CS 677: Parallel Programming for Many-core Processors
Lecture 4

Instructor: Philippos Mordohai

Webpage: www.cs.stevens.edu/~mordohai

E-mail: Philippos.Mordohai@stevens.edu

Project Proposal

- Problem description
  - What is the computation and why is it important?
  - Abstraction of computation: equations, graphic or pseudo-code, no more than 1 page
- Suitability for GPU acceleration
  - Amdahl’s Law: describe the inherent parallelism. Argue that it is close to 100% of computation.
  - Synchronization and Communication: Discuss what data structures may need to be protected by synchronization, or communication through host.
  - Copy Overhead: Discuss the data footprint and anticipated cost of copying to/from host memory.
- Intellectual Challenges
  - Generally, what makes this computation worthy of a project?
  - Point to any difficulties you anticipate at present in achieving high speedup

Amdahl’s Law

- “The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.”

- Example
  - 95% of original execution time can be sped up by 100x on GPU
  - Speed up for entire application:

$$\frac{1}{(5\% + \frac{95\%}{100})} = \frac{1}{5\% + 0.95\%} = \frac{1}{5.95\%} = 17x$$

Overview

- More Performance Considerations
  - Memory Coalescing
  - Occupancy
  - Kernel Launch Overhead
  - Instruction Performance
- Summary of Performance Considerations
  - Lectures 3 and 4

- Parallel Patterns: Reduction Trees
- Parallel Patterns: Parallel Prefix Sum (Scan)

Memory Coalescing (Part 2)

slides by

Jared Hoberock and David Tarjan

(Stanford CS 193G)

Consider the stride of your accesses

```
__global__ void foo(int* input,
                    float3* input2)
{
  int i = blockDim.x * blockIdx.x
        + threadIdx.x;
  // Stride 1
  int a = input[i];
  // Stride 2, half the bandwidth is wasted
  int b = input[2*i];
  // Stride 3, 2/3 of the bandwidth wasted
  float c = input2[i].x;
}
```

Example: Array of Structures (AoS)

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

record  *d_records;
cudaMalloc((void**)&d_records,
  ...);
```

Example: Structure of Arrays (SoA)

```
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, ...);
```

Example: SoA vs. AoS

```
__global__ void bar(record
  *AoS_data, SoA SoA_data)
{
  int i = blockDim.x * blockIdx.x
          + threadIdx.x;
  // AoS wastes bandwidth
  int key = AoS_data[i].key;
  // SoA efficient use of bandwidth
  int key_better = SoA_data.keys[i];
}
```

Memory Coalescing

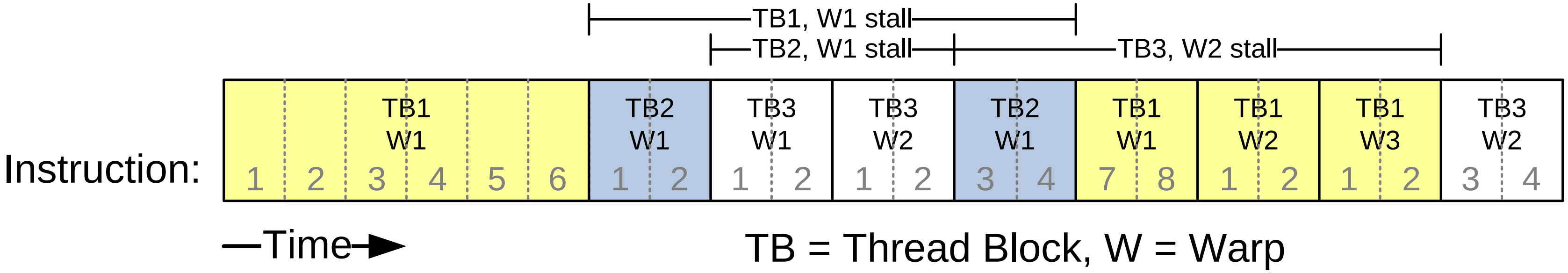
- Structure of arrays is often better than array of structures
  - Very clear win on regular, stride 1 access patterns
  - Unpredictable or irregular access patterns are case-by-case

Occupancy

slides (mostly) by
Jared Hoberock and David Tarjan
(Stanford CS 193G)
and Joseph T. Kider Jr. (UPenn)

Reminder: Thread Scheduling

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM
  - 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



TB = Thread Block, W = Warp

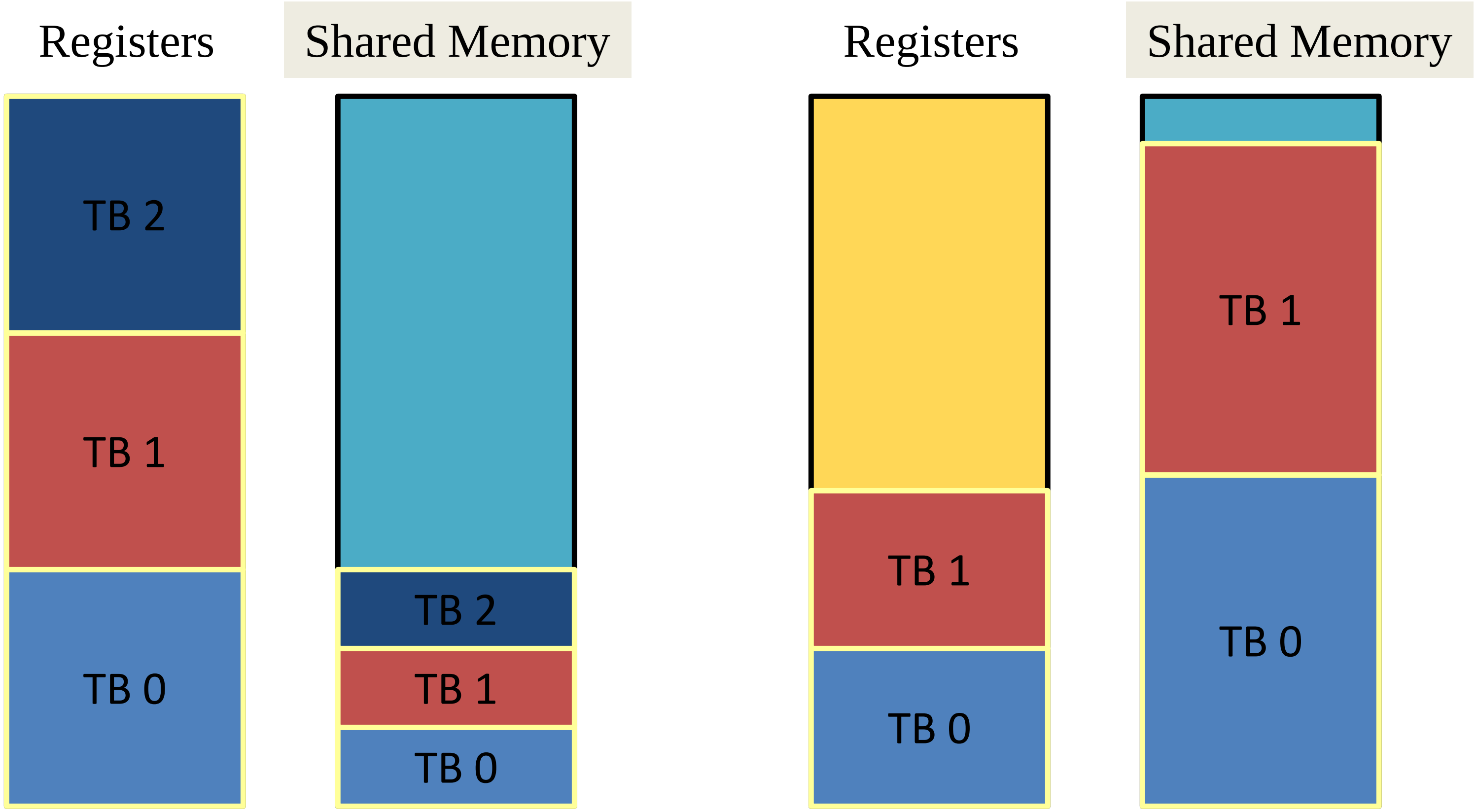
Thread Scheduling

- 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

- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep cores busy
- Occupancy = number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently
- Limited by resource usage:
  - Registers
  - Shared memory

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 N thread blocks per SM
  - If they're too small, can't fill up the SM
  - Need 128 threads / block (GT200), 192 threads/ block (GF100)

- Higher occupancy has diminishing returns for hiding latency

Grid/Block Size 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 on a multiprocessor
  - Blocks not waiting at a __syncthreads() keep hardware busy
  - Subject to resource availability – registers, shared memory
- # of blocks > 100 to scale to future devices

Register Dependency

- Read-after-write register dependency
  - Instruction’s result can be read approximately 24 cycles later
- To completely hide latency:
  - Run at least 192 threads (6 warps) per multiprocessor
    - At least 25% occupancy for compute capability 1.0 and 1.1
    - Threads do not have to belong to the same block

Register Pressure

- Hide latency by using more threads per SM
- Limiting factors:
  - Number of registers per thread
    - 8k/16k/… per SM, partitioned among concurrent threads
  - Amount of shared memory
    - 16kB/… per SM, partitioned among concurrent blocks

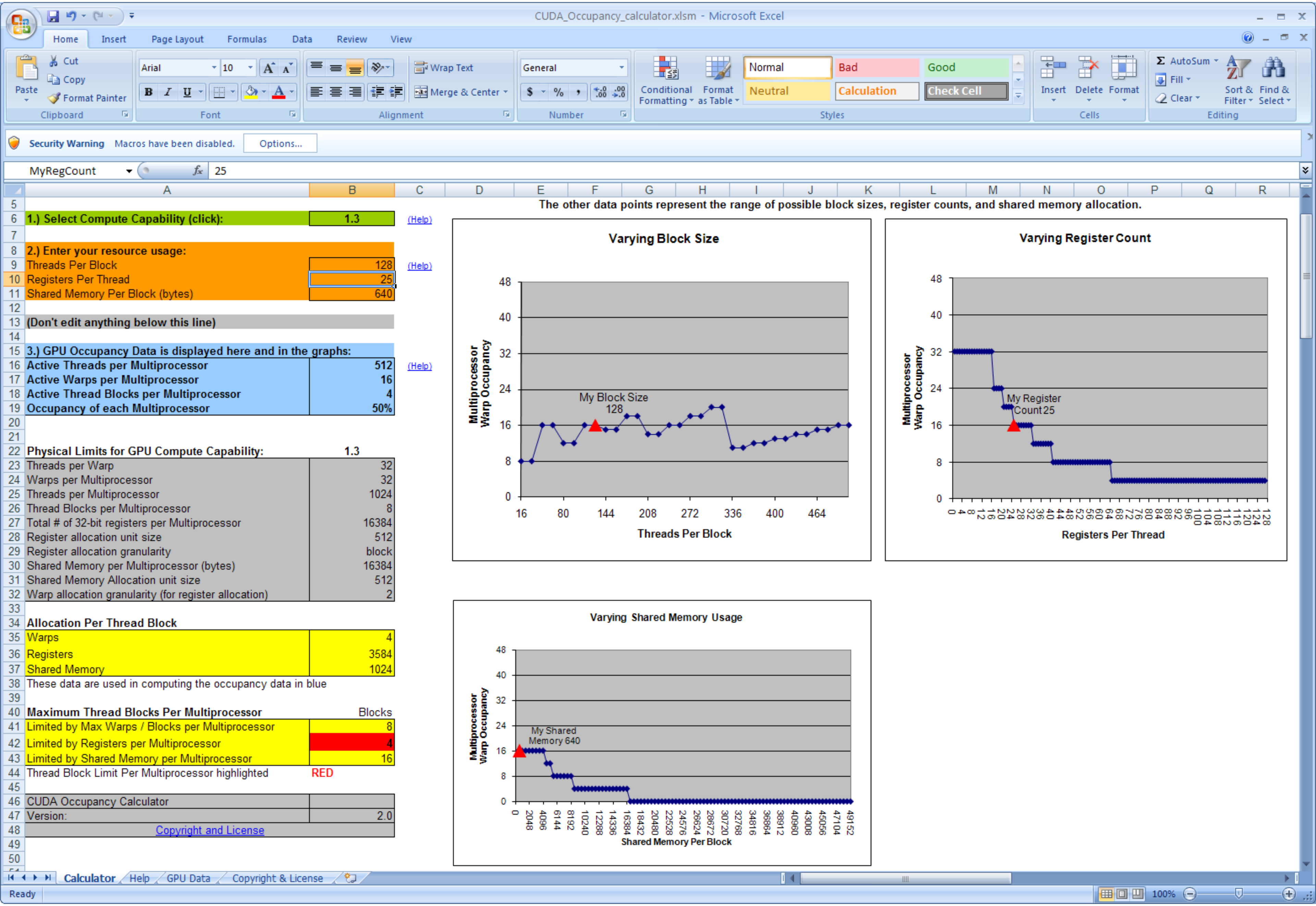
How do you know what you’re using?

- Use `nvcc -Xptxas -v` to get register and shared memory usage

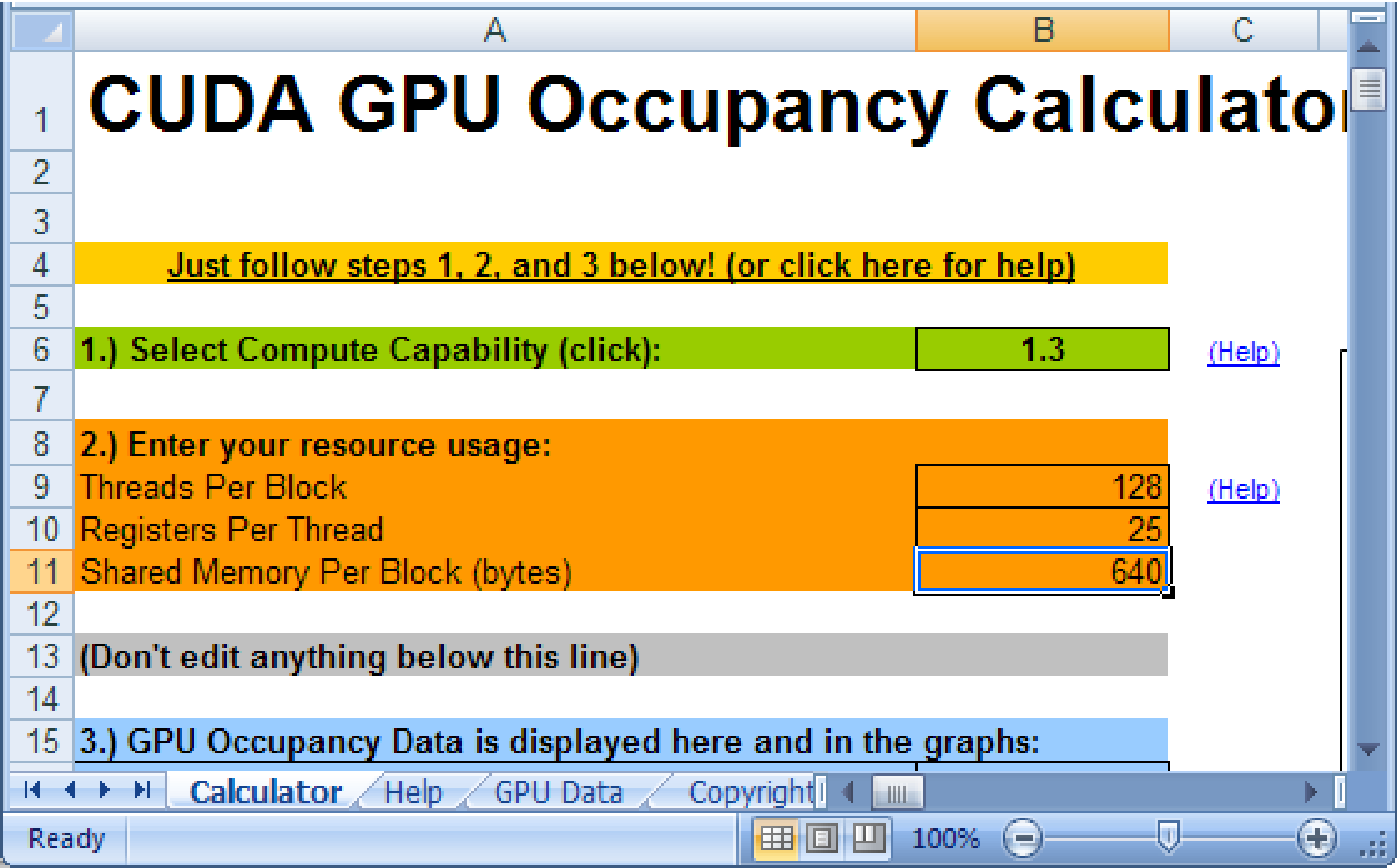
```
nvcc -Xptxas -v acos.cu
ptxas info : Compiling entry function 'acos_main'
ptxas info : Used 4 registers, 60+56 bytes lmem, 44+40 bytes
              smem, 20 bytes cmem[1], 12 bytes cmem[14]
```

  - The first number represents the total size of all the variables declared in that memory segment and the second number represents the amount of system allocated data.
  - Constant memory numbers include which memory banks have been used

- Plug those numbers into CUDA Occupancy Calculator



	A	B	C
5	The other data points represent the range of possible block sizes, register counts, and shared memory allocation.		
6	**1.) Select Compute Capability (click):**	**1.3**	(Help)
7			
8	**2.) Enter your resource usage:**		
9	Threads Per Block	128	(Help)
10	Registers Per Thread	25	
11	Shared Memory Per Block (bytes)	640	
12			
13	**(Don't edit anything below this line)**		
14			
15	**3.) GPU Occupancy Data is displayed here and in the graphs:**		
16	**Active Threads per Multiprocessor**	512	(Help)
17	**Active Warps per Multiprocessor**	16	
18	**Active Thread Blocks per Multiprocessor**	4	
19	**Occupancy of each Multiprocessor**	50%	
20			
21			
22	**Physical Limits for GPU Compute Capability:**	1.3	
23	Threads per Warp	32	
24	Warps per Multiprocessor	32	
25	Threads per Multiprocessor	1024	
26	Thread Blocks per Multiprocessor	8	
27	Total # of 32-bit registers per Multiprocessor	16384	
28	Register allocation unit size	512	
29	Register allocation granularity	block	
30	Shared Memory per Multiprocessor (bytes)	16384	
31	Shared Memory Allocation unit size	512	
32	Warp allocation granularity (for register allocation)	2	
33			
34	**Allocation Per Thread Block**		
35	Warps	4	
36	Registers	3584	
37	Shared Memory	1024	
38	These data are used in computing the occupancy data in blue		
39			
40	**Maximum Thread Blocks Per Multiprocessor**	Blocks	
41	Limited by Max Warps / Blocks per Multiprocessor	8	
42	Limited by Registers per Multiprocessor	4	
43	Limited by Shared Memory per Multiprocessor	16	
44	Thread Block Limit Per Multiprocessor highlighted	RED	
45			
46	CUDA Occupancy Calculator		
47	Version:	2.0	
48	Copyright and License		
49			
50			



CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

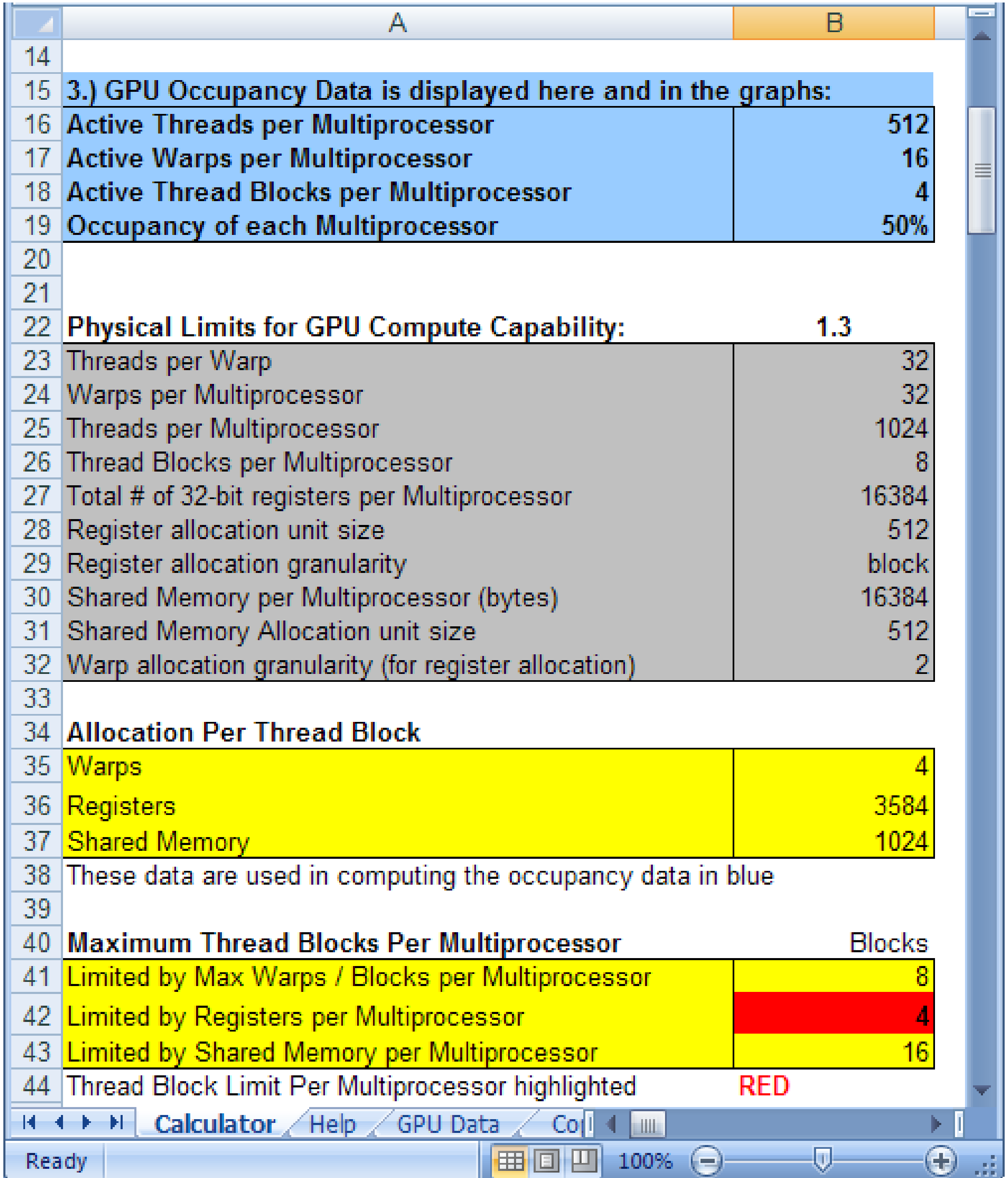
1.) Select Compute Capability (click): 1.3 (Help)

2.) Enter your resource usage:

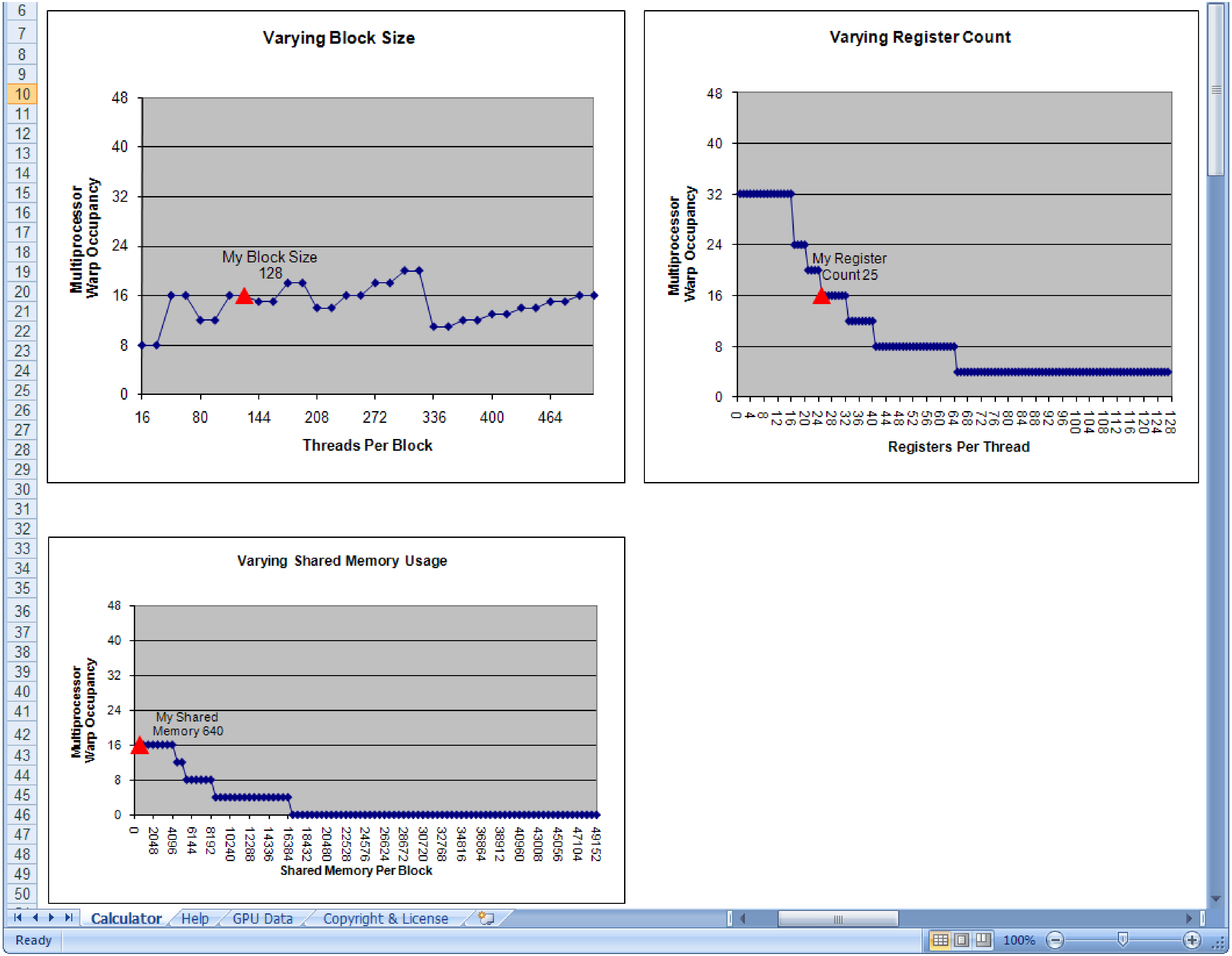
Threads Per Block	128	(Help)
Registers Per Thread	25	
Shared Memory Per Block (bytes)	640	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:



	A	B
14		
15	**3.) GPU Occupancy Data is displayed here and in the graphs:**	
16	**Active Threads per Multiprocessor**	**512**
17	**Active Warps per Multiprocessor**	**16**
18	**Active Thread Blocks per Multiprocessor**	**4**
19	**Occupancy of each Multiprocessor**	**50%**
20		
21		
22	**Physical Limits for GPU Compute Capability:**	**1.3**
23	Threads per Warp	32
24	Warps per Multiprocessor	32
25	Threads per Multiprocessor	1024
26	Thread Blocks per Multiprocessor	8
27	Total # of 32-bit registers per Multiprocessor	16384
28	Register allocation unit size	512
29	Register allocation granularity	block
30	Shared Memory per Multiprocessor (bytes)	16384
31	Shared Memory Allocation unit size	512
32	Warp allocation granularity (for register allocation)	2
33		
34	**Allocation Per Thread Block**	
35	Warps	4
36	Registers	3584
37	Shared Memory	1024
38	These data are used in computing the occupancy data in blue	
39		
40	**Maximum Thread Blocks Per Multiprocessor**	Blocks
41	Limited by Max Warps / Blocks per Multiprocessor	8
42	Limited by Registers per Multiprocessor	**4**
43	Limited by Shared Memory per Multiprocessor	16
44	Thread Block Limit Per Multiprocessor highlighted	RED


6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
Varying Block Size
Multiprocessor Warp Occupancy
48
40
32
24
16
8
0
My Block Size 128
16
80
144
208
272
336
400
464
Threads Per Block
Varying Register Count
Multiprocessor Warp Occupancy
48
40
32
24
16
8
0
My Register Count 25
0
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
68
72
76
80
84
88
92
96
100
104
108
112
116
120
124
128
Registers Per Thread
Varying Shared Memory Usage
Multiprocessor Warp Occupancy
48
40
32
24
16
8
0
My Shared Memory 640
0
2048
4096
6144
8192
10240
12288
14336
16384
18432
20480
22528
24576
26624
28672
30720
32768
34816
36864
38912
40960
43008
45056
47104
49152
Shared Memory Per Block
Calculator
Help
GPU Data
Copyright & License
Ready
100%

How to influence how many registers you use

- Pass option `-maxrregcount=X` to nvcc

- This isn't magic, won't get occupancy for free

- Use this very carefully when you are right on the edge

Optimizing Threads per Block

- Choose threads per block as multiple of warp size
  - Avoid wasting computation on under-populated warps
- Run as many warps as possible per SM
  - Hide latency
- SMs can run up to N blocks at a time

Occupancy != Performance

- Increasing occupancy does not necessarily increase performance

- BUT…

- Low-occupancy SMs cannot adequately hide latency

Parameterize your Application

- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
  - # of SMs
  - Memory bandwidth
  - Shared memory size
  - Register file size
  - Max threads per block
- ➢ Avoid local minima
  - Try widely varying configurations

Kernel Launch Overhead

slides by
Jared Hoberock and David Tarjan
(Stanford CS 193G)

Kernel Launch Overhead

- Kernel launches aren’t free
  - A null kernel launch will take non-trivial time
  - Actual time changes with HW generations and driver software…
- Independent kernel launches are cheaper than dependent kernel launches
  - Dependent launch: Some readback to the CPU
- Launching lots of small grids comes with substantial performance loss

Kernel Launch Overheads

- If you are reading back data to the CPU for control decisions, consider doing it on the GPU
- Even though the GPU is slow at serial tasks, it can do surprising amounts of work before you used up kernel launch overhead

Instruction Performance

slides by

Joseph T. Kider Jr. (Upenn)

Instruction Performance

- Instruction cycles (per warp) is the 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

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 computation
  - High arithmetic intensity programs
  - Many concurrent threads

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() and __umul24() intrinsics for 4-cycle 24-bit int multiplication
- Integer division and modulo operations are costly
  - The compiler will convert literal power-of-2 divides to shifts
    - But it may miss
  - Be explicit in cases where the compiler cannot tell that the divisor is a power of 2
    - Trick: foo % n == foo & (n-1) if n is a power of 2

Loop Transformations

Mary Hall
CS6963 University of Utah

Reordering Transformations

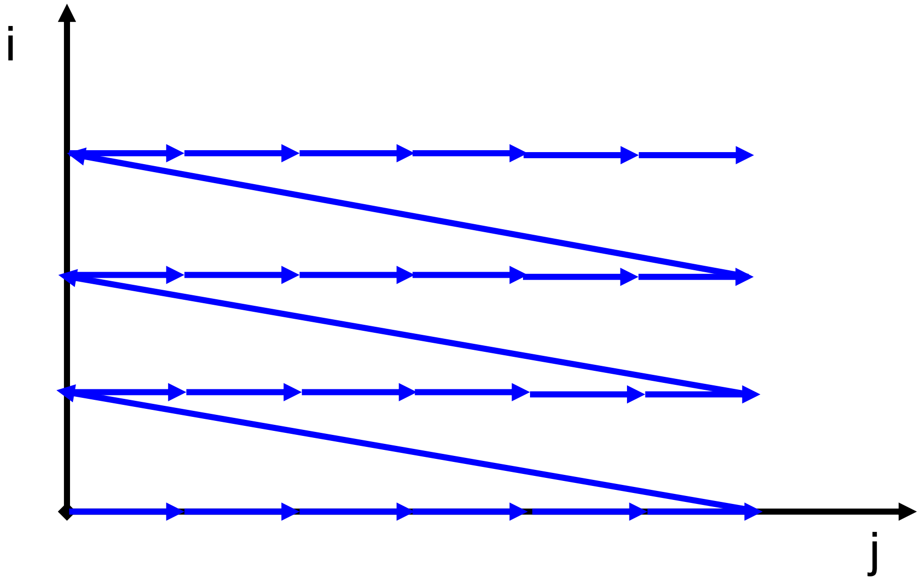
- Analyze reuse in computation
- Apply loop reordering transformations to improve locality based on reuse
- With any loop reordering transformation, always ask
  - **Safety?** (doesn't reverse dependences)
  - **Profitablity?** (improves locality)

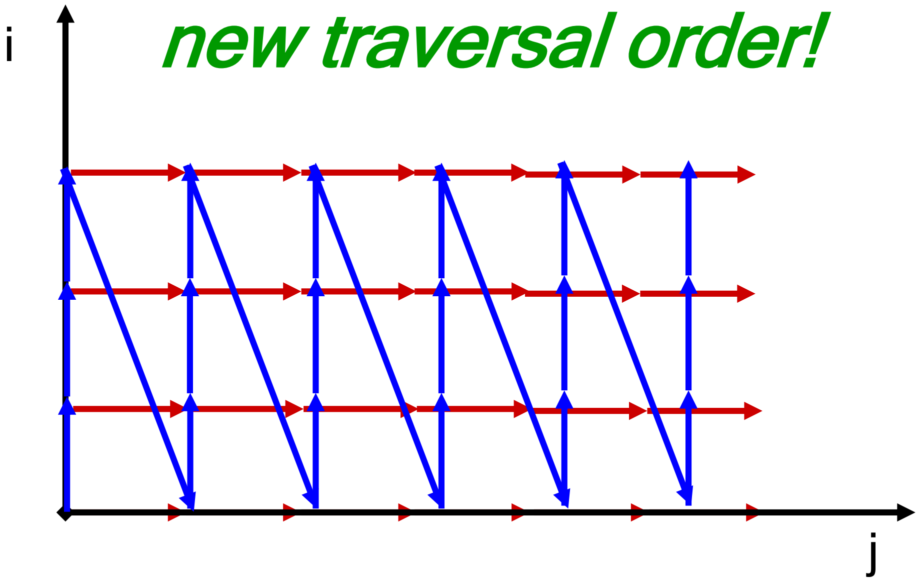
Loop Permutation: A Reordering Transformation

Permute the order of the loops to modify the traversal order

```
for (i= 0; i<3; i++)
  for (j=0; j<6; j++)
    A[i][j+1]=A[i][j]+B[j];
```

```
for (j=0; j<6; j++)
  for (i= 0; i<3; i++)
    A[i][j+1]=A[i][j]+B[j];
```





new traversal order!

Which one is better for row-major storage?

Safety of Permutation

- **Intuition:** Cannot permute two loops i and j in a loop nest if doing so reverses the direction of any dependence.

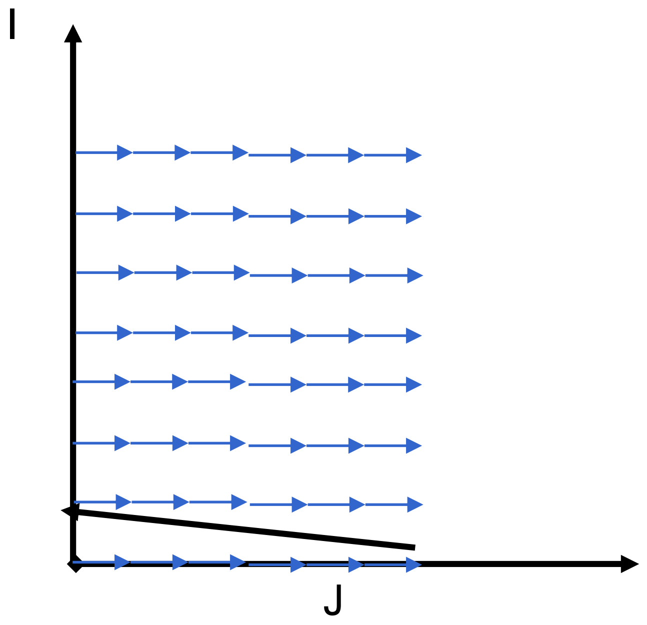
```
for (i= 0; i<3; i++)
   for (j=0; j<6; j++)
      A[i][j+1]=A[i][j]+B[j];
```

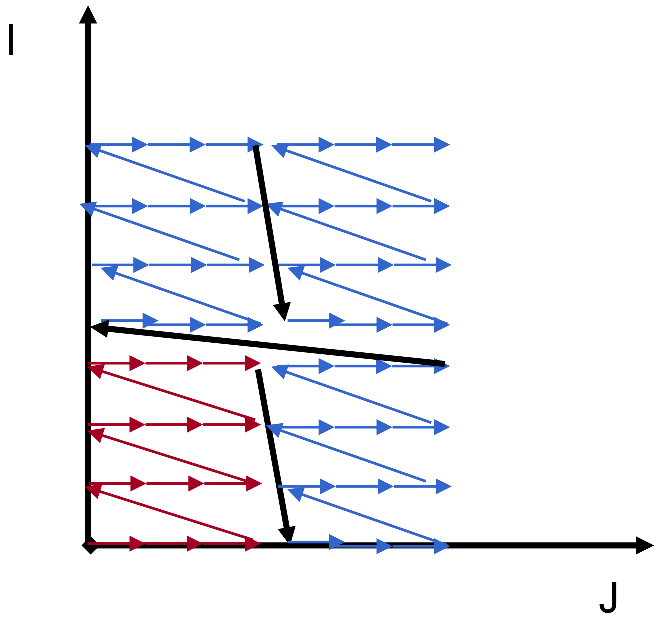
```
for (i= 0; i<3; i++)
  for (j=1; j<6; j++)
    A[i+1][j-1]=A[i][j]+B[j];
```

- Ok to permute?

Tiling (Blocking): Another Loop Reordering Transformation

- Blocking reorders loop iterations to bring iterations that reuse data closer in time





Tiling Example

```
for (j=1; j<M; j++)
      for (i=1; i<N; i++)
          D[i] = D[i] + B[j][i];
```

Strip mine

```
for (j=1; j<M; j++)
      for (ii=1; ii<N; ii+=s)
          for (i=ii; i<min(ii+s,N); i++)
                        D[i] = D[i] +B[j][i];
```

Permute

```
for (ii=1; ii<N; ii+=s)
      for (j=1; j<M; j++)
        for (i=ii; i<min(ii+s,N); i++)
                        D[i] = D[i] +B[j][i];
```

Legality of Tiling

- Tiling = strip-mine and permutation
  - Strip-mine does not reorder iterations
  - Permutation must be legal

  OR

  - strip size less than dependence distance

A Few Words On Tiling

- Tiling can be used hierarchically to compute partial results on a block of data wherever there are capacity limitations
  - Between grids if total data exceeds global memory capacity
  - Across thread blocks if shared data exceeds shared memory capacity (also to partition computation across blocks and threads)
  - Within threads if data in constant cache exceeds cache capacity or data in registers exceeds register capacity or (as in example) data in shared memory for block still exceeds shared memory capacity

Summary of Performance Considerations

Summary of Performance Considerations

- Thread Execution and Divergence
- Communication Through Memory
- Instruction Level Parallelism and Thread Level Parallelism
- Memory Coalescing
- Shared Memory Bank Conflicts
- Parallel Reduction
- Prefetching
- Loop Unrolling and Transformations
- Occupancy
- Kernel Launch Overhead
- Instruction Performance

Thread Execution and Divergence

- Instructions are issued per 32 threads (warp)
- Divergent branches:
  - Threads within a single warp take different paths
    - if-else, ...
  - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance

An Example

```
// is this barrier divergent?
for(int offset = blockDim.x / 2;
    offset > 0;
    offset >>= 1)
{
  ...
  __syncthreads();
}
```

A Second Example

```
// what about this one?
__global__ void do_i_halt(int *input)
{
  int i = ...
  if(input[i])
  {
    ...
    __syncthreads();// a divergent barrier
                    // hangs the machine
  }
}
```

Compute Capabilities

- Reminder: do not take various constants, such as size of shared memory etc., for granted since they continuously change
- Check CUDA programming guide for the features of the compute capability of your GPU

Reduction Trees

Partition and Summarize

- A commonly used strategy for processing large input data sets
  - There is no required order of processing elements in a data set (associative and commutative)
  - Partition the data set into smaller chunks
  - Have each thread to process a chunk
  - Use a reduction tree to summarize the results from each chunk into the final answer
- We will focus on the reduction tree step for now
- Google and Hadoop MapReduce frameworks are examples of this pattern

Reduction enables other techniques

- Reduction is also needed to clean up after some commonly used parallelizing transformations

- Privatization
  - Multiple threads write into an output location
  - Replicate the output location so that each thread has a private output location
  - Use a reduction tree to combine the values of private locations into the original output location

What is a reduction computation

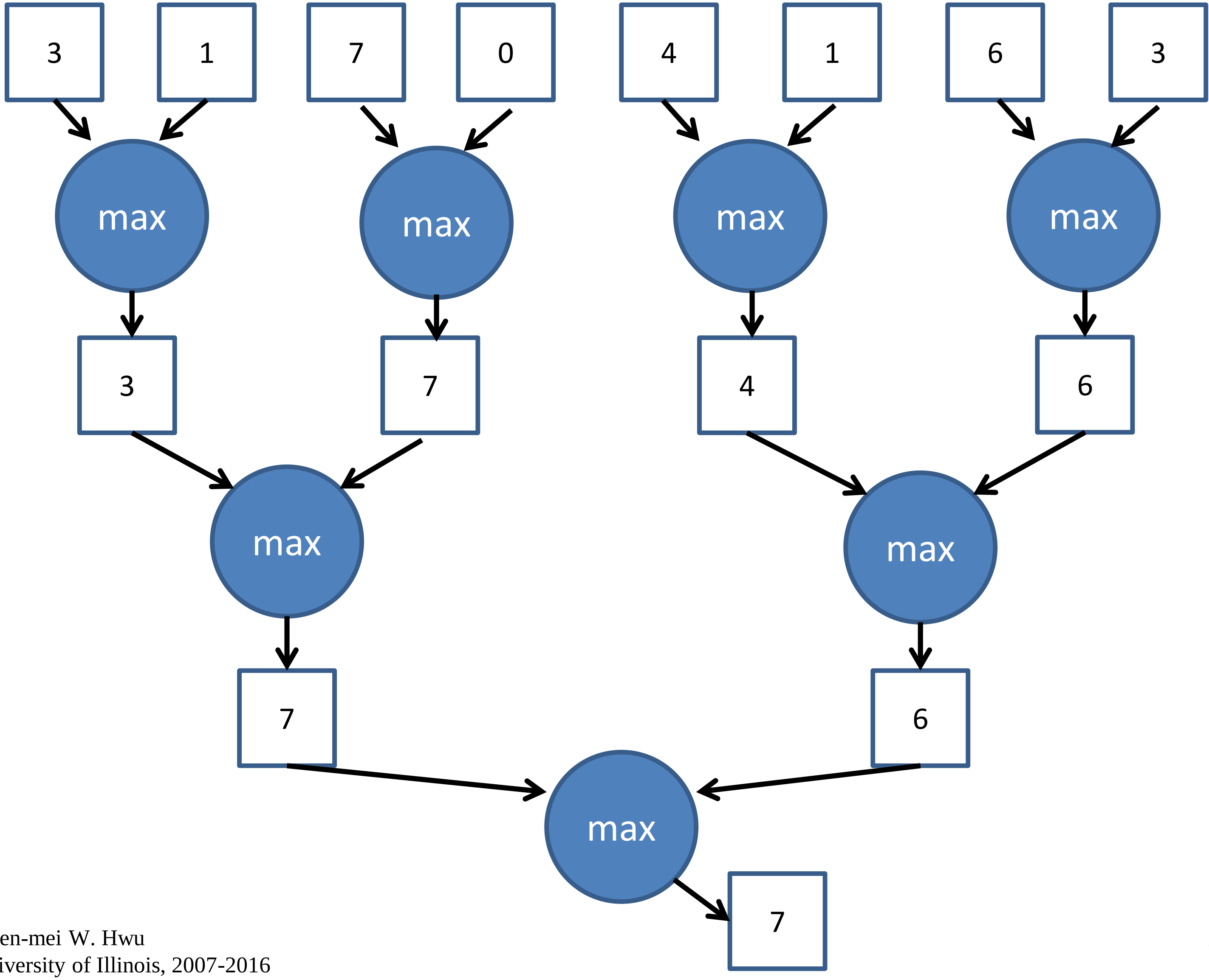
- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product
  - Often with user defined reduction operation function as long as the operation
    - Is associative and commutative
    - Has a well-defined identity value (e.g., 0 for sum)

A sequential reduction algorithm performs N operations - O(N)

- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction

- Scan through the input and perform the reduction operation between the result value and the current input value

A parallel reduction tree algorithm performs N-1 Operations in log(N) steps



© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2016

A Quick Analysis

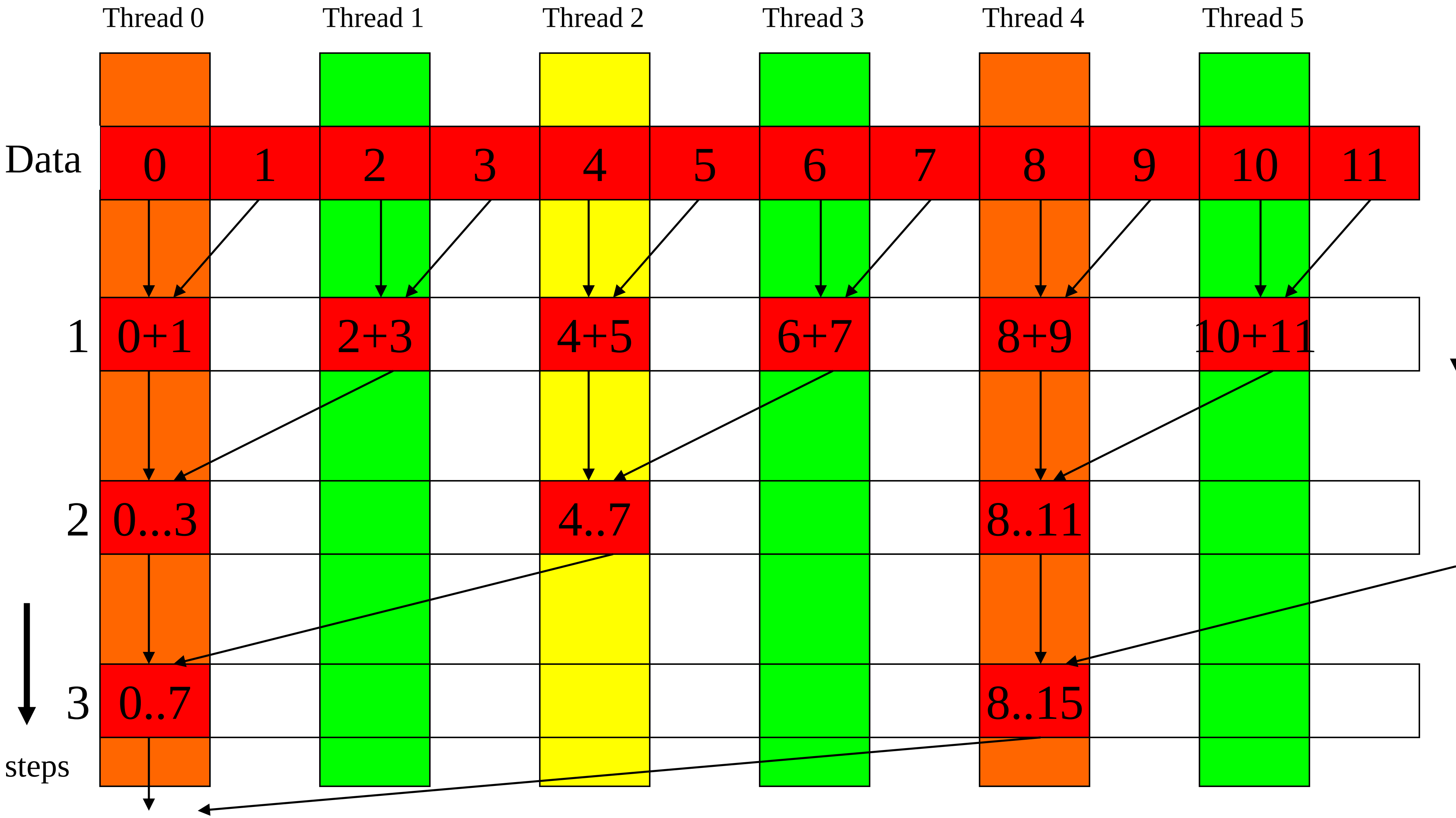
- For N input values, the reduction tree performs
  - (1/2)N + (1/4)N + (1/8)N + … (1/N) = (1- (1/N))N = N-1 operations
  - In Log (N) steps – 1,000,000 input values take 20 steps
    - Assuming that we have enough execution resources
  - Average Parallelism (N-1)/Log(N))
    - For N = 1,000,000, average parallelism is 50,000
    - However, peak resource requirement is 500,000!
- This is a work-efficient parallel algorithm
  - The amount of work done is comparable to sequential
  - Many parallel algorithms are not work efficient

A Sum Reduction Example

- Parallel implementation:
  - Recursively halve # of threads, add two values per thread in each step
  - Takes log(n) steps for n elements, requires n/2 threads

- 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 step brings the partial sum vector closer to the sum
  - The final sum will be in element 0
  - Reduces global memory traffic due to partial sum values

Vector Reduction with Branch Divergence



Thread 0 Thread 1 Thread 2 Thread 3 Thread 4 Thread 5

Data 0 1 2 3 4 5 6 7 8 9 10 11

1 0+1 2+3 4+5 6+7 8+9 10+11

2 0...3 4..7 8..11

3 0..7 8..15

steps

Partial Sum elements →

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2016

Some Observations

- In each iteration, 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 still consume execution resources
- No more than half of threads will be executing after the first step
  - All odd index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence.
    - This can go on for a while, up to 5 more steps ($1024/32=16= 2^5$), where each active warp only has one productive thread until all warps in a block retire

Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators

- Reduction satisfies this criterion

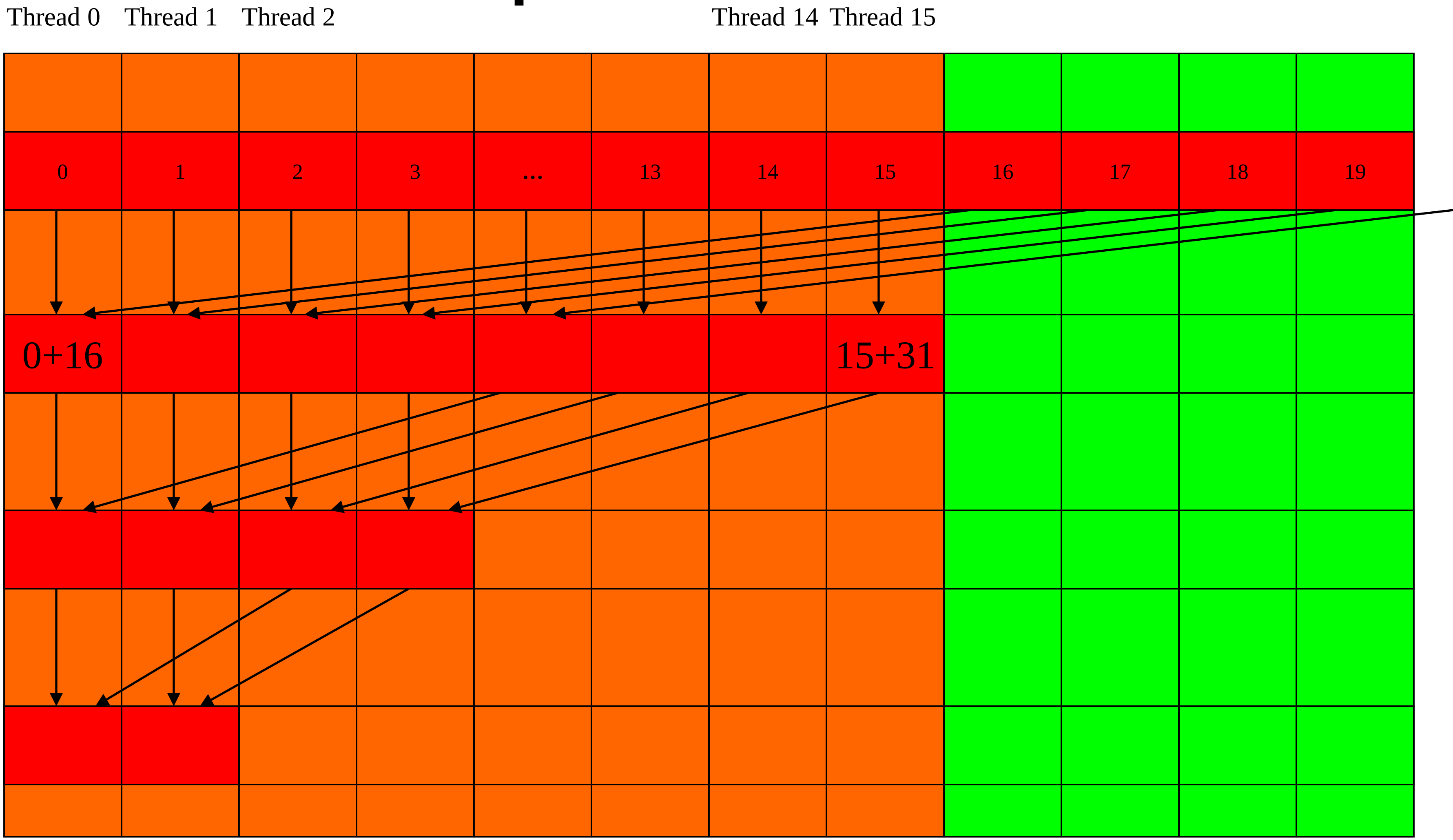
A Better Strategy

- Always compact the partial sums into the first locations in the partialSum[] array

- Keep the active threads consecutive

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2016

An Example of 16 threads



© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2016

A Better Reduction Kernel

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

A Quick Analysis

- For a 1024 thread block
  - – No divergence in the first 5 steps
  - – 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
  - – The final 5 steps will still have divergence

Parallel Algorithm Overhead

```
__shared__ float partialSum[2*BLOCK_SIZE];

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

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2016

Parallel Algorithm Overhead

```
__shared__ float partialSum[2*BLOCK_SIZE];

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

Parallel Execution Overhead

- Although the number of “operations” is N, each operation involves much more complex address calculation and intermediate result manipulation

- If the parallel code is executed on a single-thread hardware, it would be significantly slower than the code based on the original sequential algorithm

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2016

Parallel Prefix Sum (Scan)

Objectives

- Prefix Sum (Scan) algorithms
  - Frequently used for parallel work assignment and resource allocation
  - A key primitive in many parallel algorithms to convert serial computation into parallel computation
  - Based on reduction tree and reverse reduction tree

- To learn the concept of double buffering

(Inclusive) Prefix-Sum (Scan) Definition

Definition: The all-prefix-sums operation takes a binary associative operator $\oplus$, and an array of n elements

$$[x_0, x_1, \ldots, x_{n-1}],$$

and returns the array

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].$$

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array [3 1 7 0 4 1 6 3],
would return [3 4 11 11 15 16 22 25].

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2016

A Inclusive Scan Application Example

- Assume that we have a 100-inch bread to feed 10 people
- We know how much each person wants in inches
  - [3 5 2 7 28 4 3 0 8 1]
- How do we cut the bread quickly?
- How much will be left

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.
- Method 2: calculate Prefix scan
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Typical Applications of Scan

- Assigning camp slots
- Assigning farmer market space
- Allocating memory to parallel threads
- Allocating memory buffer to communication channels
- Useful for many parallel algorithms:
  - radix sort
  - quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Etc.

A Inclusive Sequential Prefix-Sum

Given a sequence $[x_0, x_1, x_2, \ldots]$

Calculate output $[y_0, y_1, y_2, \ldots]$

Such that
$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$

…

Using a recursive definition

$$y_i = y_{i-1} + x_i$$

A Work Efficient C Implementation

```
y[0] = x[0];
for (i = 1; i < Max_i; i++)
   y[i] = y [i-1] + x[i];
```

Computationally efficient:

N additions needed for N elements - O(N)

A Naïve Inclusive Parallel Scan

- Assign one thread to calculate each y element
- Have every thread to add up all x elements needed for the y element

$$y_0 = x_0$$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

“Parallel programming is easy as long as you do not care about performance.”

Parallel Inclusive Scan using Reduction Trees

- Calculate each output element as the reduction of all previous elements
  - Some reduction partial sums will be shared among the calculation of output elements
  - Based on hardware added design by Peter Kogge and Harold Stone at IBM in the 1970s – Kogge-Stone Trees

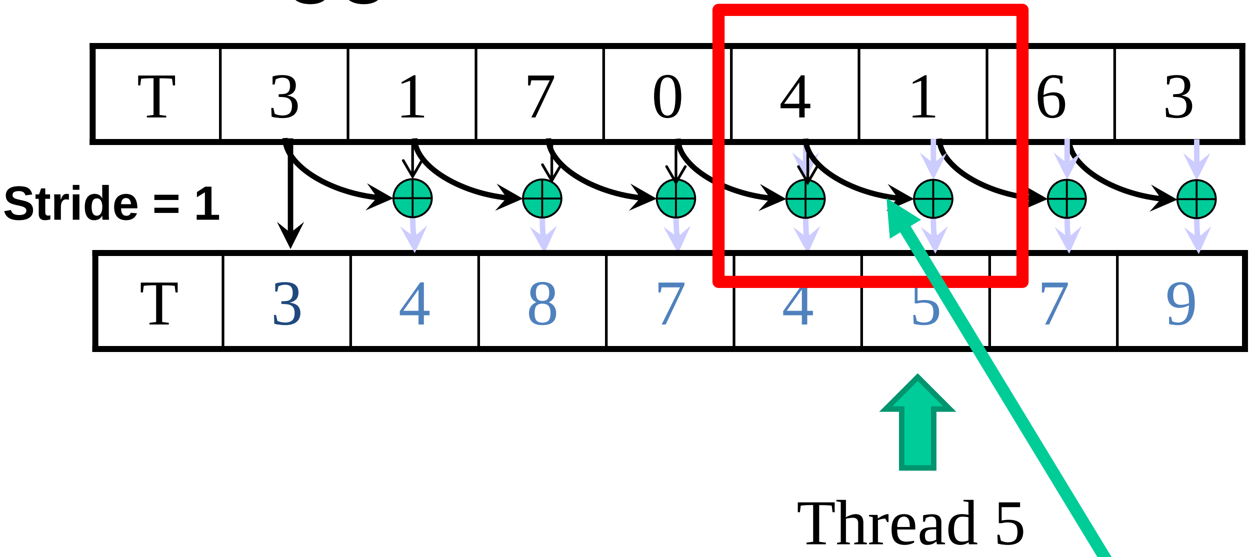
A Slightly Better Parallel Inclusive Scan Algorithm

T	3	1	7	0	4	1	6	3

1. Load input from global memory into shared memory array T

Each thread loads one value from the input (global memory) array into shared memory array T.

A Kogge-Stone Parallel Scan Algorithm



T	3	1	7	0	4	1	6	3
T	3	4	8	7	4	5	7	9

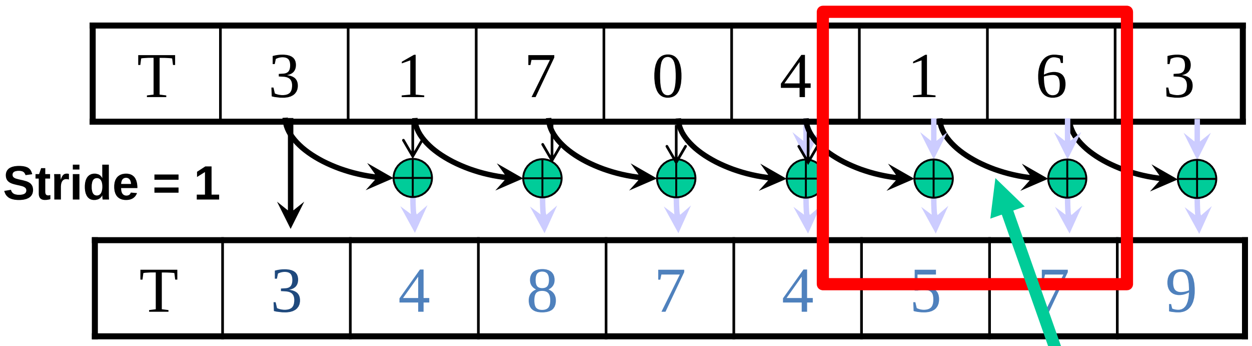
1. (previous slide)

2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads *stride* to *n-1 active:* add pairs of elements that are *stride* elements apart.

- Active threads: *stride* to *n*-1 (*n-stride* threads)
- Thread *j* adds elements *j* and *j-stride* from T and writes result into shared memory buffer T
- Each iteration requires two syncthreads
  - syncthreads(); // make sure that input is in place
  - float temp = T[j] + T[k - stride];

Iteration #1
Stride = 1

A Kogge-Stone Parallel Scan Algorithm



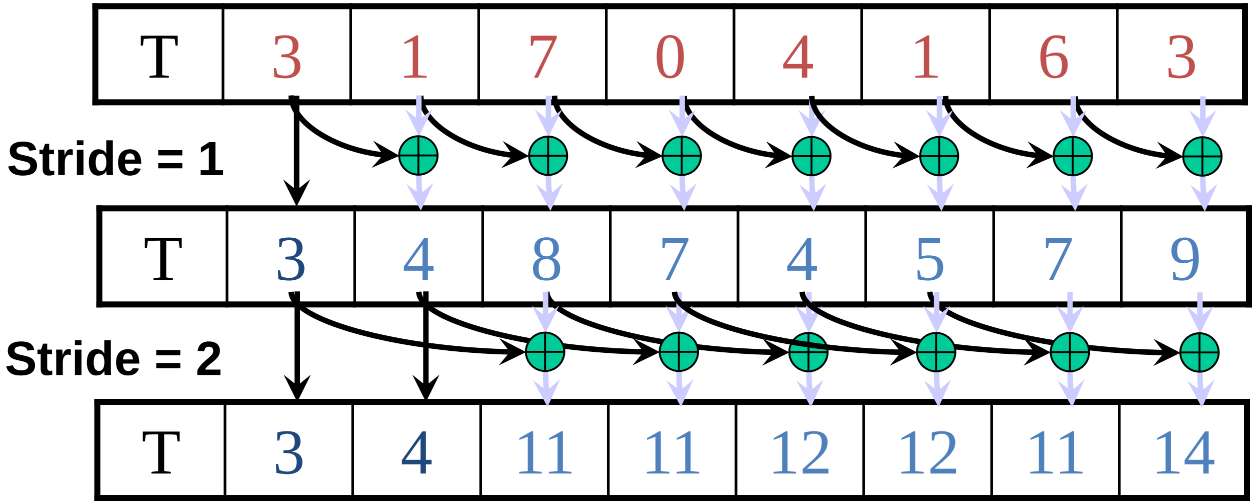
Stride = 1

T	3	1	7	0	4	1	6	3
T	3	4	8	7	4	5	7	9

- Active threads: *stride* to *n*-1 (*n-stride* threads)
- Thread *j* adds elements *j* and *j-stride* from T and writes result into shared memory buffer T
- Each iteration requires two syncthreads
  - syncthreads(); // make sure that input is in place
  - float temp = T[j] + T[j - stride];
  - syncthreads(); // make sure that previous output has been consumed
  - T[j] = temp;

Iteration #1
Stride = 1

A Kogge-Stone Parallel Scan Algorithm



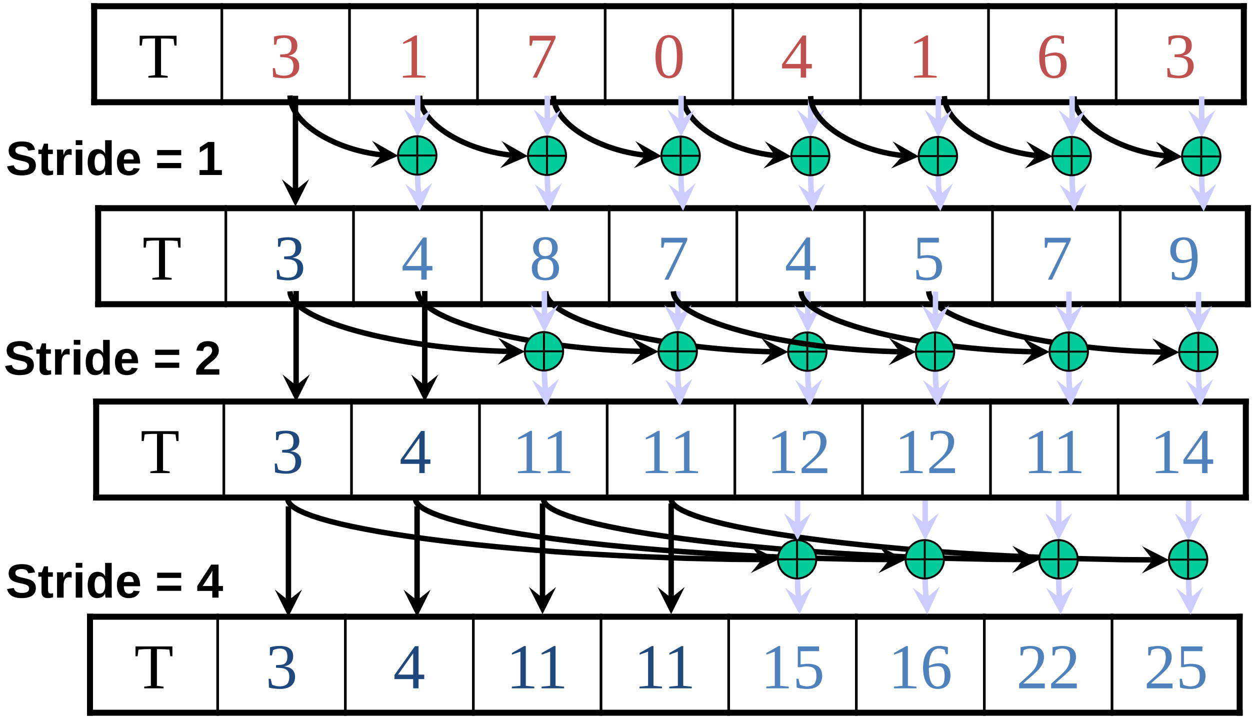
1. …

2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads *stride* to *n-1 active:* add pairs of elements that are *stride* elements apart.

Iteration #2
Stride = 2

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2016

A Kogge-Stone Parallel Scan Algorithm



1. Load input from global memory to shared memory.

2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads *stride* to *n-1 active:* add pairs of elements that are *stride* elements apart.

3. Write output from shared memory to device memory

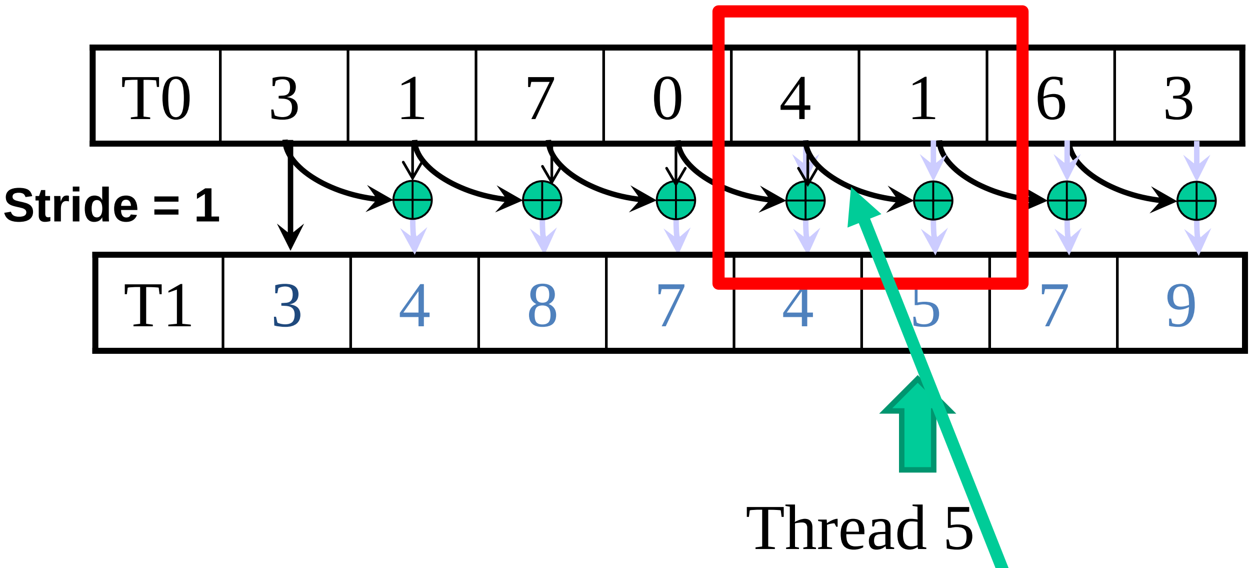
Iteration #3
Stride = 4

© David Kirk/NVIDIA and Wen-mei W. Hwu
ECE408/CS483/ECE498al University of Illinois, 2007-2016

Double Buffering

- Use two copies of data T0 and T1
- Start by using T0 as input and T1 as output
- Switch input/output roles after each iteration
  - Iteration 0: T0 as input and T1 as output
  - Iteration 1: T1 as input and T0 and output
  - Iteration 2: T0 as input and T1 as output
- This is typically implemented with two pointers, source and destination that swap their contents from one iteration to the next
- This eliminates the need for the second syncthreads

A Double-Buffered
Kogge-Stone Parallel Scan Algorithm



T0	3	1	7	0	4	1	6	3
T1	3	4	8	7	4	5	7	9

1. (previous slide)
2. Iterate log(n) times, stride from 1 to ceil(n/2.0). Threads *stride* to *n-1 active:* add pairs of elements that are *stride* elements apart.

- Active threads: *stride* to *n*-1 (*n-stride* threads)
- Thread *j* adds elements *j* and *j-stride* from T and writes result into shared memory buffer T
- Each iteration requires only one syncthreads
  - syncthreads(); // make sure that input is in place
  - float destination[j] = source[j] + source[j - stride];
  - temp = destination; destination = source; source = temp;

Iteration #1
Stride = 1

Work Efficiency Analysis

- A Kogge-Stone scan kernel executes log(n) parallel iterations
  - The steps do (n-1), (n-2), (n-4),..(n- n/2) add operations each
  - Total # of add operations: n * log(n) - (n-1) → O(n*log(n)) work

- This scan algorithm is not very work efficient
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 1,000,000 elements!
  - Typically used within each block, where n ≤ 1,024

- A parallel algorithm can be slow when execution resources are saturated due to low work efficiency

To be continued…