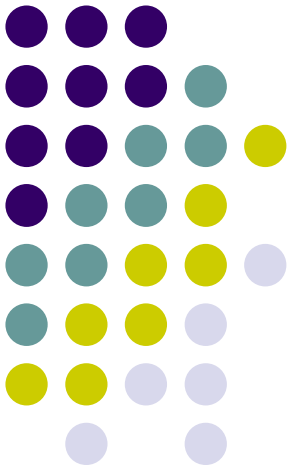
# ME964
# High Performance Computing for Engineering Applications

Control Flow in CUDA

Execution Configuration Optimization

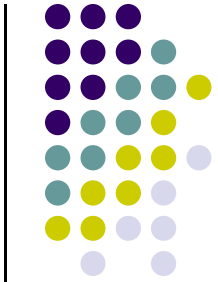
Instruction Optimization

March 3, 2011



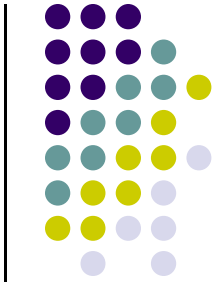
"A computer will do what you tell it to do, but that may be much different from what you had in mind"
Joseph Weizenbaum

# Before We Get Started…



- Last time
  - Discussed Midterm Project topics 3 and 4
    - Finite Element Method on the GPU. Area coordinators: Prof. Suresh and Naresh Khude
    - Sparse direct solver on the GPU (Cholesky). Area coordinator: Dan Negrut
- Today
  - Thread divergence on the GPU
  - Execution Configuration Optimization
  - Instruction Optimization
- Other issues
  - HW6 posted (due 03/22): deals with a parallel prefix scan operation
  - Midterm Projects: Four new discussion threads started, one for each topic
    - You will have to submit a one paragraph document by 11:59 PM today to commit to a project topic
      - Use “MidtermProject” drop-box in Learn@UW
    - My advice: work on Project 3 or 4 only if you want to make it your Final Project
    - Brute force collision detection is the easiest way out

# How thread blocks are partitioned

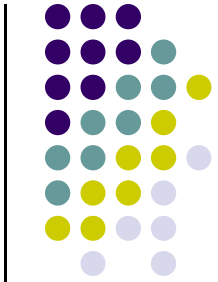


- Each thread block is partitioned into warps
  - Thread indices (indexes?) within a warp are consecutive and increasing
    - Remember: In multidimensional blocks, the x thread index runs first, followed by the y thread index, and finally followed by the z thread index
  - Warp 0 starts with Thread Idx 0

- Partitioning of threads in warps is always the same
  - You can use this knowledge in control flow
  - So far, the warp size of 32 has been kept constant from device to device and CUDA version to CUDA version

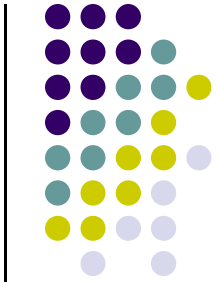
- **While you can rely on ordering among threads, DO NOT rely on any ordering among warps**
  - Remember, the concept of warp is not something you control through CUDA
  - If there are any dependencies between threads, you must __syncthreads() to get correct results

# Control Flow Instructions



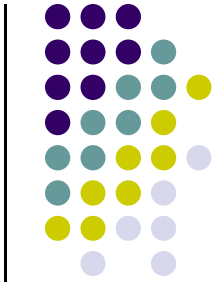
- Main performance concern with branching is *divergence*
  - Threads within a single warp take different paths
  - Different execution paths are serialized
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
  - NOTE: Don't forget that divergence can manifest *only* at the warp level. You can not discuss this concept in relation to code executed by threads in different warps

# Control Flow Instructions (cont.)



- A common case: 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

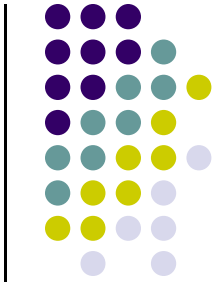
# Control Flow Instructions



- *if*, *switch*, *for*, *while* – can significantly impact the effective instruction throughput when threads of the same warp diverge

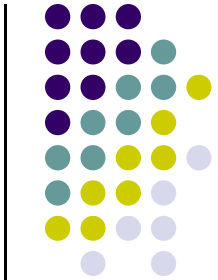
- If this happens, the execution is serialized
  - This increases the number of instructions executed for this warp
  - When all the different execution paths have completed, the threads converge back to the same execution path
  - Not only that you execute more instructions, but you also need logic associated with this process (book-keeping)

# Illustration: Parallel Reduction



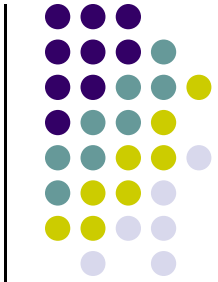
- Parallel reduction is a *very* common problem
  - Given an array of values, “reduce” them in parallel to a single value
- Examples
  - Sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array
  - Min reduction:  minimum of all values in the array
- One example where you can run into it:
  - Find the infinity norm of a very large array – used as a convergence test for a iterative solver, for instance
- Typically parallel implementation:
  - Recursively halve the number of threads, deal with 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
  - We are in the process of summing up a 512 element array
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final sum will be stored in element 0

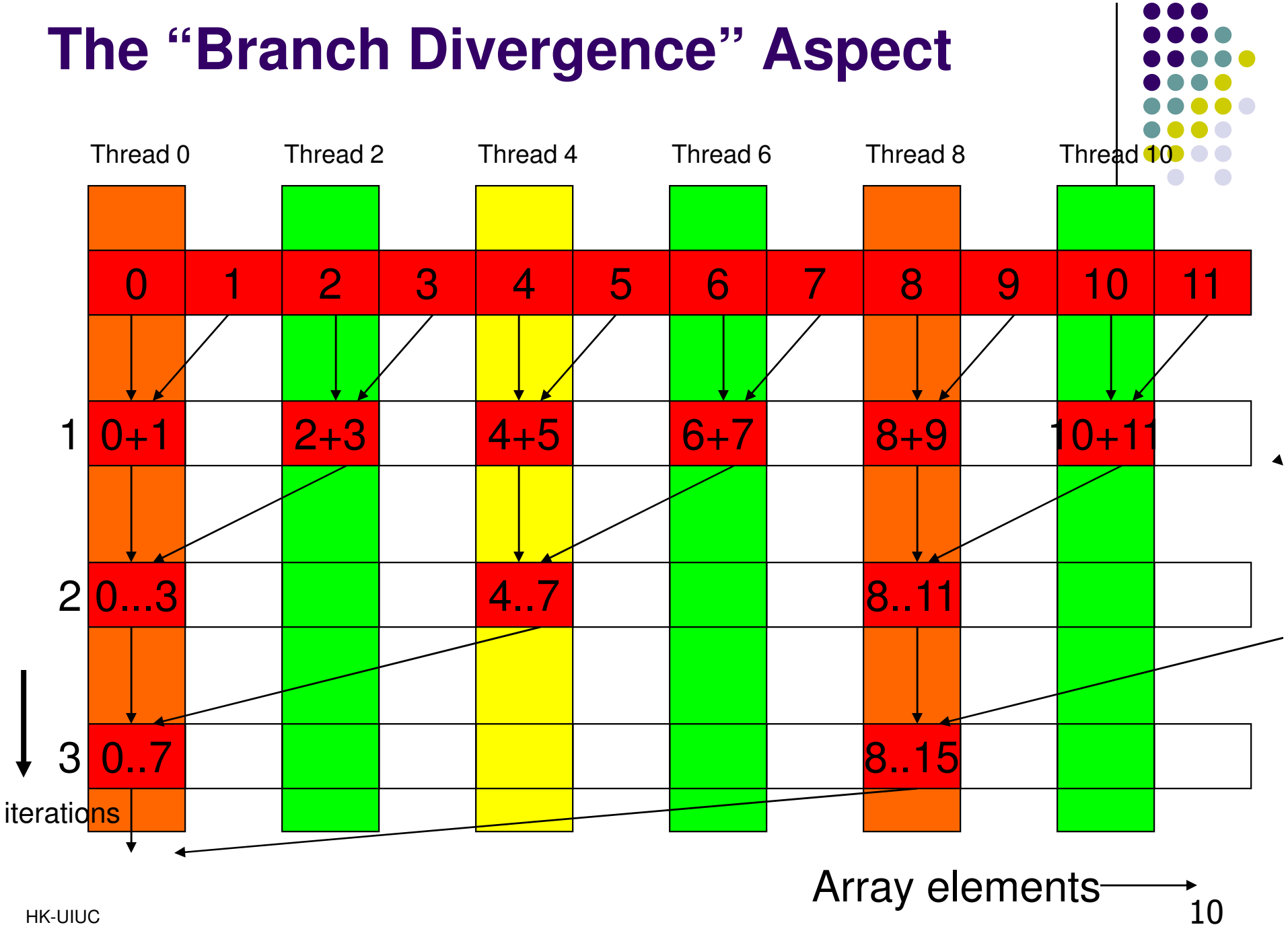
# A simple implementation



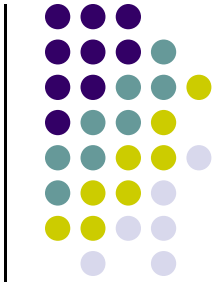
- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```
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];
}
```

# The "Branch Divergence" Aspect

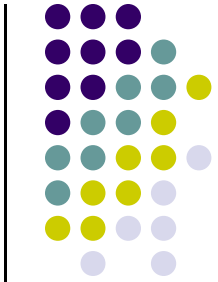


# Some Observations



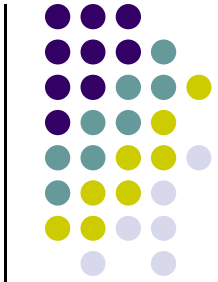
- In each iterations, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence

# Some Observations (cont.)



- No more than half of the threads will be executing at any time
  - All odd index threads are disabled right from the beginning!
  - On average, less than ¼ of the threads will be activated for all warps over time.
  - After the 5$^{th}$ 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

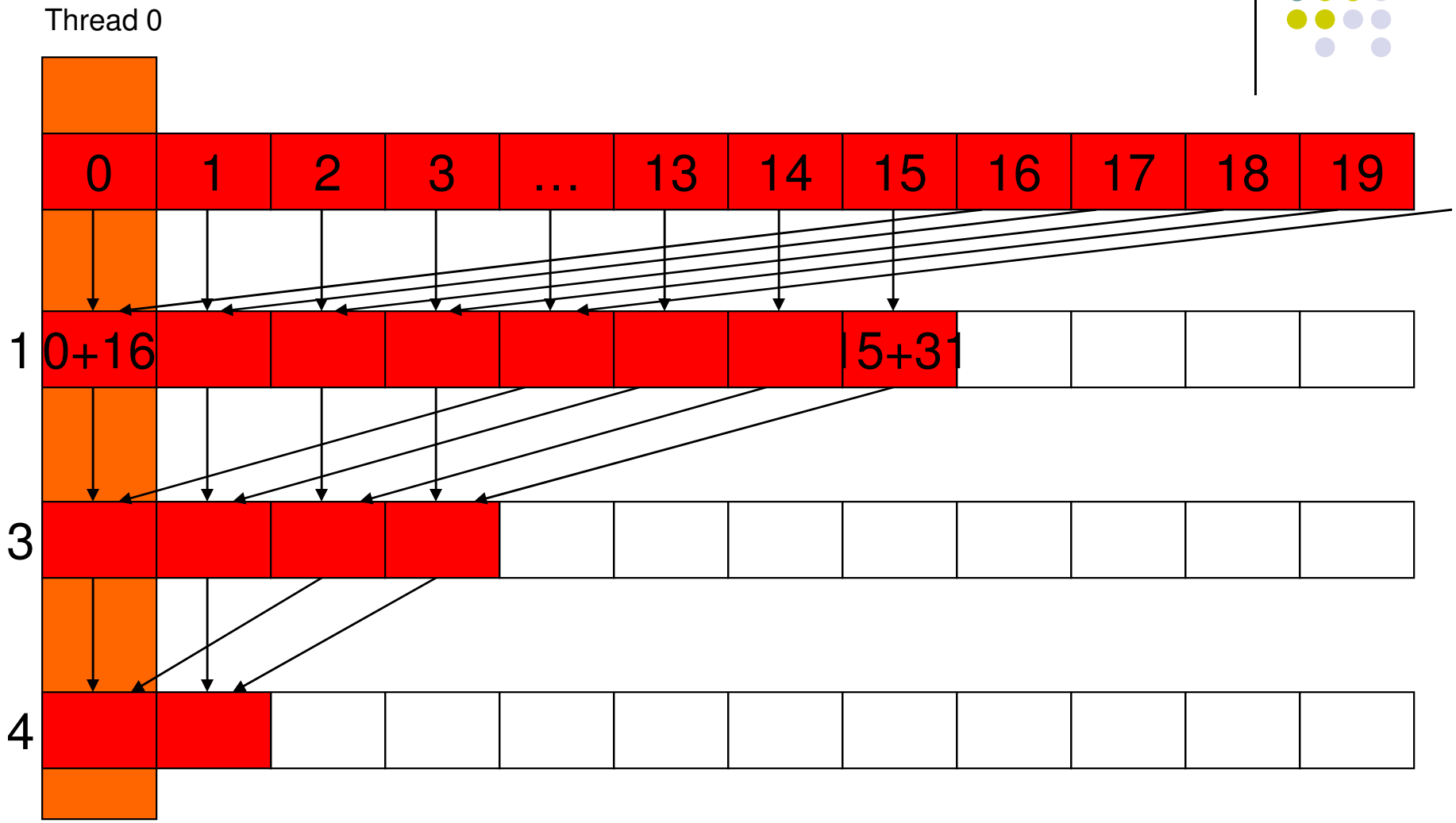
# A Better Implementation



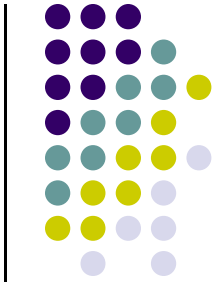
- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

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

# No Divergence until < 16 sub-sums

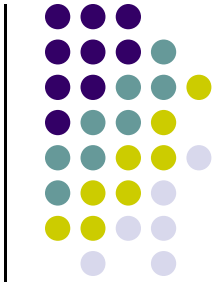


# Some Observations About the New Implementation



- 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 the block. Here's how the thread count shapes up:
    - 512 (1st iteration), 256 (2nd iteration), 128 (3rd iteration), 64 (4th iteration)
  - Better resource utilization, will likely retire warps and thus block executes faster
- Recall, no bank conflicts either

# Predicated Execution Concept
## [Looking Under the Hood]

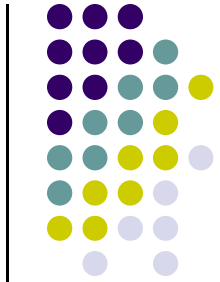


- The thread divergence can be avoided in some cases by using the concept of predication

```
<p1> LDR r1,r2,0
```

- If p1 is TRUE, the assembly code instruction above executes normally
- If p1 is FALSE, instruction treated as NOP
  - NOP – “no operation”

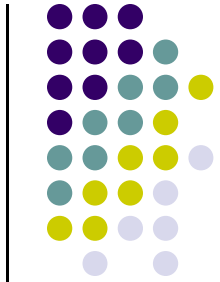
# Predication Example

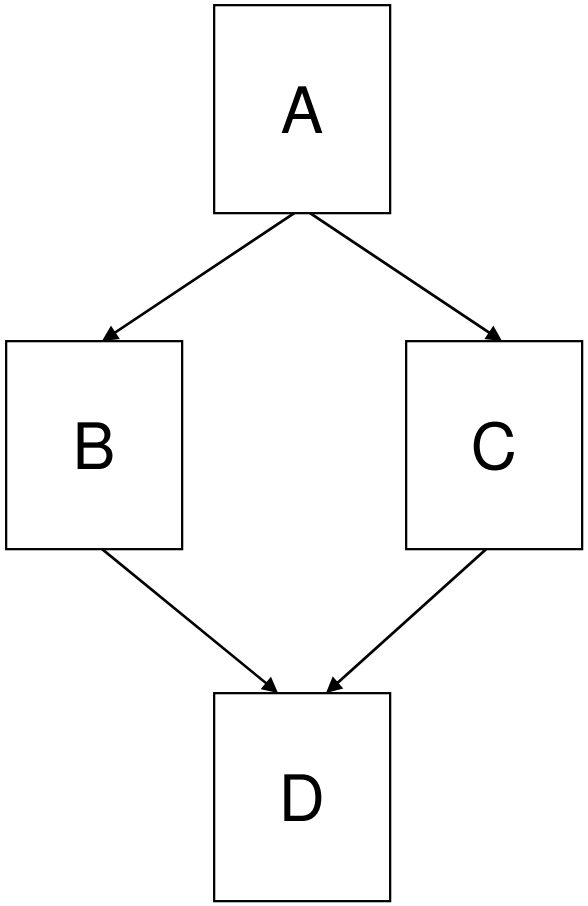


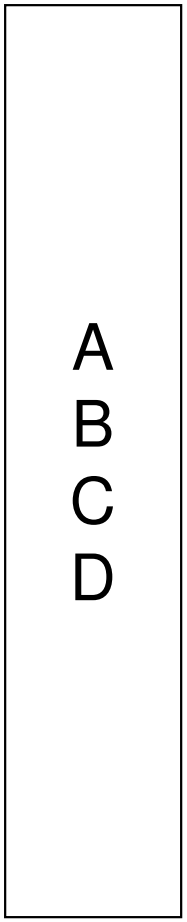
```
:
:
if (x == 10)
   c = c + 1;
:
:
```

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

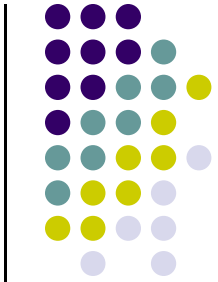
# Predication Helpful for If-Else



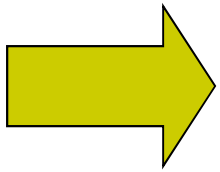


# If-else example



```
      :
      :
     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
      :
      :
```



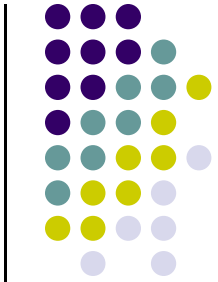
This is what gets scheduled

```
      :
      :
     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.

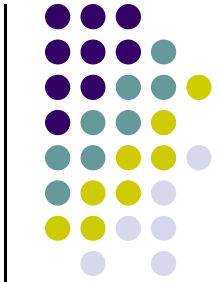
# Instruction Predication

## [Tesla C1060]



- A comparison instructions sets a condition code (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 that's the case, go ahead and predicate if the branch has <7 instructions
  - If that's not the case, only predicate if the branch has <4 instructions
  - Note: it's pretty bad if you predicate when it was obvious that there would have been no divergence

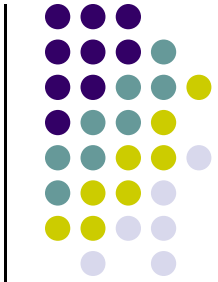
# Instruction Predication
## [Contd.]



- ALL predicated instructions take execution cycles
  - Those with false conditions don't write their output, and do not evaluate addresses or read operands
  - Saves branch instructions, so can be cheaper than serializing divergent paths

- If all this business is confusing, remember this:
  - Avoid thread divergence

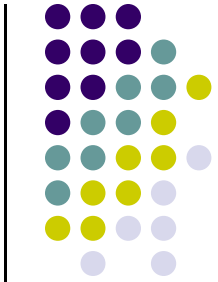
- It's not 100% clear to me, but I believe that there is no cost if a subset of threads belonging to a warp sits there and does nothing while the other warp threads are all running the same instruction



## End: Control Flow in CUDA

## Begin: Execution Configuration Optimization

# Occupancy



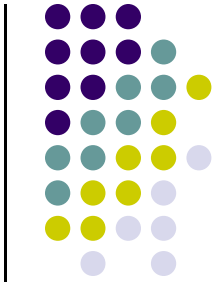
- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy

- Occupancy = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently
  - Can have 32 warps on Tesla C1060

- Limited by resource usage:
  - Registers
  - Shared memory

Source: NVIDIA

# Blocks per Grid Heuristics

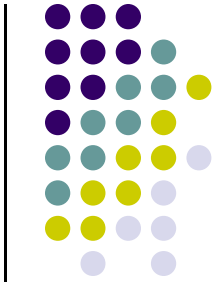


- # of blocks > # of multiprocessors
  - So all multiprocessors have at least one block to execute
- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren't waiting at a `__syncthreads()` keep the hardware busy
  - Subject to resource availability – registers, shared memory
- # of blocks > 100 to scale to future devices
  - Blocks executed in pipeline fashion
  - 1000's of blocks per grid will scale across multiple generations



Source: NVIDIA

# Register Dependency



- Read-after-write register dependency
  - Instruction's result can be read ~24 cycles later
  - Scenarios: CUDA: PTX (Parallel Thread eXecution ISA):

```
x = y + 5;
z = x + 3;
```

```
add.f32   $f3, $f1, $f2
add.f32   $f5, $f3, $f4
```

```
s_data[0] += 3;
```

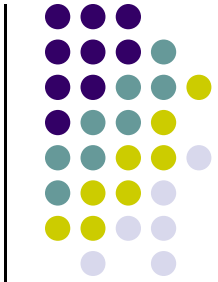
```
ld.shared.f32  $f3, [$r31+0]
add.f32        $f3, $f3, $f4
```

- To completely hide the latency:
  - Run at least 192 threads (6 warps) per multiprocessor
    - At least 25% occupancy (1.0/1.1), 18.75% (1.2/1.3)
  - Threads do not have to belong to the same thread block



Source: NVIDIA

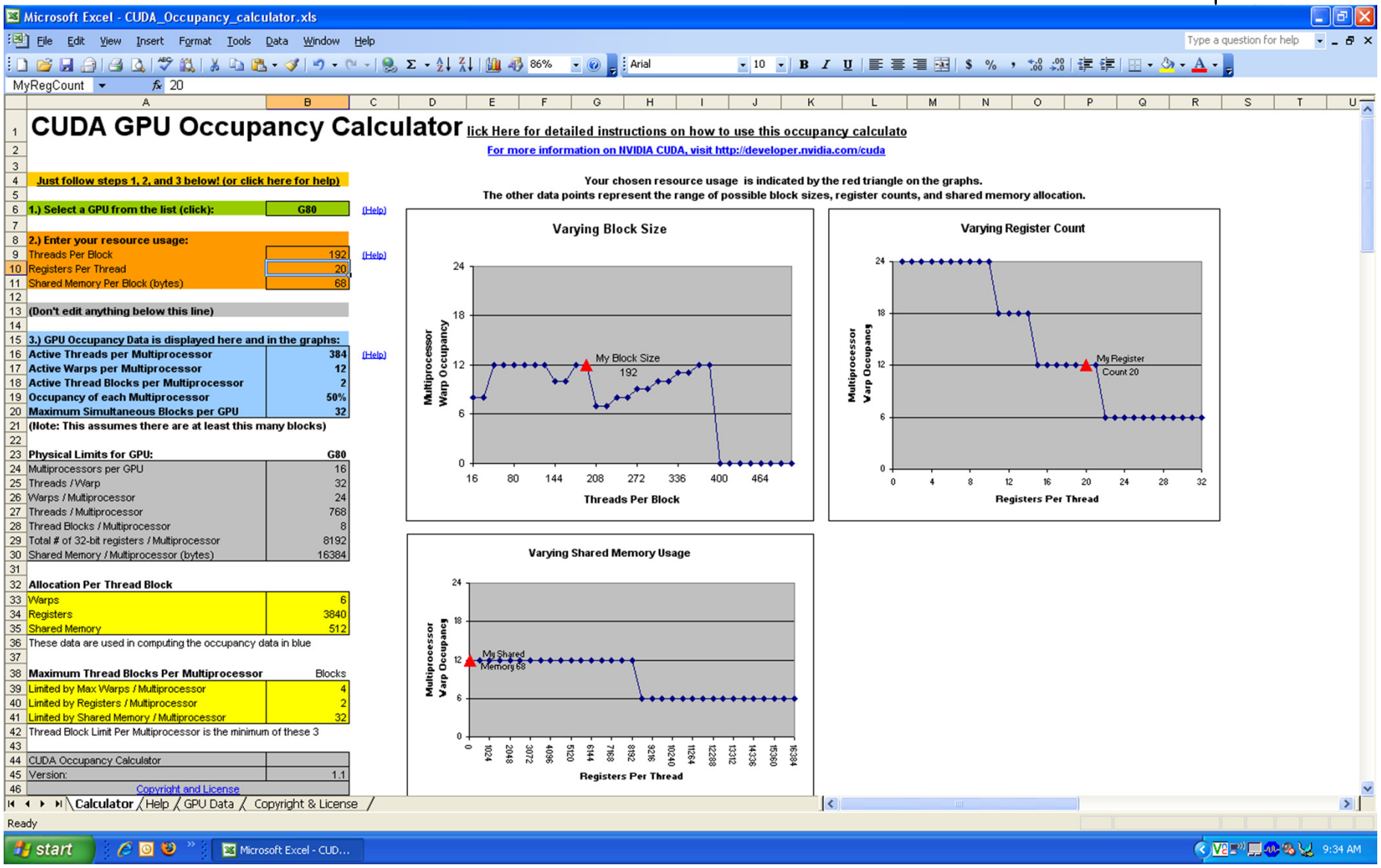
# Register Pressure



- Hide latency by using more threads per SM
- Limiting Factors:
  - Number of registers per kernel
    - 8K/16K per multiprocessor, partitioned among concurrent threads
  - Amount of shared memory
    - 16KB per multiprocessor, partitioned among concurrent threadblocks
- Compile with `-ptxas-options=-v` flag
- Use `-maxrregcount=N` flag to NVCC
  - N = desired maximum registers / kernel
  - At some point "spilling" into local memory may occur
    - Reduces performance – local memory is slow

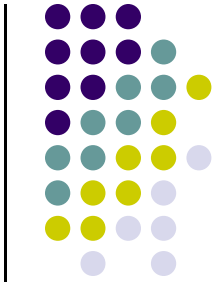
Source: NVIDIA

# Occupancy Calculator




Source: NVIDIA

# Optimizing Threads Per Block



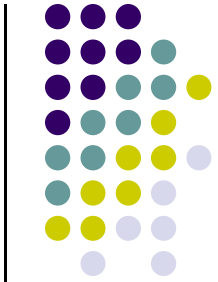
- Choose threads per block as a multiple of warp size
  - Avoid wasting computation on under-populated warps
  - Facilitates coalescing
- Want to run as many warps as possible per multiprocessor (hide latency)
- Multiprocessor can run up to 8 blocks at a time

- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads a better choice
    - Usually still enough regs to compile and invoke successfully
  - This all depends on your computation, so experiment!



Source: NVIDIA

# Occupancy != Performance



- Increasing occupancy does not necessarily increase performance

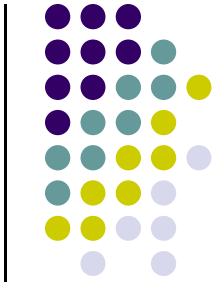
*BUT …*

- Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels
  - **In the end, it all comes down to arithmetic intensity and available parallelism**



Source: NVIDIA

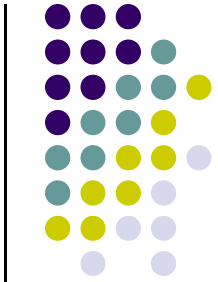
# Parameterize Your Application



- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
  - # of SMs (stream multiprocessors)
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Max. threads per block
- You can even make apps self-tuning (like FFTW and ATLAS)
  - "Experiment" mode discovers and saves optimal configuration



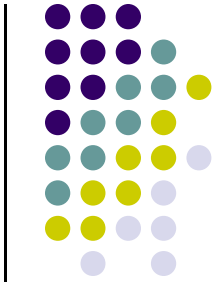
Source: NVIDIA



**End: Execution Configuration Optimization**

**Begin: Instruction Optimizations**

# CUDA Instruction Performance

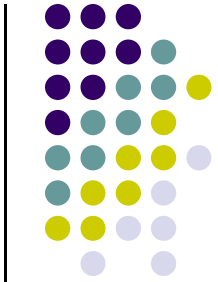


- Instruction cycles (per warp) = sum of
  - Operand read cycles
  - Instruction execution cycles
  - Result update cycles
- Therefore instruction throughput depends on
  - Nominal instruction throughput
  - Memory latency
  - Memory bandwidth
- "Cycle" refers to the multiprocessor clock rate
  - 1.3 GHz on the Tesla C1060, for example



Source: NVIDIA

# Maximizing Instruction Throughput

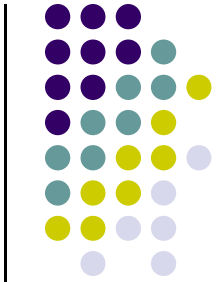


- Maximize use of high-bandwidth memory
  - Maximize use of shared memory
  - Minimize accesses to global memory
  - Maximize coalescing of global memory accesses
- Optimize performance by overlapping memory accesses with HW computation
  - You need many warps running on the SM to this end…
  - Another thing that's helpful: high arithmetic intensity programs
    - i.e. high ratio of math to memory transactions



Source: NVIDIA

# Arithmetic Instruction Throughput



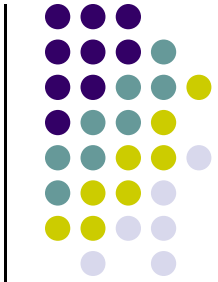
- int and float add, shift, min, max and float mul, mad: 4 cycles per warp
  - int multiply (*) is by default 32-bit
    - requires multiple cycles / warp
  - Use `__mul24()/__umul24()` intrinsics for 4-cycle 24-bit int multiply

- Integer divide and modulo are more expensive
  - Compiler will convert literal power-of-2 divides to shifts
    - But it has been documented to miss some cases
  - Be explicit in cases where compiler can't tell that divisor is a power of 2!
  - Useful trick: `foo%n==foo&(n-1)` if `n` is a power of 2



Source: NVIDIA

# Runtime Math Library

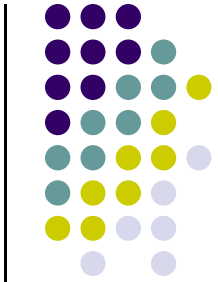


- There are two types of runtime math operations in single precision
  - `__funcf()`: direct mapping to hardware ISA
    - Fast but lower accuracy (see prog. guide for details)
    - Examples: `__sinf(x), __expf(x), __powf(x,y)`
  - `funcf()` : compile to multiple instructions
    - Slower but higher accuracy (5 ulp or less)
    - Examples: `sinf(x), expf(x), powf(x,y)`
- The `-use_fast_math` compiler option forces every `funcf()` to compile to `__funcf()`



Source: NVIDIA

# GPU Results May Not Match CPU

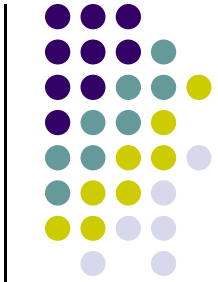


- Many variables: hardware, compiler, optimization settings
- CPU operations aren't strictly limited to 0.5 ulp
  - Sequences of operations can be more accurate due to 80-bit extended precision ALUs
  - ULP: "Unit in the Last Place" is the spacing between floating-point numbers, i.e., the value that the least significant bit (lsb) represents if it is 1. It is used as *a measure of precision* in numeric calculations
- Floating-point arithmetic is not associative!



Source: NVIDIA

# FP Math is Not Associative!

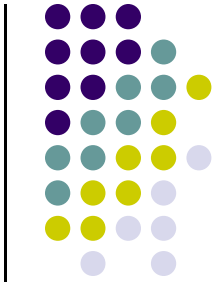


- In symbolic math, (x+y)+z == x+(y+z)
- This is not necessarily true for floating-point addition
  - Try $x = 10^{30}$, $y = -10^{30}$ and $z = 1$ in the above equation
- When you parallelize computations, you potentially change the order of operations
- Parallel results may not exactly match sequential results
  - This is not specific to GPU or CUDA – inherent part of parallel execution



Source: NVIDIA

# Control Flow Instructions



- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths must be serialized
- Avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `if (threadIdx.x > 2) { }`
    - Branch granularity < warp size
  - Example without divergence:
    - `if (threadIdx.x / WARP_SIZE > 2) { }`
    - Branch granularity is a whole multiple of warp size



Source: NVIDIA