

# An Introduction to CUDA Programming

Catch Up on CUDA



Chris Mason

Director of Product Management, Acceleware

GTC Express Webinar

Date: June 8, 2016

# About Acceleware

## Programmer Training

- CUDA and other HPC training classes
- Over 100 courses taught worldwide
- <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 & modelling
- Electromagnetics

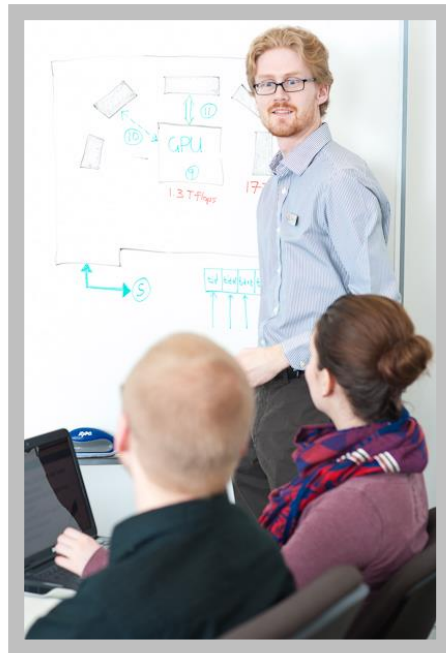


# Programmer Training

- CUDA and other HPC training classes
- 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



# 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



# Seismic Imaging & Modelling

## AxWAVE™

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

## AxRTM™

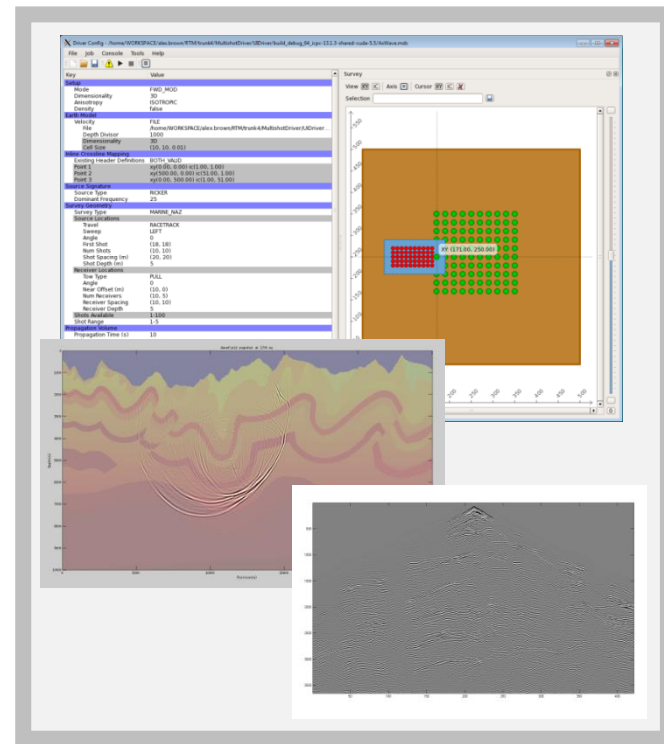
- High performance Reverse Time Migration application
- Isotropic, VTI, and TTI media

## 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

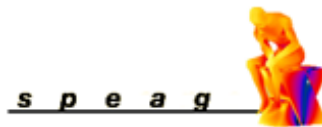


# Electromagnetics

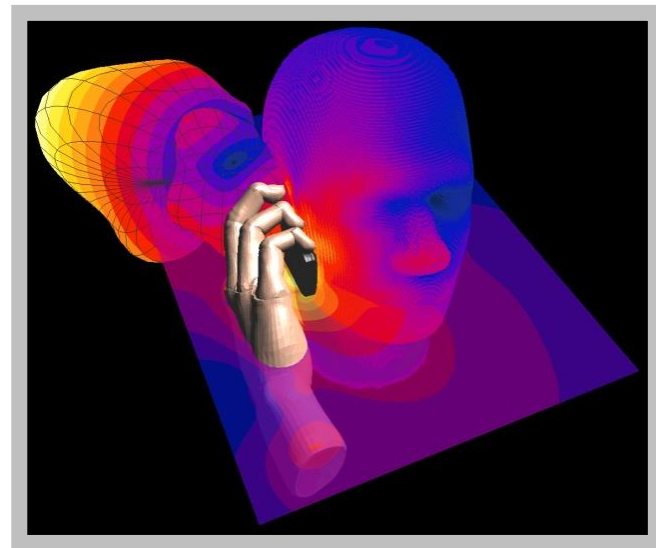
## AxFDTD™

- 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



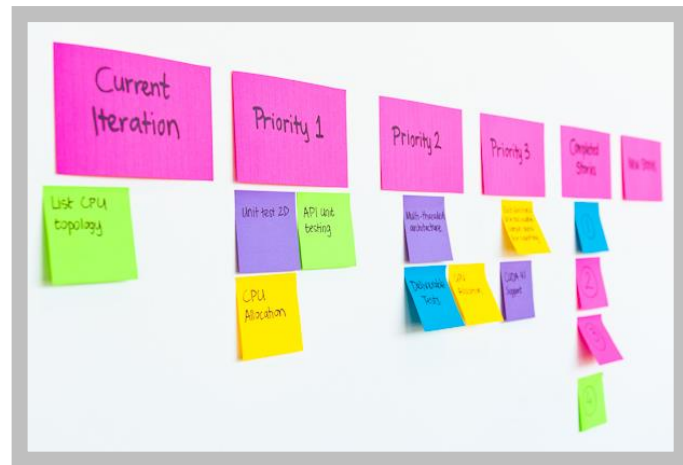
# Outline

CUDA overview

Data-parallelism

GPU programming model

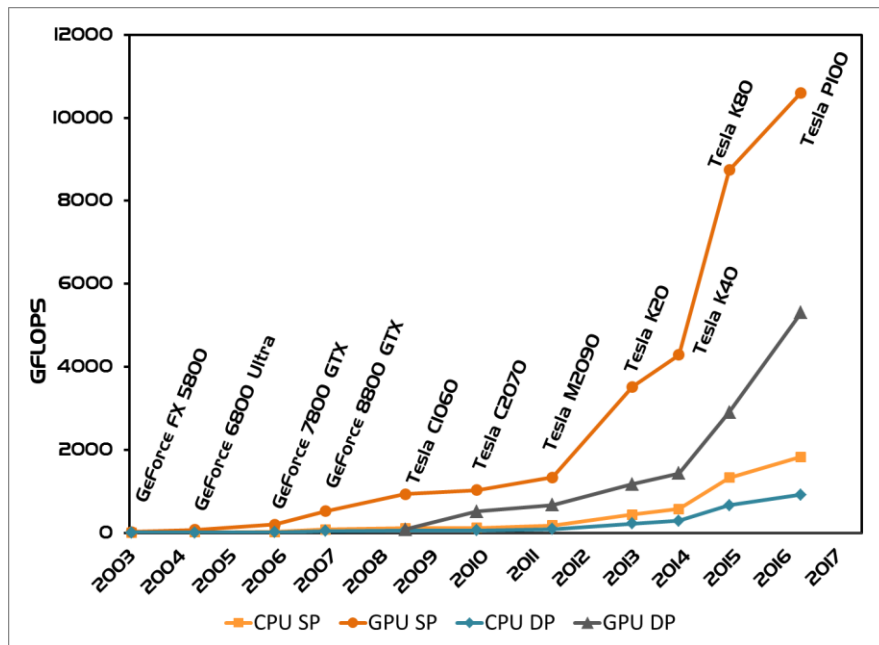
- GPU kernels
- Host vs. device responsibilities
- CUDA syntax
- Thread hierarchy



# Why use GPUs? Performance!





GPU advances are outpacing CPU advances

Continuing Moore's Law?





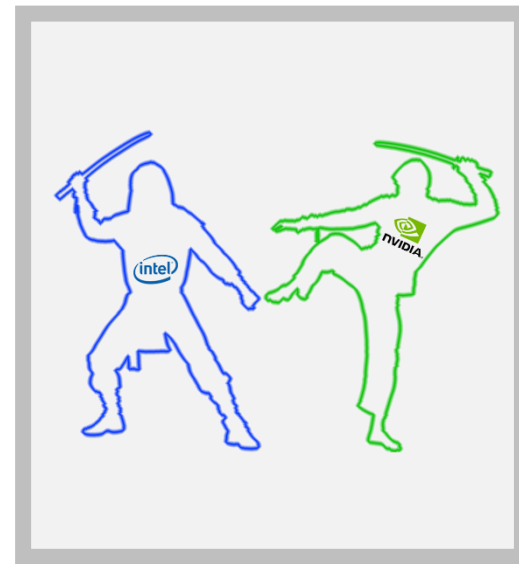
# Why use GPUs? Performance!

	Intel Xeon E5-2699v3 (Haswell-EP) 	NVIDIA Tesla M60 (Maxwell) 	NVIDIA Tesla K80 (Kepler) 	NVIDIA Tesla P100 (Pascal) 
Processing Cores	22	4096	4992	3584
Clock Frequency	2.2-3.6*	0.900-1.180 GHz	0.562-0.875 GHz	1.328-1.48 GHz
Memory Bandwidth	76.8 GB/s / socket	320 GB/s	480 GB/s	720 GB/s
Peak Tflops (single)	1.83 @ 2.6GHz	9.68 @ 1.180 GHz	8.74 @ 0.875GHz	10.6 @ 1.48GHz
Peak Tflops (double)	0.915 @ 2.6GHz	0.30 @ 1.180 GHz	2.91 @ 0.875GHz	5.3 @ 1.48GHz
Gflops/Watt (single)	9.1	32.2	29.1	35.3
Total Memory	>>24GB	16GB	24GB	16 GB

# GPU Potential Advantages

## Tesla K80 vs. Xeon E5-2699 v3

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

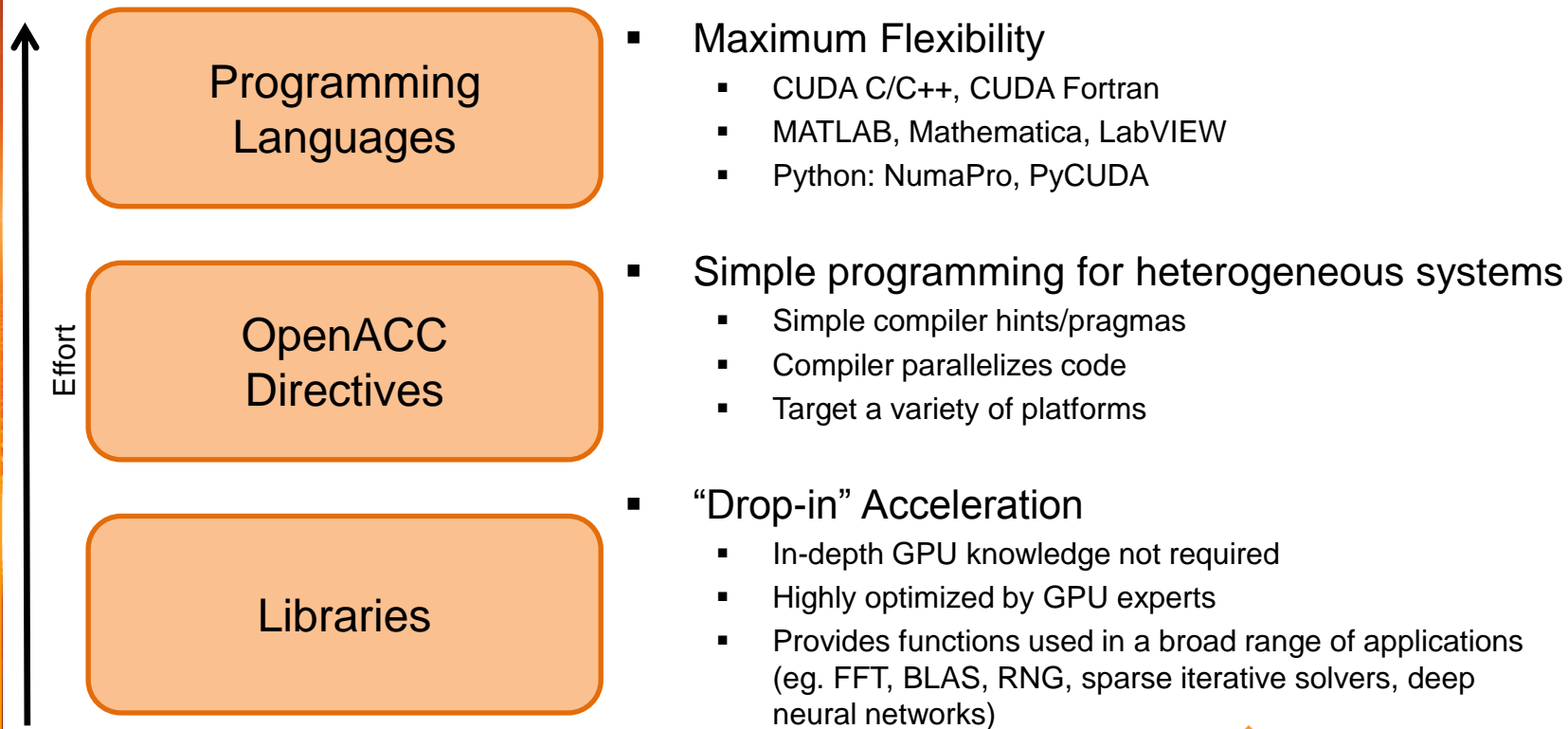


# GPU Disadvantages

- Architecture not as flexible as CPU
- Must rewrite algorithms and maintain software in GPU languages
- Discrete GPUs attached to CPU via relatively slow PCIe
  - 16GB/s bi-directional for PCIe 3.0 16x
- 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 Name	Compute Capability	GPUs	Example Features
Tesla	1.0	GeForce 8800, Quadro FX 5600, Tesla C870	Base Functionality
	1.1	GeForce 9800, Quadro FX 4700 x2	Asynchronous Memory Transfers
	1.3	GeForce GTX 295, Quadro FX 5800, Tesla C1060	Double Precision
Fermi	2.0	GeForce GTX 480, Tesla C2050	R/W Memory Cache
Kepler	3.0	GeForce GTX 680, Tesla K10	Warp Shuffle Functions PCI-e 3.0
	3.5	Tesla K20, K20X, and K40	Dynamic Parallelism
	3.7	Tesla K80	More registers and shared memory
Maxwell	5.0	GeForce GTX 750 Ti	New architecture
	5.2	GeForce GTX 970/980	More shared memory
Pascal	6.0	Tesla P100	Page Migration Engine



# CUDA Overview

Parallel computing architecture  
developed by NVIDIA



CUDA programming interface consists of:

- C language extensions to target portions of source code on the compute *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;
```

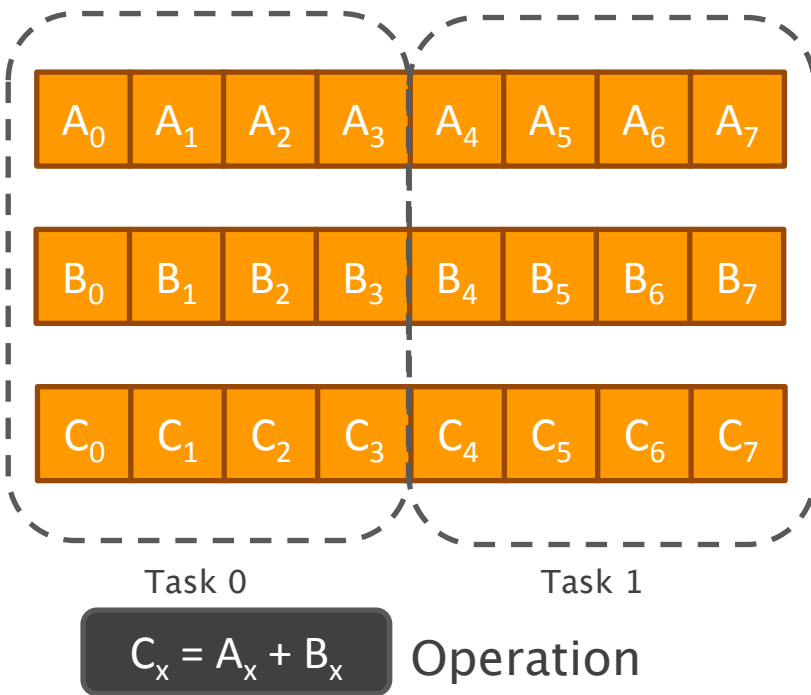
These 3 statements are  
independent!

```
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

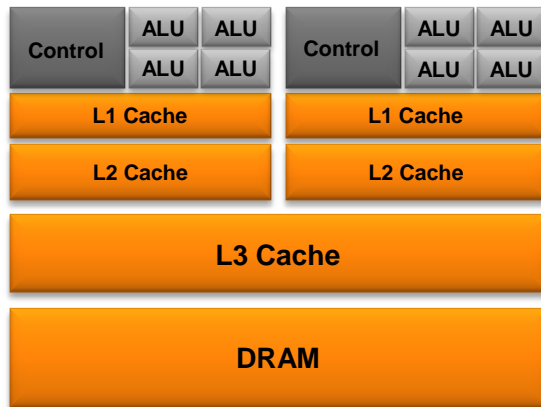


- Data set consisting of arrays A, B, and C
- Same operations performed on each element
$$C_x = A_x + B_x$$
- Two tasks operating on a subset of the arrays. Tasks 0 and 1 are independent. Could have more tasks.

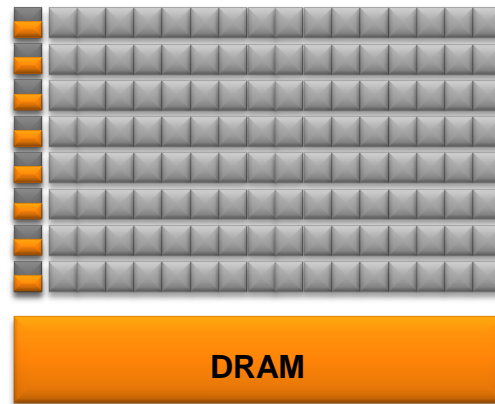
# Data-Parallel Computing on GPUs

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**

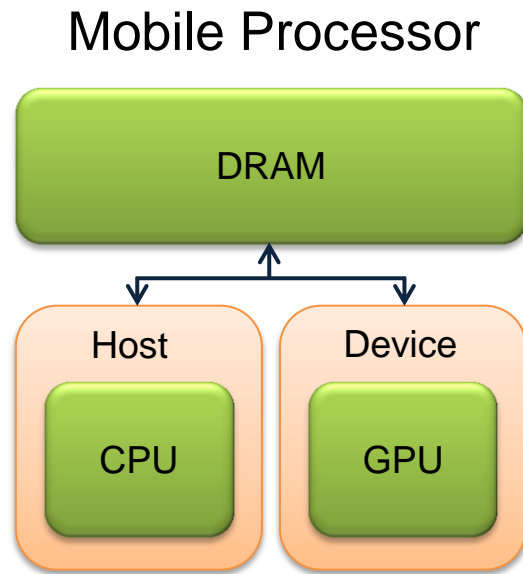
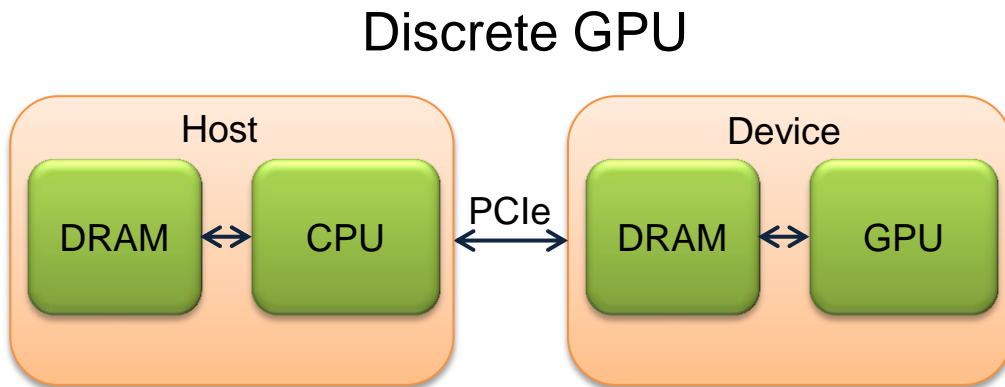


**GPU**



# 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**

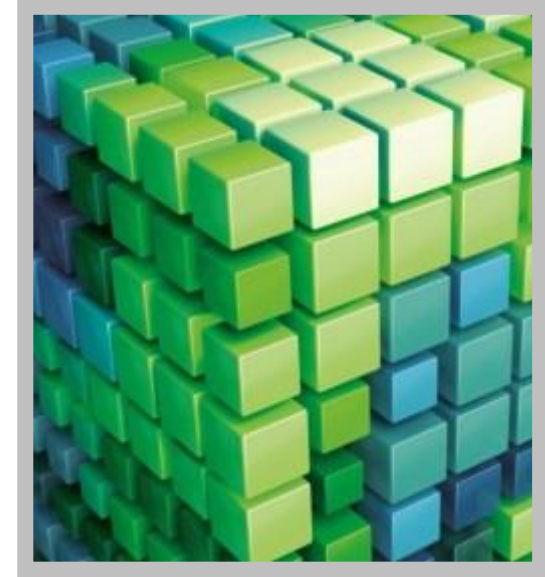
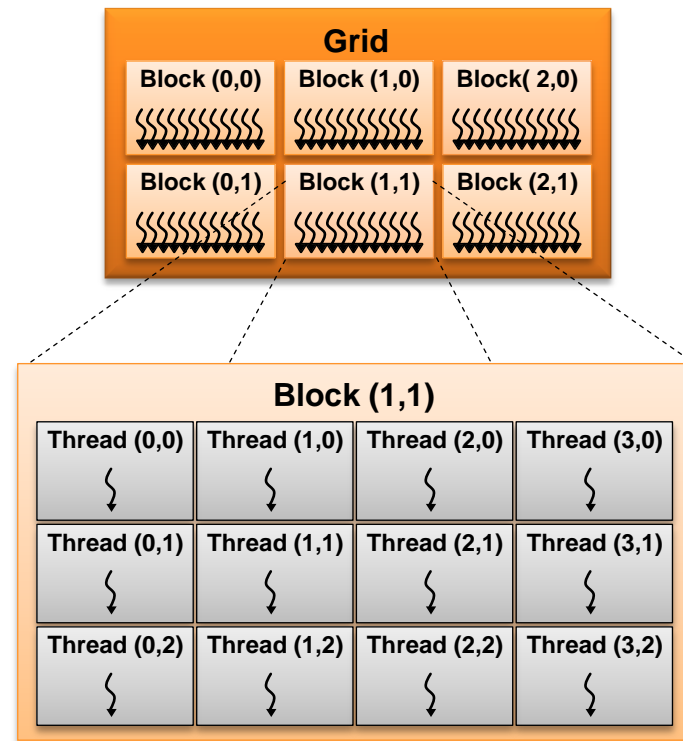


Image courtesy of NVIDIA Corp.

# CUDA Thread Hierarchy

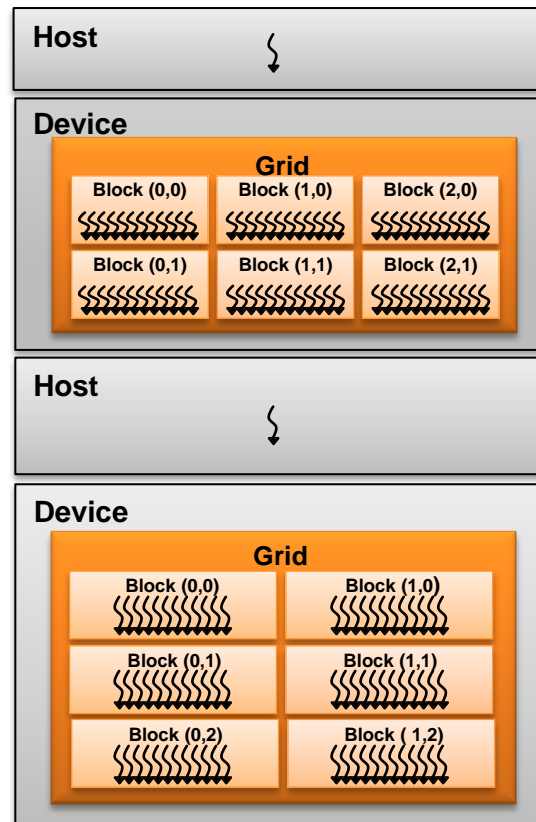
- Thread blocks and Grids can be 1D, 2D or 3D
- Dimensions set at launch time
- Thread blocks and grids do not need to have the same dimensionality
  - ie. 1D Grid of 2D Thread Blocks





# 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), or Driver API

- **This tutorial presentation uses CUDA C**
  - Uses host side C-extensions that greatly simplify code
- Driver API has a much more verbose syntax that clouds CUDA (parallel) fundamentals
- Same ability, same performance, but:
  - Historically Driver API provided slightly more control over host process interactions with GPU
  - Runtime API now supports all functionality of Driver API, and is interoperable with Driver API
  - **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)
Total Threads per Block		1024	1024
Grid Size	dGrid.x	65535	$2^{31}-1$
	dGrid.y	65535	65535
	dGrid.z	65535	65535
Block Size	dBlock.x	1024	1024
	dBlock.y	1024	1024
	dBlock.z	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 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

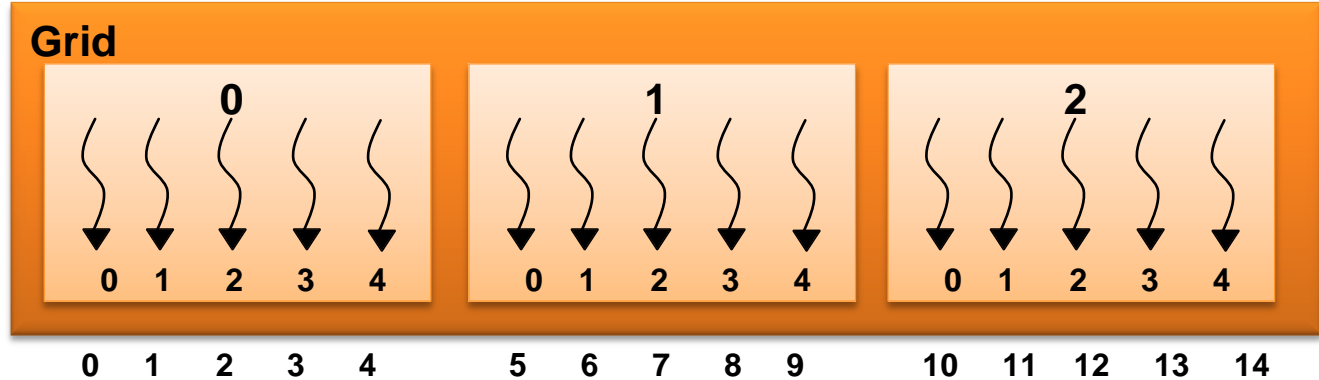
- Map local thread ID to a global array index

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

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

`blockIdx.x`

`threadIdx.x`



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

# CUDA Syntax – Error Management

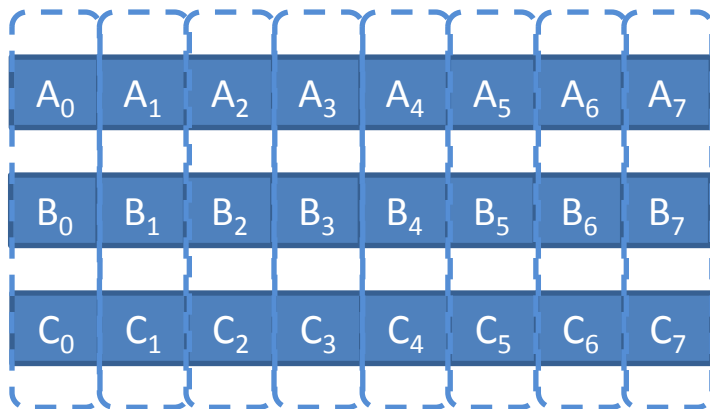
- Kernel launches have no return value!
- `cudaError_t cudaGetLastError()`
  - Returns error code for last CUDA runtime function (including kernel launches)
    - Resets global error state to `cudaSuccess`
  - In case of multiple errors, only the last one is reported
  - For kernels:
    - Asynchronous,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

```
MyKernel<<< ... >>> (...);
```

```
cudaDeviceSynchronize();  
e = cudaGetLastError();
```



# Putting It All Together



One Thread per Output

$$C_x = A_x + B_x$$

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 is evenly divisible by 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

- CUDA 6 introduced Unified Memory
- Use 'managed memory' instead of explicitly declaring memory for the host and device

```
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]);
```



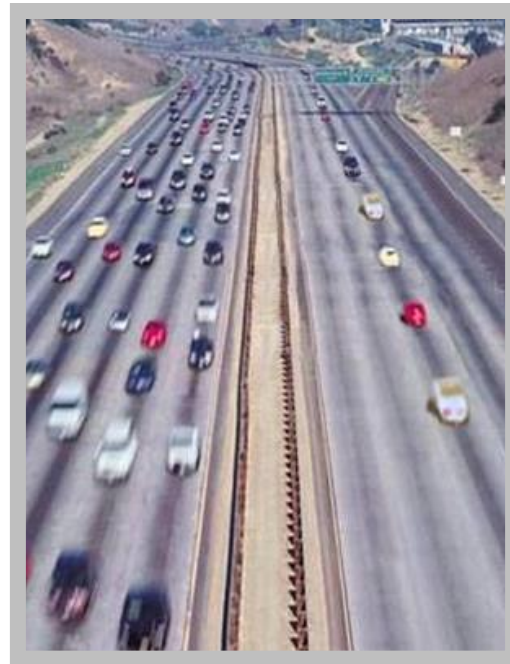
Access result from host

# 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



# Need More CUDA?

## Public Training:

- August 30 – September 2  
Calgary, Alberta

## Private Training:

- Onsite, anywhere in the world
- Tailored to your specific needs

## **NEW** Online Training:

- Access our courses from anywhere
- August 30 – September 2

## Mentoring & Services:

- Let us help develop & optimize your applications for maximum performance



Email:

[training@acceleware.com](mailto:training@acceleware.com)

# Questions?

Acceleware Ltd.

Tel: +1 403.249.9099

Email: [services@acceleware.com](mailto:services@acceleware.com)  
[training@acceleware.com](mailto:training@acceleware.com)

CUDA Blog: <http://acceleware.com/blog>

Website: <http://acceleware.com>



Chris Mason

[chris.mason@acceleware.com](mailto:chris.mason@acceleware.com)

