CUDA Profiling and Debugging

Shehzan
AccelerEyes
Summary

- AccelerEyes
- What is GPU Programming and CUDA?
- Overview of Basics
- Debugging and Profiling using CUDA Tools
- Memory Coalescing
- Shared Memory and Bank Conflicts
- Transpose
- Reduction
AccelerEyes

- GPU Software
- ArrayFire
  - Largest and fastest GPU Software library
  - v2.0, OpenCL production release coming soon
- Consulting and services
- Training

www.accelereyes.com
ArrayFire

● High-level GPU Library
● Features
  ○ Scalar and vector math
  ○ Matrix ops
  ○ Linear algebra & data analysis
  ○ Image and signal processing
  ○ Graphics
  ○ Integrate with your CUDA/OpenCL code
● Academic licenses

www.accelereyes.com
What do I do?

- Glasses.com Virtual Try On
  - [http://www.glasses.com/virtual-try-on](http://www.glasses.com/virtual-try-on)
- CUDA/OpenCL training
- ArrayFire development
CUDA

Overview
What is CUDA

- Compute Unified Device Architecture
- The Architecture
  - Expose GPU computing for general purpose
- CUDA C/C++
  - Extensions to the C/C++ language to enable heterogeneous programming
  - APIs to manage devices and memory
Overview of Basics

**kernel**: C/C++ function which executes on the device

**thread**: lightweight thread that runs on the device

**block**: collection of threads

**grid**: collection of all blocks launched
Parallel Execution

- Blocks are a group of threads
  - Each block has a unique ID which is accessed by the `blockIdx` variable
  - Threads in the same block share a very fast local memory called shared memory
  - Organized in a 1D, 2D, or 3D grid
  - You can have a maximum of 2048M x 64K x 64K grid of blocks
  - Each block executes on an SM in unspecified order
Grid: 1D/2D/3D Collection of Blocks

blockIdx.x
blockIdx.y
blockIdx.z

(3, 2, 0)
Block: 1D/2D/3D Collection of Threads

CUDA threads arranged in a 32 x 4 x 1 pattern inside each Block.
Basic Control Flow

1. Allocate memory on the device
2. Copy data from host memory to device memory
3. Launch: `kernel` <<<..>>>  
4. Retrieve results from GPU memory to CPU memory
Parallel Execution

Execution Path

The same CUDA program gets its thread blocks distributed automatically across any given SM architecture.
Memory Hierarchy

global, local, shared
Memory Hierarchy
Memory Hierarchy

Global Memory

- Created using cudaMalloc
- Available to all threads

```c
__global__ void add(int* a, int* b, int* c) {
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    c[id] = a[id] + b[id];
}
```
Memory Hierarchy

Local Memory

- Stored in registers (very fast)
- Thread local

```c
#include <cuda.h>

__global__ void add(int* a, int* b, int* c)
{
    int id = blockIdx.x*blockDim.x + threadIdx.x;
    c[id] = a[id] + b[id];
}
```
Shared Memory

- Located on the GPU's SM
- User managed
- Fast (like registers)
- Accessible by all threads in the block

```c
__global__ void add(int* a, int* b, int* c) {
    __shared__ int aValues[BLOCK_SIZE];
    __shared__ int bValues[BLOCK_SIZE];
    int id = threadIdx.x;
    int globalId = blockIdx.x * BLOCK_SIZE + threadIdx.x;

    aValues[id] = a[globalId];
    bValues[id] = b[globalId];
    c[id] = aValues[id] + bValues[id];
}
```
Debugging and Profiling

NVIDIA NSight Visual Studio Edition
Debugging

- Host side - Visual Studio Debugger, gdb
- Device side - Nsight, cuda-gdb (linux)
CUDA GDB

- CUDA-GDB
  - An extension of GDB
  - Allows you to debug on actual hardware
  - Can debug CPU and GPU code
- Compile using the -g and the -G flags
  - Includes debug info
  - `nvcc -g -G foo.cu -o foo`
- Usage of cuda-gdb is similar to gdb
Running CUDA-GDB (Linux)

- Debugging requires pausing the GPU
  - If the desktop manager is running on the GPU then it will become unusable.

- Single GPU debugging
  - Stop the desktop manager
    - On Linux: `sudo service gdm stop`
    - On Mac OS X you can log in with the `>console` user name

- Multi-GPU debugging
  - In Linux the driver excludes GPUs used by X11
  - On Mac OS X you can set which device to expose to cuda-gdb by setting the `CUDA_VISIBLE_DEVICES` environment variable
Debugging Coordinates

- **Software Coordinates**
  - Thread
  - Block
  - Kernel

- **Hardware Coordinates**
  - Lane (thread)
  - Warp
  - SM
  - Device
NVIDIA Nsight Visual Studio Edition
NVIDIA Nsight Visual Studio Edition

- Comprehensive debugging and profiling tool
- Contents:
  - GPU Debugger
  - Graphics Inspector
  - System Profiling

NVIDIA Nsight Visual Studio Documentation
NVIDIA Nsight Visual Studio Edition

- GPU Debugging
  - CUDA Debugger
  - CUDA Memcheck
  - CUDA Profiling
  - Trace
Enable Nsight Debugging

- Turn on Debug Info
  - Project->Properties->CUDA C/C++->
    - Generate GPU Debug Info
    - Generate Host Debug Info
- Best to run at highest compute available
Credits:
Vector Add, Matrix Multiply - CUDA Samples
Texture Cube - OpenGL Samples by Christophe Riccio (G-truc Creation)
NVIDIA Visual Profiler
NVIDIA Visual Profiler

- Standalone application with CUDA Toolkit
- Visualize performance
- Timeline
- Power, clock, thermal profiling
- Concurrent profiling
- NV Tools Extensions API
- nvprof - command line tool

http://docs.nvidia.com/cuda/profiler-users-guide/index.html#visual-profiler
NVIDIA Visual Profiler

Demo
Memory Coalescing

Super awesome speed up
Memory Coalescing

- Coalesce access to global memory
  - Most important performance consideration
  - Loads and stores by threads of a warp can be combined into as low as one instruction
- The concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of cache lines necessary to service all of the threads
Coalescence

- A warp requests 32 aligned, 4-byte words
- Address fall within 1 L1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

Memory addresses:

0 32 64 96 128 160 192 224 256 288 320 352 384 416 448

L1

addresses from a warp

L2

...
Coalescence

- A warp requests 32 aligned, 4-byte words
- Address fall within 1 L1 cache-line
  - Warp needs 128 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 100%

![Diagram showing memory addresses and cache lines]

Memory addresses:

- L1: 0, 32, 64, 96, 128, 160, 192, 224, 256, 288, 320, 352, 384, 416, 448
- L2: 0, 32, 64, 96, 128, 160, 192, 224, 256, 288, 320, 352, 384, 416, 448

Addresses from a warp...
Coalescence

- A warp requests 32 aligned, 4-byte words
- Address fall within 2 L1 cache-line
  - Warp needs 128 bytes
  - 256 bytes move across the bus on a miss
  - Bus utilization: 50%

addresses from a warp
L1
L2
Memory addresses
Coalescence (Non-cached)

- A warp requests 32 aligned, 4-byte words
- Address fall within 5 L2 cache-line
  - Warp needs 128 bytes
  - 160 bytes move across the bus on a miss
  - Bus utilization: 80%

Memory addresses:

```plaintext
0  32  64  96  128  160  192  224  256  288  320  352  384  416  448
```

L1 and L2 cache levels with memory addresses.
Coalescence

- A warp requests 1 4-byte word
- Address falls within 1 L1 cache-line
  - Warp needs 4 bytes
  - 128 bytes move across the bus on a miss
  - Bus utilization: 3.125%
Coalescence (Non-caching)

- A warp requests 1 4-byte words
- Address fall within 1 L1 cache-line
  - Warp needs 4 bytes
  - 32 bytes move across the bus on a miss
  - Bus utilization: 12.5%

![Memory addresses diagram]

```
0  32  64  96  128  160  192  224  256  288  320  352  384  416  448
```

addresses from a warp
Coalescence

- A warp requests 32 scattered 4-byte words
- Address fall within N L1 cache-line
  - Warp needs 128 bytes
  - N * 128 bytes move across the bus on a miss
  - Bus utilization: 128 / (N * 128)

addresses from a warp

L1

L2

Memory addresses
Coalescence (Non-caching)

- A warp requests 32 scattered 4-byte words
- Address fall within N L1 cache-line
  - Warp needs 128 bytes
  - N * 32 bytes move across the bus on a miss
  - Bus utilization: \[ \frac{128}{N \times 32} \] addresses from a warp
Shared Memory
Shared Memory

● Acts as a user-controlled cache
● Declared using the __shared__ qualifier
● Accessible from all threads in the block
● Lifetime of the block
● Allocate statically or at kernel launch.

__shared__ float myVariable[32]; // static

// ... or specify at launch:
extern __shared__ float myVar[];

// ...
myKernel<<<blocks, threads, shared_bytes>>>(parameters);
Shared Memory

- Inter-thread communication within a block
- Cache data to reduce redundant global memory access
- Improve global memory access patterns
- Divided into **32 32-bit banks**
  - Can be accessed simultaneously
  - Requests to the same bank are serialized
Matrix Transpose

Get 90% Bandwidth
Matrix Transpose

- Inherently parallel
  - Each element independent of another
- Simple to implement

1  2  3  4
5  6  7  8
9 10 11 12
13 14 15 16

1  5  9 13
2  6 10 14
3  7 11 15
4  8 12 16
Matrix Transpose [CPU Transpose]

```c
for(int i = 0; i < rows; i++)
    for(int j = 0; j < cols; j++)
        transpose[i][j] = matrix[j][i]
```

- Easy
- $O(n^2)$
- Slow!!!!!!
Matrix Transpose
[Naive GPU Transpose]

- GPU Transpose
  - Launch 1 thread per element
  - Compute index
  - Compute transposed index
  - Copy data to transpose matrix
- O(1) using Parallel compute
- Essentially one memcpy from global-to-global
  - It should be fast, shouldn’t it?
Matrix Transpose
[Naive GPU Transpose]

```c
__global__ void matrixTranspose(float *_a, float *_b) {
    int i = blockIdx.y * blockDim.y + threadIdx.y; // row
    int j = blockIdx.x * blockDim.x + threadIdx.x; // col

    int index_in = i*cols+j; // (i,j) from matrix A
    int index_out = j*rows+i; // transposed index

    b[index_out] = a[index_in];
}
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>-2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>4</td>
<td>-1</td>
<td>3</td>
</tr>
</tbody>
</table>

_b[......]  _a[......]
Matrix Transpose

[Naive GPU Transpose]

- Problems?
Matrix Transpose
[Naive GPU Transpose]

● Problems?
  ○ Non-coalesced memory

● Improvements?
GMEM Access Pattern in NT

READ - Coalesced memory access  

WRITE - Uncoalesced memory access

Good!

Bad!
Matrix Transpose

[Naive GPU Transpose]

- Problems?
  - Non-coalesced memory

- Improvements?
  - Use shared memory
  - Use coalesced memory access
Matrix Transpose  [GPU Transpose]

- Use Shared Memory
  - Allows temporary storage of data
  - Use coalesced memory access to global memory

- Walkthrough
  - Compute input index (same as in naive transpose)
  - Copy data to shared memory
  - Compute output index
    - Remember, coalesced memory access
    - Hint, transpose only in shared memory
  - Copy data from shared memory to output
__syncthreads();
re-indexing
Shared Memory Transpose

Copy from SMEM to GMEM in order of BIT

Copy from GMEM to SMEM (row-by-row)
__global__ void matrixTransposeShared(const float *a, float *b)
{
  __shared__ float mat[BLOCK_SIZE_X][BLOCK_SIZE_Y];
  int bx = blockIdx.x * BLOCK_SIZE_X;
  int by = blockIdx.y * BLOCK_SIZE_Y;
  int i = by + threadIdx.y;   int j = bx + threadIdx.x;  //input
  int ti = bx + threadIdx.y;   int tj = by + threadIdx.x;  //output
  if(i < rows && j < cols)
    mat[threadIdx.x][threadIdx.y] = a[i * cols + j];
  __syncthreads();       //Wait for all data to be copied
  if(tj < cols && ti < rows)
    b[ti * rows + tj] = mat[threadIdx.y][threadIdx.x];
}
Matrix Transpose

- Problem?
Matrix Transpose  [GPU Transpose]

● Problem?
  ○ Why are we not even close to max bandwidth?
  ○ Hint, think “banks”

● Solution?
Matrix Transpose [GPU Transpose]

- **Problem?**
  - Why are we not even close to max bandwidth?
  - Hint, think “banks”

- **Solution?**
  - Remove bank conflicts
Bank Conflicts
Banks

- Shared Memory is organized into 32 banks
- Consecutive shared memory locations fall on different banks

```c
__shared__ float tile[64];
```
Banks

- Access to different banks by a **warp** executes in parallel.

```c
__shared__ float tile[64];
int tidx = threadidx.x;
float foo = tile[tidx] - 3;
```
Banks

- Access to the same element in a bank is also executed in parallel.

```c
__shared__ float tile[64];
int tidx = threadIdx.x;
int bar = tile[tidx - tidx % 2];
```
Banks

- Access to the different elements in a bank is executed serially.
- “2 way bank conflict”

```c
__shared__ float tile[64];
int tidx = threadidx.x;
tmp = tile[tidx + tidx % 2*31];
```
- Access to the different elements in a bank is also executed serially.
- 32 way bank conflict

```c
_b[index_out] = tile[tx][ty];
```
__global__ void matrixTransposeShared(const float *a, float *b)
{
    __shared__ float mat[BLOCK_SIZE_X][BLOCK_SIZE_Y];
    int bx = blockIdx.x * BLOCK_SIZE_X;
    int by = blockIdx.y * BLOCK_SIZE_Y;
    int i  = by + threadIdx.y;
    int j  = bx + threadIdx.x; //input
    int ti = bx + threadIdx.y;
    int tj = by + threadIdx.x; //output

    if(i < rows && j < cols)
        mat[threadIdx.x][threadIdx.y] = a[i * cols + j];
    __syncthreads();       //Wait for all data to be copied
    if(tj < cols && ti < rows)
        b[ti * rows + tj] = mat[threadIdx.y][threadIdx.x];
}
Transpose

- No Bank conflicts
Transpose

- 32-way Bank conflict!!

Add Unused column

Thread 0

Thread 31
Banks

- Resolving bank conflict

```cpp
__shared__ float tile[BLOCKSIZE][BLOCKSIZE+1];
_b[index_out] = tile[tx][ty];
```
__global__ void matrixTransposeSharedwBC(const float *a, float *b)
{
    __shared__ float mat[BLOCK_SIZE_X][BLOCK_SIZE_Y + 1];
    //Rest is same as shared memory version
}
Matrix Transpose [GPU Transpose]

- Very very close to production ready!
- More ways to improve?
  - More work per thread - Do more than one element
  - Loop unrolling
Transpose: Loop Unrolled

- More work per thread:
  - Threads should be kept light
  - But they should also be saturated
  - Give them more operations

- Loop unrolling
  - Allocate operation in a way that loops can be unrolled by the compiler for faster execution
  - Warp scheduling
    - Kernels can execute 2 instructions simultaneously as long as they are independent
Transpose: Loop Unrolled

- Use same number of blocks, shared memory
- Reduce threads per block by factor (side)

Block Size X = 4
Block Size Y = 4
Threads/Block = 16
Total blocks = 2
Shared mem = 4 x 4

Block Size X = 4 -> TILE
Block Size Y = 1 -> SIDE
Threads/Block = 4
Total blocks = 2
Shared mem = TILE x TILE
Transpose: Loop Unrolled

- **Walkthrough**
- **Host:**
  - Same number of blocks
  - Compute new threads per block
- **Device:**
  - Allocate same shared memory
  - Compute input indices similar to before
  - Copy data to shared memory using loop (k)
    - Unrolled index: add k to y
  - Compute output indices similar to before
  - Copy data from shared memory into global memory
    - Unrolled index: add k to y
const int TILE = 32; const int SIDE = 8;

__global__ void matrixTransposeUnrolled(const float *a, float *b)
{
  __shared__ float mat[TILE][TILE + 1];
  int x = blockIdx.x * TILE + threadIdx.x;
  int y = blockIdx.y * TILE + threadIdx.y;

#pragma unroll
  for(int k = 0; k < TILE ; k += SIDE) {
    if(x < rows && y + k < cols)
      mat[threadIdx.y + k][threadIdx.x] = a[((y + k) * rows) + x];
  }

__syncthreads();
  //continuing on next slide
}
const int TILE = 32; const int SIDE = 8;

__global__ void matrixTransposeUnrolled(const float *a, float *b)
{
    //continuing from previous slide
    __syncthreads();
    x = blockIdx.y * TILE + threadIdx.x;
    y = blockIdx.x * TILE + threadIdx.y;
    #pragma unroll
    for(int k = 0; k < TILE; k += SIDE)
    {
        if(x < cols && y + k < rows)
        {
            b[(y + k) * cols + x] = mat[threadIdx.x][threadIdx.y + k];
        }
    }
}
## Performance for 4k x 4k Matrix Transpose (K20)

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (ms)</th>
<th>Bandwidth (GB/s)</th>
<th>Step Speedup</th>
<th>Speed Up vs CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>166.2</td>
<td>0.807</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naive Transpose</td>
<td>2.456</td>
<td>54.64</td>
<td>67.67</td>
<td>67.67</td>
</tr>
<tr>
<td>Coalesced Memory</td>
<td>1.712</td>
<td>78.37</td>
<td>1.434</td>
<td>97.08</td>
</tr>
<tr>
<td>Bank Conflicts</td>
<td>1.273</td>
<td>105.38</td>
<td>1.344</td>
<td>130.56</td>
</tr>
<tr>
<td>Loop Unrolling</td>
<td>0.870</td>
<td>154.21</td>
<td>1.463</td>
<td>191.03</td>
</tr>
</tbody>
</table>

Device to Device Memcpy: 167.10 GB/s
Transpose

Let’s review your predictions!
Reduction
Reduce

Algorithm to apply a reduction operation on a set of elements to get a result.

Example:
SUM(10, 13, 9, 14) = 10+13+9+14 = 46
MAX(10, 13, 9, 14) = 14
Serial Reduce

- Loop through all elements
- Number of steps: $N - 1$

Serial Code:

```c
int sum = array[0];
for(i=1;i<n;i++) {
    sum += array[i];
}
```

Number of Steps: $4 - 1 = 3$
Parallel reduce

Operations can be applied for Parallel reduce

- **Binary**
  - example: $a*b$, $a+b$, $a\&b$, $a|b$
  - not binary: $!(a)$, $(a)!$

- **Associative**
  - example: $a*b$, $a+b$, $a\&b$, $a|b$
  - non associative: $a/b$, $a^b$

Example:
Reduce[$(10,13,9,14) +$]

Number of steps: $\log_2 4 = 2$
Parallel Reduce on GPU

- Parallel reduce is applied to a part of the whole array in each block.
- Multiple blocks help in:
  - Maximising Occupancy by keeping SMs busy.
  - Processing very large arrays.

- Parallel reduce is not arithmetic intensive, it takes only 1 Flop per thread (1 add) so it is completely memory bandwidth bounded.
Parallel Reduce on GPU

Need a way to communicate partial results between blocks

- Global sync is not practical due to the overhead of sync across so many cores

Solution: Call the reduce kernel recursively to reduce the results from previous reduce.
Serial reduce vs Parallel reduce

Serial Reduce:
- Each iteration is dependant on the previous iteration.
- Number of steps taken is $n-1$.
- Runtime complexity is $O(n)$

Parallel Reduce:
- Has smaller number steps $\log_2 n$.
- Faster than a serial implementation.
- Runtime complexity : $O(\log n)$
Method 1: Interleaved Addressing

Step 1

Step 2

Step 3

Step 4

656
Method 1: Interleaved Addressing

```c
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // Each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + tid;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // Do reduction in shared mem
    for (unsigned s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```
# Performance for 32M elements (GTX 770)

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
<th>Bandwidth (GB/s)</th>
<th>Step Speedup</th>
<th>Speed Up vs CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>8.8</td>
<td>15.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 0</td>
<td>7.90</td>
<td>16.98</td>
<td>1.11</td>
<td>1.11</td>
</tr>
</tbody>
</table>
Method 1: Interleaved Addressing

- Problem?
Method 1: Interleaved Addressing

- Problem?
  - Interleaved addressing
  - Divergent warps

- Solution?
Method 1: Interleaved Addressing

● Problem?
  ○ Interleaved addressing
  ○ Divergent warps

● Solution?
  ○ Non-divergent branches
Warps

Divergence
Warp

A thread block is broken down to 32-thread warps. Warps are executed physically in a SM(X).

Total number of warps in a block: ceil(T/W_{size})

Up to 1024 Threads

32 threads each

Up to 64 Warps/SM
Warp

Each thread in a warp execute one common instruction at a time

- Warps with diverging threads execute each branch serially

```c
if (threadIdx.x < 4)
{
    x++;
}
else
{
    x--;
}
```
Warp

Each thread in a warp execute one **common** instruction at a time

- Warps with diverging threads execute each branch serially

```c
if (threadIdx.x < 4)
{
    x++;  // Yellow
}
else
{
    x--;  // Red
}
```
Warp

Each thread in a warp execute one **common** instruction at a time

- Warps with diverging threads execute each branch serially

```plaintext
if (threadIdx.x < 4)
{
    x++;  
}
else
{
    x--; 
}
```
Each thread in a warp execute one common instruction at a time

- Warps with diverging threads execute each branch serially

```c
if (threadIdx.x < WARP_SIZE )
{
    x++; 
}
else 
{
    x--; 
}
```
Each thread in a warp execute one **common** instruction at a time

- Warps with diverging threads execute each branch serially

```c
if (threadIdx.x < WARP_SIZE )
{
    x++; 
}
else
{
    x--; 
}
```
Warps - Take aways

● Try to make threads per blocks to be a multiple of a warp (32)
  ○ incomplete warps disable unused cores (waste)
  ○ 128-256 threads per blocks is a good starting point

● Try to have all threads in warp execute in lock step
  ○ divergent warps will use time to compute all paths as if they were in serial order
...back to reductions
Method 2: Interleaved addressing with non divergent branch

Problem: Thread Divergence

if (tid % (2*s) == 0)
    sdata[tid] += sdata[tid+s];

Solution: Replace with non divergent branch

- uses half the number of threads as before

int index = 2 * s * tid;
if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
}
__syncthreads();

thread 3 and 53 are idle
### Performance for 32M elements (GTX 770)

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
<th>Bandwidth (GB/s)</th>
<th>Step Speedup</th>
<th>Speed Up vs CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>8.8</td>
<td>15.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 0</td>
<td>7.90</td>
<td>16.98</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>Stage 1</td>
<td>6.26</td>
<td>21.45</td>
<td>1.26</td>
<td>1.41</td>
</tr>
</tbody>
</table>
Problem:
Bank conflict
  ○ Each thread accesses adjacent memory locations resulting in shared memory bank conflict.

Method 2

Solution:
○ Resolve bank conflicts.

Diagram:
- Threads: t0, t1, t2, t3, t4, t5
- Memory bank locations: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
- Bank conflicts indicated by circles at memory locations.
Method 3: Removing Bank Conflicts

Step 1

Final Step 4
Method 3:
Sequential reduction

Replace the interleaved addressing in for loop of Method 2

```c
for(unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

With reversed loop thread id based indexing in Method 3

```c
for(unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```
# Performance for 32M elements (GTX 770)

<table>
<thead>
<tr>
<th>Stage</th>
<th>Time (ms)</th>
<th>Bandwidth (GB/s)</th>
<th>Step Speedup</th>
<th>Speed Up vs CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>8.8</td>
<td>15.25</td>
<td></td>
<td>1.11</td>
</tr>
<tr>
<td>Stage 0</td>
<td>7.90</td>
<td>16.98</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>Stage 1</td>
<td>6.26</td>
<td>21.45</td>
<td>1.26</td>
<td>1.41</td>
</tr>
<tr>
<td>Stage 2</td>
<td>4.70</td>
<td>28.54</td>
<td>1.33</td>
<td>1.87</td>
</tr>
</tbody>
</table>
Method 3: Sequential Reduction

- **Problem:**
  - In the first iteration itself half of the threads in each block are wasted, only half of them perform the reduction

- **Solution:**
  - Reduce the number of threads to half in each block.
  - Make each thread read 2 elements to the shared memory.
  - Perform first reduction during first read.
Until now each thread loaded one element to the shared memory.

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

Halve the number of threads. Make each thread read in two values from global memory, perform reduction and write result to shared memory

```
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```
# Performance for 32M elements (GTX 770)

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
<th>Bandwidth (GB/s)</th>
<th>Step Speedup</th>
<th>Speed Up vs CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>8.8</td>
<td>15.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 0</td>
<td>7.90</td>
<td>16.98</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>Stage 1</td>
<td>6.26</td>
<td>21.45</td>
<td>1.26</td>
<td>1.41</td>
</tr>
<tr>
<td>Stage 2</td>
<td>4.70</td>
<td>28.54</td>
<td>1.33</td>
<td>1.87</td>
</tr>
<tr>
<td>Stage 3 (2 elements / thread)</td>
<td>2.84</td>
<td>47.22</td>
<td>1.65</td>
<td>3.10</td>
</tr>
</tbody>
</table>
Method X: Multiple adds / thread

- Replace single add with a loop.
- Use a counter TILE to define the number to adds per thread
  - defining TILE as global constant will allow loop unrolling
  - preferable set TILE as power of 2
Method X: Multiple adds / thread

//in global space
const int TILE = 4;

//in kernel
extern __shared__ int smem[];
int idx = blockIdx.x * blockDim.x * TILE + threadIdx.x;
int tid = threadIdx.x;
if(idx < n) {
    smem[tid] = 0;
    for(int c = 0; c < TILE; c++) {
        //can use #pragma unroll here
        if(idx + c * blockDim.x < n)
            smem[tid] += d_idata[idx + c * blockDim.x];
    }
}
syncthreads();
Method 5 : 
Last Warp Unroll

Write a device function “warpReduce” to be called by all threads with threadIdx.x < 32

__device__ void warpReduce(volatile int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}

Observe that volatile is used to declare sdata, so that the compiler doesn't reorder stores to it and induce incorrect behavior.
Method 5: Last Warp Unroll

Rewrite inner loop as:

```c
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s)
        sdata[tid] += sdata[tid + s];
    __syncthreads();
}
if (tid < 32) warpReduce(sdata, tid);
```
## Performance for 32M elements (GTX 770)

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
<th>Bandwidth (GB/s)</th>
<th>Step Speedup</th>
<th>Speed Up vs CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>8.8</td>
<td>15.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stage 0</td>
<td>7.90</td>
<td>16.98</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td>Stage 1</td>
<td>6.26</td>
<td>21.45</td>
<td>1.26</td>
<td>1.41</td>
</tr>
<tr>
<td>Stage 2</td>
<td>4.70</td>
<td>28.54</td>
<td>1.33</td>
<td>1.87</td>
</tr>
<tr>
<td>Stage 3 (2 elements / thread)</td>
<td>2.84</td>
<td>47.22</td>
<td>1.65</td>
<td>3.10</td>
</tr>
<tr>
<td>Stage 4 (32 elements / thread)</td>
<td>0.91</td>
<td>147.89</td>
<td>3.13</td>
<td>9.70</td>
</tr>
</tbody>
</table>
Method 6: Complete Unroll

● Problem:
  ○ Now we come down to inner for loop lowering the performance.

● Solution:
  ○ Unroll the for loop entirely.
  ○ Possible only if the block size is known beforehand.
  ○ Block size in GPU limited to 512 or 1024 threads.
  ○ Also make block sizes power of 2 (preferably multiples of 32).
Method 6: Complete Unroll

But block sizes is not known at compile time.

- Solution:
  - Use Templates
  - CUDA supports C++ template parameters on device and host functions
  - Specify block size as a function template parameter:
Method 6: Complete Unroll

Loop Unroll:

```c
if (blockSize >= 1024) {
    if (tid < 512) { sdata[tid] += sdata[tid + 512]; } __syncthreads();
} if (blockSize >= 512) {
    if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads();
} if (blockSize >= 256) {
    if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads();
} if (blockSize >= 128) {
    if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads();
}
if (tid< 32) warpReduce<blockSize>(sdata, tid.x);
```

The block size is known at compile time, so all the code in red is evaluated at compile time.
In the main host code add:

```c
// number of threads in the block = 256
reduce<256><<<< dimGrid, dimBlock, smemSize >>>>(d_idata, d_odata);
```
Method 6:
Complete Unroll

- Also template the device warpReduce function

//Using templates, blockSize will be defined at compile time

```cpp
template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
    sdata[tid] += sdata[tid + 32];
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid +  8];
    sdata[tid] += sdata[tid +  4];
    sdata[tid] += sdata[tid +  2];
    sdata[tid] += sdata[tid +  1];
}
```
## Performance for 32M elements (GTX 770)

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
<th>Bandwidth (GB/s)</th>
<th>Step Speedup</th>
<th>Speed Up vs CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong> (Intel Core i7 4770K)</td>
<td>8.8</td>
<td>15.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stage 0</strong></td>
<td>7.90</td>
<td>16.98</td>
<td>1.11</td>
<td>1.11</td>
</tr>
<tr>
<td><strong>Stage 1</strong></td>
<td>6.26</td>
<td>21.45</td>
<td>1.26</td>
<td>1.41</td>
</tr>
<tr>
<td><strong>Stage 2</strong></td>
<td>4.70</td>
<td>28.54</td>
<td>1.33</td>
<td>1.87</td>
</tr>
<tr>
<td><strong>Stage 3</strong></td>
<td>2.84</td>
<td>47.22</td>
<td>1.65</td>
<td>3.10</td>
</tr>
<tr>
<td>(2 elements / thread)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stage 4</strong></td>
<td>0.91</td>
<td>147.89</td>
<td>3.13</td>
<td>9.70</td>
</tr>
<tr>
<td>(32 elements / thread)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stage 5</strong></td>
<td>0.87</td>
<td>154.18</td>
<td>1.04</td>
<td>10.11</td>
</tr>
<tr>
<td>(32 elements / thread)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance Comparison

![Graph showing reduction benchmarks with data points for stages 0 to 5.]
How to make the most out of it?

- Do all the **extra credit** possible, the grade really doesn't matter.
- Practice speaking about your assignments. It really really helps in your interview and job.
- **Final projects** - break the bank on this one.
- Write excellent **blogs**. Ask Patrick/3rd person to review them.
- Live by **Git** (thanks Patrick!). Use local repos for other classes.
- See CUDA Samples (they are really good!)
- Share your knowledge
Job Advice

- Do side projects, extra work - *driven by motivation*
- Learn to write makefile
  - Good practice: try converting projects to Linux based
  - It's really not hard
  - Programs run much faster on Linux
- Maintain a coding style
- Be ready to forget everything you did in school
- [http://www.hackermeter.com](http://www.hackermeter.com)
Job Advice

- Contribute to open source projects, take part actively in forums.
  - Having an accepted contribution may mean more to development-centered companies than research.
  - Forums get you noticed. Coders have high respect for people who share knowledge.

- Do not settle for a job/role you do not want
  - It’s a burden you will not enjoy
  - Don’t go after the money
Job Advice

● We are hiring!
  ○ Email me: shehzan@accelereyes.com
  ○ or visit: www.accelereyes.com
Resources

- https://developer.nvidia.com/content/efficient-matrix-transpose-cuda-cc
Acknowledgements

- NVIDIA Documentation and website
- AccelerEyes
- UPenn CIS 565 - Patrick Cozzi