



# 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™

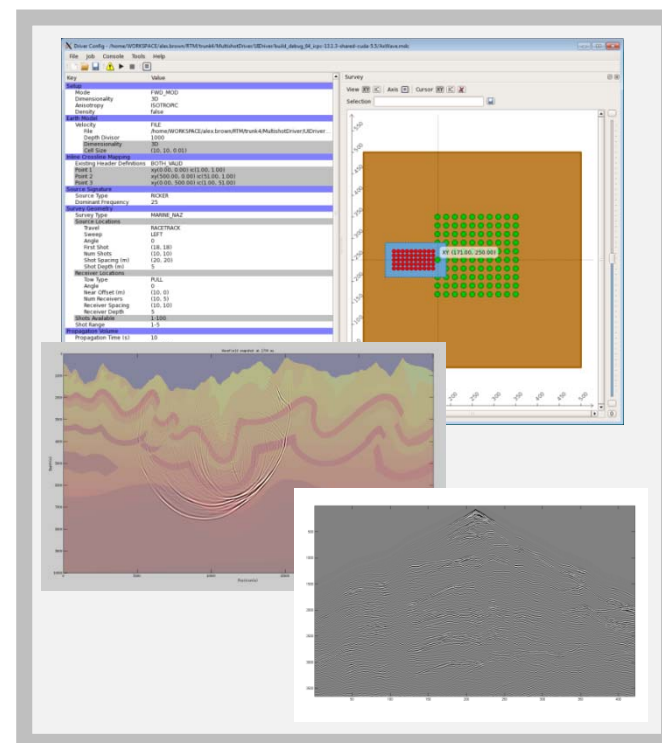
- High performance Reverse Time Migration application
- Isotropic, VTI and TTI media

## 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

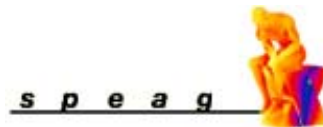


# Electromagnetics

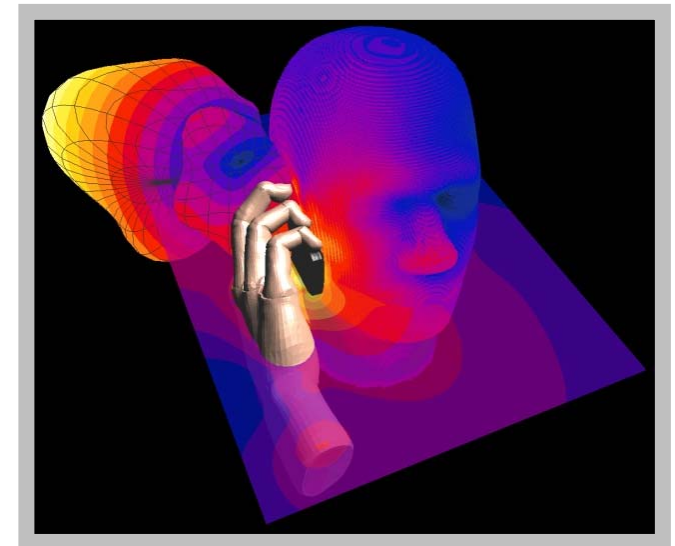
## AxFDTD™

- 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

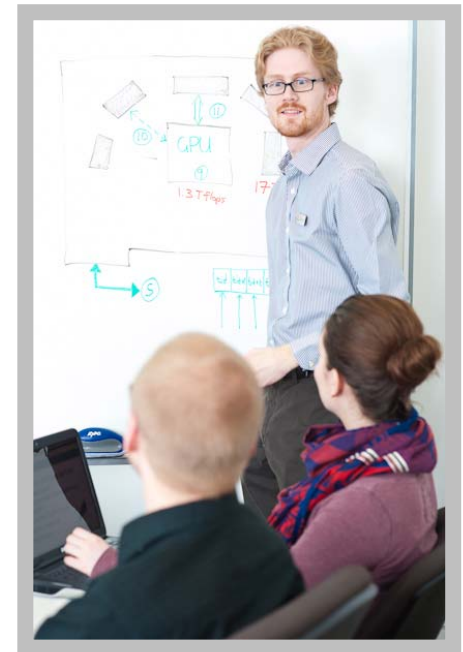
Industry	Application	Work Completed	Results
Finance	Option Pricing	Debugged & optimized existing CUDA code Implemented the Leisen-Reimer version of the binomial model for stock option pricing	30-50x performance improvement compared to single-threaded CPU code
Security & Defense	Detection System	Replaced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

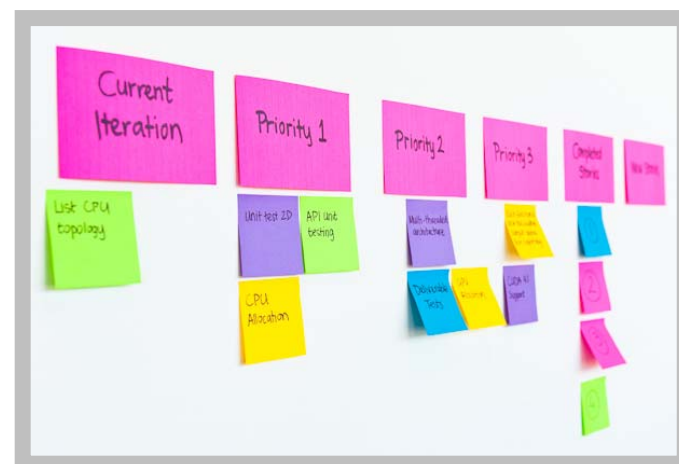
Jason Gauci, Software Engineer  
Lockheed Martin





# Outline

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



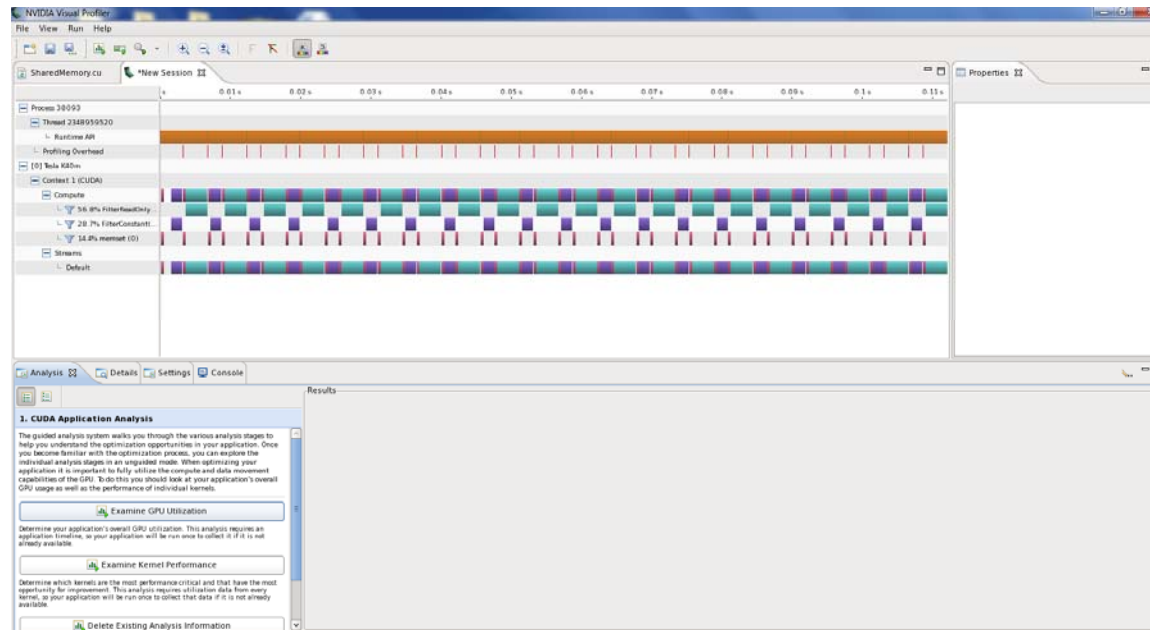
# Profiling Tools

- 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

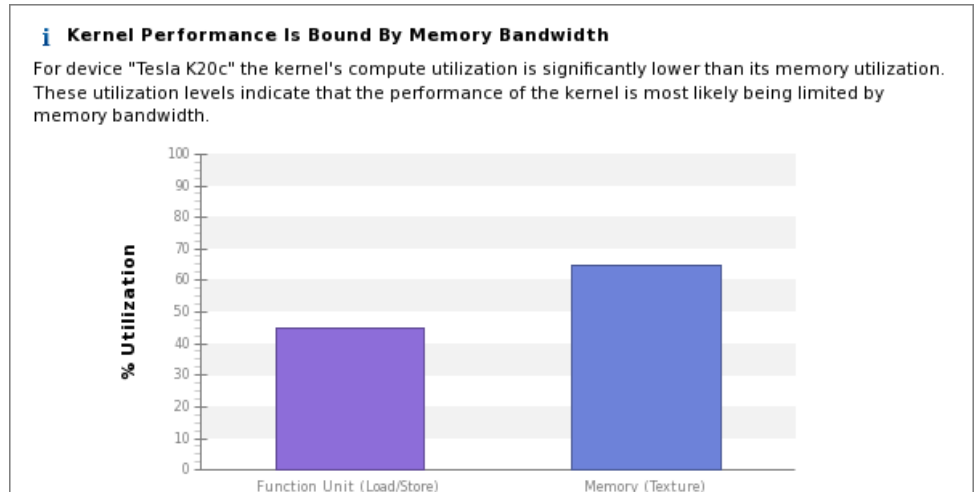
The screenshot shows the 'Analysis' tab in the NVIDIA Nsight Visual Studio Edition. The left sidebar has two sections: '1. CUDA Application Analysis' and '2. Performance-Critical Kernels'. Under '2. Performance-Critical Kernels', there is a text block explaining that the results on the right show kernels ordered by potential for performance improvement. Below this text is a button labeled 'Perform Kernel Analysis'. To the right of the sidebar is the 'Results' pane, which contains a section titled 'Kernel Optimization Priorities'. This section explains that the following kernels are ordered by optimization importance based on execution performance compared to lower ranked kernels. Below this explanation is a table with two columns: 'Rank' and 'Description'.

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

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.

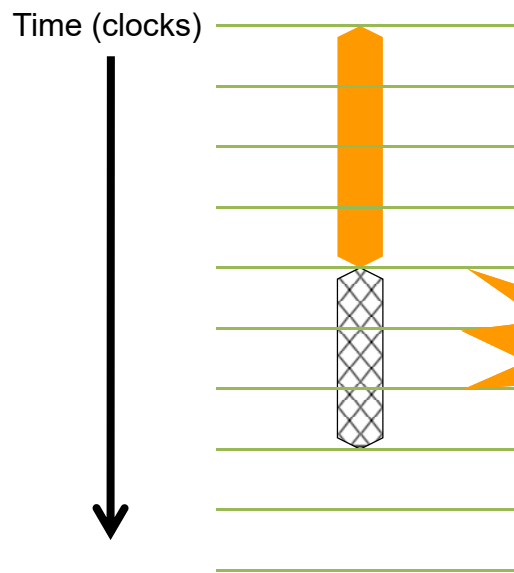
# Guided Performance Analysis

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



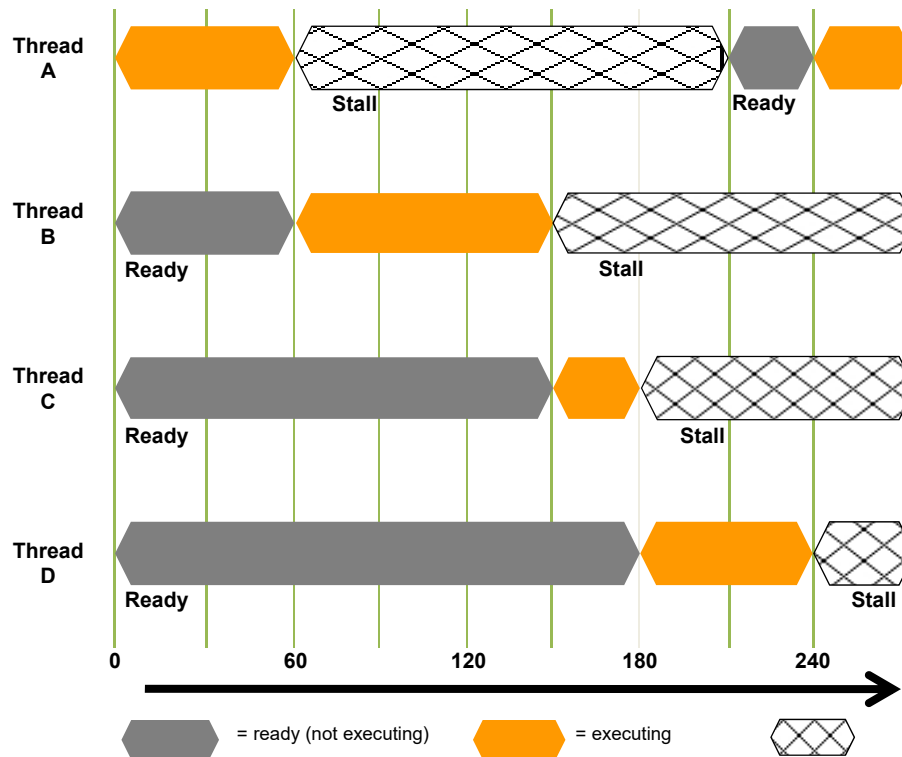
# Processor Stalls

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



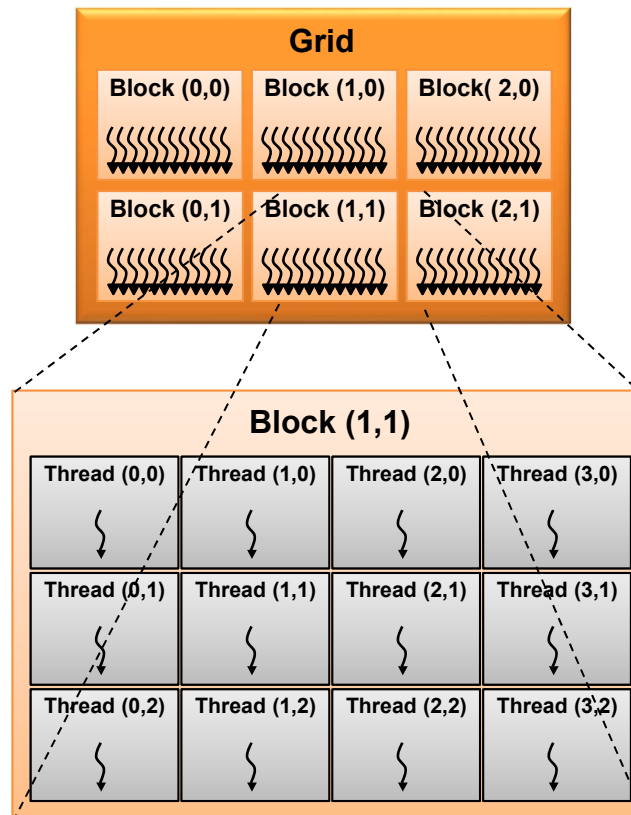
```
float sum(float* a, int const N)
{
    float sum = 0.0f;
    for(int i = 0; i < N; i++)
    {
        float temp = a[i];
        sum += temp;
    }
    return sum;
}
```

# Hiding Latency in GPUs



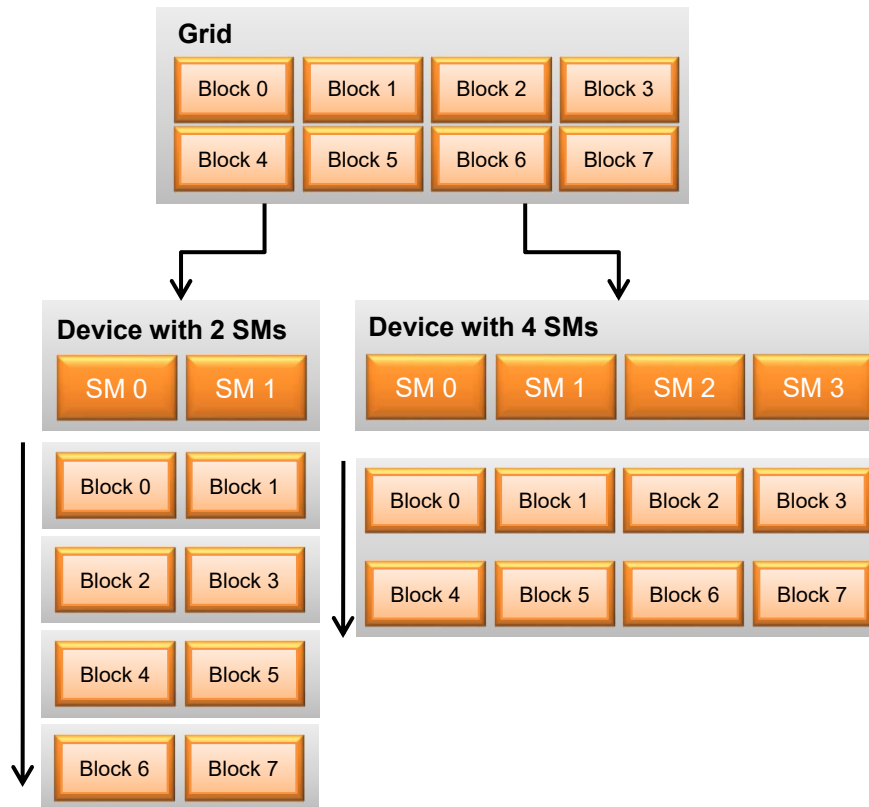
- GPUs minimize the impacts of stalls by interleaving execution of many threads on a single core
- Data-parallel model:
  - Can always execute threads from different blocks independently
  - Can execute threads within a block independently
    - Unless constrained by `__syncthreads()`

# 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!



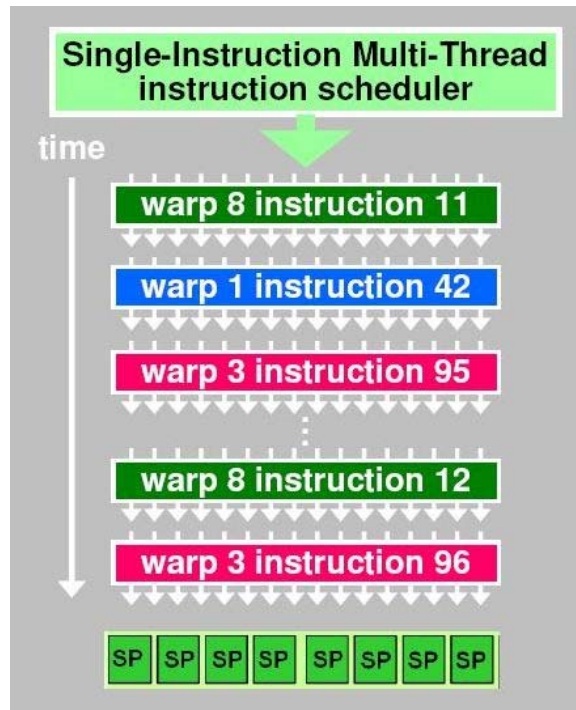
# What is a Warp?

- 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

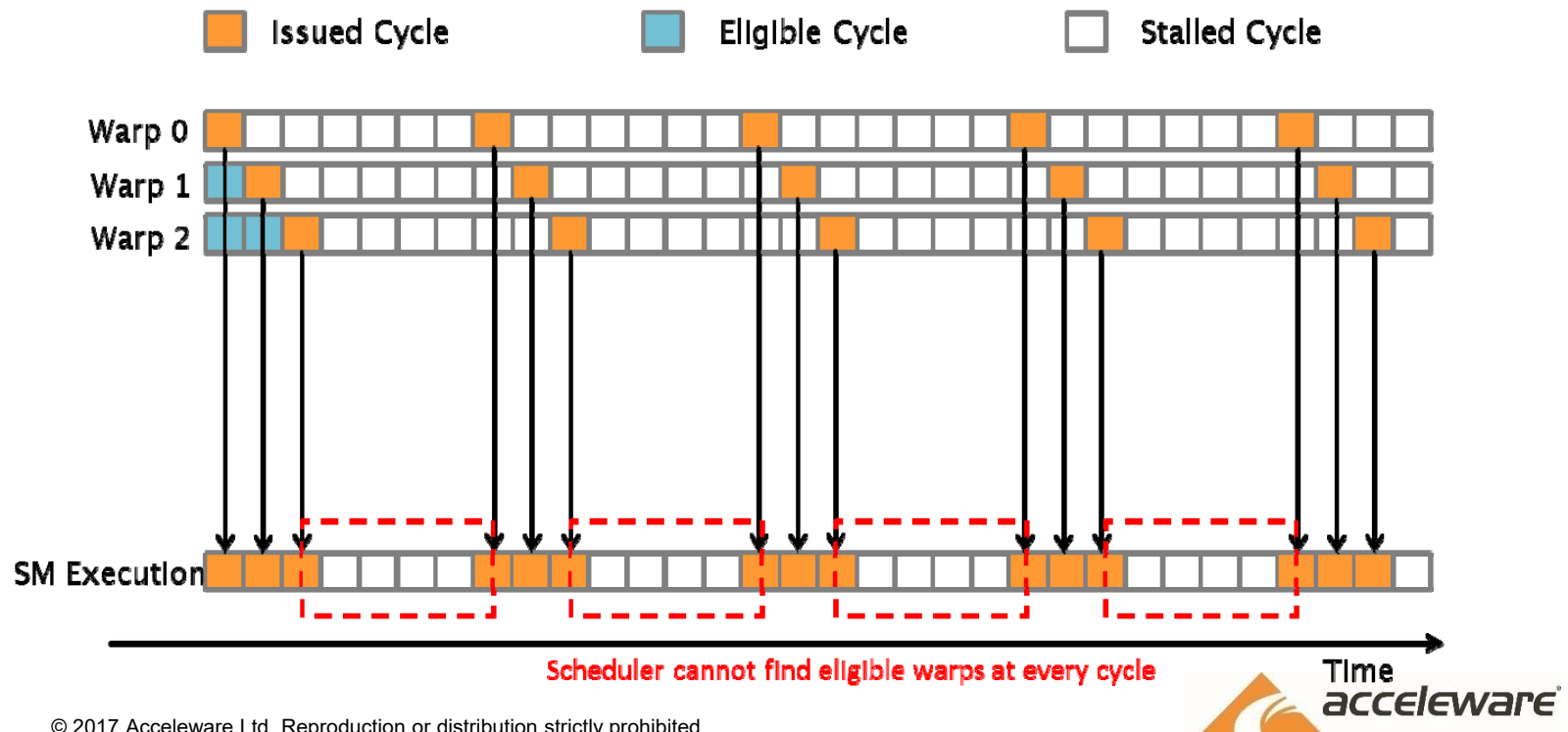


\*Image courtesy of NVIDIA

- 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!

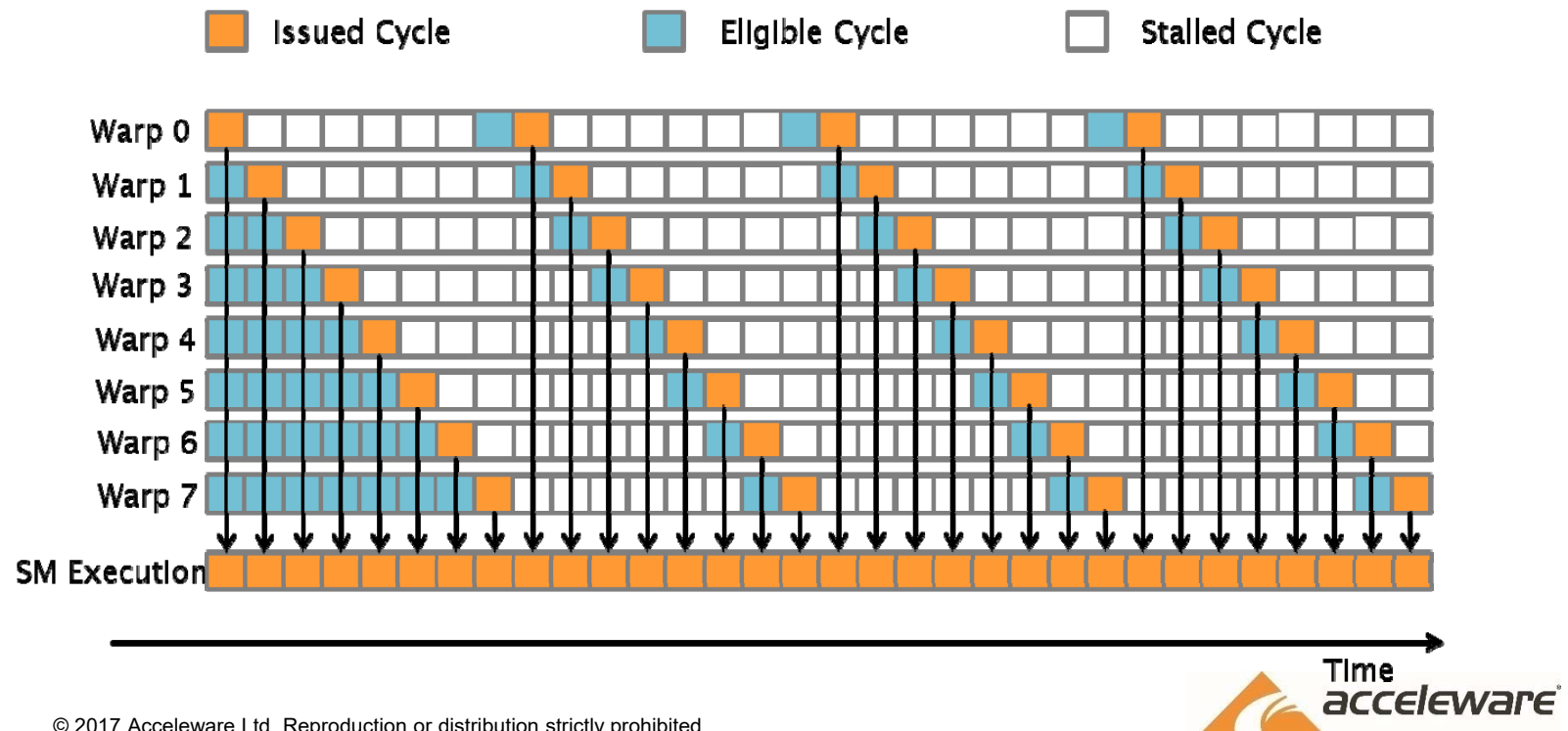
# Masking Latency

- 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



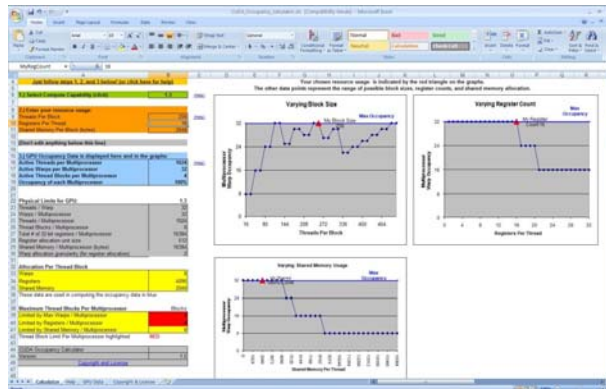
- Tesla P100 (Compute 6.0)
  - 256 threads/block
  - 18KB of shared memory/block
  - 32 registers/thread
- How many blocks can run at the same time on a streaming multiprocessor?



# 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



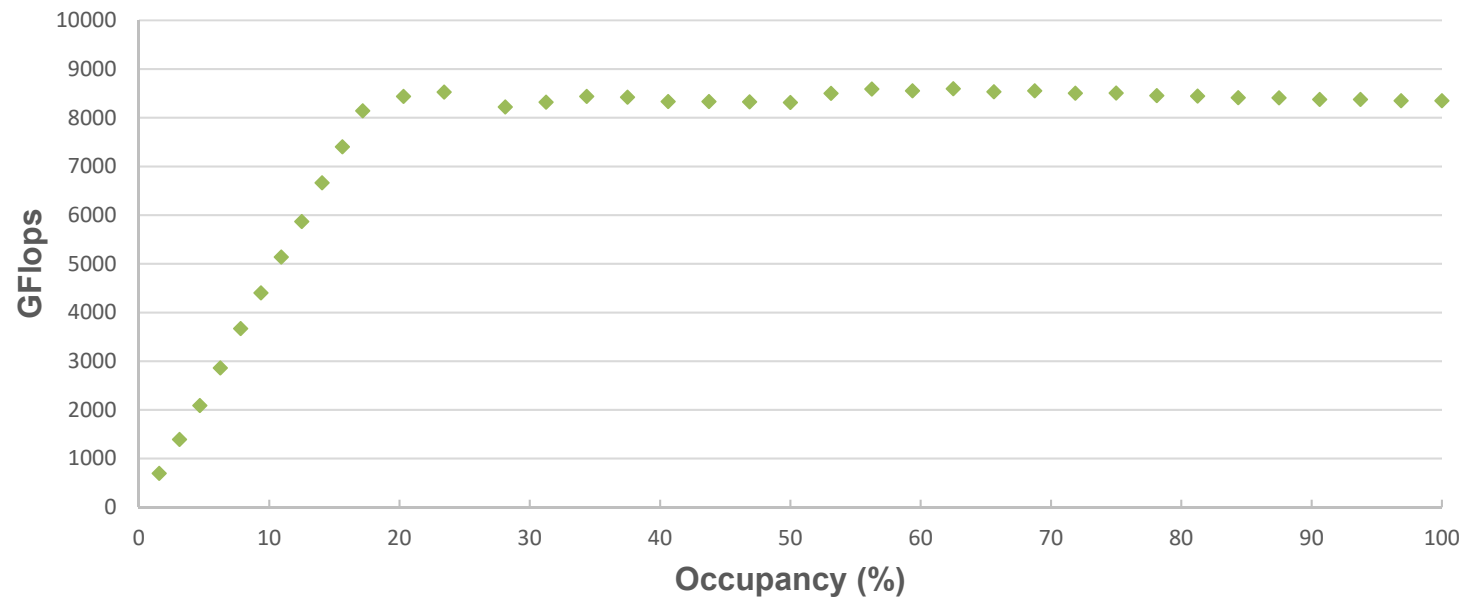
- We can run 3 blocks at a time
  - Limited by shared memory to 3 blocks
  - 3 blocks x 256 threads = 768 threads
  - $768 / 2048$  max threads = 37.5% occupancy
- CUDA Occupancy Calculator and Profilers do this calculation for you!

# Occupancy and Performance

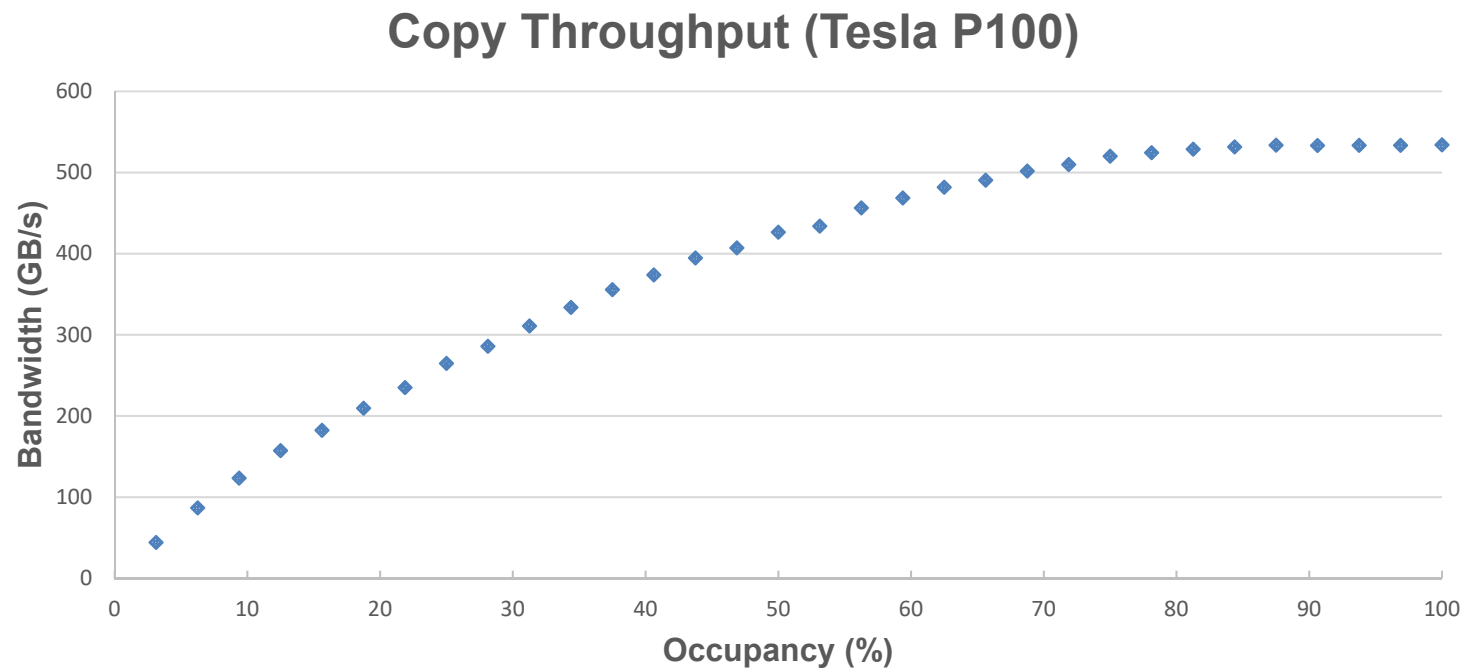
- Switching to other warps to mask latency is key to performance!
  - Need 'enough' other work to mask
    - May not need 100% occupancy!
    - Once you've reached the threshold, additional occupancy won't improve performance
  - 'Enough' occupancy depends on the code
    - Memory latency is typically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

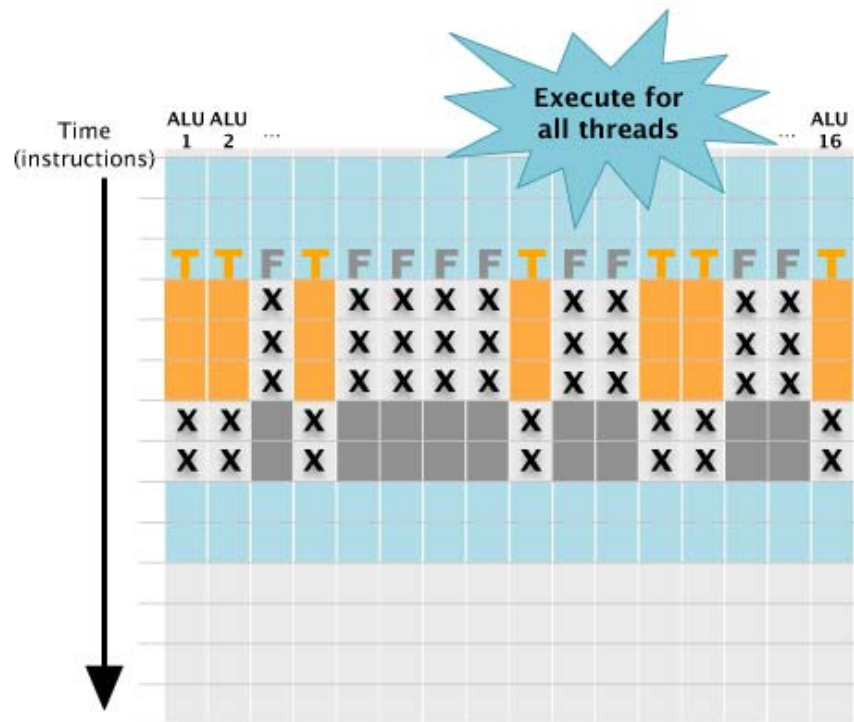
# Efficiency and Branching

- 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++) {...}  
    ...  
}
```

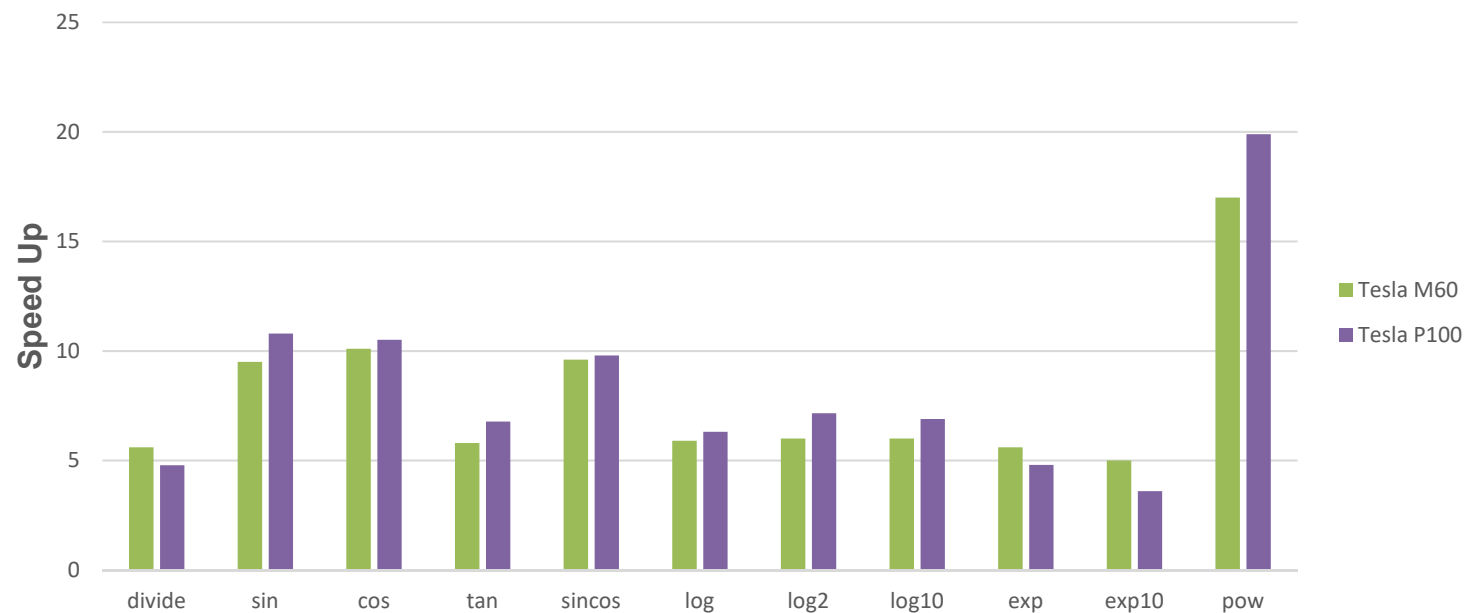
- Control unrolling with **#pragma unroll**
  - `#pragma unroll 1` prevents unrolling
  - `#pragma unroll N` unrolls the loop N times
  - `#pragma unroll` unrolls the loop completely

# '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+

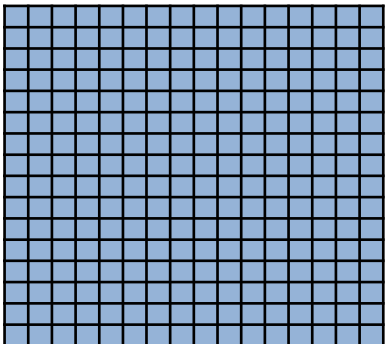
- Allows threads within a warp to share data
  - Use this functionality instead of shared memory
  - A single shuffle instruction can replace shared memory write/sync/read sequence
- Any thread can read any other thread's data within a warp
  - `__shfl()`
- Support for useful patterns
  - `__shfl_up()`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

# Memory Access Patterns

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

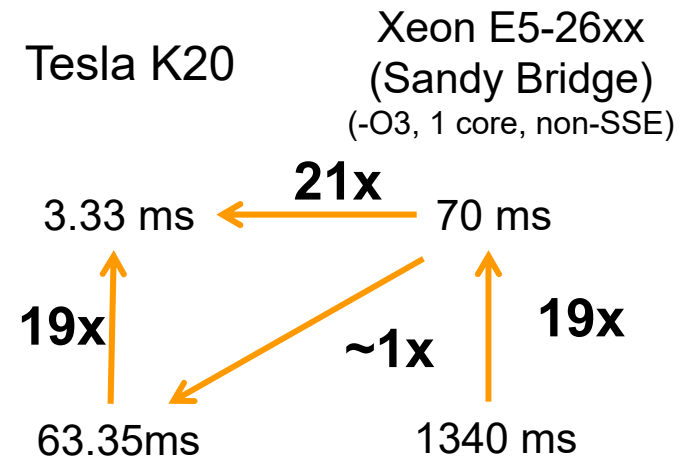
Columns = 8192

Rows = 8192

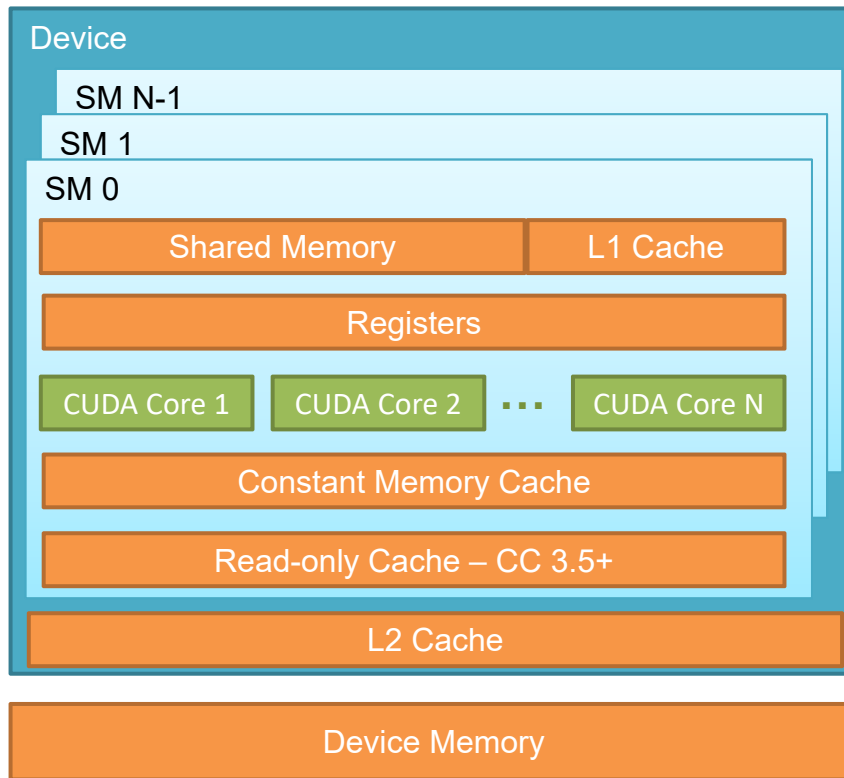


```
for(row)
  for(column)
    buf[x,y] ++;
```

```
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

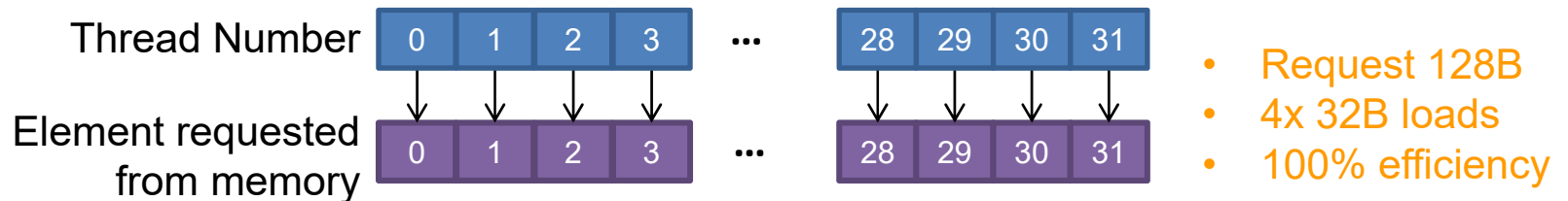


# Global Memory

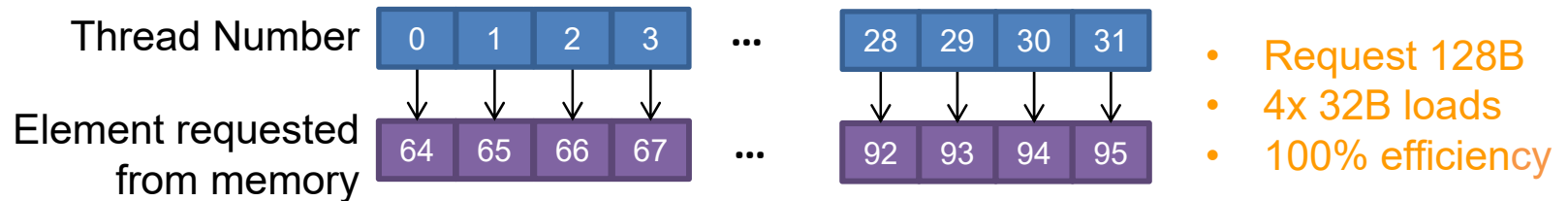
- Global memory is most efficient when threads in a warp access an aligned contiguous region of memory
  - Individual thread requests can be combined (or **coalesced**) into fewer and larger transactions
- Can be quite inefficient otherwise
  - Minimum transaction size across the bus
  - For Kepler GPUs, 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];
```



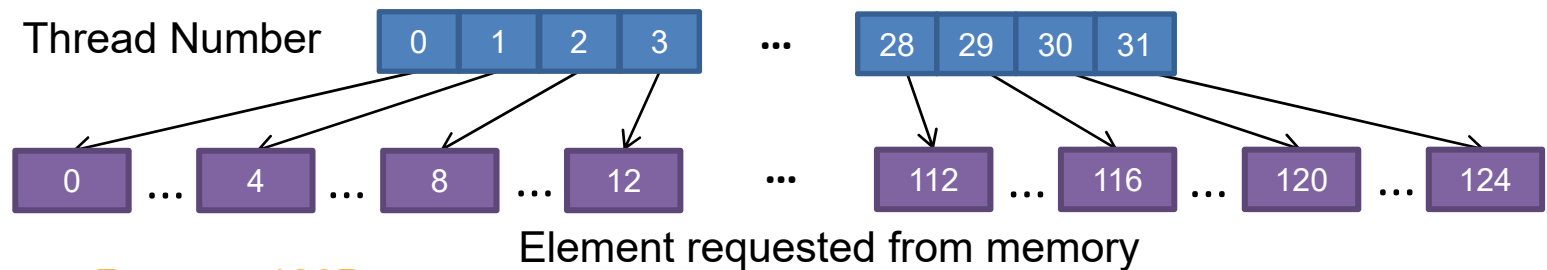
```
int idx = threadIdx.x + blockIdx.x * blockDim.x;  
float temp = input[idx+64];
```



## 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];
```

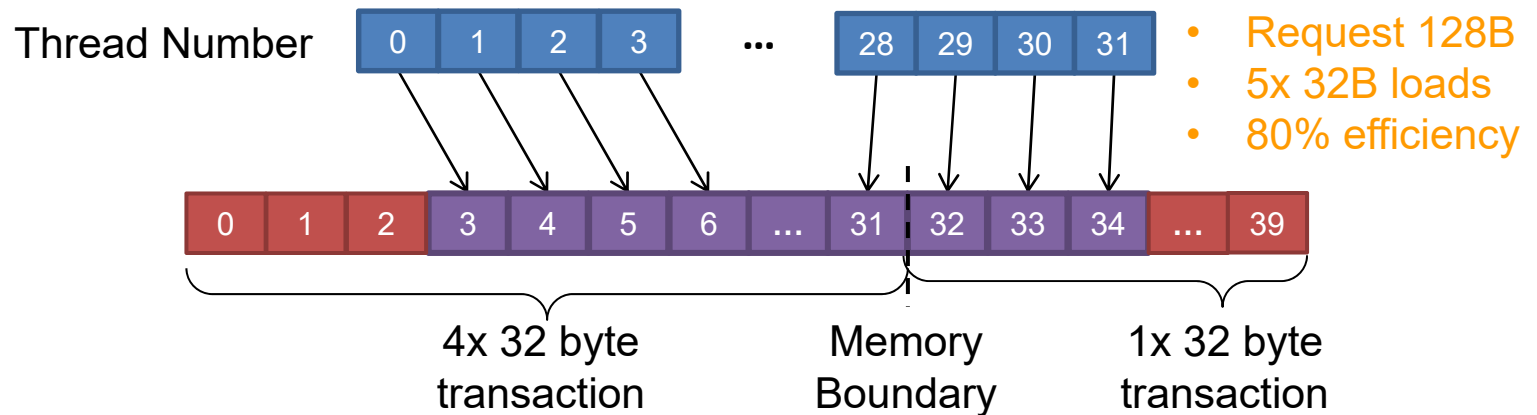


- Request 128B
- 16x 32B loads
- 25% efficiency

## Access Pattern Examples (3)

- Example: Contiguous but not aligned to memory boundary

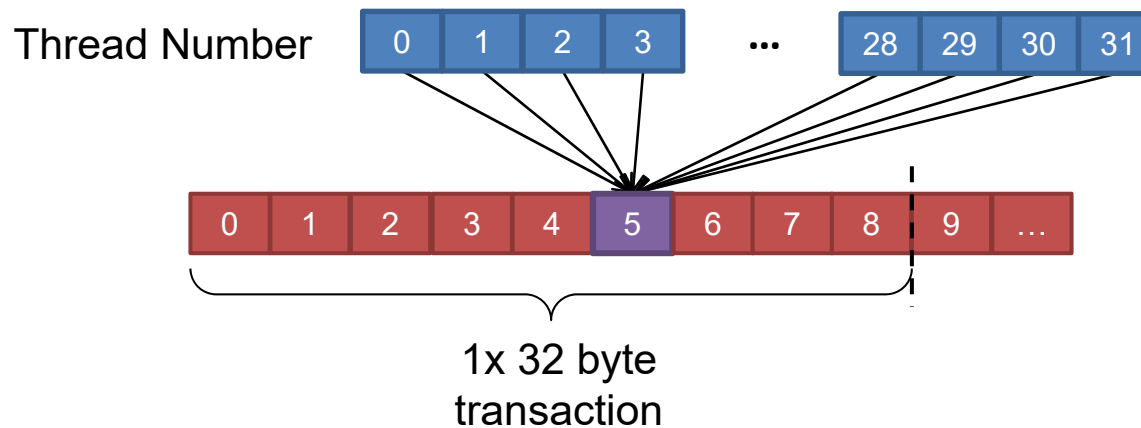
```
int idx = threadIdx.x + blockIdx.x * blockDim.x;  
float temp = input[idx + 3];
```



# 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

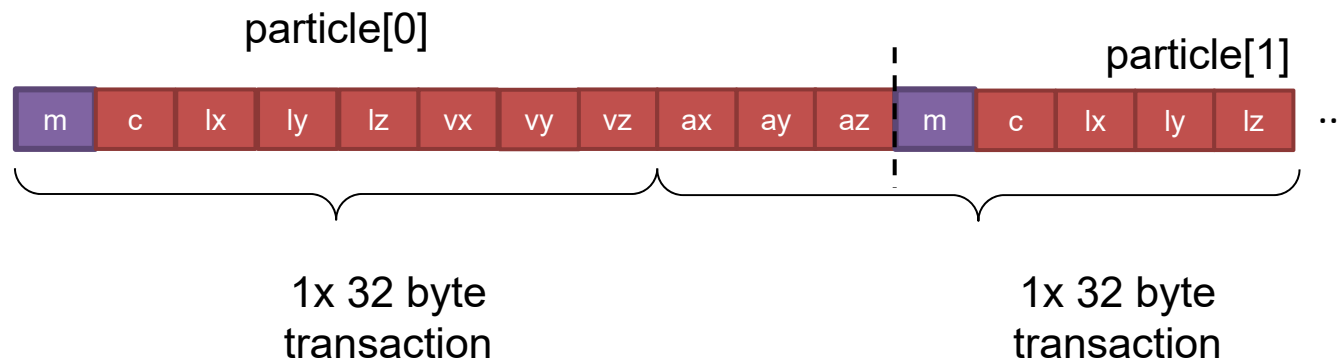
- Assume the following structure

```
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



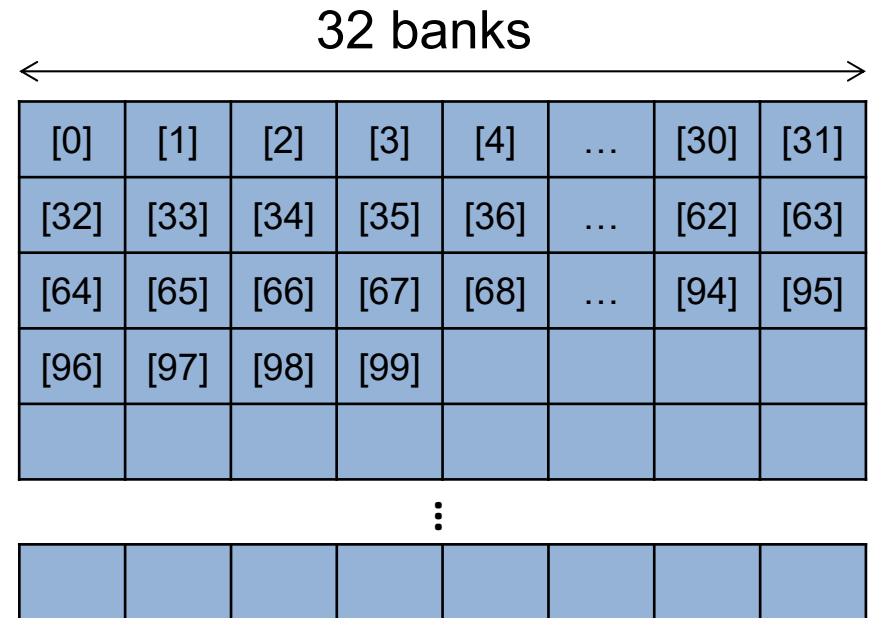
# Shared Memory Operation

- 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

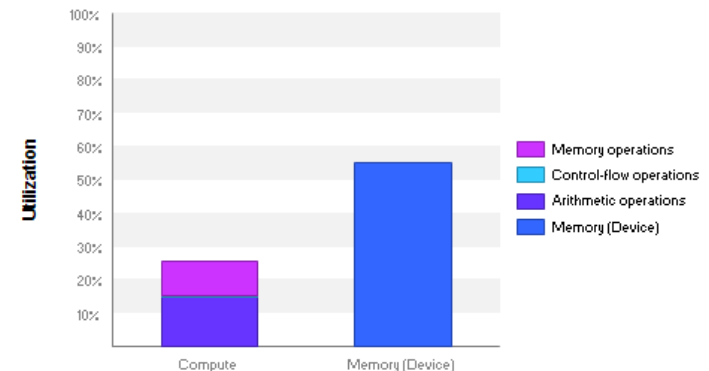
- Prefer constant memory when:
  - Data fits in 64KB
  - Values are broadcast to all threads in a warp
- Prefer read-only cache when:
  - Read-only data where constant memory isn't a good fit
  - For read-only strided accesses from global memory
    - eg. Access Pattern Example (2)

# 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

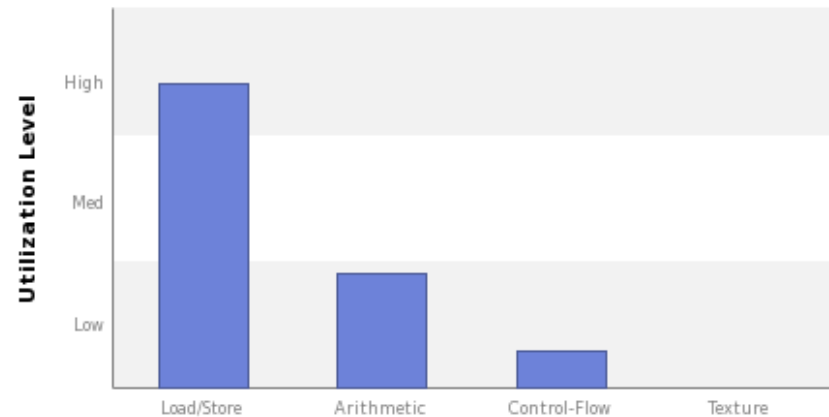
Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is potentially limited by overuse of the "Load/Store"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

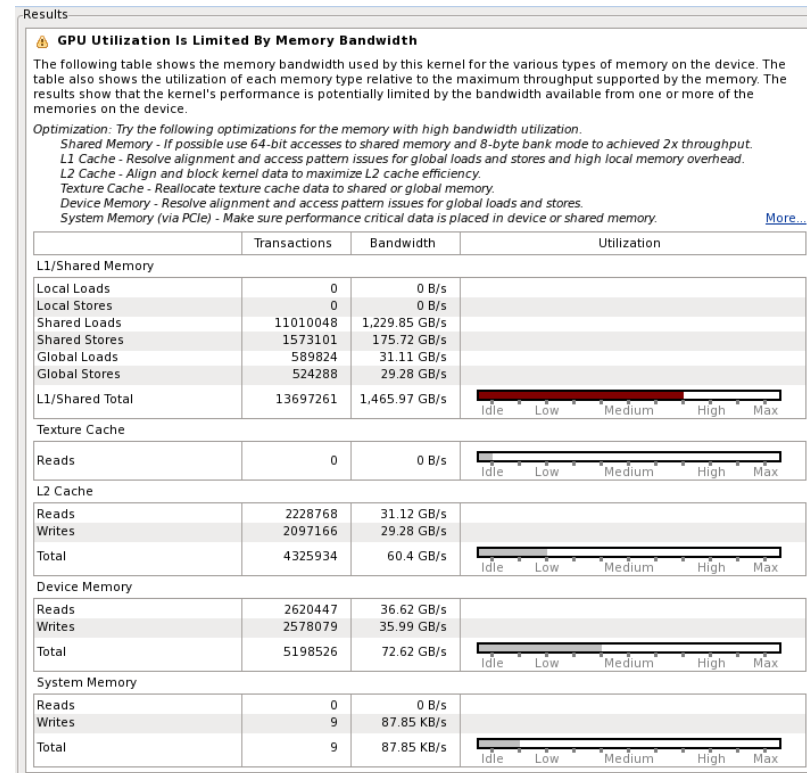
Texture - Texture operations.





# 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

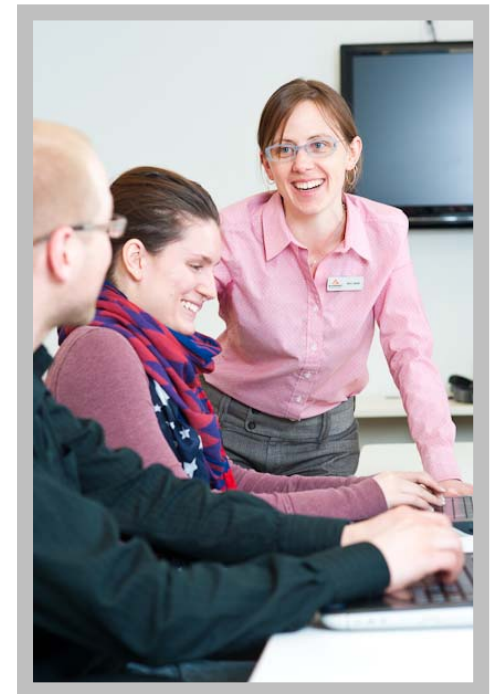
## Scheduled CUDA Courses (also available online)

- 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](mailto:services@acceleware.com)

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

Website: <http://acceleware.com>



Chris Mason

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