The GPU Teaching Kit is licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License.
Warps as Scheduling Units

- Each block is divided into 32-thread warps
  - An implementation technique, not part of the CUDA programming model
  - Warps are scheduling units in SM
  - Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
  - The number of threads in a warp may vary in future generations
Warps in Multi-dimensional Thread Blocks

- The thread blocks are first linearized into 1D in row major order
  - In x-dimension first, y-dimension next, and z-dimension last
Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0

- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation

- DO NOT rely on any ordering within or between warps
  - If there are any dependencies between threads, you must __syncthreads() to get correct results (more later).
SIMD Execution Among Threads in a Warp

- All threads in a warp must execute the same instruction at any point in time

- This works efficiently if all threads follow the same control flow path
  - All if-then-else statements make the same decision
  - All loops iterate the same number of times
Branch Divergence in Warps

• occurs when threads inside warps branches to different execution paths.

50% performance loss
Control Divergence

- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - Some take the then-path and others take the else-path of an if-statement
  - Some threads take different number of loop iterations than others
- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - During the execution of each path, all threads taking that path will be executed in parallel
  - The number of different paths can be large when considering nested control flow statements
Dealing With Branch Divergence

• A common case: avoid divergence when branch condition is a function of thread ID
  – Example with divergence:
    • If (threadIdx.x > 2) { }
    • This creates two different control paths for threads in a block
  – Example without divergence:
    • If (threadIdx.x / WARP_SIZE > 2) { }
    • Also creates two different control paths for threads in a block
    • Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

• There is a big body of research for dealing with branch divergence
Control Divergence Examples

- Divergence can arise when branch or loop condition is a function of thread indices
- Example kernel statement with divergence:
  - if (threadIdx.x > 2) { }
  - This creates two different control paths for threads in a block
  - Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- Example without divergence:
  - If (blockIdx.x > 2) { }
  - Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path
Example: Vector Addition Kernel

Device Code

// Compute vector sum $C = A + B$
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
Analysis for vector size of 1,000 elements

- Assume that block size is 256 threads
  - 8 warps in each block

- All threads in Blocks 0, 1, and 2 are within valid range
  - i values from 0 to 767
  - There are 24 warps in these three blocks, none will have control divergence

- Most warps in Block 3 will not control divergence
  - Threads in the warps 0-6 are all within valid range, thus no control divergence

- One warp in Block 3 will have control divergence
  - Threads with i values 992-999 will all be within valid range
  - Threads with i values of 1000-1023 will be outside valid range

- Effect of serialization on control divergence will be small
  - 1 out of 32 warps has control divergence
  - The impact on performance will likely be less than 3%
Parallel Reduction (max / sum / etc.)
One Parallel Reduction Kernel

__shared__ float partialSum[SIZE];

partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x + threadIdx.x];
unsigned int t = threadIdx.x;
for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {

    __syncthreads();
    if (t % (2 * stride) == 0)
        partialSum[t] += partialSum[t+stride];
}


One Parallel Reduction Kernel

__shared__ float partialSum[SIZE];

partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x + threadIdx.x];
unsigned int t = threadIdx.x;
for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {

    __syncthreads();
    if (t % (2 * stride) == 0)
        partialSum[t] += partialSum[t+stride];
}

\[\text{Diagram showing the reduction process, with data elements and steps.}\]
One Parallel Reduction Kernel

__shared__ float partialSum[SIZE];

partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x + threadIdx.x];
unsigned int t = threadIdx.x;
for (unsigned int stride = 1; stride < blockDim.x; stride *= 2) {
    __syncthreads();
    if (t % (2 * stride) == 0)
        partialSum[t] += partialSum[t+stride];
}

![Diagram of parallel reduction kernel with data elements and steps]
A Better Parallel Reduction Kernel

__shared__ float partialSum[SIZE];

partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x + threadIdx.x];
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x/2; stride >= 1; stride >>= 1) {
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
A Better Parallel Reduction Kernel

__shared__ float partialSum[SIZE];

partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x + threadIdx.x];
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x/2; stride >= 1; stride >>= 1) {
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 5</th>
<th>Thread 6</th>
<th>Thread 7</th>
<th>Thread 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>13</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of underutilization

Computational Resource Utilization

32 warps, 32 threads per warp, round-robin scheduling