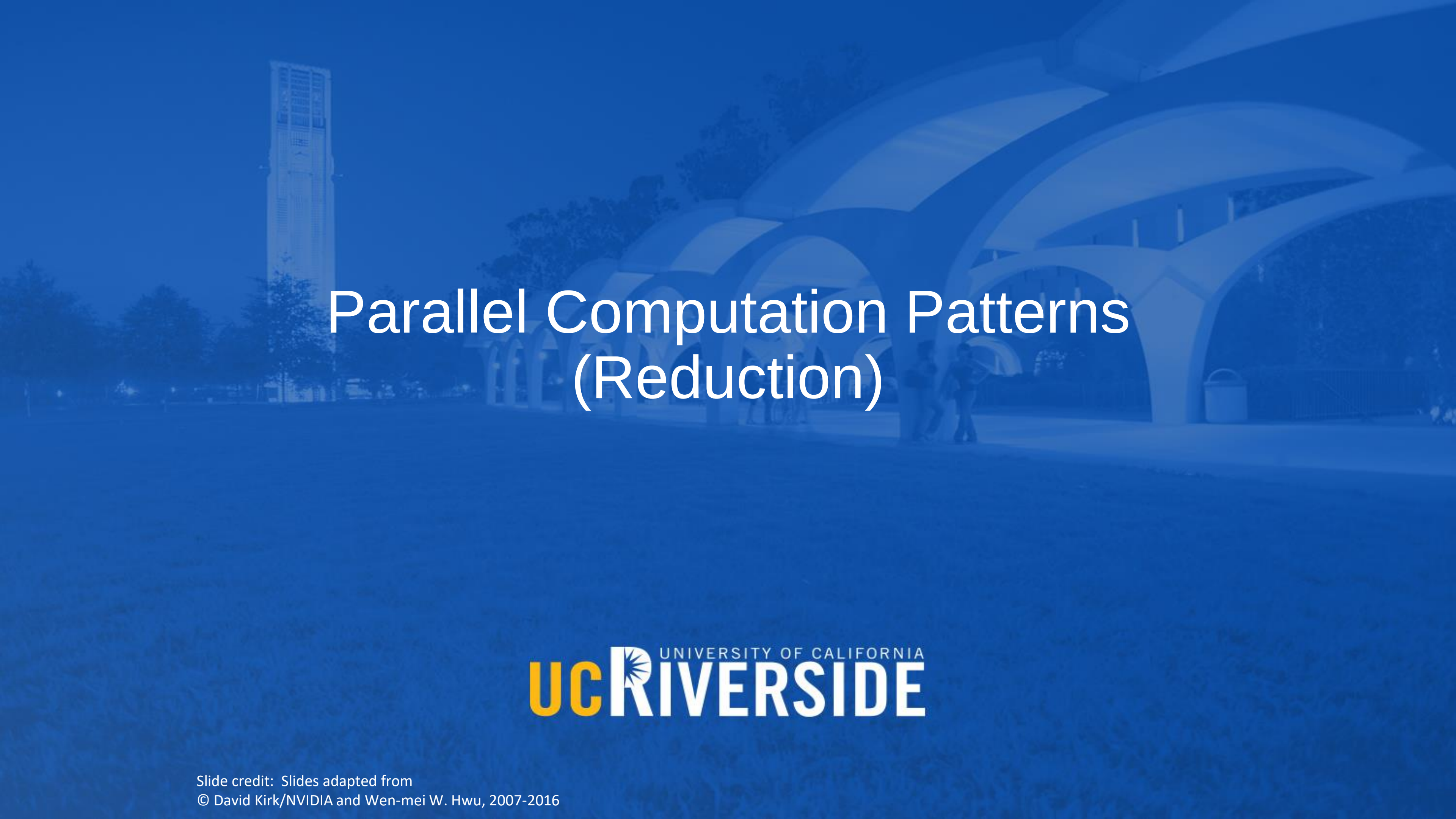
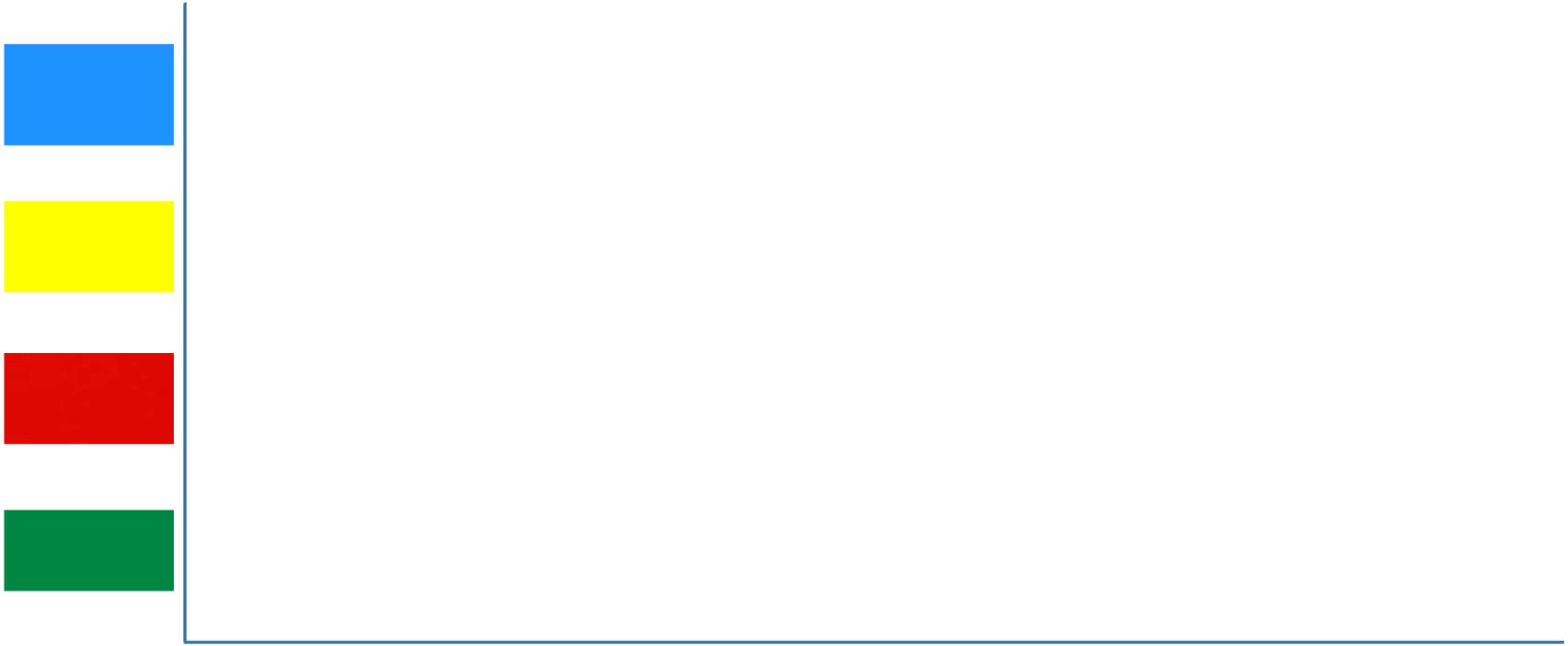


Parallel Computation Patterns (Reduction)

UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

Slide credit: Slides adapted from
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2016

Choose you team, Team with second highest number wins

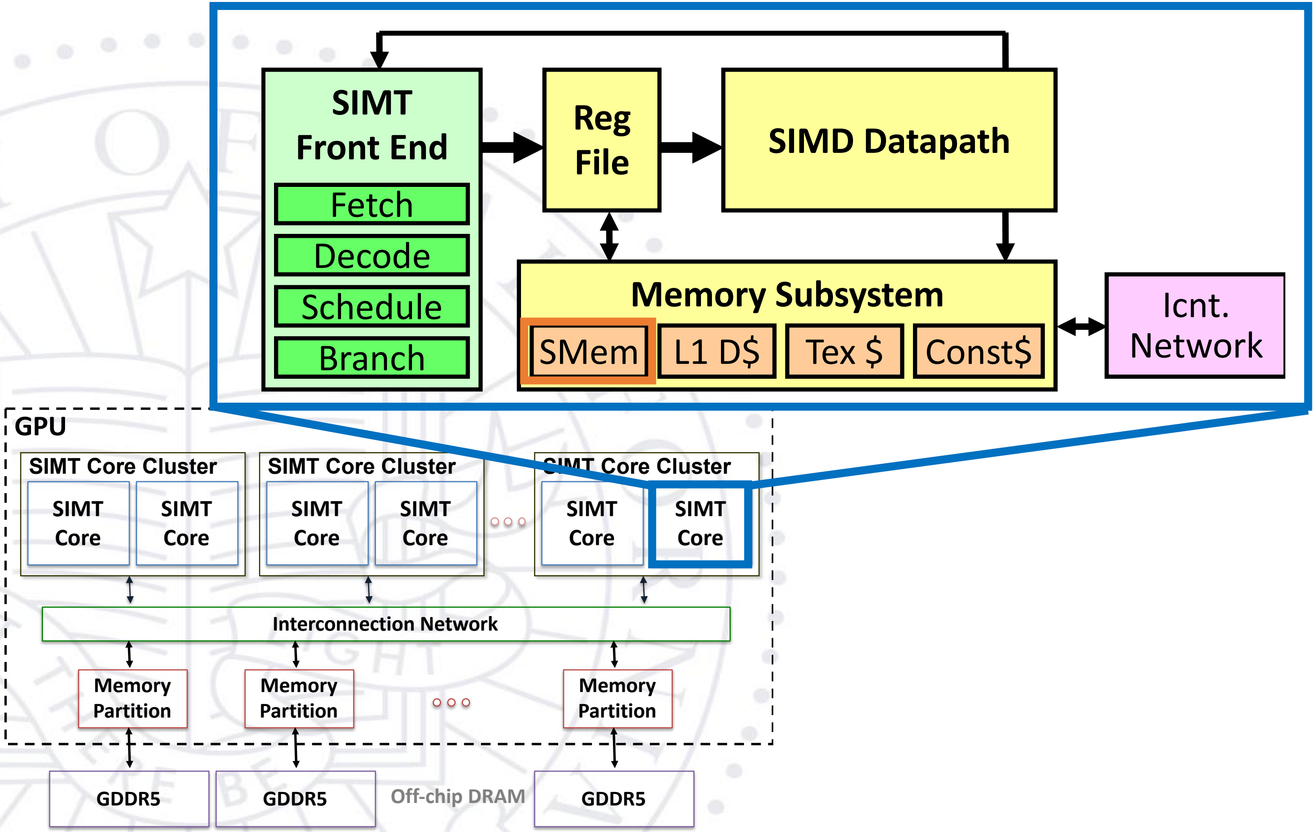


Logistics

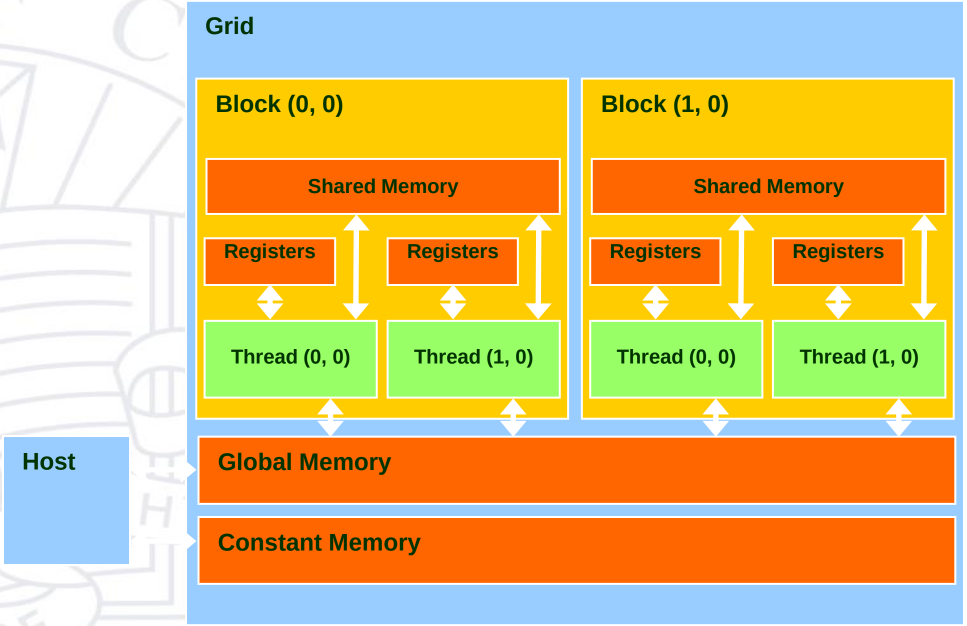
- Lab1 due this Friday
- Friday discussion will start lab 2
- From the school –
  - **In spring 2020, students can withdraw from one or more classes without a “W” grade through the end of week eight of instruction without an associate dean’s approval**

CUDA Memories

Hardware View of CUDA Memories



Programmer View of CUDA Memories



Declaring CUDA Variables

Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables reside in a register
  - Except per-thread arrays that reside in global memory

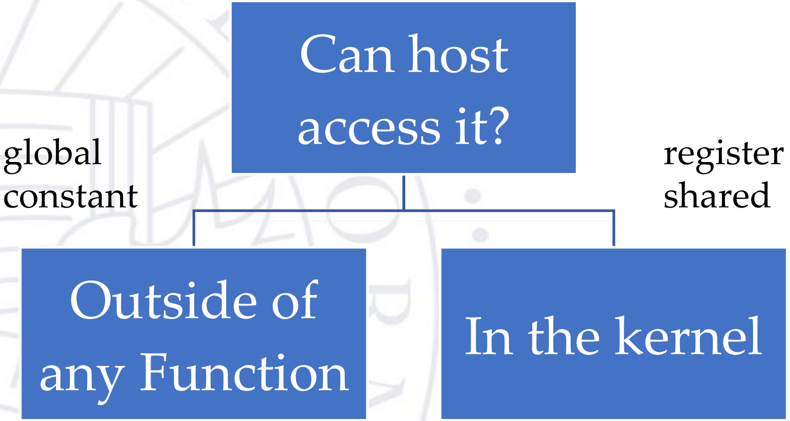
Example:
Shared Memory Variable Declaration

```
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
{

   __shared__  float ds_in[TILE_WIDTH][TILE_WIDTH];

  …
}
```

Where to Declare Variables?



Can host access it?

global
constant

Outside of any Function

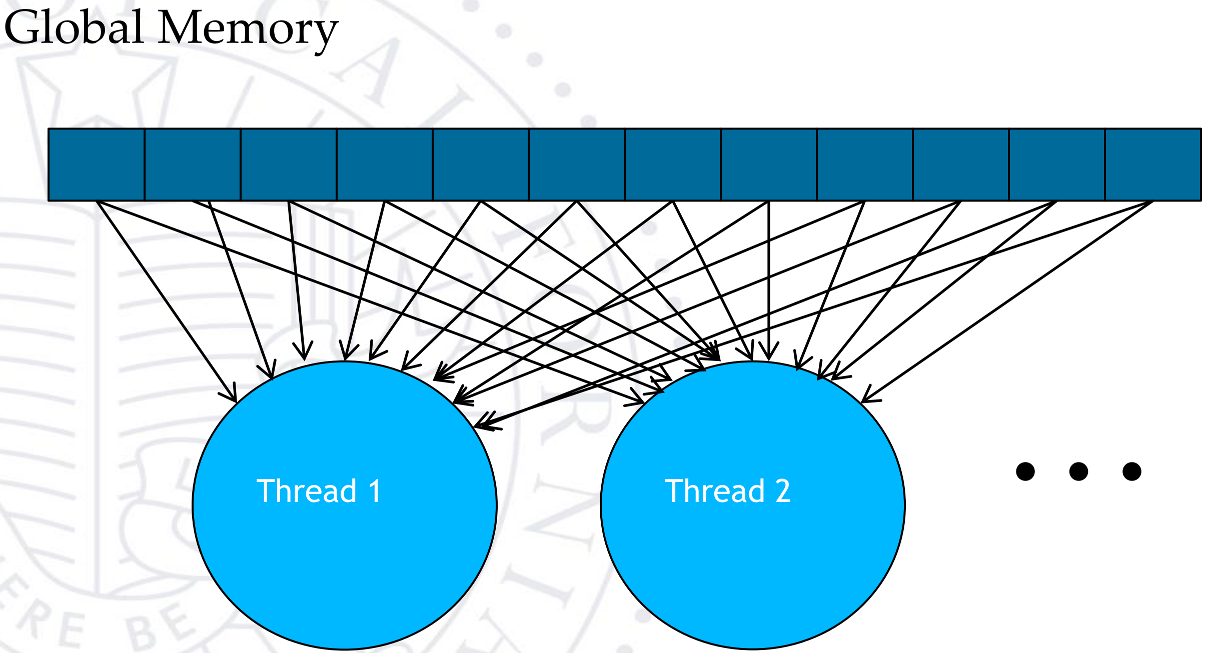
register
shared

In the kernel

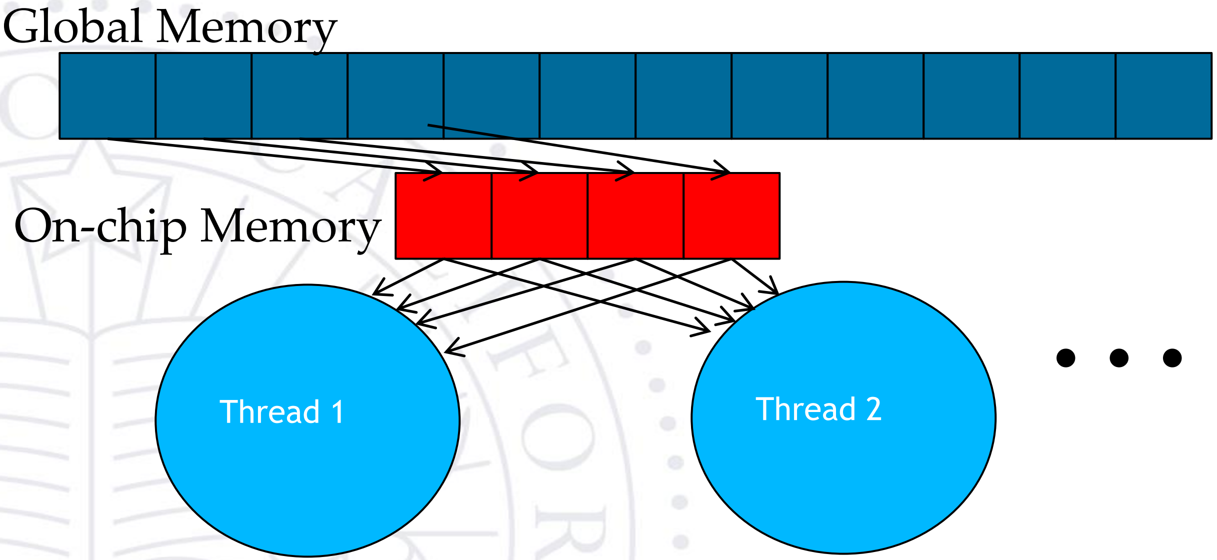
Shared Memory in CUDA

- A special type of memory whose contents are explicitly defined and used in the kernel source code
    - One in each SM
    - Accessed at much higher speed (in both latency and throughput) than global memory
    - Scope of access and sharing - thread blocks
    - Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
    - Accessed by memory load/store instructions
    - A form of scratchpad memory in computer architecture

Global Memory Access Pattern of the Basic Matrix Multiplication Kernel



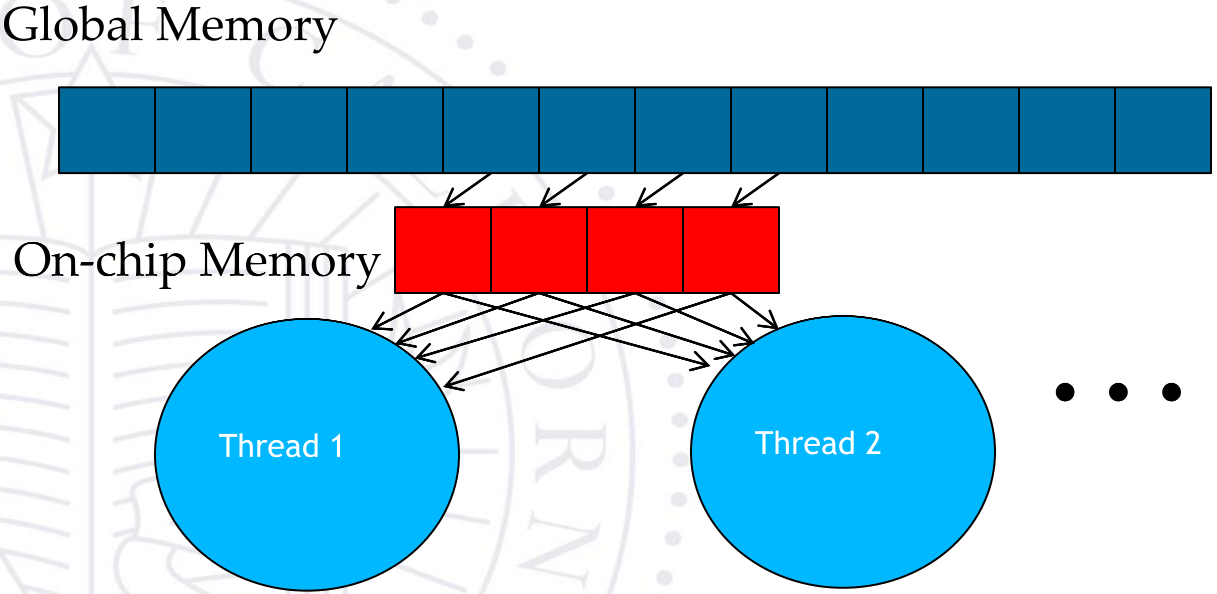
Tiling/Blocking - Basic Idea



Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time

Tiling/Blocking - Basic Idea



Key Takeaways

- Shared Memory is a programmer specified SM memory
- Located inside of an SM Core
- Threads within a single thread block have access to the same space
- Across thread blocks can not be accessed
- Static declaration through __shared__ int var[numElements]
- Dynamic declaration through func<<gridDim,BlockDim,SharedMemSize>>(args)

When poll is active, respond at **PollEv.com/marcuschow119**

Would you use shared memory if your data gets used once? why or why not?

Parallel Computation Patterns - Reduction

“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
- E.G., Google and Hadoop MapReduce frameworks support this strategy
- We will focus on the reduction tree step for now

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 (privatization)
  - Use a reduction tree to combine the values of private locations into the original output location

What is a reduction computation?

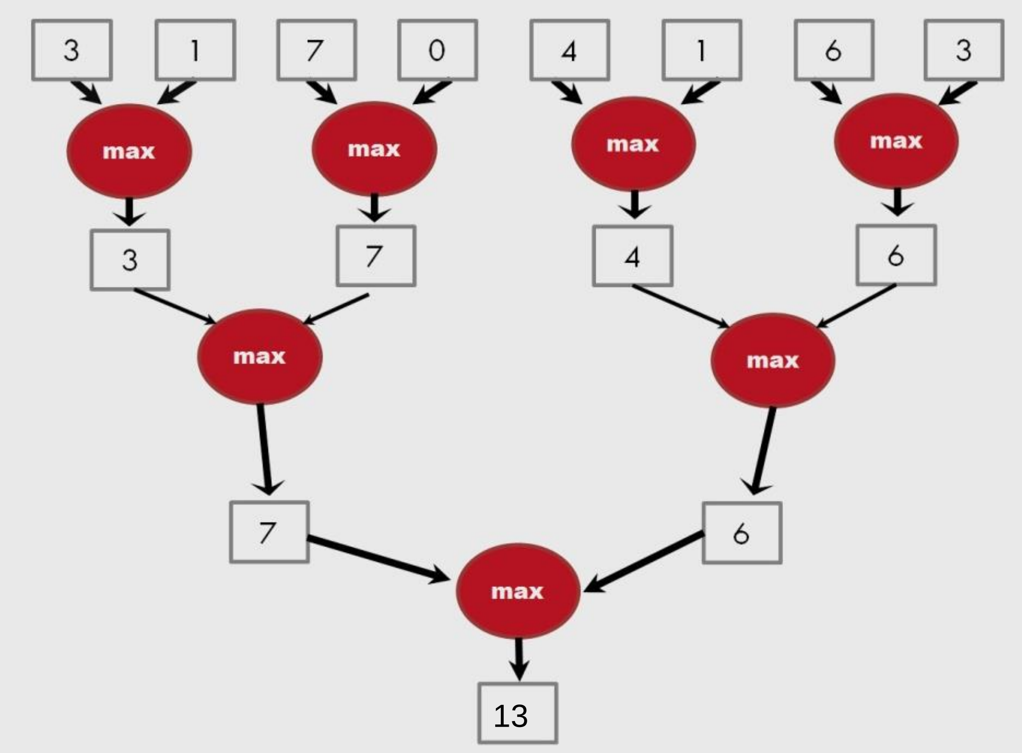
- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product
- Often used with a user defined reduction operation function as long as the operation
  - Is associative and commutative
  - Has a well-defined identity value (e.g., 0 for sum)
  - For example, the user may supply a custom “max” function for 3D coordinate data sets where the magnitude for the each coordinate data tuple is the distance from the origin.

An Efficient Sequential Reduction 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

- Iterate through the input and perform the reduction operation between the result value and the current input value
  - N reduction operations performed for N input values
  - Each input value is only visited once – an O(N) algorithm
  - This is a computationally efficient algorithm.

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



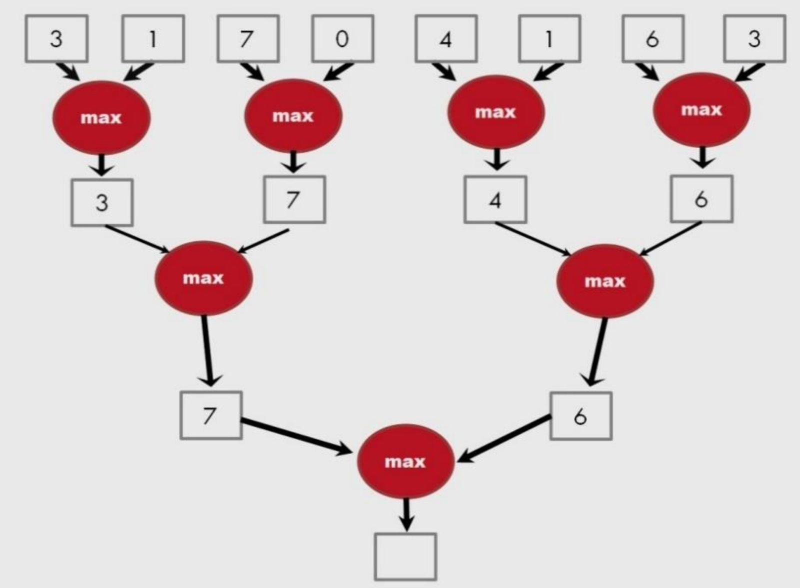
Work Efficiency Analysis

- 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 not resource efficient
- This is a ***work-efficient*** parallel algorithm
  - The amount of work done is comparable to the an efficient sequential algorithm
  - Many parallel algorithms are not work efficient

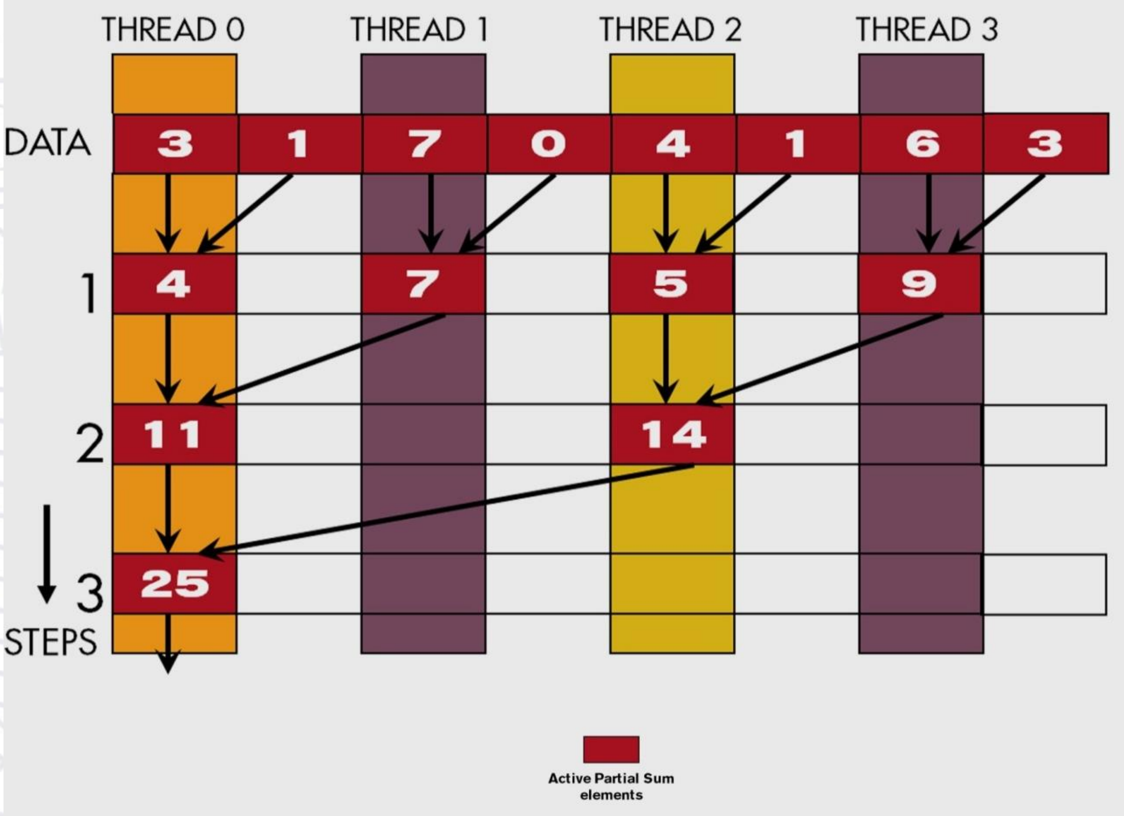
Basic reduction kernel

Parallel Sum Reduction

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

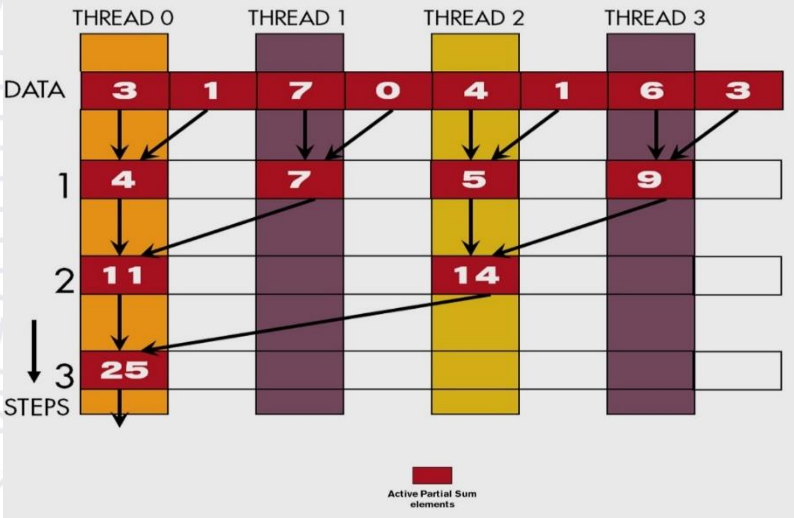


A Parallel Sum Reduction Example



A Naive Thread to Data Mapping

- Each thread is responsible for an even-index location of the partial sum vector (location of responsibility)
- After each step, half of the threads are no longer needed
- One of the inputs is always from the location of responsibility
- In each step, one of the inputs comes from an increasing distance away

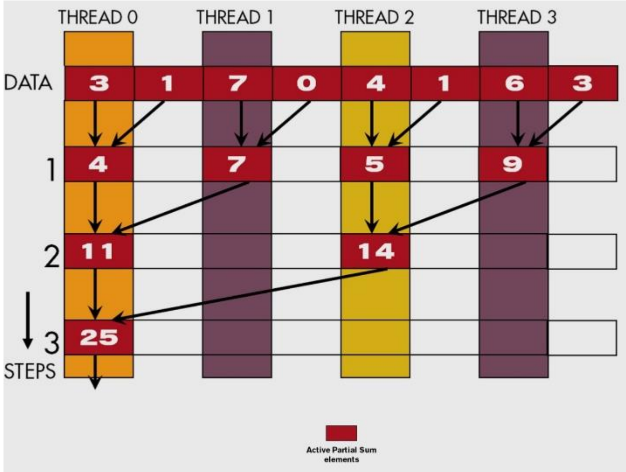


A Simple Thread Block Design

- Each thread block takes 2*BlockDim.x input elements
- Each thread loads 2 elements into shared memory

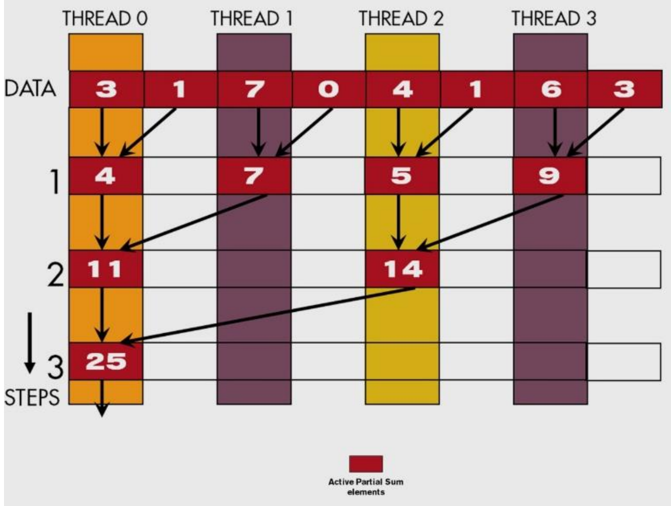
```
__shared__ float partialSum[2*BLOCK_SIZE];

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];
```



The Reduction Steps

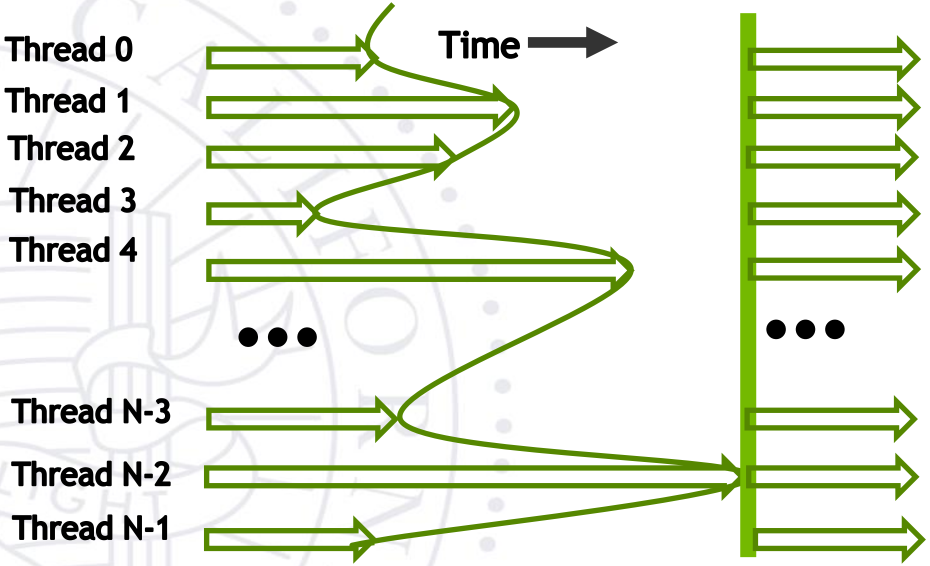
```
for (unsigned int stride = 1;
     stride <= blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % stride == 0)
    partialSum[2*t]+= partialSum[2*t+stride];
}
```



Barrier Synchronization

- __syncthreads() is needed to ensure that all elements of each version of partial sums have been generated before we proceed to the next step
- __syncthreads() synchronizes all threads within the *block*

Barrier Synchronization



Back to the Global Picture

- At the end of the kernel, Thread 0 in each block writes the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x
- There can be a large number of such sums if the original vector is very large
  - The host code may iterate and launch another kernel
- If there are only a small number of sums, the host can simply transfer the data back and add them together
- Alternatively, Thread 0 of each block could use atomic operations to accumulate into a global sum variable.

When poll is active, respond at **PollEv.com/marcuschow119**

What are some drawback to this implementation of reduction?

UC RIVERSIDE
UNIVERSITY OF CALIFORNIA

A better reduction model

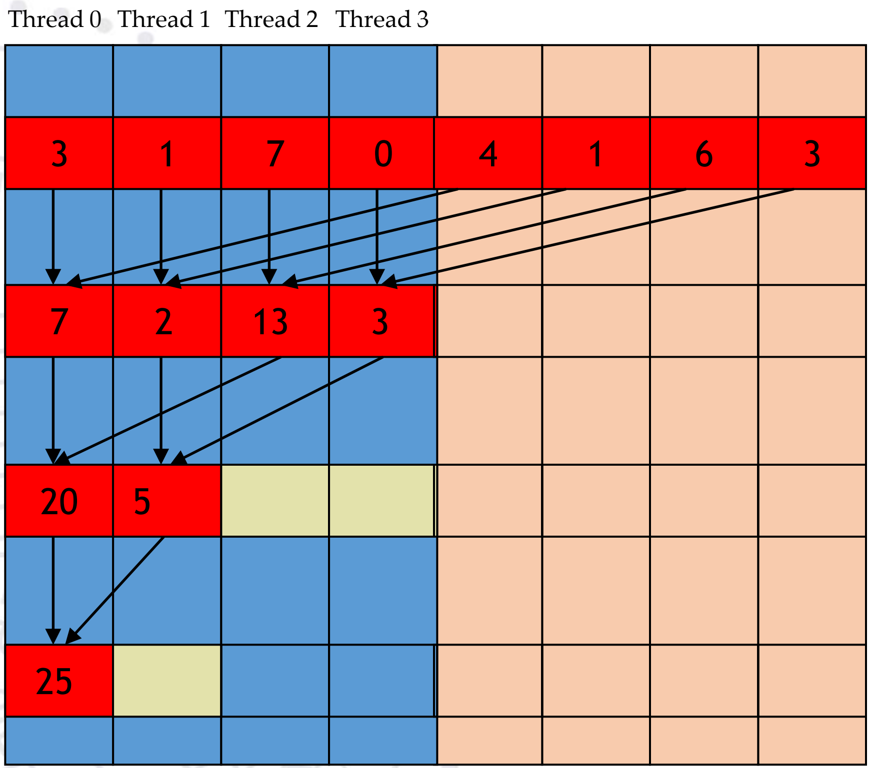
Some Observations on the naïve reduction kernel

- 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
- Half or fewer 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 6 more steps (stride = 32, 64, 128, 256, 512, 1024), 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
- Keep the active threads consecutive
  - Always compact the partial sums into the front locations in the partialSum[ ] array

An Example of 4 threads



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
    - All threads in each warp either all active or all inactive
  - The final 5 steps will still have divergence

When poll is active, respond at **PollEv.com/marcuschow119**

Are there any drawbacks to this implementation?

Start the presentation to see live content. Still no live content? Install the app or get help at **PollEv.com/app**