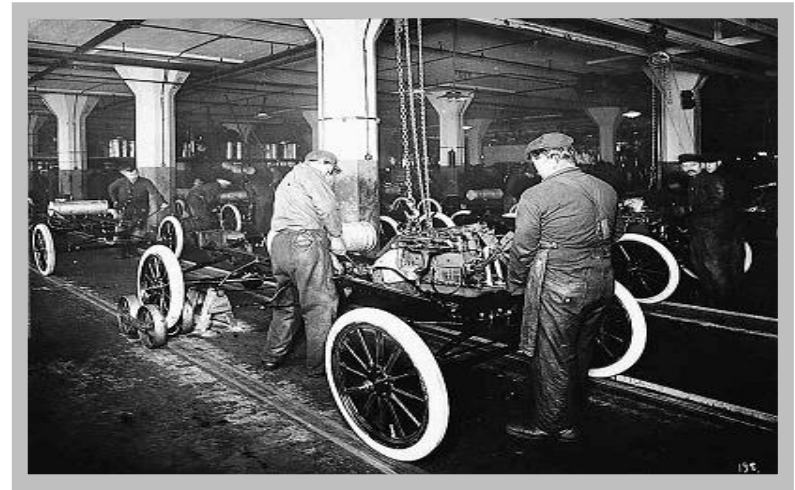


Concurrent Kernels



Pipelining

- Pipelining – a set of data processing elements connected in series
 - The output of one element is the input of the next
- Computing equivalent of assembly lines

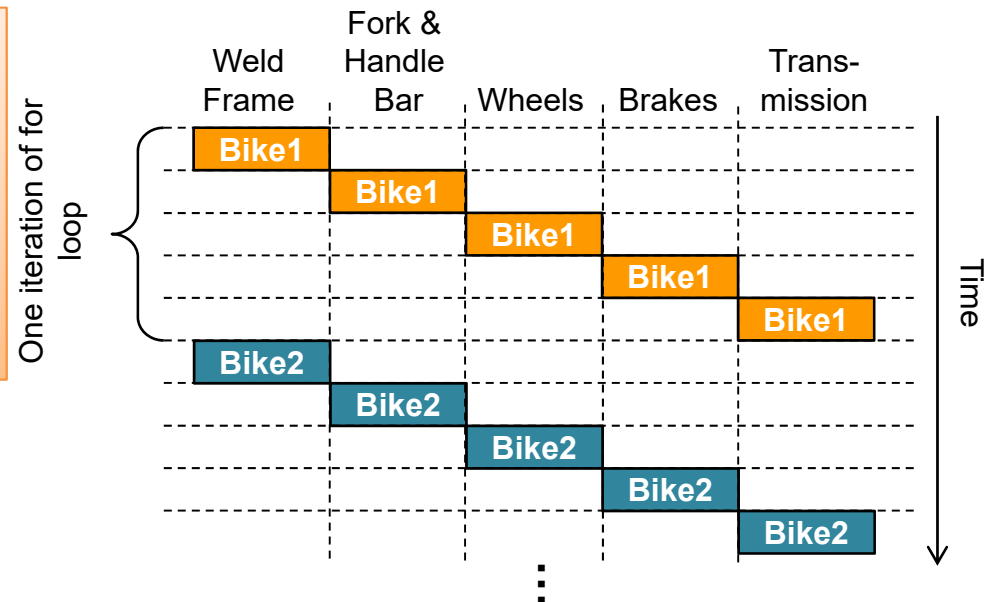


Pipelining in CUDA

- Pipelining is common in CUDA as you have multiple resources, with each resource specialized for different tasks:
 - CPU – serial algorithms, I/O
 - GPU SMs – parallel algorithms
 - GPU DMA engines – CPU→GPU transfers

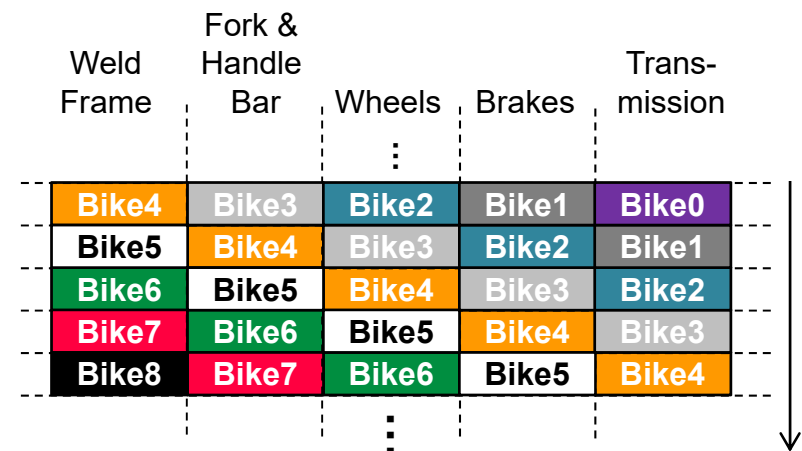
Implementing A Software Pipeline

```
for(int bike = 0;  
    bike < BICYCLES;  
    ++bike)  
{  
    WeldFrame(bike);  
    AttachForkAndHandleBar(bike);  
    InstallWheelsAndTires(bike);  
    InstallBrakes(bike);  
    InstallChainAndTransmission(bike);  
}
```



Implementing A Software Pipeline

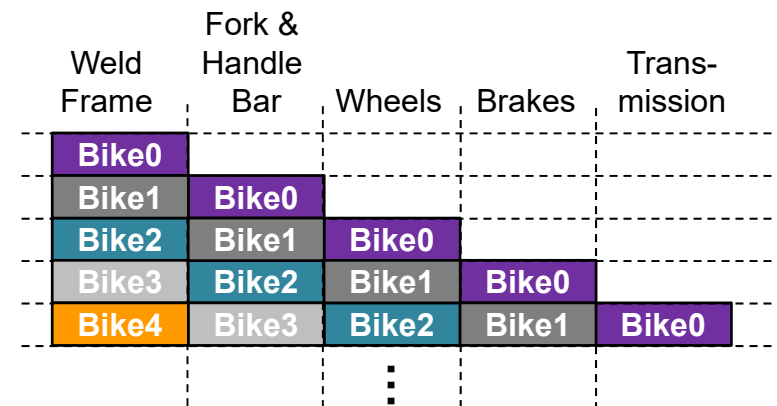
```
for(int bike = 0;
    bike < BICYCLES;
    ++bike)
{
    WeldFrameAsync(bike+2);
    AttachForkAndHandleBarAsync(bike+1);
    InstallWheelsAndTiresAsync(bike);
    InstallBrakesAsync(bike-1);
    InstallChainAndTransmissionAsync(bike-2);
}
```



Implementing a Software Pipeline

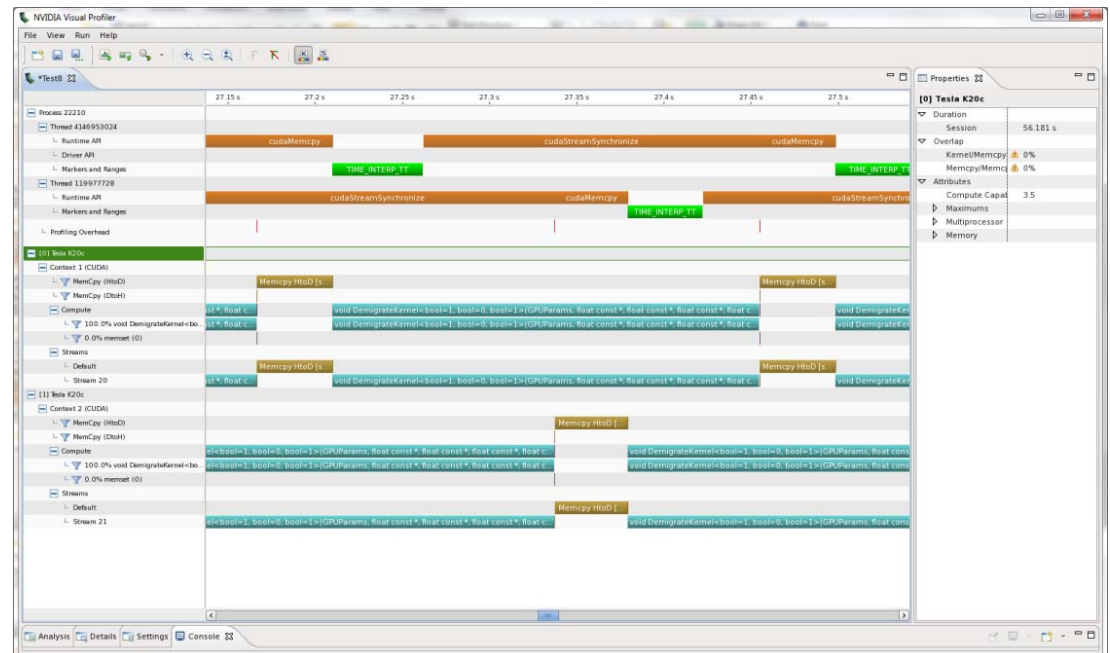
- One bicycle is completed every iteration, but 5 are under construction at the same time.
- Need to prime (and empty) the pipeline

```
WeldFrame(0);  
  
WeldFrame(1);  
AttachForkAndHandleBar(0);  
  
WeldFrame(2);  
AttachForkAndHandleBar(1);  
InstallWheelsAndTires(0);  
...  
for(int bike = 2; bike < BICYCLES-2;  
    bike++)
```



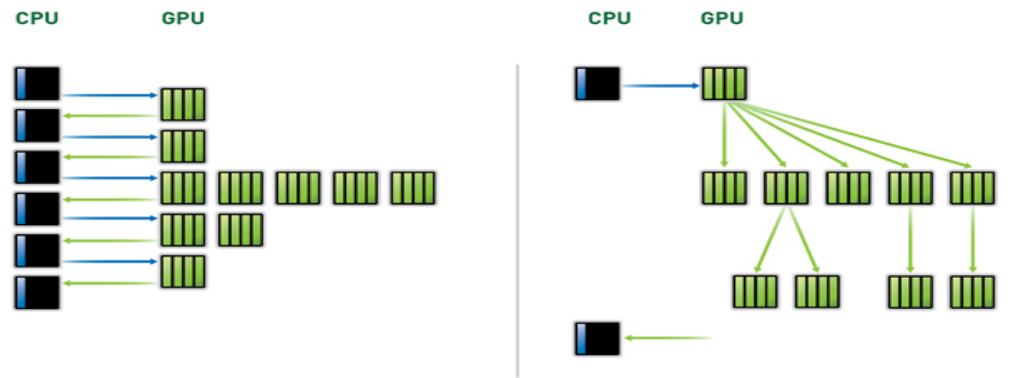
Profilers and Asynchronous Operations

Timeline views
useful for visualizing
actual execution
order of
asynchronous
operations



Dynamic Parallelism

- Allows threads in GPU kernels to launch other GPU kernels
 - GPUs can operate more autonomously from host CPU



CPU Kernel Launches vs. Dynamic Parallelism

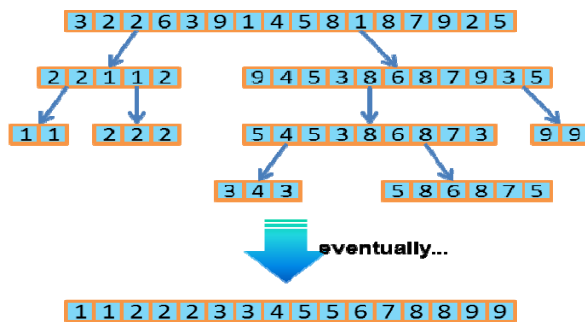
src: <http://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quick-sort-a-familiar-comp-sci-code/>

© 2017 Acceleware Ltd. Reproduction or distribution strictly prohibited.



Dynamic Parallelism

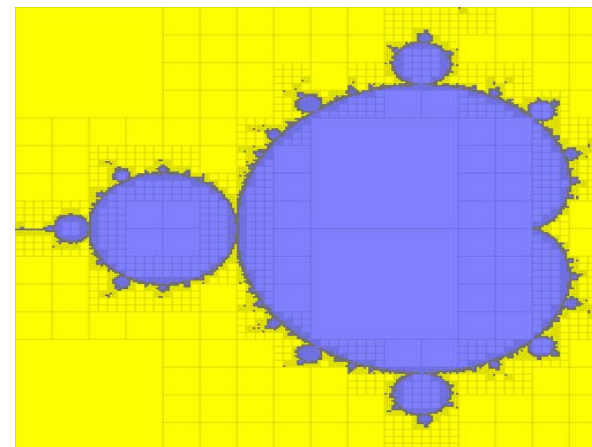
- Dynamic Parallelism is well-suited to algorithms that use recursive subdivision



Quicksort Algorithm

src: <http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/>

© 2017 Acceleware Ltd. Reproduction or distribution strictly prohibited.



Mariani-Silver Mandelbrot Set Algorithm



Dynamic Parallelism

- Consider Dynamic Parallelism for batch parallelism
 - Write a GPU kernel with 1 thread handling each batch of data
 - That thread can launch kernels for parallel computations on the batch

Dynamic Parallelism (Continued)

- Potential advantages of Dynamic Parallelism:
 - Simpler code for recursive problems
 - Eliminates device to host data transfers required for host to launch subsequent kernels
 - More efficient
 - Launch kernels from GPU as needed without any need for host synchronization that might introduce false data dependencies
 - eg. Non-Dynamic QuickSort implementation you need to wait for all kernels at one level of the tree are complete before launching any kernels at the next level!

Dynamic Parallelism

- Use <<<>>> syntax from within a kernel
- Synchronization
 - `cudaDeviceSynchronize()` synchronizes kernels launched from current thread block
 - All work launched by a thread block is implicitly synchronized when the block exits
 - Parent doesn't finish until children have finished

```
__global__ void childKernel()
{
    printf("Hello ");
}

__global__ void parentKernel()
{
    cudaError_t e;
    childKernel<<<1,1>>>();
    cudaDeviceSynchronize();
    e = cudaGetLastError();
    if(e) return;

    printf("World!\n");
}

void host_code()
{
    parentKernel<<<1,1>>>();
    ...
}
```

Dynamic Parallelism & Memory Model

- Parent and child grids share the same global and constant memory storage
 - You can pass pointers to global/constant memory to child kernels
- Thread-private variables still have thread scope
 - Can't pass pointers to thread-private variables to child kernels
- Shared memory still has thread block scope
 - Can't pass pointers to shared memory to child kernels

```
__global__ void Child(int* );
__global__ void Child2(int );

__constant__ int constantX[10];

__global__ void ParentKernel(int *globalX)
{
    __shared__ int sharedX[10];
    int localX = 5;

    Child<<<1,1>>>(sharedX);           x
    Child<<<1,1>>>(&localX);           x
    Child2<<<1,1>>>(sharedX[0]);       ✓
    Child2<<<1,1>>>(localX);           ✓

    Child<<<1,1>>>(globalX);           ✓
    Child<<<1,1>>>(constantX);         ✓
}
```

Dynamic Parallelism

- `cudaDeviceSynchronize()`
 - Parent blocks may pause execution and backup state (program counters, registers/shared memory contents) to host memory and yield to allow child kernels to make forward progress



Dynamic Parallelism

- Kernels launched within a thread-block are executed in order by default
 - Use streams/events to allow concurrent execution of kernels
- Streams and Events are supported
 - A stream and event created by a thread has thread block scope
 - Using streams and events created on the host has undefined behavior
- Create Streams and Events using flags:
 - `cudaStreamCreateWithFlags()` with `cudaStreamNonBlocking` flag
 - `cudaEventCreateWithFlags()` with `cudaEventDisableTiming` flag

Dynamic Parallelism & Streams

```
__global__ void parentKernel()
{
    cudaStream_t stream1, stream2;
    cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking);
    cudaStreamCreateWithFlags(&stream2, cudaStreamNonBlocking);

    // launch child
    if(threadIdx.x == 0)
    {
        foo<<<1,1,0,stream1>>>();
    }
    else if(threadIdx.x == 32)
    {
        bar<<<1,1,0,stream2>>>();
    }
    ...
}
```

Acceleware CUDA Training

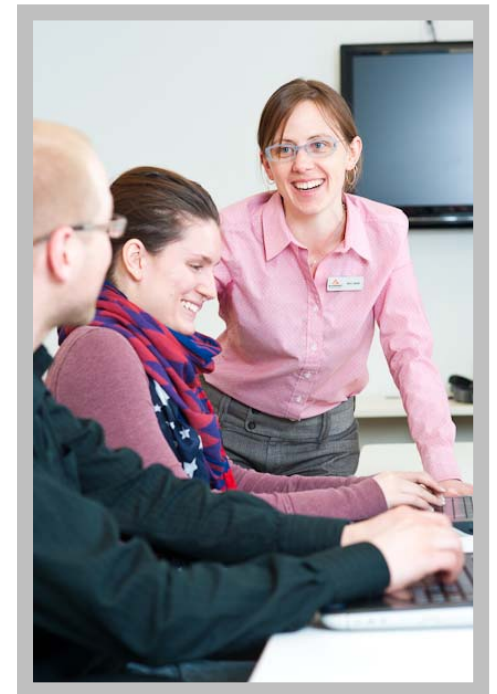
Scheduled CUDA Courses (also available online)

- 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



Essential CUDA Optimization Techniques

S7706 – Session 4 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™

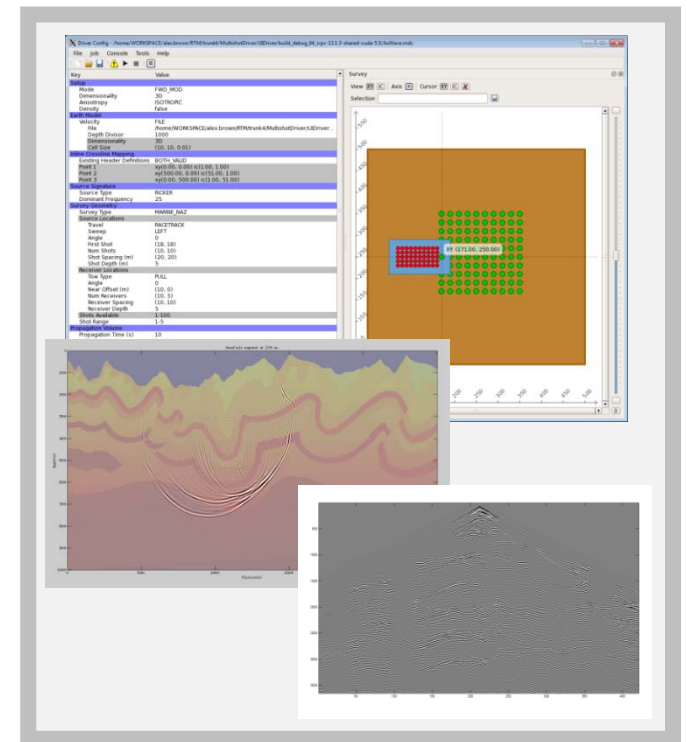
- High performance Reverse Time Migration application
- Isotropic, VTI and TTI media

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

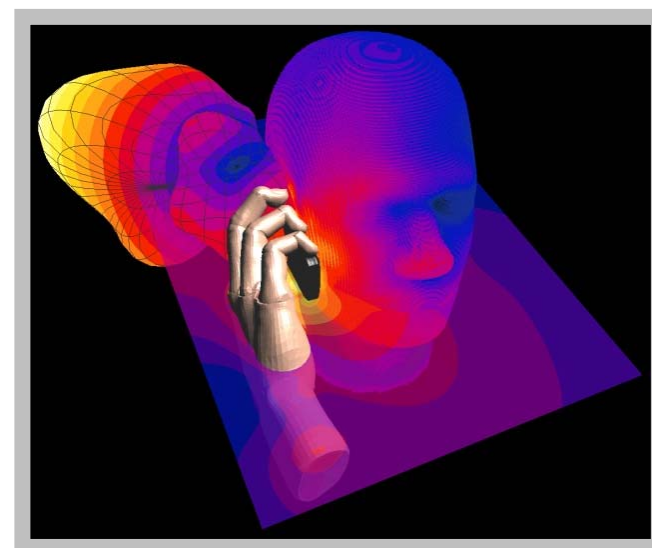
AxFDTD™

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

Available from:



Agilent Technologies



Consulting Services

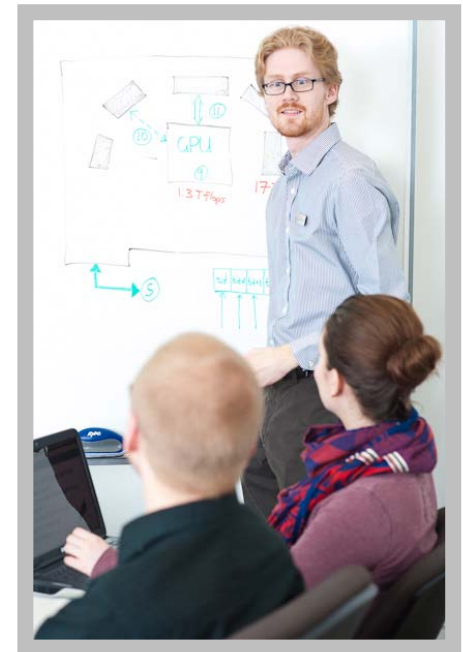
Industry	Application	Work Completed	Results
Finance	Option Pricing	Debugged & optimized existing CUDA code Implemented the Leisen-Reimer version of the binomial model for stock option pricing	30-50x performance improvement compared to single-threaded CPU code
Security & Defense	Detection System	Replaced 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

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



- Profiling
- Optimizations for latency bound kernels
- Optimizations for compute bound kernels
- Optimizations for memory bound kernels

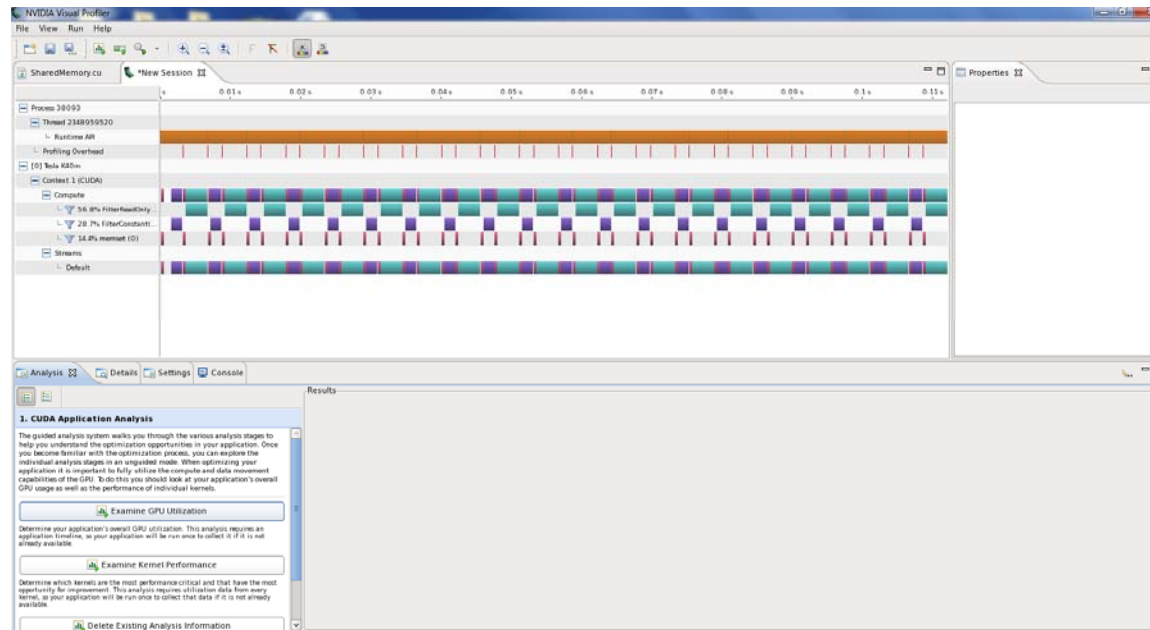


Profiling Tools

- Use profiling tools to identify performance bottlenecks
 - Standalone NVIDIA Visual Profiler (nvvp)
 - nvprof command-line profiler
 - Some functionality integrated into NVIDIA Nsight Visual Studio Edition/NVIDIA Nsight Eclipse Edition

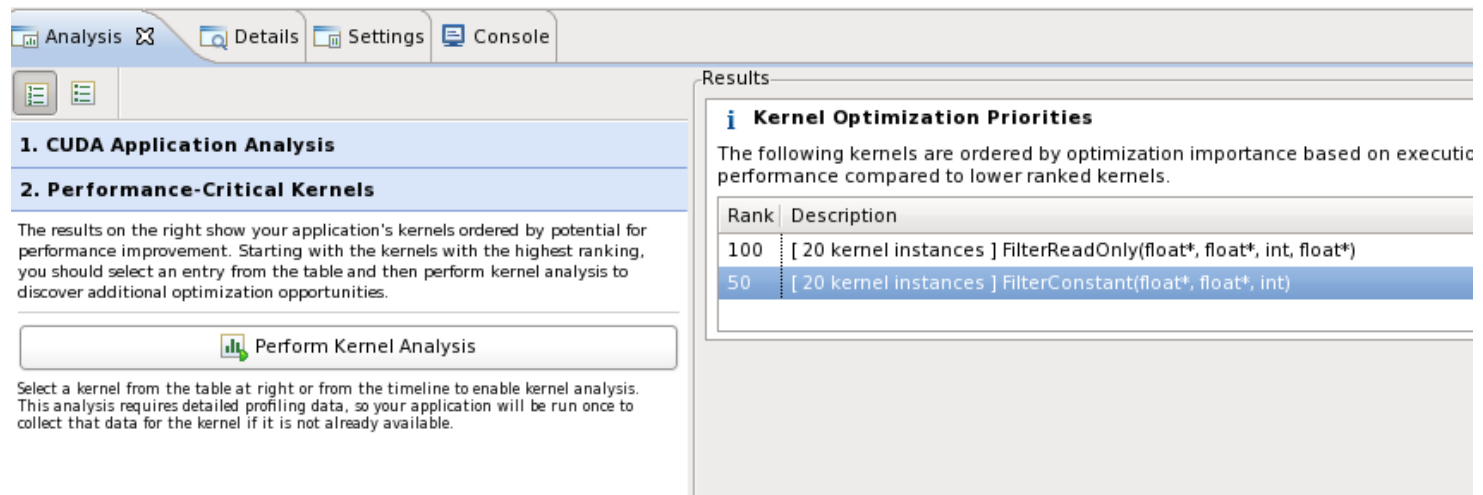
NVIDIA Visual Profiler

- CPU/GPU Timeline View – Are my GPUs Busy?



Guided Performance Analysis


- Profiler lists kernels which are best optimization targets
 - Ranked based on execution times and achieved occupancy



1. CUDA Application Analysis

2. Performance-Critical Kernels

The results on the right show your application's kernels ordered by potential for performance improvement. Starting with the kernels with the highest ranking, you should select an entry from the table and then perform kernel analysis to discover additional optimization opportunities.

 Perform Kernel Analysis

Select a kernel from the table at right or from the timeline to enable kernel analysis. This analysis requires detailed profiling data, so your application will be run once to collect that data for the kernel if it is not already available.

Results

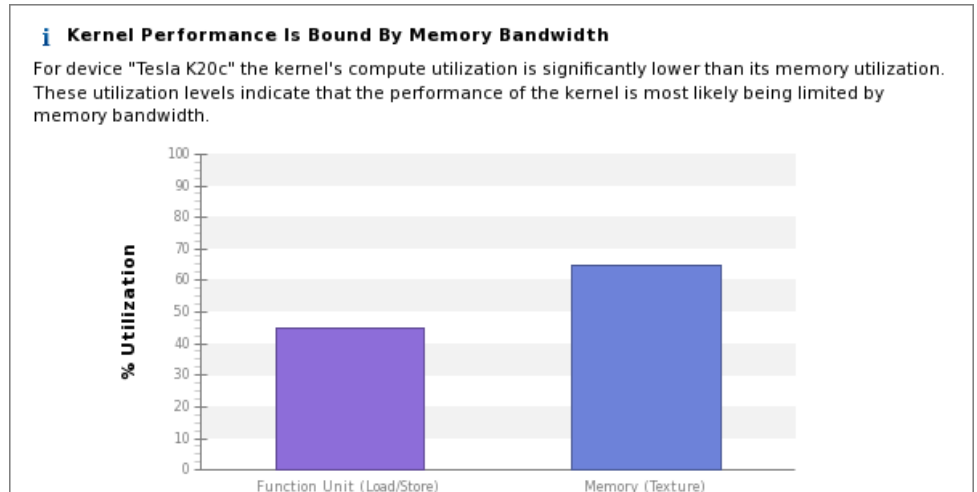
i Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution performance compared to lower ranked kernels.

Rank	Description
100	[20 kernel instances] FilterReadOnly(float*, float*, int, float*)
50	[20 kernel instances] FilterConstant(float*, float*, int)

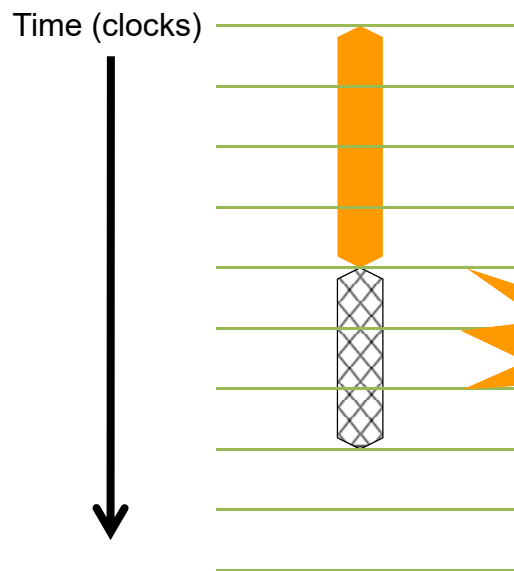
Guided Performance Analysis

- Identifies what is likely limiting performance of a kernel
 - Latency
 - Compute Resources
 - Memory Bandwidth



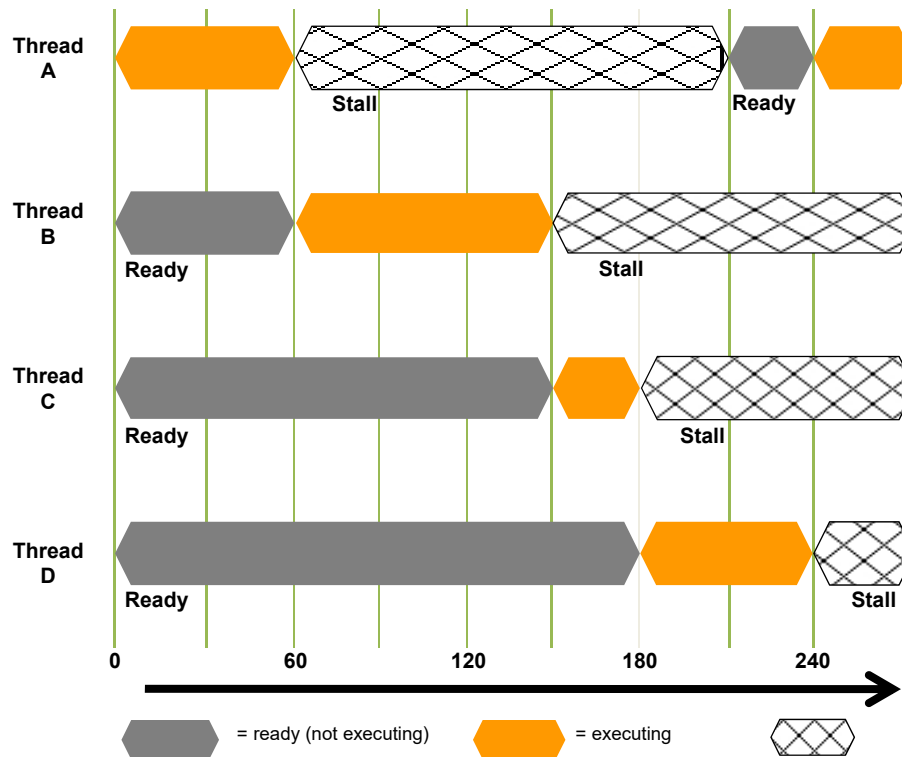
Processor Stalls

- Stalls (or idling) occurs when a processor cannot execute the next instruction due to a dependency on the previous instruction



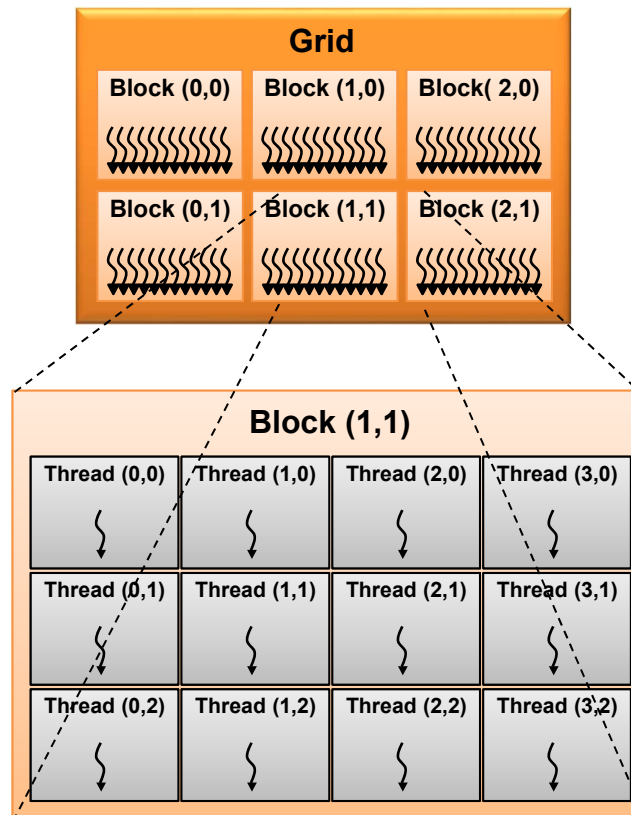
```
float sum(float* a, int const N)
{
    float sum = 0.0f;
    for(int i = 0; i < N; i++)
    {
        float temp = a[i];
        sum += temp;
    }
    return sum;
}
```

Hiding Latency in GPUs



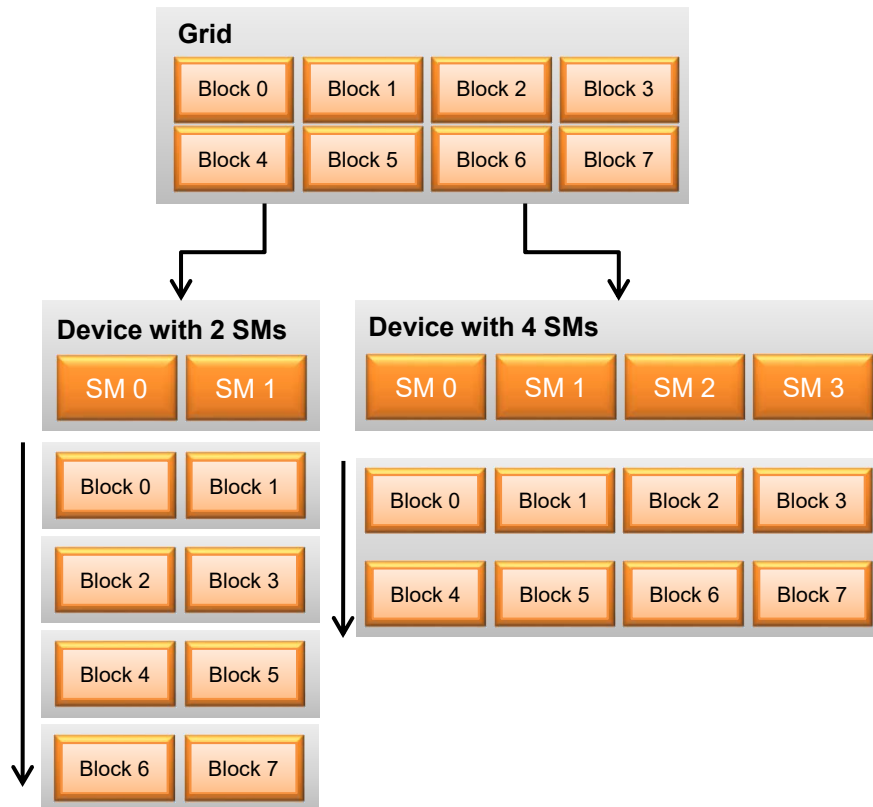
- GPUs minimize the impacts of stalls by interleaving execution of many threads on a single core
- Data-parallel model:
 - Can always execute threads from different blocks independently
 - Can execute threads within a block independently
 - Unless constrained by `__syncthreads()`

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)
- A block will execute on one (and only one) multiprocessor
 - However, a multiprocessor can execute multiple blocks
- Blocks must be independent!

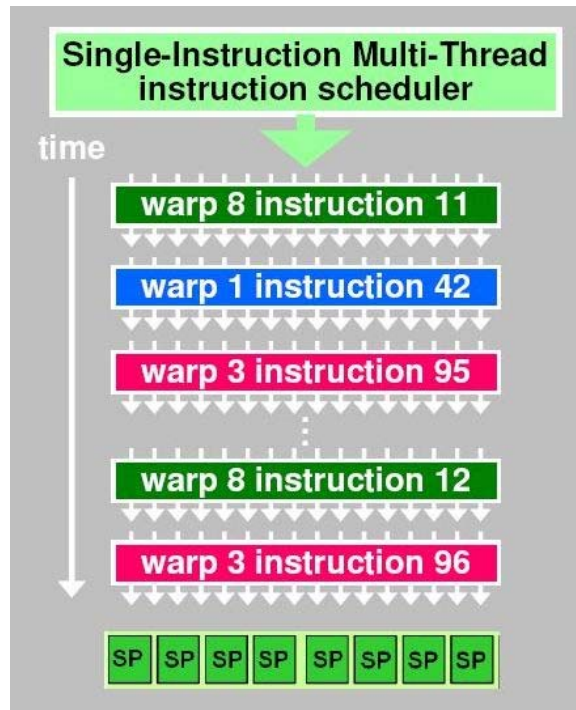
What is a Warp?

- The SM creates, manages, schedules, and executes threads in groups of parallel threads called *warps*
 - For current GPUs the warp size is 32 threads
 - `warpSize` is a built in variable available in kernels like `blockDim`
 - Threads are assigned to a warp based on `threadIdx.x` first, followed by `threadIdx.y` and then `threadIdx.z`
- Individual threads composing a warp start together at the same program address but are otherwise free to branch and execute independently
- When a multiprocessor is given one *or more* thread blocks to execute, it splits them into warps that get scheduled by the SM

Resource Allocation

- When a block is scheduled to run on a multiprocessor, resources in that SM are allocated on an exclusive basis
 - Shared memory – per block
 - Registers – per block, with unique registers for each thread within the block
- This enables zero-overhead switching between warps at any point in their execution

Warp Execution

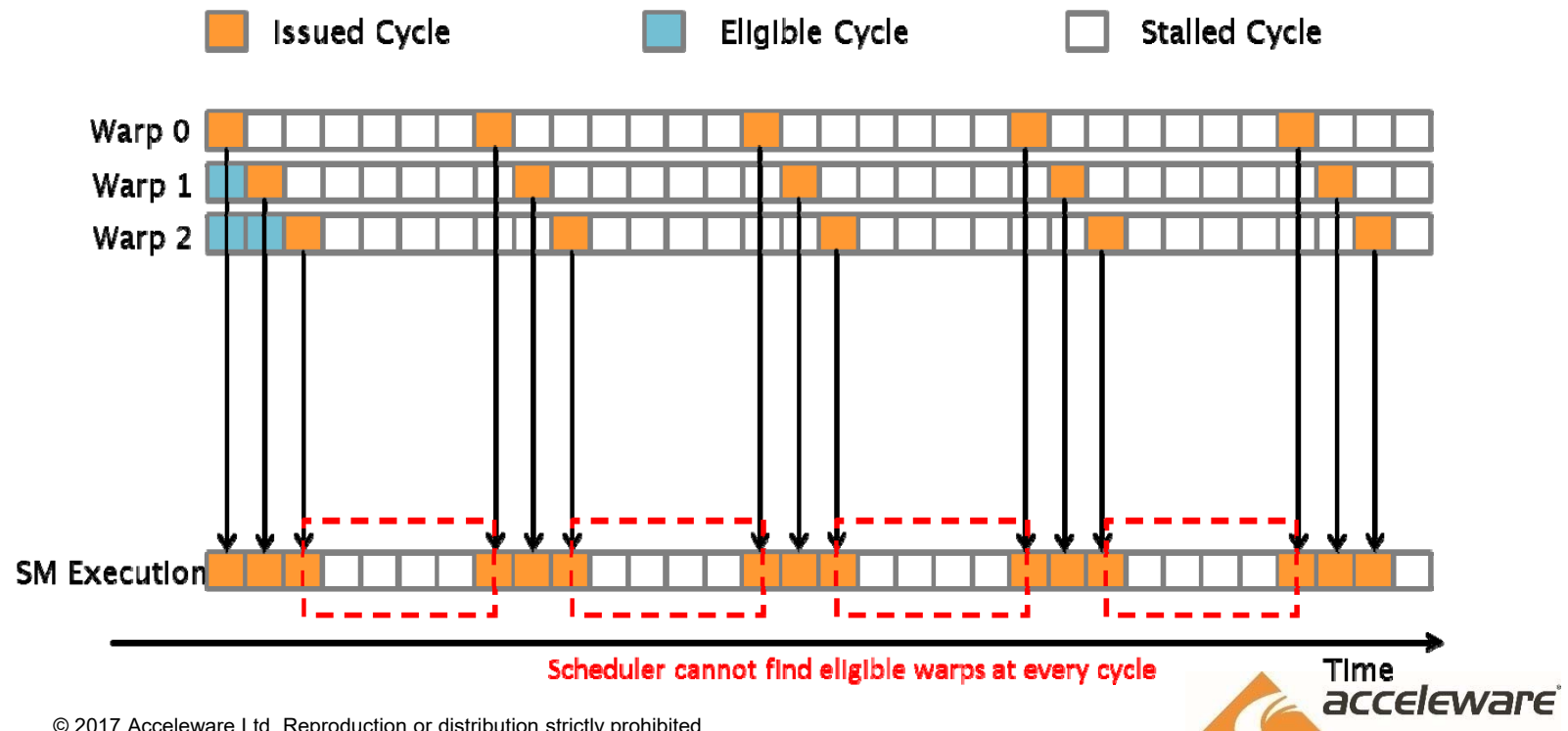


*Image courtesy of NVIDIA

- Scheduler evaluates all available warps to find one that is ready to execute an instruction
 - Warps are eligible if:
 - Resources are available to execute the next instruction
 - Arguments for instruction are ready
- The scheduler re-evaluates all available warps to find one that is ready to execute an instruction
- This effectively hides latency if:
 - There is a high ratio between arithmetic and memory operations/synchronization
 - OR There are enough warps to select from
- Instructions for a warp are still executed in order!

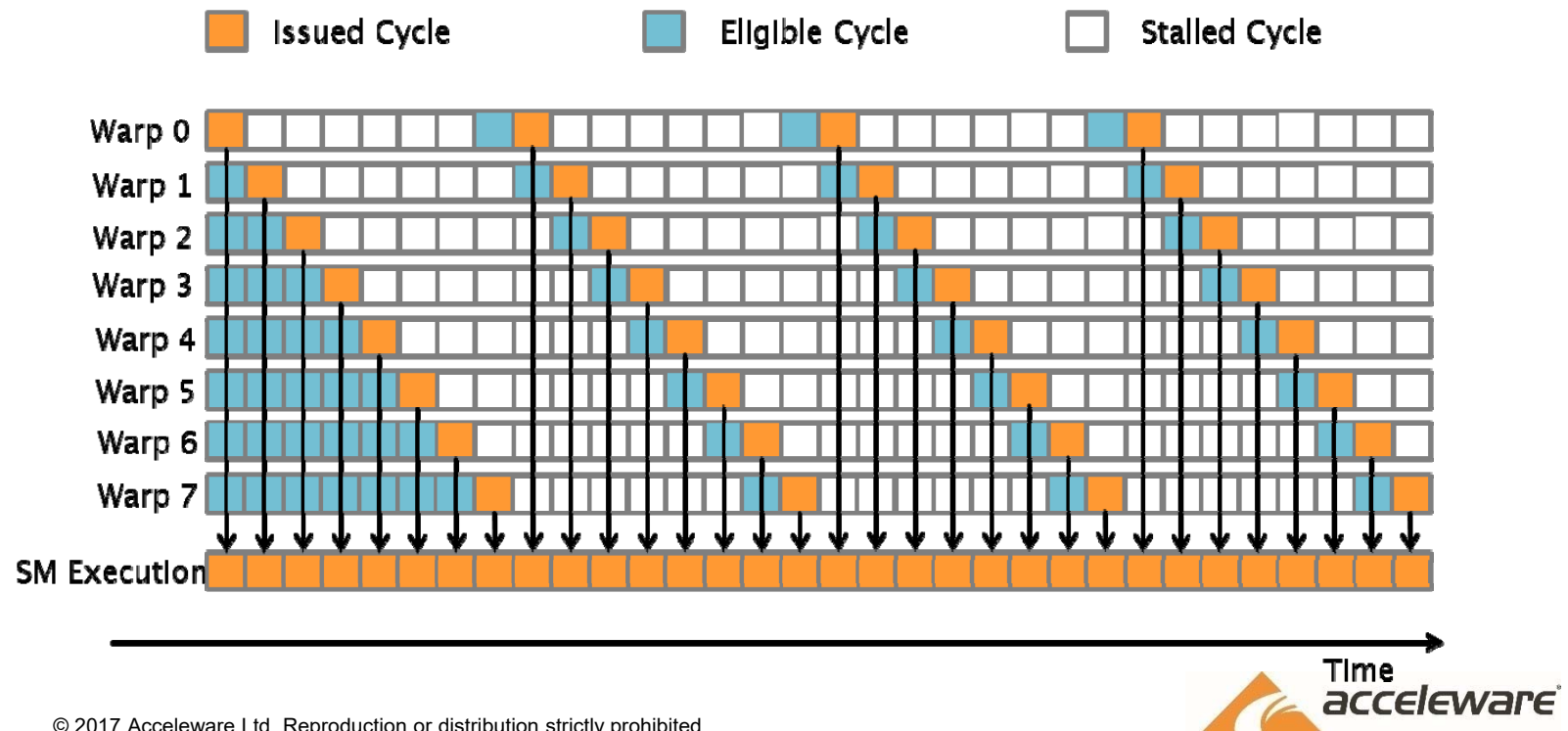
Masking Latency

- GPUs mask latency by having a lot of work in flight



Masking Latency

- Running more warps on an SM increases the ability to mask latency



Occupancy

- **Occupancy** = active warps / maximum active warps multiprocessor
- Determined by: block size, resource usage and hardware limits

Limitations	CC 2.x	CC 3.0-3.5	CC 3.7	CC 5.x	CC 6.x
Max # of threads per block	1024	1024	1024	1024	1024
Warp size (# of threads)	32	32	32	32	32
# 32-bit Registers/SM	32K	64K	128K	64K	64K
Maximum Shared memory/SM (KB)	48	48	112	64/96	64/96
Max # active blocks/SM	8	16	16	32	32
Max # active warps/SM	48	64	64	64	64
Max # active threads/SM	1536	2048	2048	2048	2048

Occupancy Calculation Example

Limitations	CC 6.0
Max # of threads per block	1024
Warp size (# of threads)	32
# 32-bit registers/SM	64K
Shared memory/SM (KB)	64
Max # active blocks/SM	32
Max # active warps/SM	64
Max # active threads/SM	2048

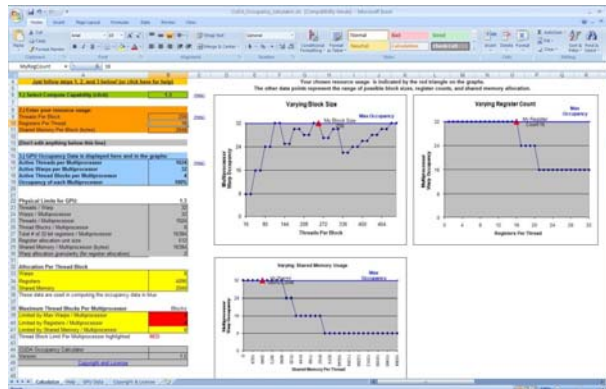


- Tesla P100 (Compute 6.0)
 - 256 threads/block
 - 18KB of shared memory/block
 - 32 registers/thread
- How many blocks can run at the same time on a streaming multiprocessor?

Occupancy Calculation Continued

- Each block is 256 threads and the maximum number of threads per SM is 2048
 - Limit of $2048/256 = 8$
- Each block requires 256 x 32 registers and an SM has 65536 registers
 - Limit of $65536/(256*32) = 8$
- Each block requires 18KB of shared memory and an SM has 64KB of shared memory
 - Limit of $64KB/18KB = 3$
- A streaming multiprocessor can run a maximum of 32 blocks
- The limit is the MIN (8, 8, 3, 32) = 3, and limited by shared memory

Occupancy Calculation Concluded



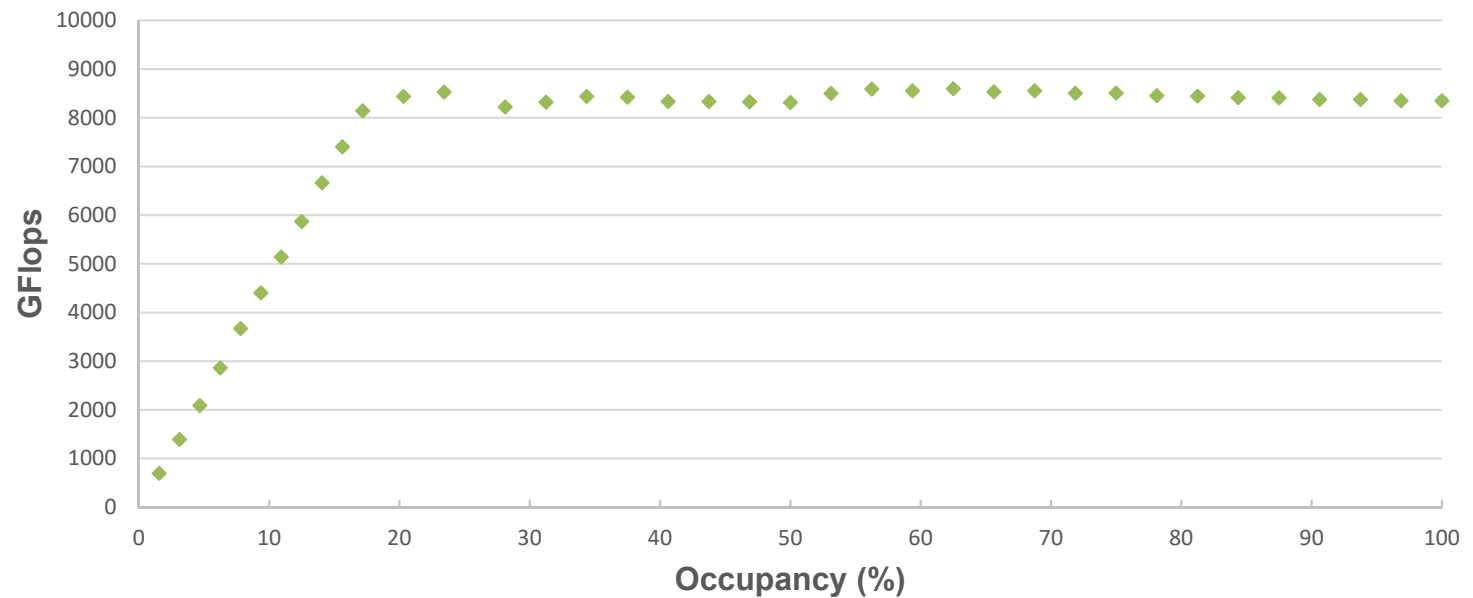
- We can run 3 blocks at a time
 - Limited by shared memory to 3 blocks
 - 3 blocks x 256 threads = 768 threads
 - $768 / 2048$ max threads = 37.5% occupancy
- CUDA Occupancy Calculator and Profilers do this calculation for you!

Occupancy and Performance

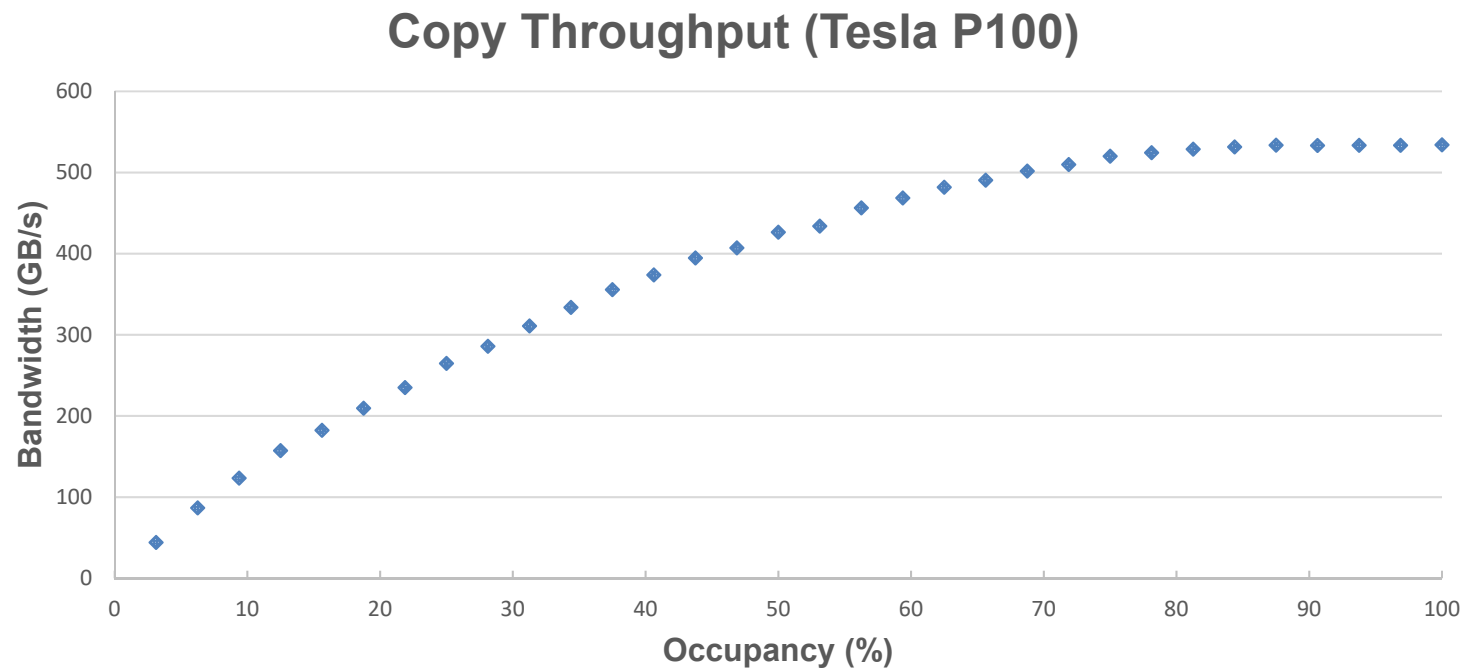
- Switching to other warps to mask latency is key to performance!
 - Need 'enough' other work to mask
 - May not need 100% occupancy!
 - Once you've reached the threshold, additional occupancy won't improve performance
 - 'Enough' occupancy depends on the code
 - Memory latency is typically higher than arithmetic latency

Occupancy Examples

Compute Bound Kernel (Tesla P100)



Occupancy Examples



Optimizing Latency Bound Kernels

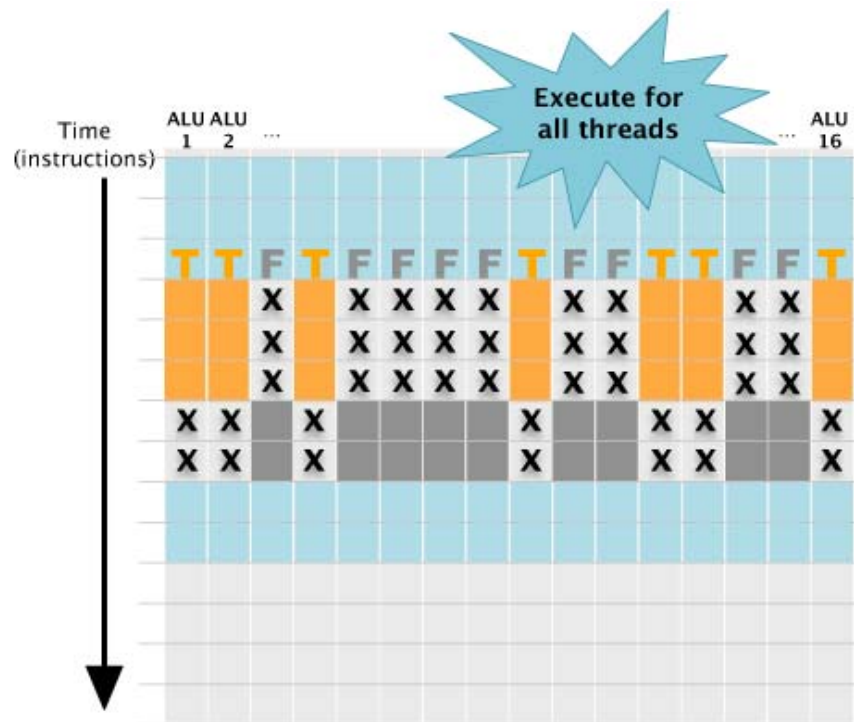
- Low theoretical occupancy
 - Adjust block size, register usage, shared memory usage
 - Occupancy Calculator can help!
- Low achieved occupancy
 - eg. Launching 33 blocks of a kernel that runs with 2 blocks/SM on a GPU with 16 SMs
 - Theoretical occupancy is 100%
 - Achieved occupancy is ~50%

Efficiency and Branching

- GPU executes instruction for the entire warp
- Full efficiency is realized when all 32 threads of a warp agree on their execution path
- If threads of a warp **diverge** via a conditional branch, the warp serially executes each branch path
 - Only the active threads in the warp execute the instructions for a path. Threads from the warp on other paths are idle.
 - Therefore, hardware is underutilized when this occurs
 - Branch divergence occurs only within a warp!

```
if(threadIdx.x & 0x01)
    c = a + b;
else
    c = a - b;
```

Efficiency and Branching



```
int idx = threadIdx.x;
int x = a[idx];

if (x < 0) {
    f = sin(x * M_PI);
    f *= f;
    q = 2*M_SQRT2*f;
} else {
    f = exp(x * M_PI);
    q = sqrt(f);
}

b[idx] = f;
c[idx] = q;
```

Iteration/Loop Unrolling (cont.)

- For small loop bodies reduce the loop overhead by unrolling
- For known trip counts the compiler will unroll the loop
 - Ensure trip count is known at compile time by templating the kernel

```
template<int FILTER_SIZE> __global__ void filter(...)  
{  
    ...  
    for(int i = 0; i < FILTER_SIZE; i++) {...}  
    ...  
}
```

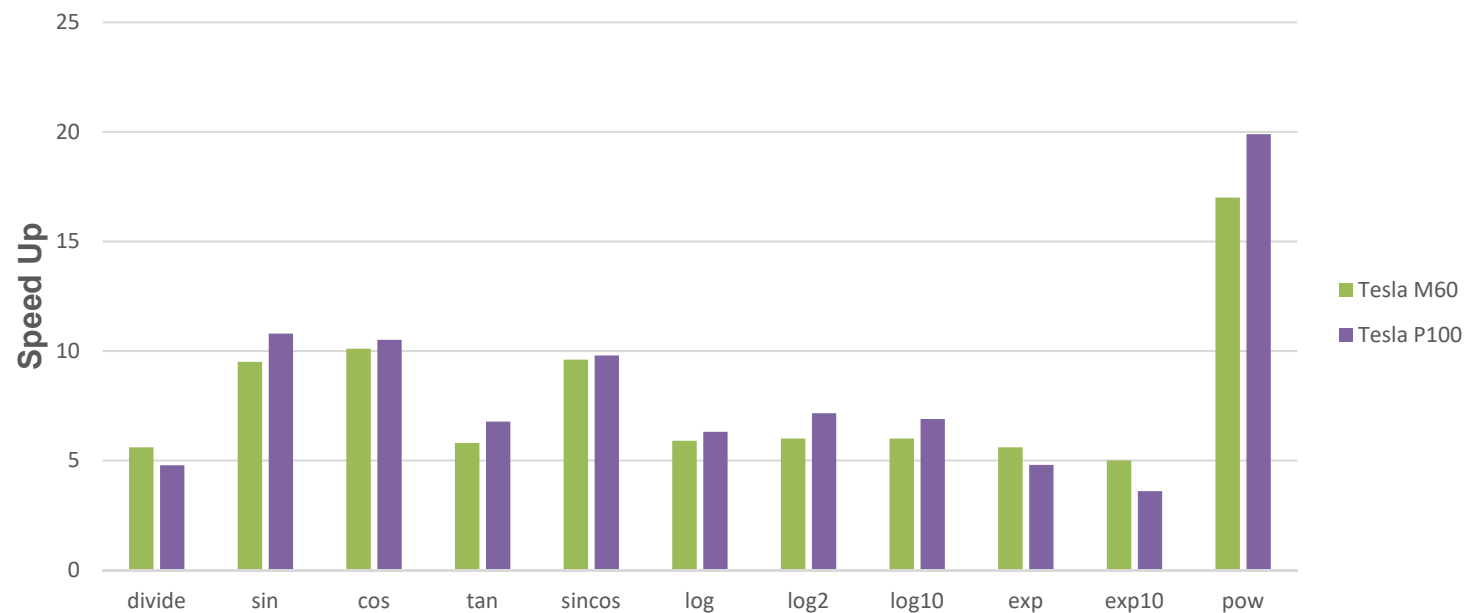
- Control unrolling with **#pragma unroll**
 - `#pragma unroll 1` prevents unrolling
 - `#pragma unroll N` unrolls the loop N times
 - `#pragma unroll` unrolls the loop completely

'Fast' Math

- Consider intrinsic function
 - Approximate versions of many math library functions
 - Slightly reduced accuracy, higher throughput
 - `__sinf()` vs. `sinf()`
- Compiler flags
 - `--use_fast_math` – Redirect all math functions to intrinsics
 - `-prec-div=false`
 - `-prec-sqrt=false`

Runtime Math Library

Single Precision Fast Math Library



Shuffle Instruction on CC 3.0+

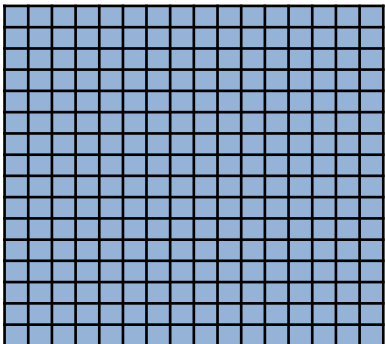
- Allows threads within a warp to share data
 - Use this functionality instead of shared memory
 - A single shuffle instruction can replace shared memory write/sync/read sequence
- Any thread can read any other thread's data within a warp
 - `__shfl()`
- Support for useful patterns
 - `__shfl_up()`
 - `__shfl_down()`
 - `__shfl_xor()`

Memory Access Patterns

- Simple task:
 - Traverse and increment contiguous 2D memory (stored as row-major)

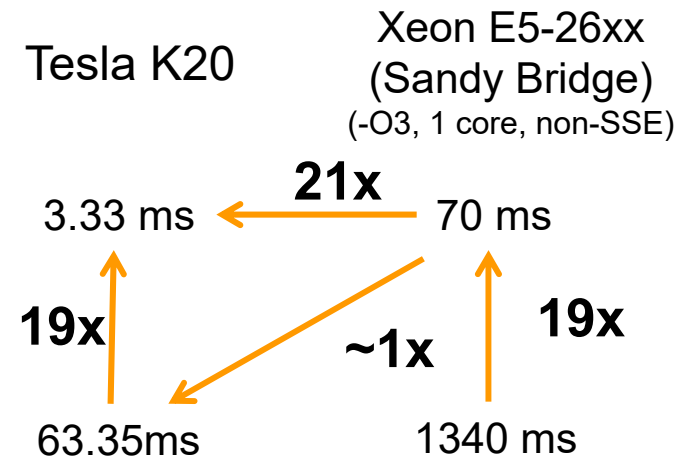
Columns = 8192

Rows = 8192

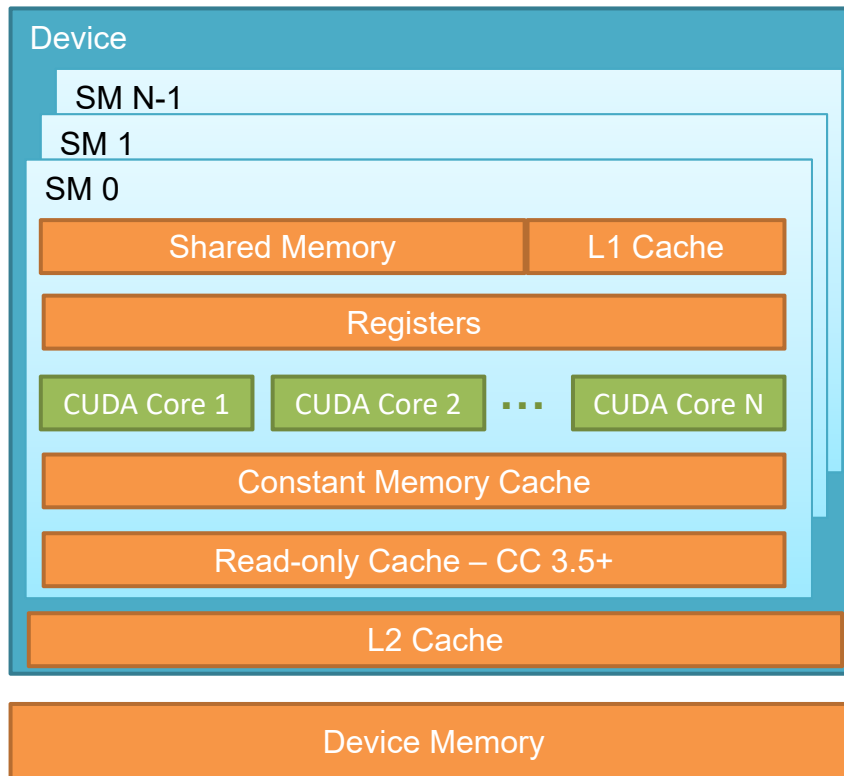


```
for(row)
  for(column)
    buf[x,y] ++;
```

```
for(column)
  for(row)
    buf[x,y] ++;
```



CUDA Memory Spaces (Review)



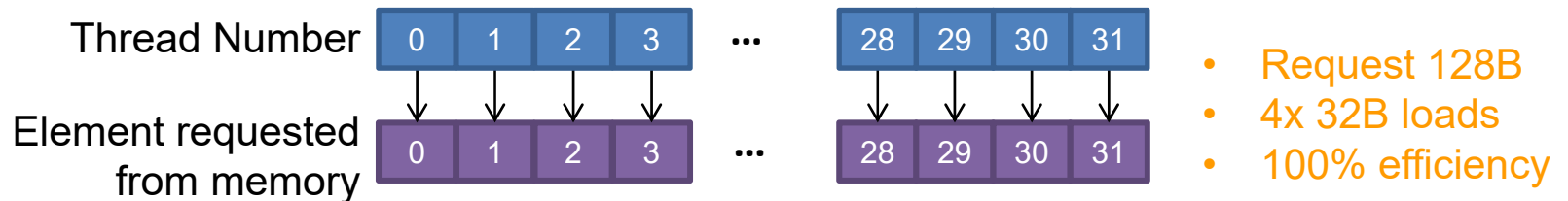
- Many memory resources with different performance characteristics
 - SM Resources
 - Share Memory
 - L1 Cache
 - Registers
 - Constant Cache
 - Read-Only Cache/Texture
 - Device Resources
 - L2 Cache
 - Device Memory

Global Memory

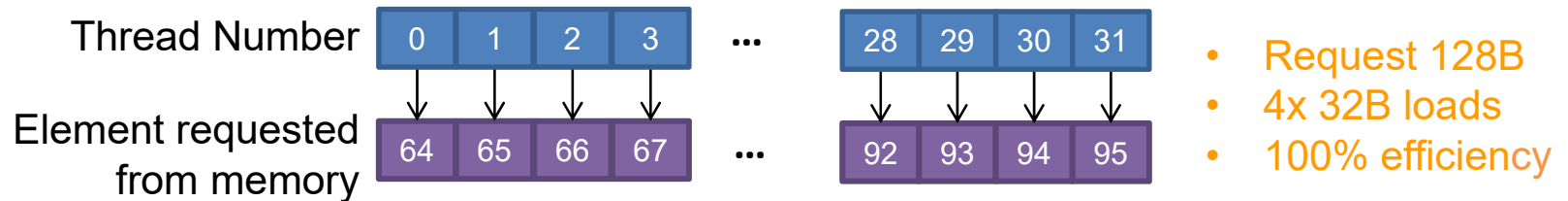
- Global memory is most efficient when threads in a warp access an aligned contiguous region of memory
 - Individual thread requests can be combined (or **coalesced**) into fewer and larger transactions
- Can be quite inefficient otherwise
 - Minimum transaction size across the bus
 - For Kepler GPUs, minimum transaction size is a 32B aligned L2 cache line
- Specifics depend on GPU architecture, word size (1,2,4,8 byte words), and access pattern

Access Pattern Examples (1)

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;  
float temp = input[idx];
```



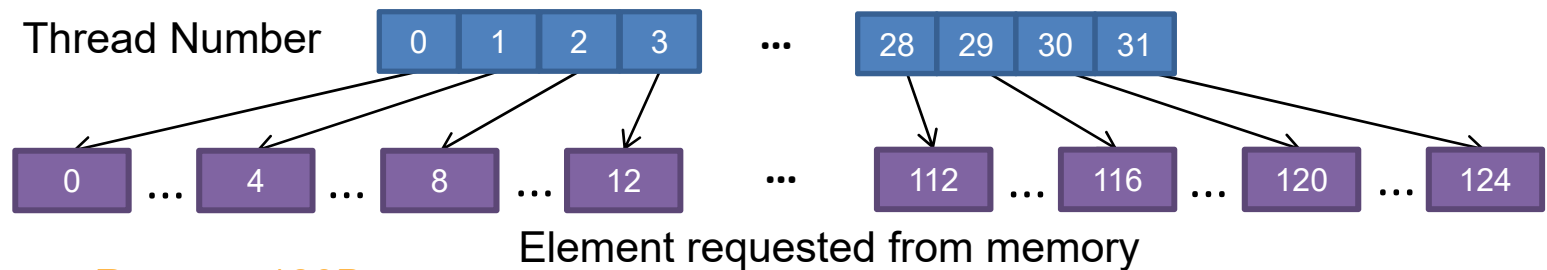
```
int idx = threadIdx.x + blockIdx.x * blockDim.x;  
float temp = input[idx+64];
```



Access Pattern Examples (2)

- Example: Aligned to memory boundary but not contiguous

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;  
float temp = input[idx*4];
```

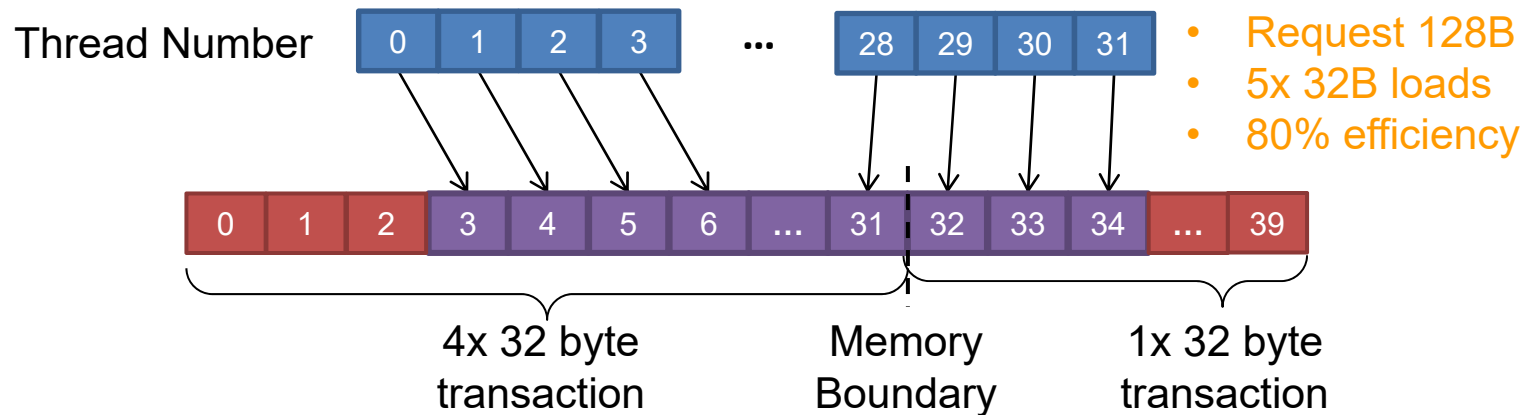


- Request 128B
- 16x 32B loads
- 25% efficiency

Access Pattern Examples (3)

- Example: Contiguous but not aligned to memory boundary

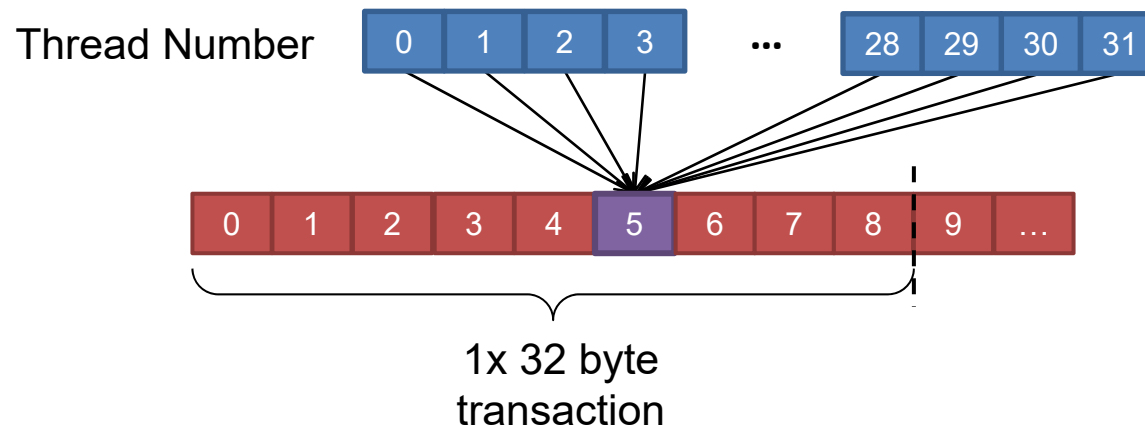
```
int idx = threadIdx.x + blockIdx.x * blockDim.x;  
float temp = input[idx + 3];
```



Access Pattern Examples (4)

■ Example: Broadcast

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;  
float temp = input[blockIdx.x];
```



- Request 4B
- 1x 32B load
- 12.5% efficiency

Global Memory – Design Considerations

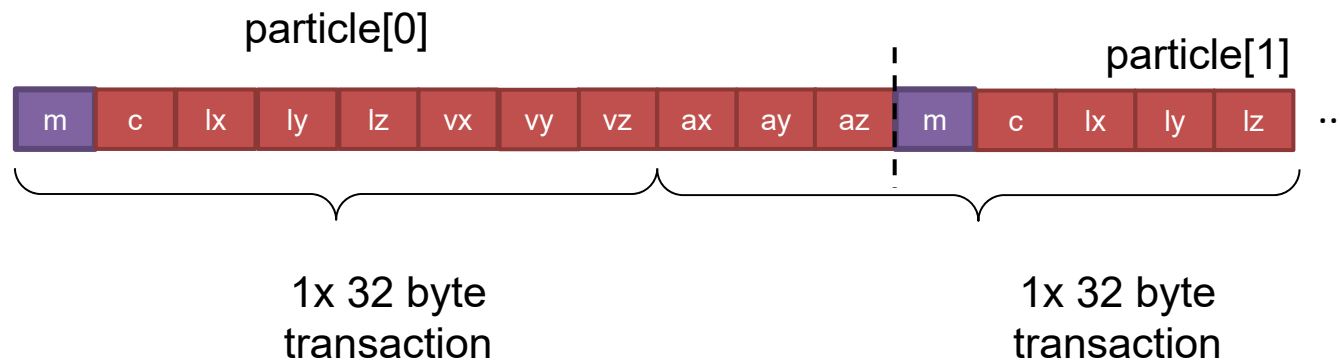
- Assume the following structure

```
typedef struct
{
    float m, c;          // Mass and Charge
    float lx, ly, lz;    // Location
    float vx, vy, vz;    // Velocity
    float ax, ay, az;    // Acceleration
} particle;
```

- Passed into the following kernel

```
__global__ MyKernel (particle *p)
{
    ...
    float temp = p[gIdx].m * 2;
}
```

Global Memory Access Pattern



- Assuming non-caching load
- Request 128B
- 32x 32B loads
- 12.5% efficiency

Access Pattern Comments

- Threads in a warp are accessing memory with a stride of `sizeof(particle)`
 - Not coalesced!
- Consider using a structure of arrays (SoA) data structure instead
 - 12 arrays of floats, one for each property
 - Generally preferred for CPU vectorization anyway!
 - Read-only cache

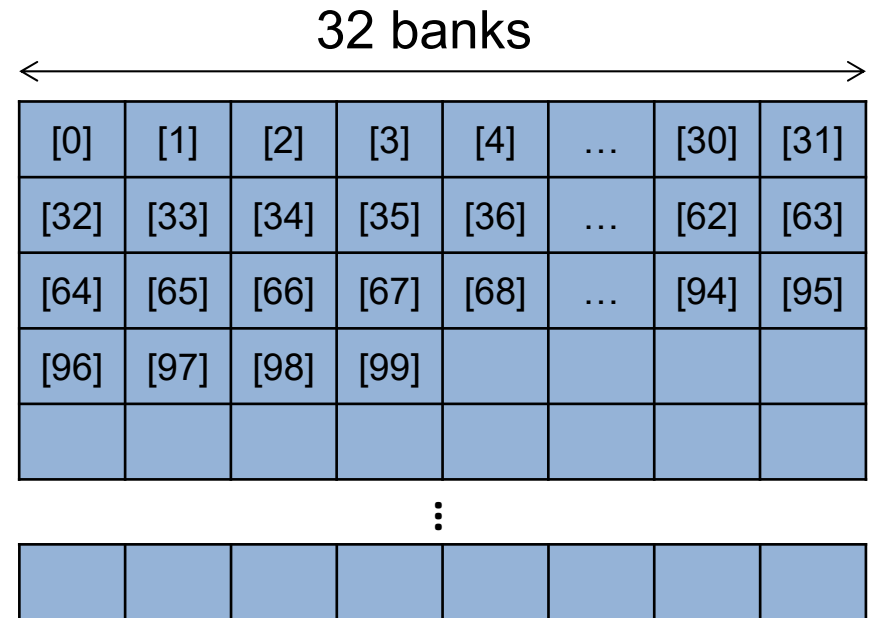
Shared Memory Operation

- Shared memory organized as **banks** each with a width of one word
 - Word is 4B or 8B depending on the architecture
 - Each bank can read/write 1 word per cycle
- Reads – fetch words and distribute amongst threads
 - Words can be broadcast to multiple threads
 - Banks can only deliver one word/cycle
 - Requests for different words in the same bank are called **bank conflicts**
- Writes
 - Multiple threads writing to same address cause bank conflicts

Shared Memory Banks (CC 2.x, CC 5.x, and CC 6.x)

- Shared memory has 32 banks
 - Words are 32-bits wide
 - Successive 32-bit words are assigned to successive banks
- CC 3.x (Kepler) also has 32 banks but has a word size of 64-bits
 - For more information on the Kepler shared memory architecture see:
<http://www.acceleware.com/blog/maximizing-shared-memory-bandwidth-nvidia-kepler-gpus>

```
__shared__ float array[100];
```



Bank Conflicts Example (CC 2.x, CC 5.x, and CC 6.x)

Bank Conflicts

```
float temp = array[32*threadIdx.x];
```

← 32 banks →

[0]	[1]	...	[30]	[31]
[32]	[33]	...	[62]	[63]
[64]	[65]	...	[94]	[95]
[96]	[97]			

No Bank Conflicts

```
float temp = array[threadIdx.x];
```

← 32 banks →

[0]	[1]	...	[30]	[31]
[32]	[33]	...	[62]	[63]
[64]	[65]	...	[94]	[95]
[96]	[97]			

Avoiding Bank Conflicts

- For best shared memory performance:
 - A) Access all 32 banks (i.e. each thread accesses a unique bank)
 - or*
 - B) Access the same bank *and* the same element in the bank
 - Words are broadcast (multicast) to all requesting threads
- Sometimes possible to avoid bank conflicts with padding

Read-Only Cache

- 12-48KB per SM
 - Traditionally the texture cache
- How to access:

```
__global__ void kernel (const float* __restrict__ input, ...)  
{  
    float temp = input[0]; // Stored in Read-Only Cache  
}
```

```
__global__ void kernel (float* input,...)  
{  
    float temp = __ldg(&input[0]); // Stored in Read-Only Cache  
}
```


Constant Memory vs. Read-Only Cache

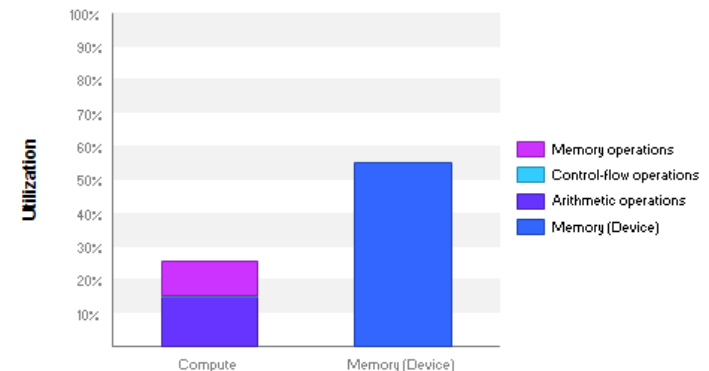
- Prefer constant memory when:
 - Data fits in 64KB
 - Values are broadcast to all threads in a warp
- Prefer read-only cache when:
 - Read-only data where constant memory isn't a good fit
 - For read-only strided accesses from global memory
 - eg. Access Pattern Example (2)

Summary - Latency Limited Kernels

- Low theoretical occupancy
 - Adjust block size, register usage, shared memory usage
 - Occupancy Calculator can help!
- Low achieved occupancy
 - eg. Launching 33 blocks of a kernel that runs with 2 blocks/SM on a GPU with 16 SMs
 - Theoretical occupancy is 100%
 - Achieved occupancy is ~50%
 - Consider concurrent kernel execution

i Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "NVS 4200M". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



Summary - Compute Bound Kernels

- Branching/warp divergence
- Loop unrolling
- Fast math
- Shuffle

⚠ GPU Utilization Is Limited By Function Unit Usage

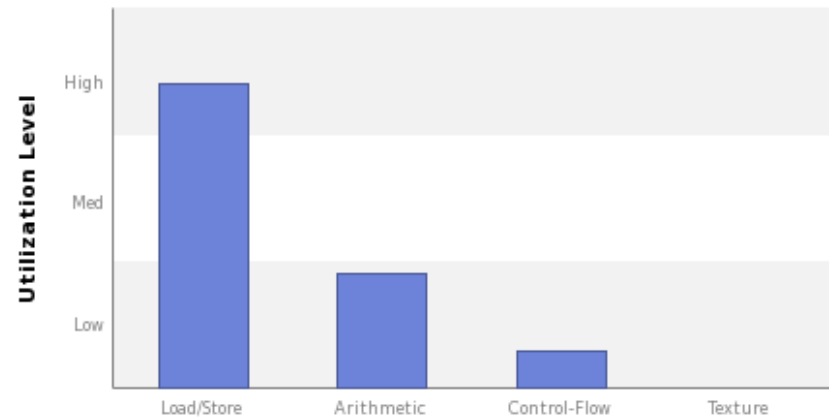
Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is potentially limited by overuse of the "Load/Store" function unit.

Load/Store - Load and store instructions for local, shared, global, constant, etc. memory.

Arithmetic - All arithmetic instructions including integer and floating-point add and multiply, logical and binary operations, etc.

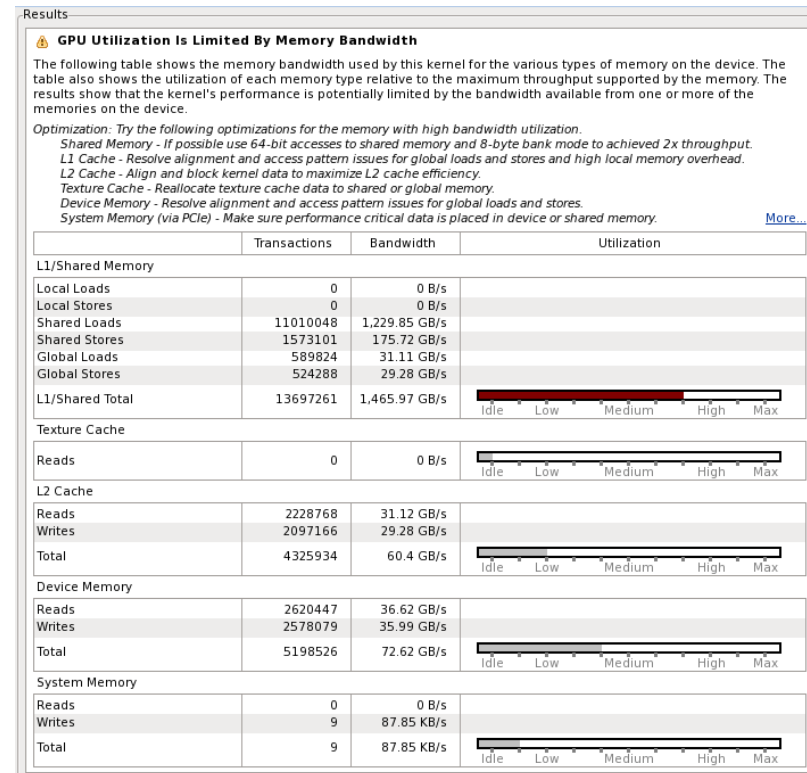
Control-Flow - Direct and indirect branches, jumps, and calls.

Texture - Texture operations.



Summary - Memory Bound Kernels

- Global Memory
 - Access patterns
 - Data structures
- Shared Memory
 - Bank conflicts
 - 64-bit accesses
- Constant memory vs. Read-Only Cache



Acceleware CUDA Training

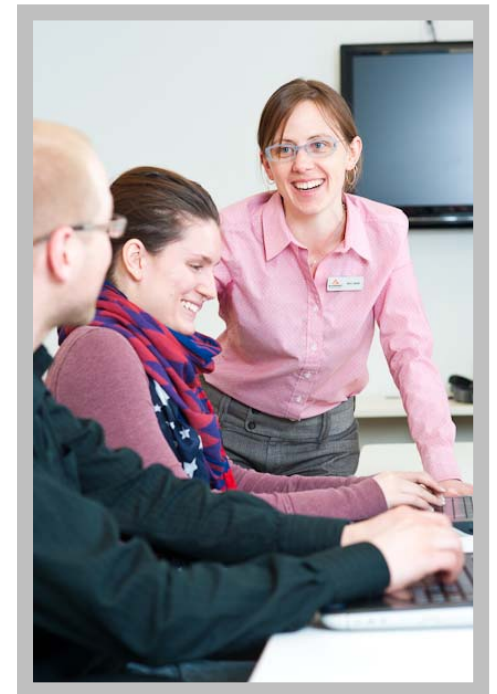
Scheduled CUDA Courses (also available online)

- 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