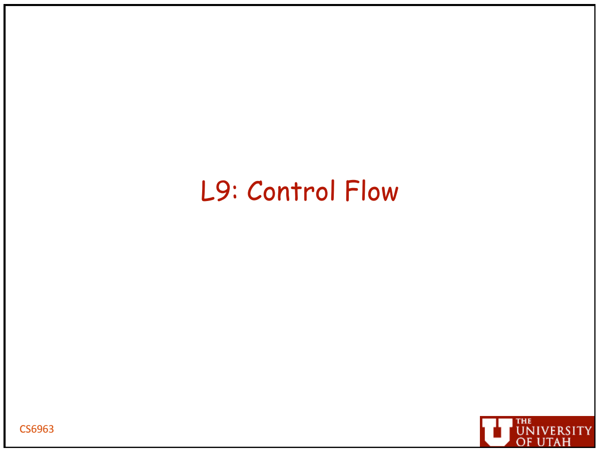
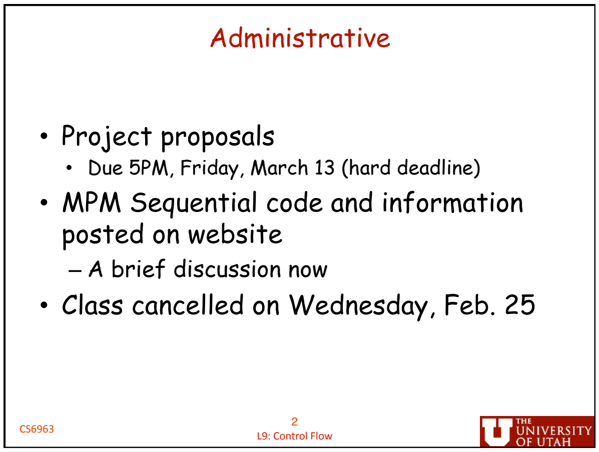
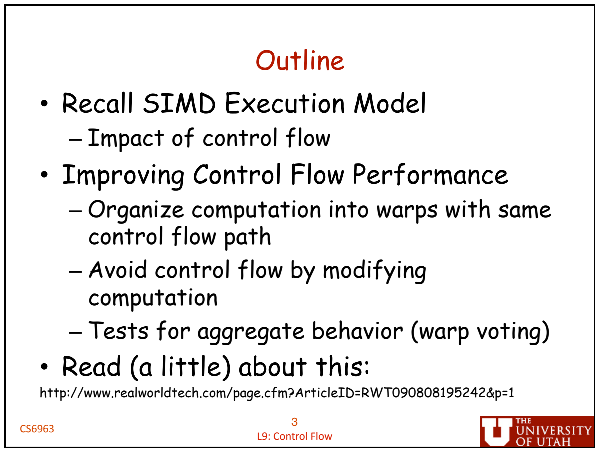


L9: Control Flow



Administrative

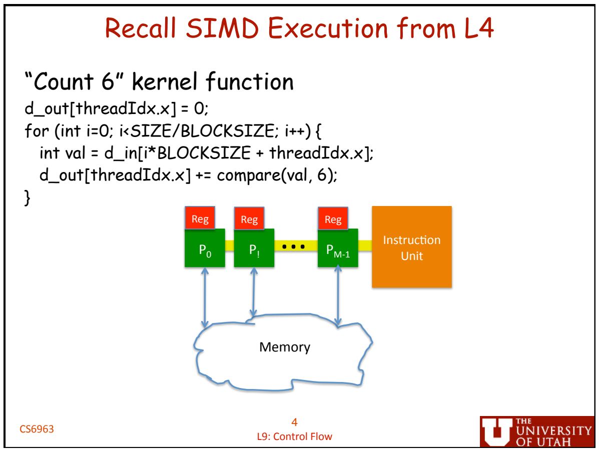
- Project proposals
  - Due 5PM, Friday, March 13 (hard deadline)
- MPM Sequential code and information posted on website
  - A brief discussion now
- Class cancelled on Wednesday, Feb. 25



Outline

- Recall SIMD Execution Model
  - Impact of control flow
- Improving Control Flow Performance
  - Organize computation into warps with same control flow path
  - Avoid control flow by modifying computation
  - Tests for aggregate behavior (warp voting)
- Read (a little) about this:

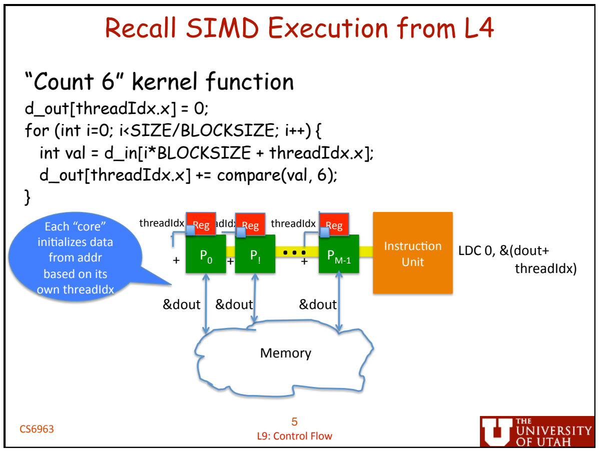
http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242&p=1



Recall SIMD Execution from L4

"Count 6" kernel function

```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
   int val = d_in[i*BLOCKSIZE + threadIdx.x];
   d_out[threadIdx.x] += compare(val, 6);
}
```



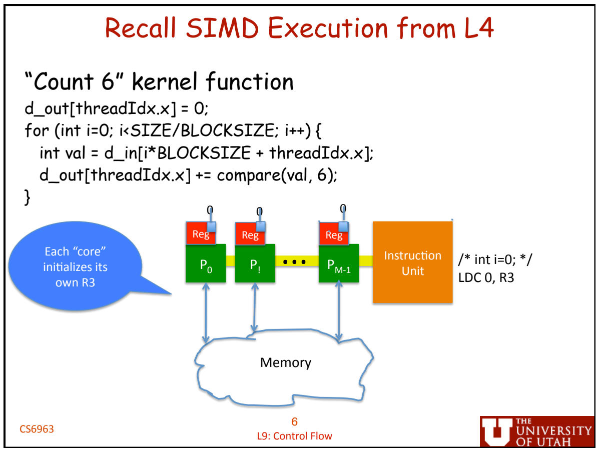
Recall SIMD Execution from L4

"Count 6" kernel function

```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
   int val = d_in[i*BLOCKSIZE + threadIdx.x];
   d_out[threadIdx.x] += compare(val, 6);
}
```

Each "core" initializes data from addr based on its own threadIdx

LDC 0, &(dout+ threadIdx)



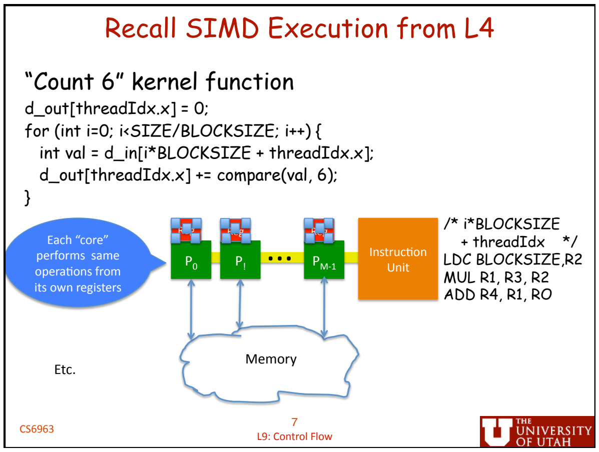
Recall SIMD Execution from L4

"Count 6" kernel function

```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
   int val = d_in[i*BLOCKSIZE + threadIdx.x];
   d_out[threadIdx.x] += compare(val, 6);
}
```

Each "core" initializes its own R3

```
/* int i=0; */
LDC 0, R3
```



Recall SIMD Execution from L4

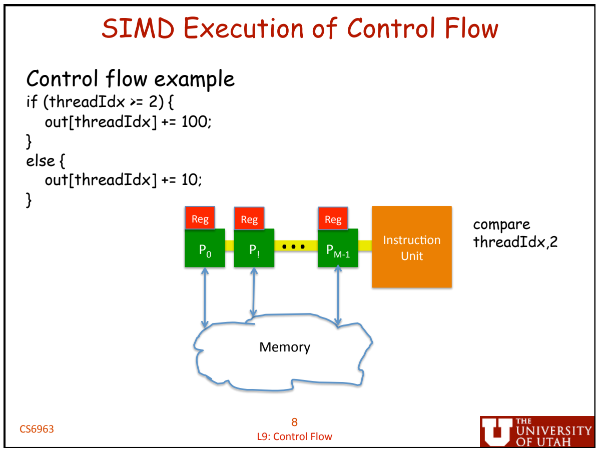
"Count 6" kernel function

```
d_out[threadIdx.x] = 0;
for (int i=0; i<SIZE/BLOCKSIZE; i++) {
   int val = d_in[i*BLOCKSIZE + threadIdx.x];
   d_out[threadIdx.x] += compare(val, 6);
}
```

Each "core" performs same operations from its own registers

```
/* i*BLOCKSIZE
   + threadIdx    */
LDC BLOCKSIZE,R2
MUL R1, R3, R2
ADD R4, R1, RO
```

Etc.

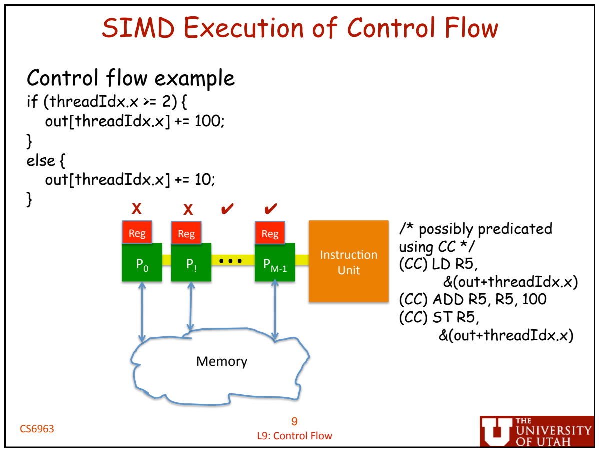


SIMD Execution of Control Flow

Control flow example

```
if (threadIdx >= 2) {
   out[threadIdx] += 100;
}
else {
   out[threadIdx] += 10;
}
```

```
compare
threadIdx,2
```

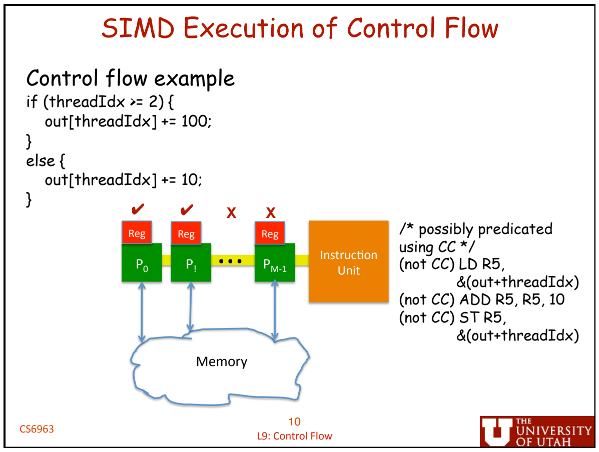


SIMD Execution of Control Flow

Control flow example

```
if (threadIdx.x >= 2) {
   out[threadIdx.x] += 100;
}
else {
   out[threadIdx.x] += 10;
}
```

```
/* possibly predicated
using CC */
(CC) LD R5,
       &(out+threadIdx.x)
(CC) ADD R5, R5, 100
(CC) ST R5,
       &(out+threadIdx.x)
```

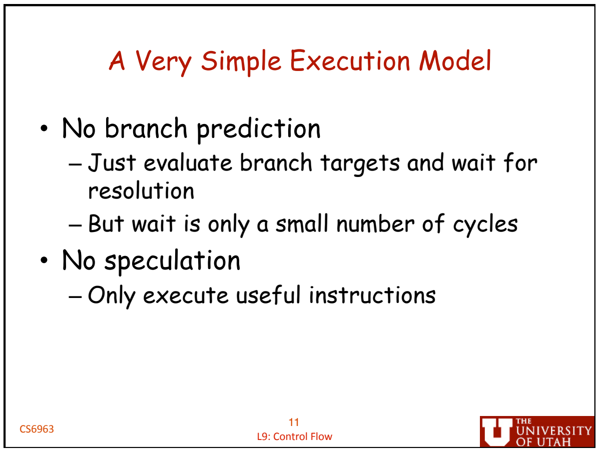


SIMD Execution of Control Flow

Control flow example

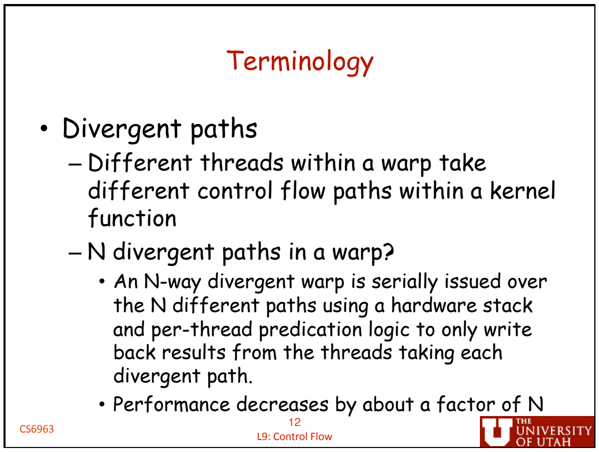
```
if (threadIdx >= 2) {
   out[threadIdx] += 100;
}
else {
   out[threadIdx] += 10;
}
```

```
/* possibly predicated
using CC */
(not CC) LD R5,
       &(out+threadIdx)
(not CC) ADD R5, R5, 10
(not CC) ST R5,
       &(out+threadIdx)
```



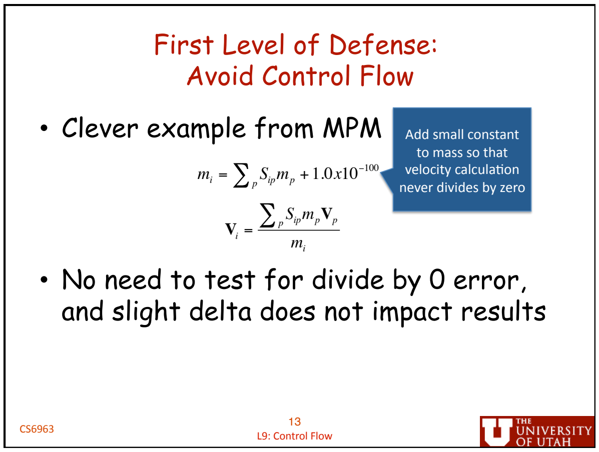
A Very Simple Execution Model

- No branch prediction
  - Just evaluate branch targets and wait for resolution
  - But wait is only a small number of cycles
- No speculation
  - Only execute useful instructions



Terminology

- Divergent paths
  - Different threads within a warp take different control flow paths within a kernel function
  - N divergent paths in a warp?
    - An N-way divergent warp is serially issued over the N different paths using a hardware stack and per-thread predication logic to only write back results from the threads taking each divergent path.
    - Performance decreases by about a factor of N



First Level of Defense: Avoid Control Flow

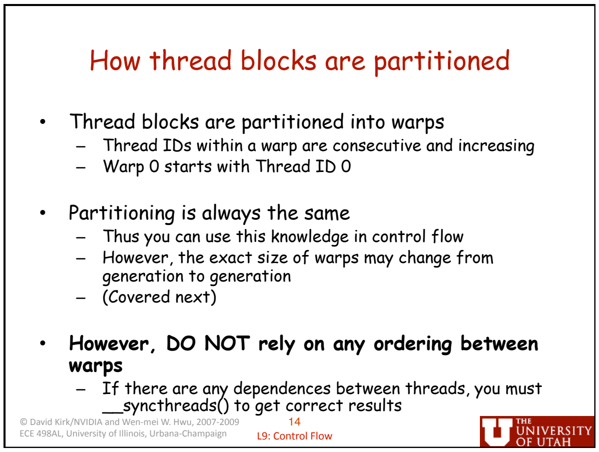
- Clever example from MPM

$$m_i = \sum_p S_{ip} m_p + 1.0x10^{-100}$$

$$\mathbf{V}_i = \frac{\sum_p S_{ip} m_p \mathbf{V}_p}{m_i}$$

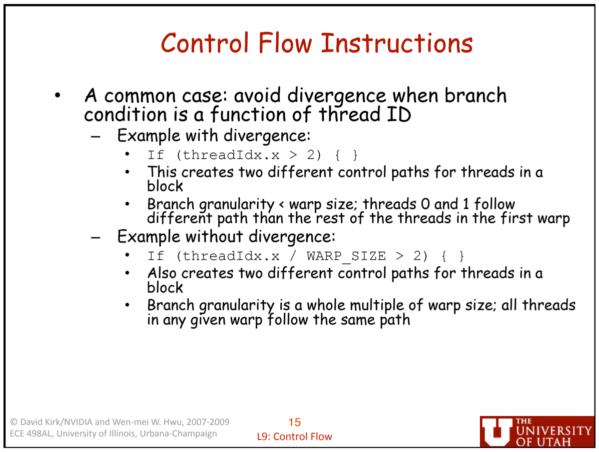
Add small constant to mass so that velocity calculation never divides by zero

- No need to test for divide by 0 error, and slight delta does not impact results



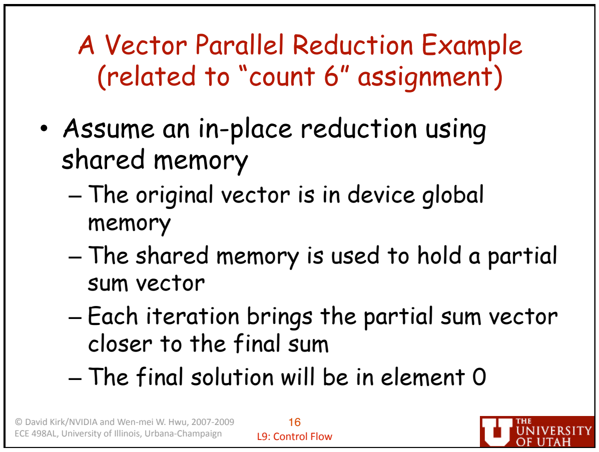
How thread blocks are partitioned

- 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 dependences between threads, you must __syncthreads() to get correct results



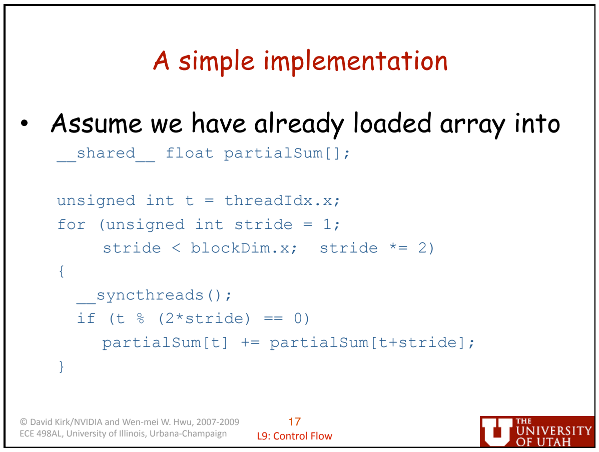
Control Flow Instructions

- 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



A Vector Parallel Reduction Example (related to "count 6" assignment)

- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is 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

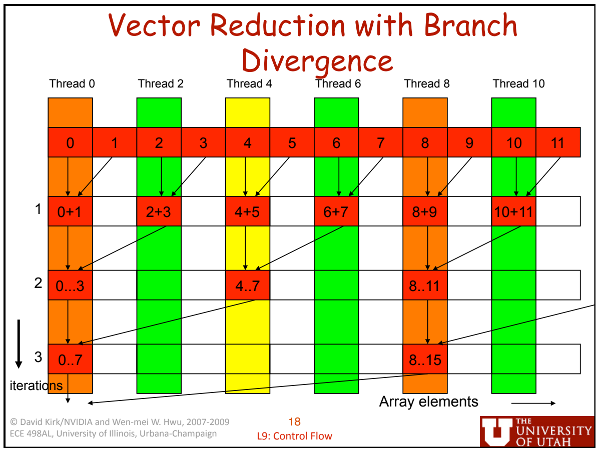


A simple implementation

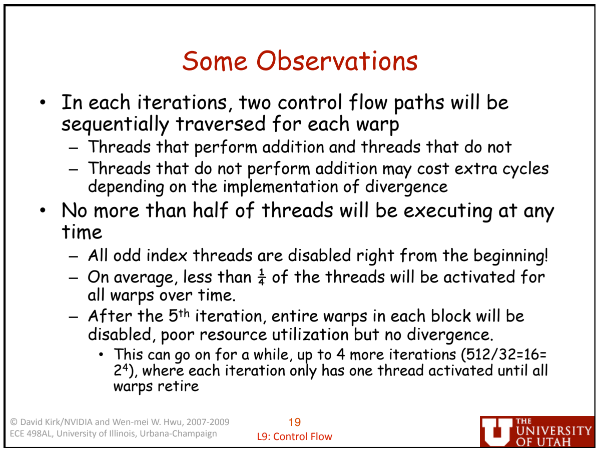
- Assume we have already loaded array into

```
__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];
}
```

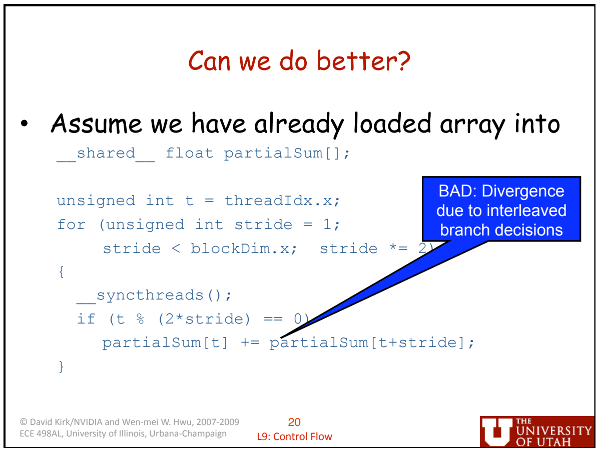


Vector Reduction with Branch Divergence



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 5th 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



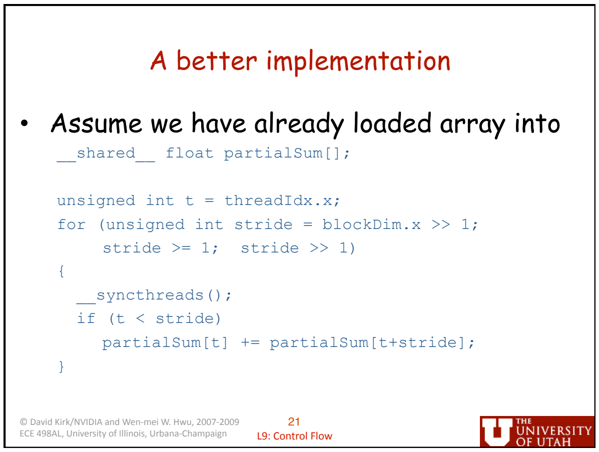
Can we do better?

- 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];
}
```

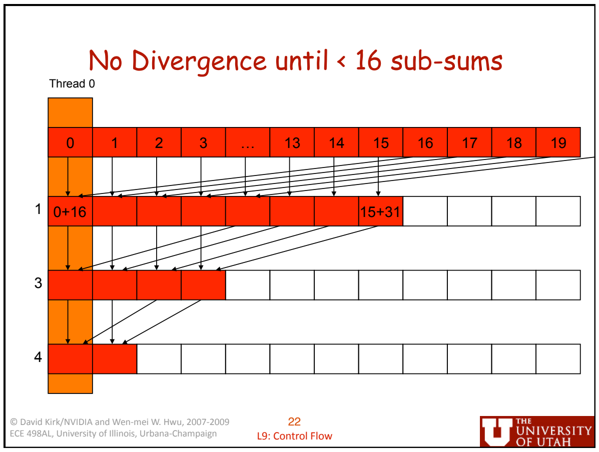
BAD: Divergence due to interleaved branch decisions



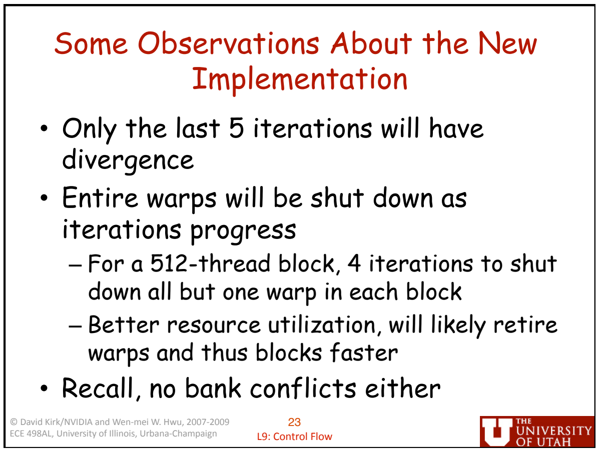
A better implementation

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

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

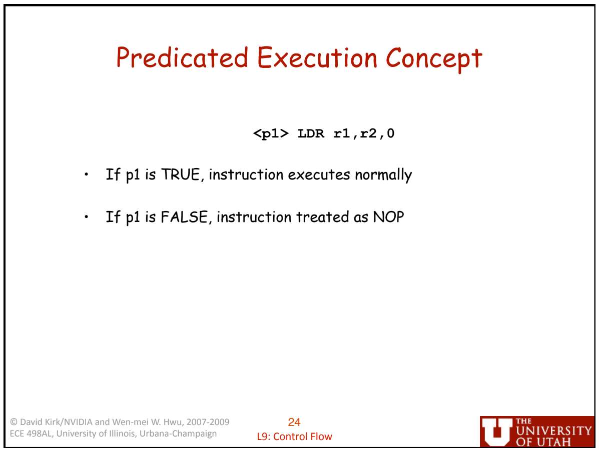


No Divergence until < 16 sub-sums



Some Observations About the New Implementation

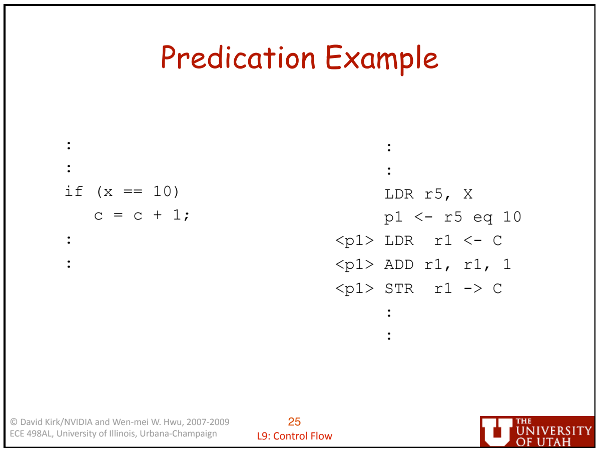
- Only the last 5 iterations will have divergence
- Entire warps will be shut down as iterations progress
  - For a 512-thread block, 4 iterations to shut down all but one warp in each block
  - Better resource utilization, will likely retire warps and thus blocks faster
- Recall, no bank conflicts either



Predicated Execution Concept

`<p1> LDR r1,r2,0`

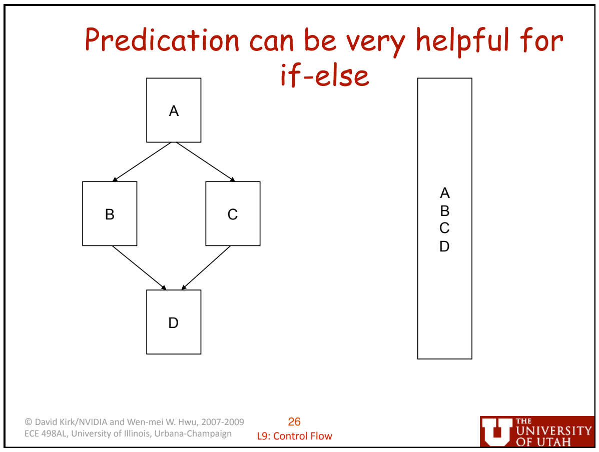
- If p1 is TRUE, instruction executes normally
- If p1 is FALSE, instruction treated as NOP



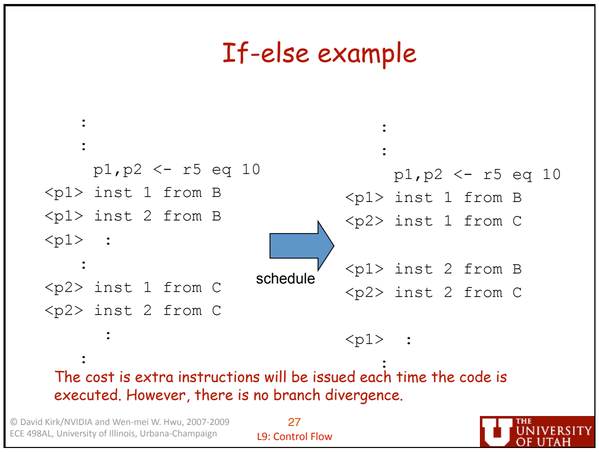
Predication Example

```
:
:
if (x == 10)
   c = c + 1;
:
:
```

```
      :
      :
      LDR r5, X
      p1 <- r5 eq 10
<p1> LDR  r1 <- C
<p1> ADD r1, r1, 1
<p1> STR  r1 -> C
      :
      :
```



Predication can be very helpful for if-else



If-else example

```
    :
    :
     p1,p2 <- r5 eq 10
<p1> inst 1 from B
<p1> inst 2 from B
<p1>   :
    :
<p2> inst 1 from C
<p2> inst 2 from C
      :
    :
```

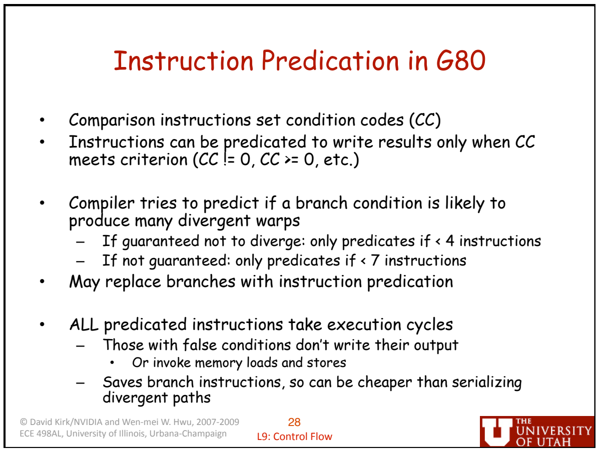
schedule

```
      :
      :
     p1,p2 <- r5 eq 10
<p1> inst 1 from B
<p2> inst 1 from C

<p1> inst 2 from B
<p2> inst 2 from C

<p1>   :
      :
```

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

Warp Vote Functions (Compute Capability > 1.2)

- Can test whether condition on all threads in a warp evaluates to same value

  int __all(int predicate):

  evaluates predicate for all threads of a warp and returns non-zero iff predicate evaluates to non-zero for ***all*** of them.

  int __any(int predicate):

  evaluates predicate for all threads of a warp and returns non-zero iff predicate evaluates to non-zero for ***any*** of them.

Using Warp Vote Functions

- Can tailor code for when none/all take a branch.
- Eliminate overhead of branching and predication.
- Particularly useful for codes where most threads will be the same
  - Example 1: looking for something unusual in image data
  - Example 2: dealing with boundary conditions

Summary of Lecture

- Impact of control flow on performance
  - Due to SIMD execution model for threads
- Strategies for avoiding control flow
  - Eliminate divide by zero test (MPM)
  - Warp vote function
- Group together similar control flow paths into warps
  - Example: "tree" reduction