

# ECE 498AL

## Lecture 10: Control Flow

## Objective

- To understand the implications of control flow on
  - Branch divergence overhead
  - SM execution resource utilization
- To learn better ways to write code with control flow
- To understand compiler/HW predication designed to reduce the impact of control flow
  - There is a cost involved.

## Quick terminology review

- *Thread*: concurrent code and associated state executed on the CUDA device (in parallel with other threads)
  - The unit of parallelism in CUDA
- *Warp*: a group of threads executed *physically* in parallel in G80
- *Block*: a group of threads that are executed together and form the unit of resource assignment
- *Grid*: a group of thread blocks that must all complete before the next phase of the program can begin

## How thread blocks are partitioned

- Thread blocks are partitioned into warps
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0
- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
  - (Covered next)
- **However, DO NOT rely on any ordering between warps**
  - If there are any dependencies between threads, you must `__syncthreads()` to get correct results

# Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized in G80
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- 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
    - Branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp
  - 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

# Parallel Reduction

- Given an array of values, “reduce” them to a single value in parallel
- Examples
  - sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array
- Typically parallel implementation:
  - Recursively halve # threads, add two values per thread
  - Takes  $\log(n)$  steps for  $n$  elements, requires  $n/2$  threads

## A Vector Reduction Example

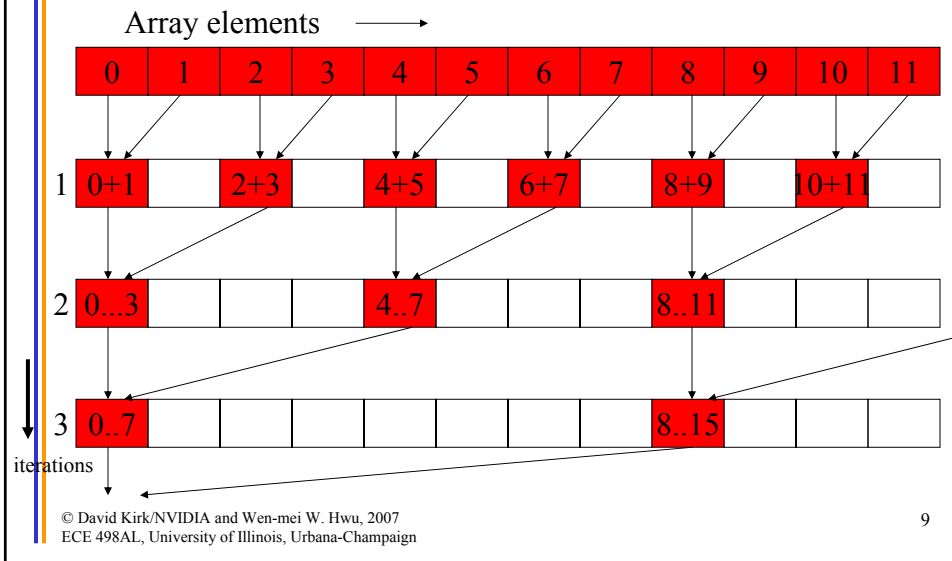
- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0

## A simple implementation

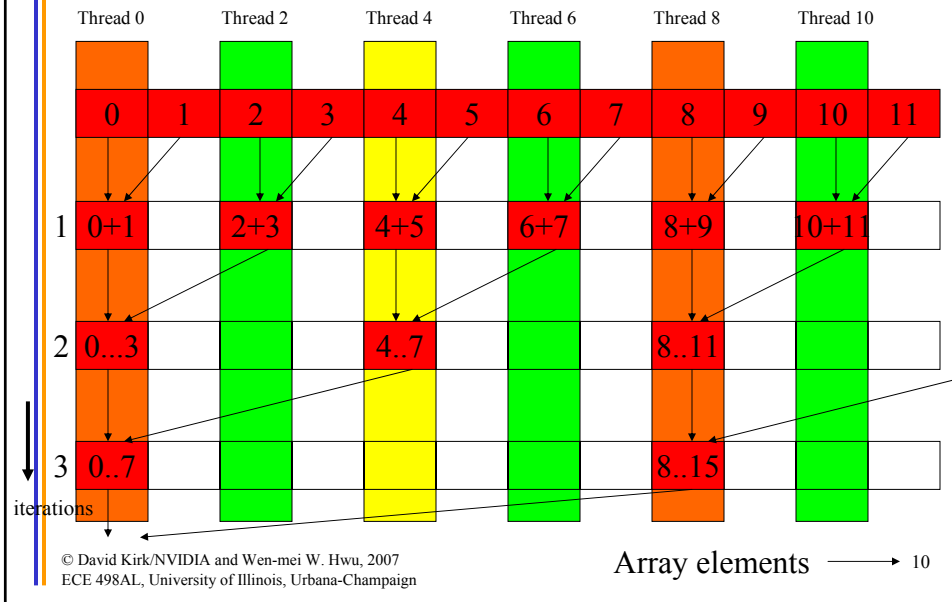
- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```
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];
}
```

# Vector Reduction with Bank Conflicts



# Vector Reduction with Branch Divergence



## Some Observations

- In each iterations, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence
- No more than half of threads will be executing at any time
  - All odd index threads are disabled right from the beginning!
  - On average, less than  $\frac{1}{4}$  of the threads will be activated for all warps over time.
  - After the 5<sup>th</sup> iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    - This can go on for a while, up to 4 more iterations ( $512/32=16=2^4$ ), where each iteration only has one thread activated until all warps retire

## Short comings of the implementation

- Assume we have already loaded array into

```
- __shared__ float partialSum[]  
  
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];  
}
```

BAD: Divergence  
due to interleaved  
branch decisions

BAD: Bank  
conflicts due to  
stride

## A better implementation

- Assume we have already loaded array into

```
- __shared__ float partialSum[]
```

```
unsigned int t = threadIdx.x;
```

```
for (unsigned int stride = blockDim.x;
```

```
    stride > 1; stride >> 1)
```

```
{
```

```
    __syncthreads();
```

```
    if (t < stride)
```

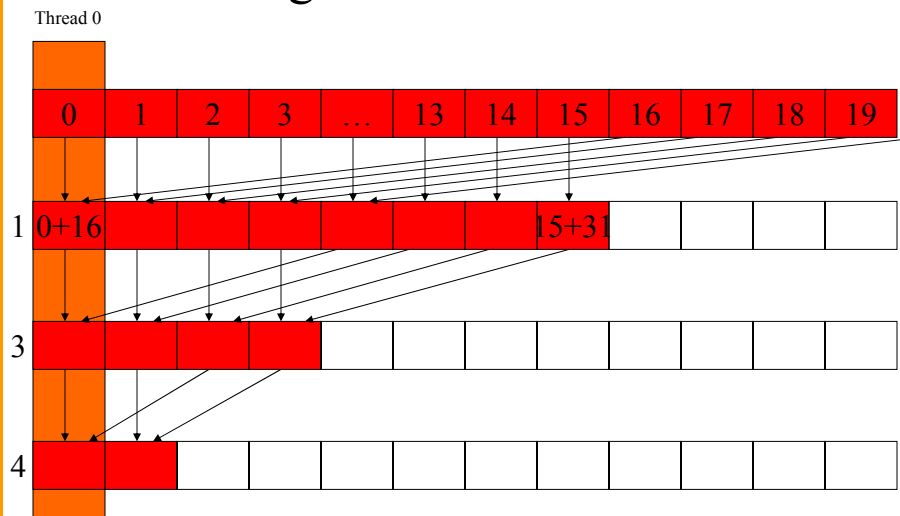
```
        partialSum[t] += partialSum[t+stride];
```

```
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007  
ECE 498AL, University of Illinois, Urbana-Champaign

13

## No Divergence until < 16 sub-sums



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007  
ECE 498AL, University of Illinois, Urbana-Champaign

14

## Some Observations About the New Implementation

- Only the last 5 iterations will have divergence
- Entire warps will be shut down as iterations progress
  - For a 512-thread block, 4 iterations to shut down all but one warps in each block
  - Better resource utilization, will likely retire warps and thus blocks faster
- Recall, no bank conflicts either

## A Potential Further Refinement but bad idea

- For last 6 loops only one warp active (i.e. tid's 0..31)
  - Shared reads & writes SIMD synchronous within a warp
  - So skip `__syncthreads()` and

```
unsigned int tid = threadIdx.x;
for (unsigned int d = n>>2; d>0; d>>=1)
    __syncthreads();
    if (tid < d)
        shared[tid] += shared[tid + d];
}
__syncthreads();
if (tid <= 32) { // unrolled
    shared[tid] += shared[tid + 16];
    shared[tid] += shared[tid + 8];
    shared[tid] += shared[tid + 4];
    shared[tid] += shared[tid + 2];
    shared[tid] += shared[tid + 1];
}
```

This would not work properly  
is warp size decreases; need  
`__syncthreads()` between each  
statement!  
However, having  
`__syncthreads()` in if  
statement is problematic.



## Predicated Execution Concept

`<p1> LDR r1, r2, 0`

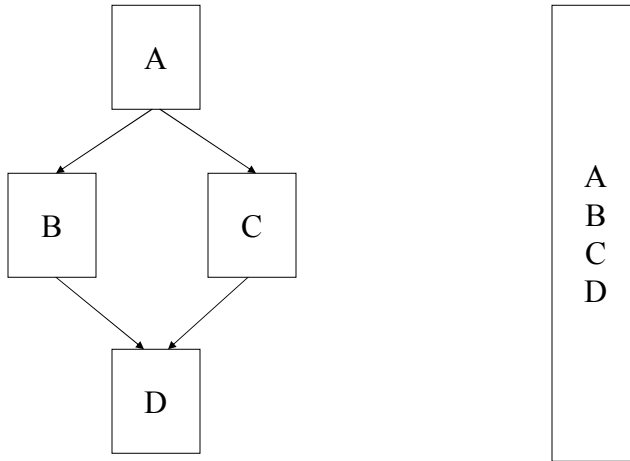
- If p1 is TRUE, instruction executes normally
- If p1 is FALSE, instruction treated as NOP

## Predication Example

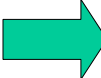
```

:                               :
:                               :
if (x == 10)                   LDR r5, X
    c = c + 1;                 p1 <- r5 eq 10
:                               <p1> LDR r1 <- C
:                               <p1> ADD r1, r1, 1
:                               <p1> STR r1 -> C
:                               :
:                               :
```

## Predication very helpful for if-else



## If-else example

|                                                                                                                                                              |                                                                                                     |                                                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> : : p1,p2 &lt;- r5 eq 10 &lt;p1&gt; inst 1 from B &lt;p1&gt; inst 2 from B &lt;p1&gt; : : &lt;p2&gt; inst 1 from C &lt;p2&gt; inst 2 from C : : </pre> |  <p>schedule</p> | <pre> : : p1,p2 &lt;- r5 eq 10 &lt;p1&gt; inst 1 from B &lt;p2&gt; inst 1 from C &lt;p1&gt; inst 2 from B &lt;p2&gt; inst 2 from C &lt;p1&gt; : : </pre> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|

The cost is extra instructions will be issued each time the code is executed. However, there is no branch divergence.

# Instruction Predication in G80

- Comparison instructions set condition codes (CC)
- Instructions can be predicated to write results only when CC meets criterion (CC != 0, CC >= 0, etc.)
- Compiler tries to predict if a branch condition is likely to produce many divergent warps
  - If guaranteed not to diverge: only predicates if < 4 instructions
  - If not guaranteed: only predicates if < 7 instructions
- May replace branches with instruction predication
- ALL predicated instructions take execution cycles
  - Those with false conditions don't write their output
    - Or invoke memory loads and stores
  - Saves branch instructions, so can be cheaper than serializing divergent paths