ASCI A24
DAY3: GPU PROGRAMMING

Part 2: Advanced Topics
PROGRAMMING IN CUDA
# CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int var;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td>int array_var[10];</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>shared</strong> int shared_var;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int global_var;</td>
<td>device</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>constant</strong> int constant_var;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>
Memory Spaces in CUDA

- Host
  - Shared data (per block)
  - Private data
  - Global data

- Grid
  - Block (0, 0)
    - Shared Memory
    - Registers
    - Thread (0, 0)
    - Thread (1, 0)
  - Block (1, 0)
    - Shared Memory
    - Registers
    - Thread (0, 0)
    - Thread (1, 0)

- Device Memory
- Constant Memory
- Texture Memory
Device Memory

- CPU and GPU have separate memory spaces
  - Data is moved across PCI-e bus
  - Use functions to allocate/set/copy memory on GPU
- Pointers are just addresses
  - Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must exercise care when dereferencing:
    - Dereferencing CPU pointer on GPU will likely crash
    - Same for vice versa
Other memories

- Constant memory – ReadOnly
  - Data resides in device memory
  - Manually managed
  - Small (e.g., 64KB)
  - Use when all threads in a block read the same address
    - Serializes otherwise

- Textures* – ReadOnly
  - Dimension restricted by hardware
  - Data resides in device memory
    - Different read path, includes specialized caches
    - Specialized spatial locality
  - Dedicated API for reading and processing
    - Normalized (float) coordinates
    - Interpolated values
    - Specified out-of-bounds behavior

http://cuda-programming.blogspot.nl/2013/02(texture-memory-in-cuda-what-is-texture.html
ECC (Error-Correcting Code)

- All major internal memories are ECC protected
  - Register file, L1 cache, L2 cache
- DRAM protected by ECC (on Tesla only)
- ECC is a must have for many computing applications
CUDA: SYNCHRONIZATION AND ATOMICS
Thread Scheduling

- Order of threads within a block is undefined!
  - Threads are grouped in warps (32)
    - AMD calls it “a wavefront” (64)
- Order in which thread blocks are mapped and scheduled is undefined!
  - Blocks run to completion on one SM without preemption
  - Can run in any order
    - Any possible interleaving of blocks should be valid
  - Can run concurrently OR sequentially
Global synchronization

- We launch many more blocks than physical SM’s.
- Each block might/should have more threads than the SM’s cores

```c
__global__ void my_kernel() {
    step1; // compute some values in a global array
    // wait for *all* threads to finish
    __my_global_barrier();
    step2; // use the array
}
```

```c
int main() {
    dim3 blockSize(32, 32);
    dim3 gridSize(100, 100, 100);
    my_kernel<<<gridDim, blockDim>>>();
}
```
Global synchronization

• Q: How do we do global synchronization with these scheduling semantics?
  • NOT POSSIBLE on the device itself
  • POSSIBLE by finishing a grid, and starting a new one!
    • Split work in two different kernels

\[ \text{step1}<<<\text{grid1,blk1}>>>(...) ; \]
// CUDA ensures that all writes from step1 are complete.
\[ \text{step2}<<<\text{grid2,blk2}>>>(...) ; \]

• Global, constant, and texture memories are persistent!
  • We don’t need extra data copying.
Memory consistency

- Device (global) memory is not coherent!
  - No insurance that different writes will be visible to other threads
- Share data between streaming multiprocessors
  - Potential write hazards!
- Use **atomics** to allow consistency for global (and shared memory) variables!
- Evolution:
  - Fermi has reasonable atomics for both shared and global memory
  - Kepler increases *global memory atomics* performance vs. Fermi
  - Maxwell uses native support for shared memory atomics
    - Much faster than Fermi and Kepler
Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
  - Ordering is still arbitrary
- Different types of atomic instructions
  - Atomic Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Both device memory and shared memory
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;
}
Example: Histogram

// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter atomically

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
CUDA: MEMORY COALESCING
Coalescing

traditional multi-core
optimal memory access pattern

thread 0

$\begin{align*}
\text{t = 0} & \rightarrow \text{address 0} \\
\text{t = 1} & \rightarrow \text{address 1}
\end{align*}$

thread 1

$\begin{align*}
\text{t = 0} & \rightarrow \text{address 2} \\
\text{t = 1} & \rightarrow \text{address 3}
\end{align*}$

thread 2

$\begin{align*}
\text{t = 0} & \rightarrow \text{address 4} \\
\text{t = 1} & \rightarrow \text{address 5}
\end{align*}$

thread 3

$\begin{align*}
\text{t = 0} & \rightarrow \text{address 6} \\
\text{t = 1} & \rightarrow \text{address 7}
\end{align*}$

many-core GPU
optimal memory access pattern

thread 0

$\begin{align*}
\text{t = 0} & \rightarrow \text{address 0} \\
\text{t = 1} & \rightarrow \text{address 1}
\end{align*}$

thread 1

$\begin{align*}
\text{t = 0} & \rightarrow \text{address 2} \\
\text{t = 1} & \rightarrow \text{address 3}
\end{align*}$

thread 2

$\begin{align*}
\text{t = 0} & \rightarrow \text{address 4} \\
\text{t = 1} & \rightarrow \text{address 5}
\end{align*}$

thread 3

$\begin{align*}
\text{t = 0} & \rightarrow \text{address 6} \\
\text{t = 1} & \rightarrow \text{address 7}
\end{align*}$
Memory Coalescing

- **Memory coalescing** refers to combining multiple memory accesses into a single transaction.
Consider the stride of your accesses

```c
__global__ void foo(int* input, float3* input2) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    // Stride 1, full bandwidth used!
    int a = input[i];

    // Stride 2, 50% of the bandwidth is wasted
    int b = input[2*i];

    // Stride 3, 67% of the bandwidth is wasted
    float c = input2[i].x;
}
```
Example: Array of Structures (AoS)

```c
struct record {
    int key;
    int value;
    int flag;
};

record *d_AoS_data;
cudaMalloc((void**)&d_AoS_data, ...);

kernel {
    threadID = blockDim.x * blockIdx.x + threadIdx.x;
    // ...
    d_AoS_data[threadID].value += i; // wastes bandwidth!
    // ...
}
```
Example: Structure of Arrays (SoA)

```c
struct SoA {
    int* keys;
    int* values;
    int* flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);

kernel {
    threadID = blockDim.x * blockIdx.x + threadIdx.x;
    ...
    d_SoA_data.values[threadID] += i; // full bandwidth!
    ...
}
```
Example: SoA vs AoS

```c
__global__ void bar(record* AoS_data,
                     SoA  SoA_data) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    // AoS wastes bandwidth
    int key1 = AoS_data[i].key;

    // SoA efficient use of bandwidth
    int key2 = SoA_data.keys[i];
}
```
Memory Coalescing

- Structure of arrays is often better than array of structures
- Stride 1 access patterns are preferred!
  - Other patterns can still get benefits
- Unpredictable/irregular access patterns
  - Case-by-case performance impact

- No coalescing => performance loss 10 – 30x!
CUDA: USING SHARED MEMORY
Using shared memory

- Equivalent with providing software caching
  - **Explicit**: Load data to be re-used in shared memory
  - Use it for computation
  - **Explicit**: Store results back to global memory
- All threads in a block share memory
  - Load/Store: using all threads
  - **Barrier**: `__syncthreads`
    - Guard against using uninitialized data – not all threads have finished loading data to shared memory
    - Guard against corrupting live data – not all threads have finished computing
A Common Programming Strategy

- Partition data into subsets that fit into shared memory
A Common Programming Strategy

- Handle each data subset with one thread block
A Common Programming Strategy

- Load the subset from device memory to shared memory, using multiple threads to exploit memory-level parallelism.
A Common Programming Strategy

- Perform the computation on the subset from shared memory
A Common Programming Strategy

- Copy the result from shared memory back to device memory
Caches vs. Shared Memory

• Since Fermi, NVIDIA GPUs feature BOTH hardware **L1 caches** and **shared memory** per SM
  • They share the same space
    • $\frac{3}{4}$ Cache + $\frac{1}{4}$ Shared Memory  OR
    • $\frac{1}{4}$ Cache + $\frac{3}{4}$ Shared Memory

• **L1 Cache**
  • Hardware caching enabled
    • The HW decides what goes in or out and when

• **Shared memory**
  • Software manages what goes in/out
  • Allows more complex access patterns to be cached
Matrix multiplication example

• $C = A \times B$
  • $C(i,j) = \text{sum}(\text{dot(row}(A,i),\text{col}(B,j)))$

• Parallelization strategy
  • Each thread computes one C element
  • 2D kernel
Matrix multiplication implementation

```c
__global__ void mat_mul(float *a, float *b,
                         float *c, int width)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    float result = 0;

    // do dot product between row of a and column of b
    for(int k = 0; k < width; k++) {
        result += a[row*width+k] * b[k*width+col];
    }

    c[row*width+col] = result;
}
```
# Matrix multiplication performance

<table>
<thead>
<tr>
<th>Loads per dot product term</th>
<th>2 (a and b) = 8 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOPS</td>
<td>2 (multiply and add)</td>
</tr>
<tr>
<td>AI</td>
<td>2 / 8 = 0.25</td>
</tr>
<tr>
<td>Performance GTX 580</td>
<td>1581 GFLOPs</td>
</tr>
<tr>
<td>Memory bandwidth GTX 580</td>
<td>192 GB/s</td>
</tr>
<tr>
<td>Attainable performance</td>
<td>192 * 0.25 = 48 GFLOPS</td>
</tr>
<tr>
<td>Maximum efficiency</td>
<td>3.0 % of theoretical peak</td>
</tr>
</tbody>
</table>
Data reuse

- Each input element in A and B is read WIDTH times

- IDEA:
  - Load elements into shared memory
  - Have several threads use local version to improve memory bandwidth
Using shared memory

- Partition kernel loop into phases
- In each thread block, load a tile of both matrices into shared memory each phase
- Each phase, each thread computes a partial result
Matrix multiply with shared memory

```c
__global__ void mat_mul(float *a, float *b,
                        float *c, int width) {

    // shorthand
    int tx = threadIdx.x, ty = threadIdx.y;
    int bx = blockIdx.x, by = blockIdx.y;

    // allocate tiles in shared memory
    __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];

    // calculate the row & column index from A,B
    int row = by*blockDim.y + ty;
    int col = bx*blockDim.x + tx;

    float result = 0;
```
Matrix multiply with shared memory

// loop over input tiles in phases, p = crt. phase
for(int p = 0; p < width/TILE_WIDTH; p++) {
    // collaboratively load tiles into shared memory
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(p*TILE_WIDTH + ty)*width + col];
    __syncthreads();

    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; k++) {
        result += s_a[ty][k] * s_b[k][tx];
    }
    __syncthreads();
}

c[row*width+col] = result;
Use of Barriers in mat_mul

• Two barriers per phase:
  • __syncthreads after all data is loaded into shared memory
  • __syncthreads after all data is read from shared memory
    • Second __syncthreads in phase p guards the load in phase p+1

• Formally, __synchthreads is a barrier for shared memory for a block of threads:

“void __syncthreads();
waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to __syncthreads() are visible to all threads in the block.”
# Matrix multiplication performance

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>shared memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global loads</td>
<td>$2N^3 \times 4$ bytes</td>
<td>$(2N^3 / \text{TILE_WIDTH}) \times 4$ bytes</td>
</tr>
<tr>
<td>Total ops</td>
<td>$2N^3$</td>
<td>$2N^3$</td>
</tr>
<tr>
<td>AI</td>
<td>0.25</td>
<td>$0.25 \times \text{TILE_WIDTH}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance GTX 580</th>
<th>1581 GFLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory bandwidth GTX 580</td>
<td>192 GB/s</td>
</tr>
<tr>
<td>AI needed for peak</td>
<td>$1581 / 192 = 8.23$</td>
</tr>
<tr>
<td>TILE_WIDTH required to achieve peak</td>
<td>$0.25 \times \text{TILE_WIDTH} = 8.23$,\n  $\text{TILE_WIDTH} = 32.9$</td>
</tr>
</tbody>
</table>
CUDA: WARPS AND OCCUPANCY
Thread Scheduling

- SMs implement zero-overhead warp scheduling
  - A warp is a group of 32 threads that runs concurrently on an SM
  - At any time, the number of warps concurrently executed by an SM is limited by its number of cores.
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

[Diagram showing thread block and warp scheduling]

TB = Thread Block, W = Warp
Stalling warps

• What happens if all warps are stalled?
  • No instruction issued → performance lost

• Most common reason for stalling?
  • Waiting on global memory

• If your code reads global memory every couple of instructions
  • You should try to maximize occupancy
Occupancy

• What determines occupancy?
• Limited resources!
  • Register usage per thread
  • Shared memory per thread block
Resource Limits (1)

- Pool of registers and shared memory per SM
  - Each thread block grabs registers & shared memory
  - If one or the other is fully utilized, no more thread blocks
Resource Limits (2)

- Can only have $P$ thread blocks per SM
  - If they’re too small, can’t fill up the SM
  - Need 128 threads / block on gt200 (4 cycles/instruction)
  - Need 192 threads / block on Fermi (6 cycles/instruction)

- Higher occupancy has diminishing returns for hiding latency
Hiding Latency with more threads

Throughput, 32-bit words

![Graph showing throughput versus threads per multiprocessor. The x-axis represents threads per multiprocessor, ranging from 0 to 1024. The y-axis represents throughput in GB/s, ranging from 0 to 100. The graph shows an initial increase in throughput as the number of threads increases, reaching a peak around 512 threads, before stabilizing at a higher throughput compared to the initial value.](image-url)
How do you know what you’re using?

- Use compiler flags to get register and shared memory usage
  - “nvcc -Xptxas -v”
- Use the NVIDIA Profiler
- Plug those numbers into CUDA Occupancy Calculator

- Maximize occupancy for improved performance
  - Empirical rule! Don’t overuse!
1. Select Compute Capability (click): 1.3

2. Enter your resource usage:
   - Threads Per Block: 128
   - Registers Per Thread: 28
   - Shared Memory Per Block (bytes): 640

The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

**GPU Occupancy Data is displayed here and in the graphs:**

- Active Threads per Multithread: 512
- Active Threads per Multithread (8x): 1024
- Active Threads per Multithread (16x): 2048
- Active Threads per Multithread (32x): 4096

**Physical Limits for GPU Compute Capability:** 1.3

<table>
<thead>
<tr>
<th>Threads per Warp</th>
<th>Warps per Multithreads</th>
<th>Warps per Multithread (8x)</th>
<th>Warps per Multithread (16x)</th>
<th>Warps per Multithread (32x)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread Blocks per Multithread</th>
<th>Total # of registers per Multithread</th>
<th>Register allocation unit size</th>
<th>Register allocation granularity</th>
<th>Shared Memory per Multithread (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Allocation Per Thread Block**

<table>
<thead>
<tr>
<th>Wafps</th>
<th>Registers</th>
<th>Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3684</td>
<td>1024</td>
</tr>
</tbody>
</table>

These data are used in computing the occupancy data in blue.

**Maximum Thread Blocks Per Multithread**

<table>
<thead>
<tr>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Limited by Max Warps/Blocks per Multithread: 8
Limited by Registers per Multithread: 28
Limited by Shared Memory per Multithread: 640

**Thread Block Limit Per Multithread highlighted** RED

**CUDA Occupancy Calculator**

Version: 2.0

Copyright and License
Thread divergence

• “I heard GPU branching is expensive. Is this true?”

```c
__global__ void Divergence(float* dst, float* src) {
    float value = 0.0f;

    if ( threadIdx.x % 2 == 0 )
        // active threads : 50%
        value = src[0] + 5.0f;
    else
        // active threads : 50%
        value = src[0] - 5.0f;

    dst[index] = value;
}
```
Execution

```c
unsigned int index = ( blockDim.x * blockIdx.x ) + threadIdx.x;
float value = 0.0f;

if ( threadIdx.x % 2 == 0 )
{
    value = PathA( src );
}
else
{
    value = PathB( src );
}

dst[index] = value;
```

Worst case performance loss:
50% compared with the non divergent case.
Another example

Not all ALUs do useful work!
Worst case: 1/8 peak performance

(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```c
float x = A[i];
if (x > 0) {
    float tmp = exp(x, 5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}

// resume unconditional code
result[i] = x;
```
Performance penalty?

- Depends on the amount of divergence
  - Worst case: 1/32 performance
    - When each thread does something different
- Depends on whether branching is data- or ID- dependent
  - If ID – consider grouping threads differently
  - If data – consider sorting
- Non-diverging warps => NO performance penalty
  - In this case, branches are not expensive …
CUDA: STREAMS
What are streams?

- **Stream** = a sequence of operations that execute on the device in the order in which they are issued by the host code.
  - Same stream: In-Order execution
  - Different streams: Out-of-Order execution

- **Default stream** = Synchronizing stream
  - No operation in the default stream can begin until all previously issued operations in any stream on the device have completed.
  - An operation in the default stream must complete before any other operation in any stream on the device can begin.
Default stream: example

cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
increment<<<1,N>>>(d_a);
CpuFunction(b);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);

- All operations happen in the same stream
- Device (GPU)
  - Synchronous execution
    - all operations execute (in order), one after the previous has finished
  - Unaware of CpuFunction()
- Host (CPU)
  - Launches increment and regains control
  - *May* execute CpuFunction *before* increment has finished
  - Final copy starts *after* both increment and CpuFunction() have finished
Non-default streams

• Enable asynchronous execution and overlaps
  • Require special creation/deletion of streams
    • `cudaStreamCreate(&stream1)`
    • `cudaStreamDestroy(stream1)`
  • Special memory operations
    • `cudaMemcpyAsync(deviceMem, hostMem, size, cudaMemcpyHostToDevice, stream1)`
  • Special kernel parameter (the 4\(^{th}\) one)
    • `increment<<1, N, 0, stream1>>(d_a)`

• Synchronization
  • All streams
    • `cudaDeviceSynchronize()`
  • Specific stream:
    • `cudaStreamSynchronize(stream1)`
Computation vs. communication

//Single stream, numBytes = 16M, numElements = 4M
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
kernel<<blocks,threads>>>(d_a, firstElement);
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);

C1060 (pre-Fermi): 12.9ms

C2050 (Fermi): 9.9ms
Computation-communication overlap[1]*

for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, stream[i]);
    kernel<<blocks,threads,0,stream[i]>>(d_a, offset);
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, stream[i]);
}

C1060 (pre-Fermi): 13.63 ms (worse than sequential)

C2050 (Fermi): 5.73 ms (better than sequential)

Computation-communication overlap[^2]*

```c
for (int i = 0; i < nStreams; ++i) offset[i] = i * streamSize;
for (int i = 0; i < nStreams; ++i)
    cudaMemcpyAsync(&d_a[offset[i]], &a[offset[i]], streamBytes,
                     cudaMemcpyHostToDevice, stream[i]);

for (int i = 0; i < nStreams; ++i)
    kernel<<<blocks, threads, 0, stream[i]>>>(d_a, offset);

for (int i = 0; i < nStreams; ++i)
    cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
                     cudaMemcpyDeviceToHost, stream[i]);
```

C1060 (pre-Fermi): 8.84 ms (better than sequential)

![Diagram](http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/)

C2050 (Fermi): 7.59 ms (better than sequential, worse than v1)
CUDA: SHARED MEMORY BANK CONFLICTS
Shared Memory Banks

- Shared memory is banked
  - Only matters for threads within a warp
  - Full performance with some restrictions
    - Threads can each access different banks
    - Or can all access the same value

- Consecutive words are in different banks
- If two or more threads access the same bank but different value, we get bank conflicts
Bank Addressing Examples: OK

- No Bank Conflicts

- No Bank Conflicts
Bank Addressing Examples: BAD

- **2-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 8
  - Thread 9
  - Thread 10
  - Thread 11
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 3
  - Bank 4
  - Bank 5
  - Bank 6
  - Bank 7
  - Bank 15

- **8-way Bank Conflicts**
  - Thread 0
  - Thread 1
  - Thread 2
  - Thread 3
  - Thread 4
  - Thread 5
  - Thread 6
  - Thread 7
  - Thread 15
  - Bank 0
  - Bank 1
  - Bank 2
  - Bank 7
  - Bank 8
  - Bank 9
  - Bank 15
Trick to Assess Performance Impact

- Change all shared memory reads to the same value
- All broadcasts = no conflicts
- Will show how much performance could be improved by eliminating bank conflicts

- The same doesn’t work for shared memory writes
  - So, replace shared memory array indices with `threadIdx.x`
  - (Could also be done for the reads)
CUDA: MANY OTHER FEATURES
HyperQ

CPU Cores Simultaneously Run Tasks on Kepler

FERMI
1 MPI Task at a Time

KEPLER
32 Simultaneous MPI Tasks
Dynamic parallelism

Dynamic Parallelism

GPU Adapts to Data, Dynamically Launches New Threads

CPU → Fermi GPU

CPU → Kepler GPU
Dynamic parallelism

Stream Queue
Ordered queues of grids

One-way Flow

Work Distributor
Tracks blocks issued from grids
16 Active Grids

SM  SM  SM  SM

CUDA-Created Work

Stream Queues
Ordered queues of grids

Grid Management Unit
Pending & suspended grids
1000's of pending grids

Two-way link allows pausing dispatch

Work Distributor
Actively dispatching grids
32 Active Grids
GPUDirect

Direct Transfers between GPU and 3rd Party Devices

Server 1

Server 2
SUMMARY AND CONCLUSIONS
Summary and conclusions

• CUDA is a high-performance programming model
  • Simple semantics of the hardware
  • Easy to obtain functionality
  • Increasingly difficult to optimize

• Choose optimizations wisely
  • Amdahl’s Law
  • Roofline mode

• Heterogeneous computing is the way of the future
  • CPUs and GPUs are closer than you think in terms of performance
  • Using them jointly is key to improve the performance of more applications
BACKUP SLIDES
// For algorithms where the amount of work per item
// is highly non-uniform, it often makes sense to
// continuously grab work from a queue.

__global__
void workq(int* work_q, int* q_counter, 
            int queue_max, int* output)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int q_index = atomicInc(q_counter, queue_max);
    int result = do_work(work_q[q_index]);
    output[q_index] = result;
}
Using shared memory

// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread’s global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
Using shared memory

// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input) {
    // compute this thread's global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i > 0) {
        // each thread loads two elements from device memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}

The next thread also reads input[i]
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i];

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}
Using shared memory: coalescing

```c
__global__ void adj_diff(int *result, int *input) {
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
    // each thread reads 1 device memory elt, stores it in s_data
    s_data[threadIdx.x] = input[i]; // COALESCED ACCESS!

    // avoid race condition: ensure all loads are complete
    __syncthreads();

    if(threadIdx.x > 0) {
        result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
    } else if(i > 0) {
        // I am thread 0 in this block: handle thread block boundary
        result[i] = s_data[threadIdx.x] - input[i-1];
    }
}
```