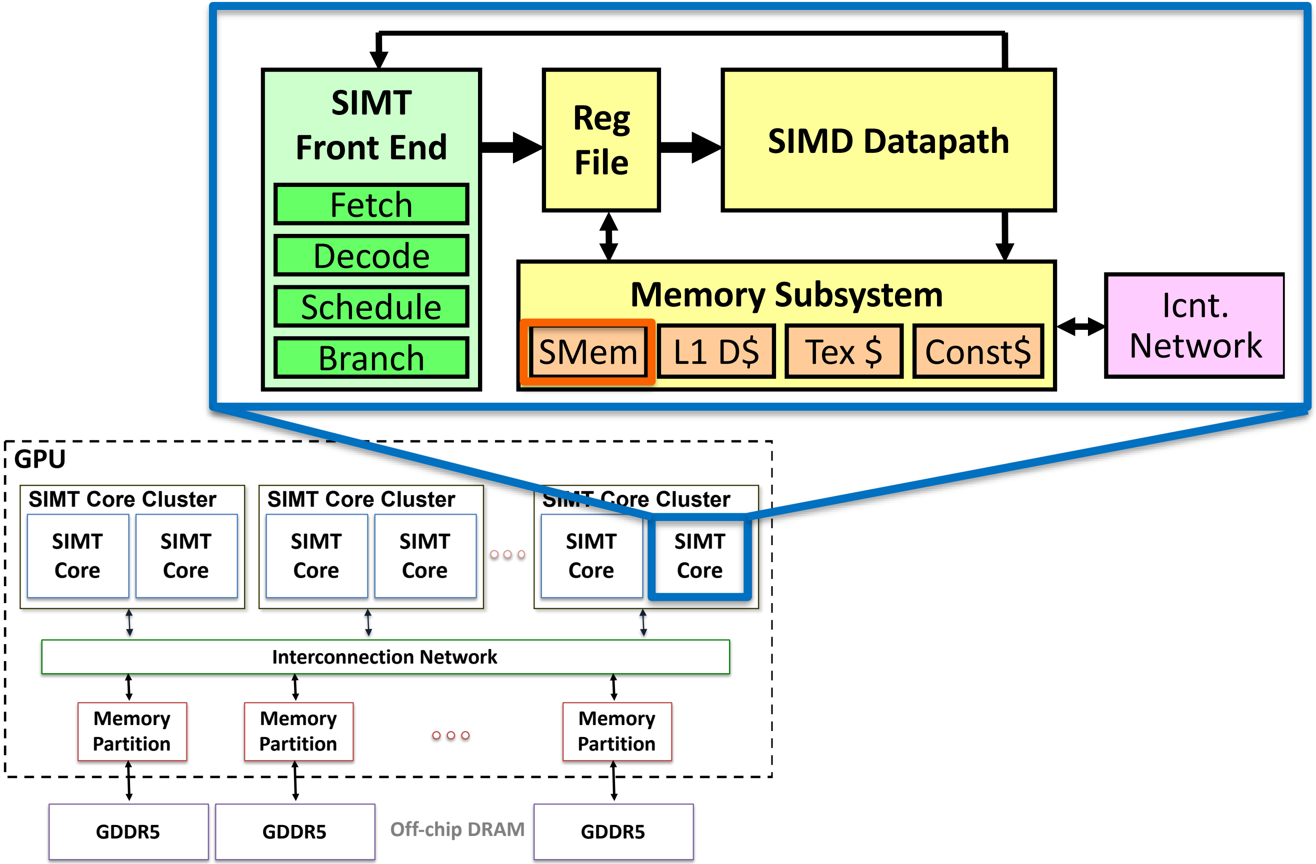
Parallel Computation Patterns (Reduction)

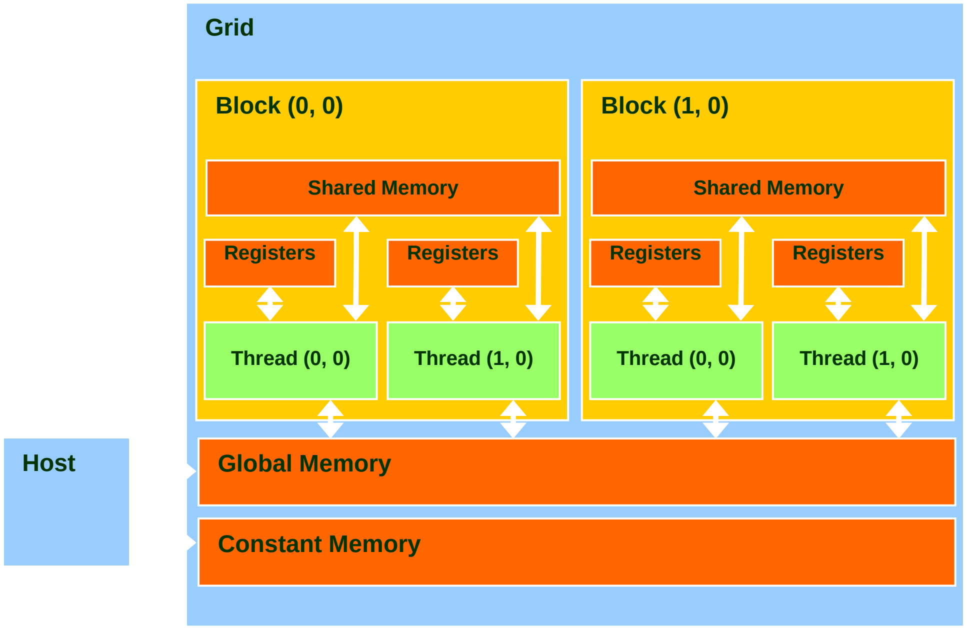
UNIVERSITY OF CALIFORNIA, RIVERSIDE

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

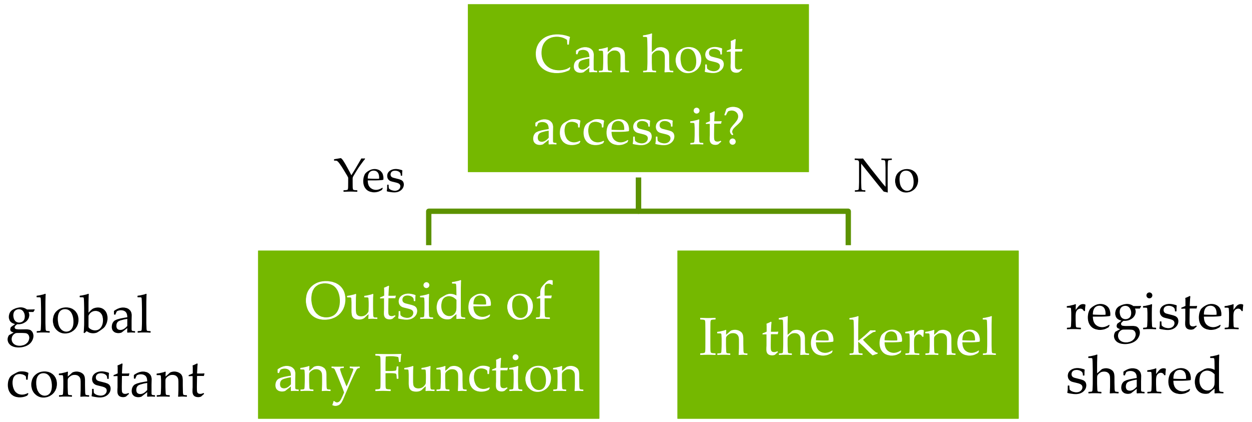
- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables reside in a register
  - Except per-thread arrays that reside in global memory

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?

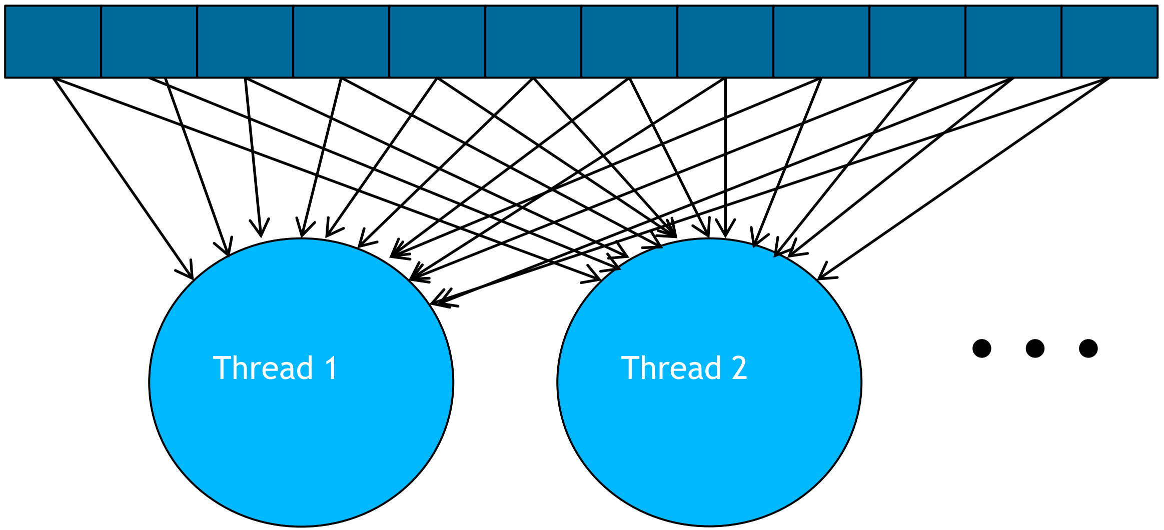


Shared Memory in CUDA

- 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

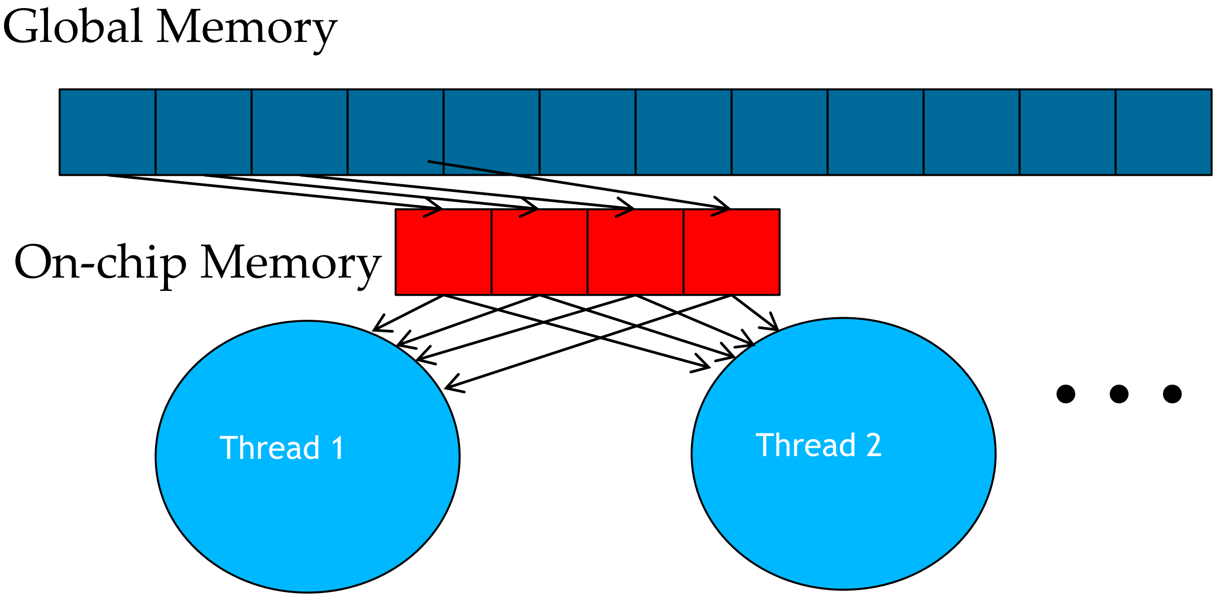
Example Global Memory Access Pattern

Global Memory



Each thread access every element in array

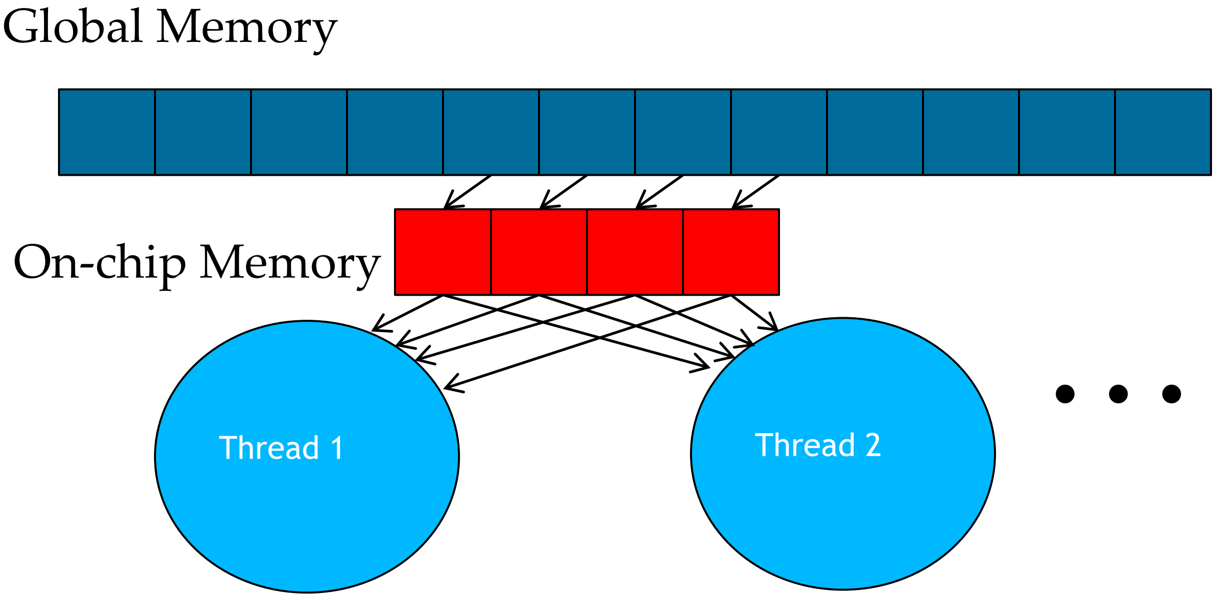
Optimization: Tiling/Blocking - Basic Idea



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



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
  - Also in distributed ML training. Reduce operation used heavily in collective communication

- 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
  - Also commonly used in ML training

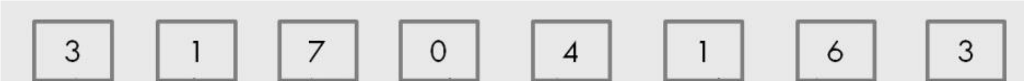
- Privatization (Optimization technique for reduction, more on this later)
  - Multiple threads write into an output location
  - Replicate the output location so that each thread has a private output location (privatization)
  - Use a reduction tree to combine the values of private locations into the 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 each coordinate data tuple is the distance from the origin.

An Efficient Sequential Reduction O(N)

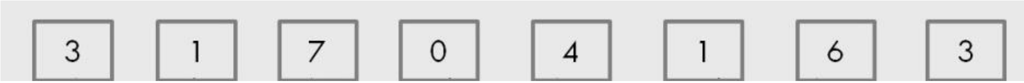


An Efficient Sequential Reduction O(N)

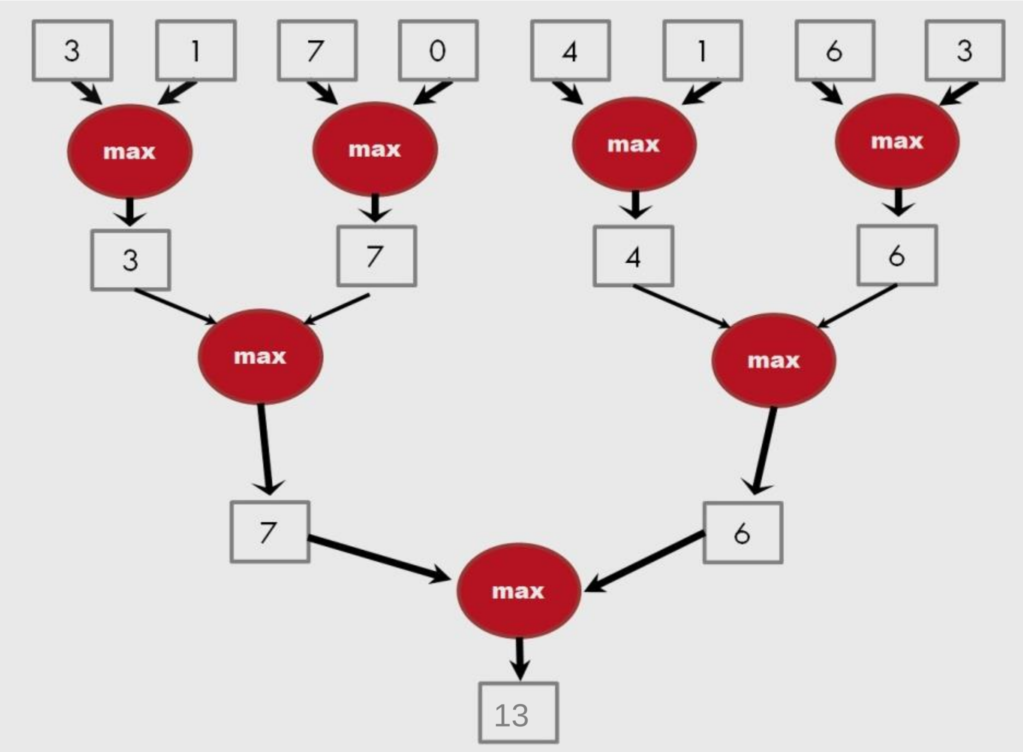
- Initialize the result as an identity value for the reduction operation
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction

- 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



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



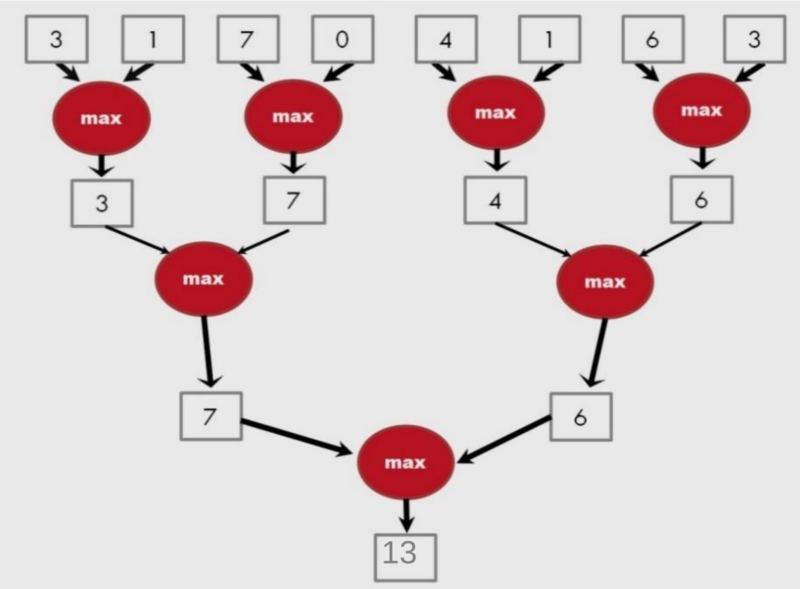
Work Efficiency Analysis

- How many operations do we perform?
- 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 an efficient sequential algorithm
  - Many parallel algorithms are not work efficient

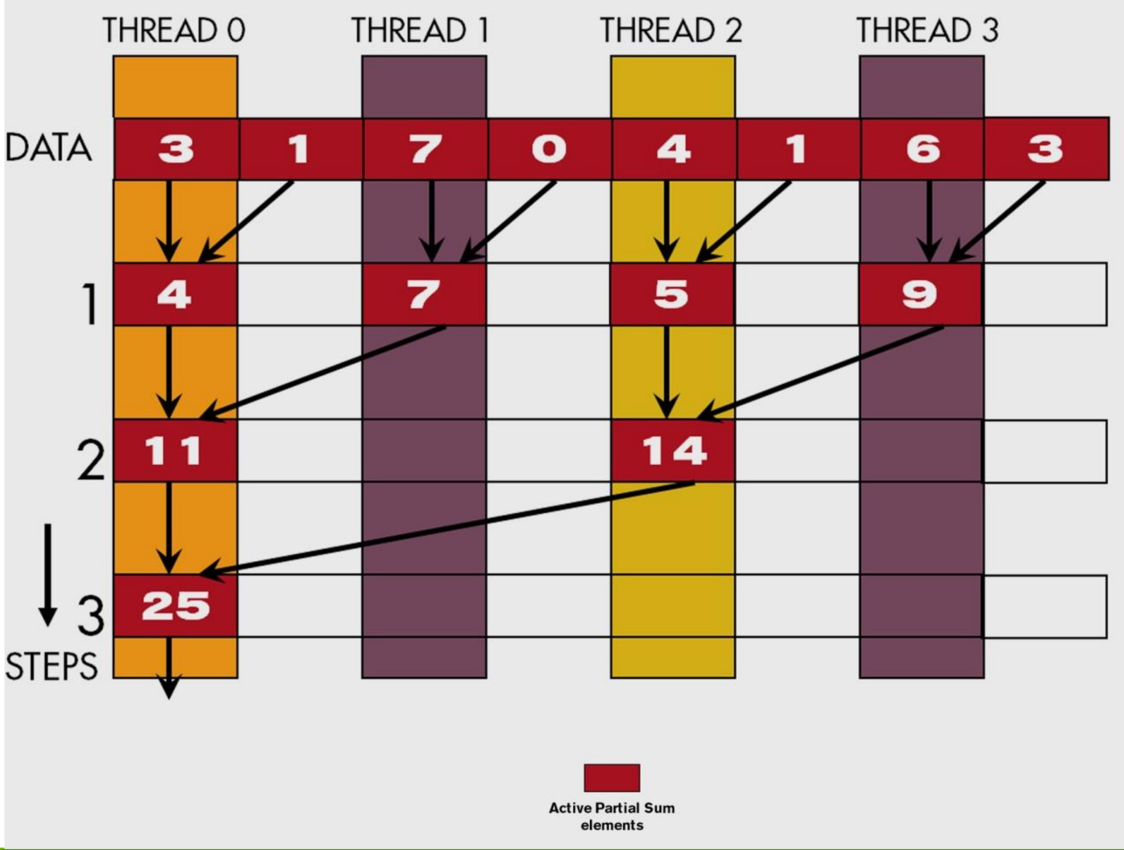
BASIC REDUCTION KERNEL

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

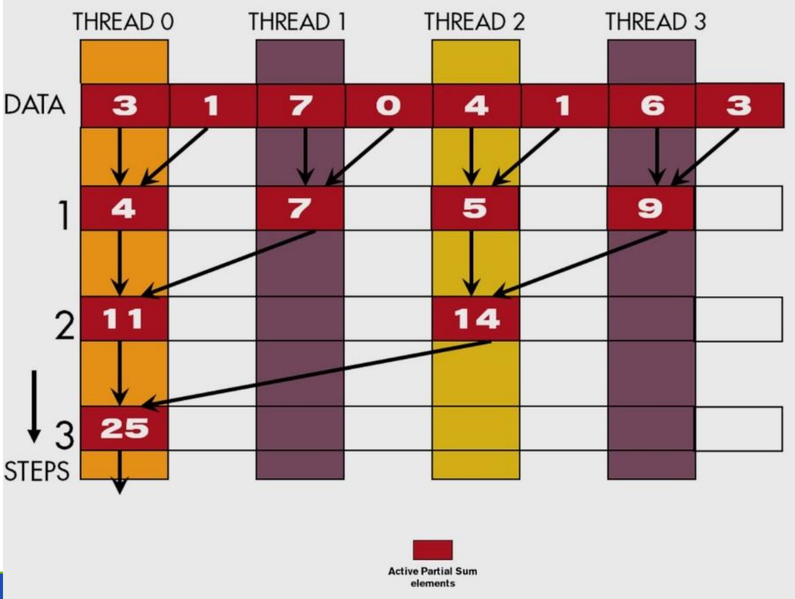


A Parallel Sum Reduction Example



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

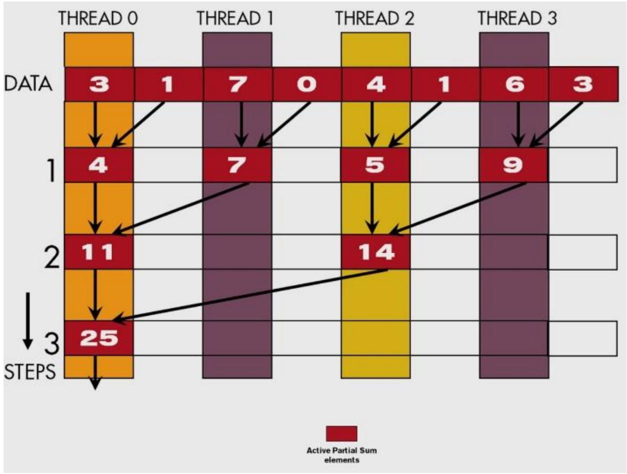
- Each thread block takes 2*BlockDim.x input elements

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

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;
```

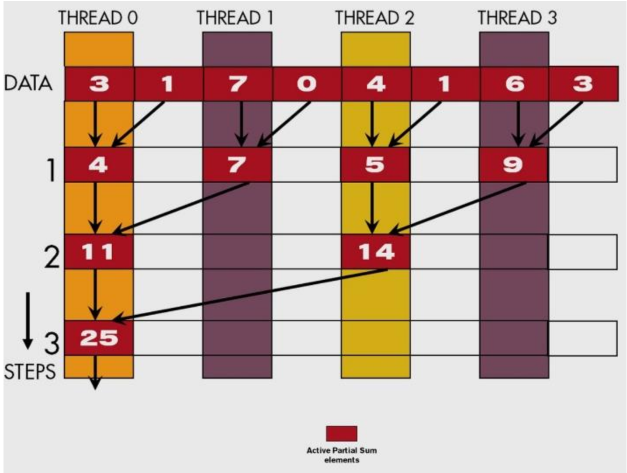
- Each thread loads 2 elements into shared memory

```
partialSum[t] = input[start + t];
partialSum[blockDim.x+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];
}
```

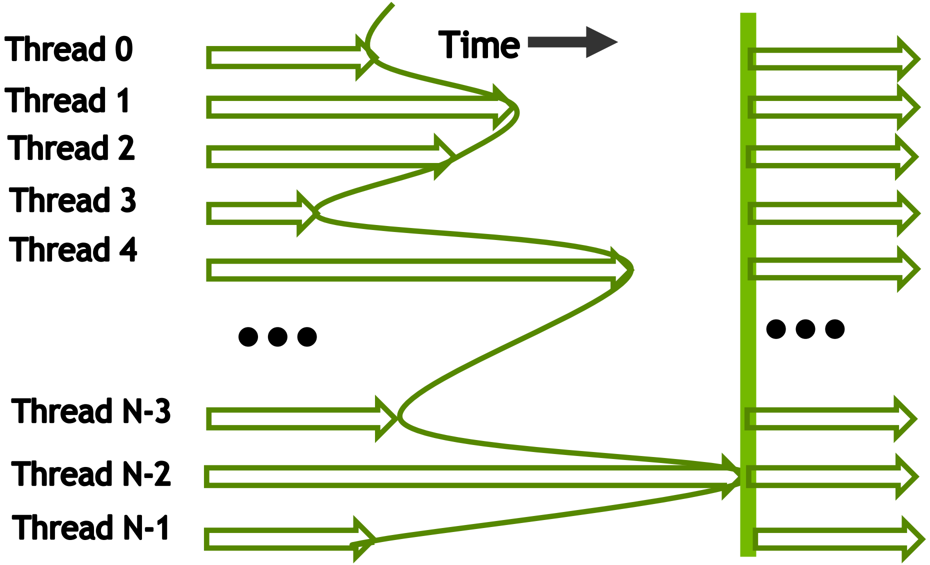


Barrier Synchronization

- __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



Handling partialSum[]

- 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.

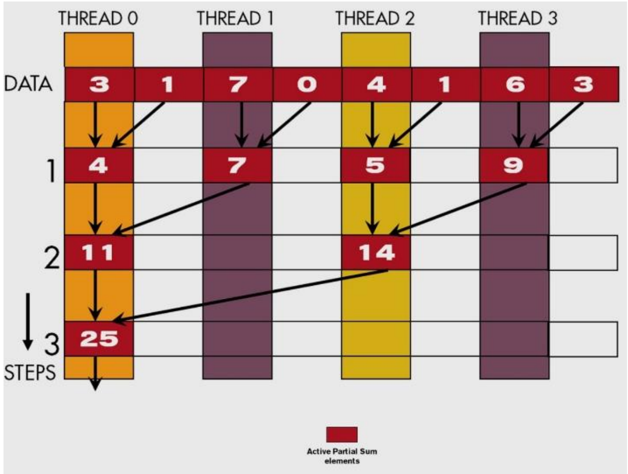
A BETTER REDUCTION MODEL

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

A Quick Analysis

- For a 1024 thread block
- 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

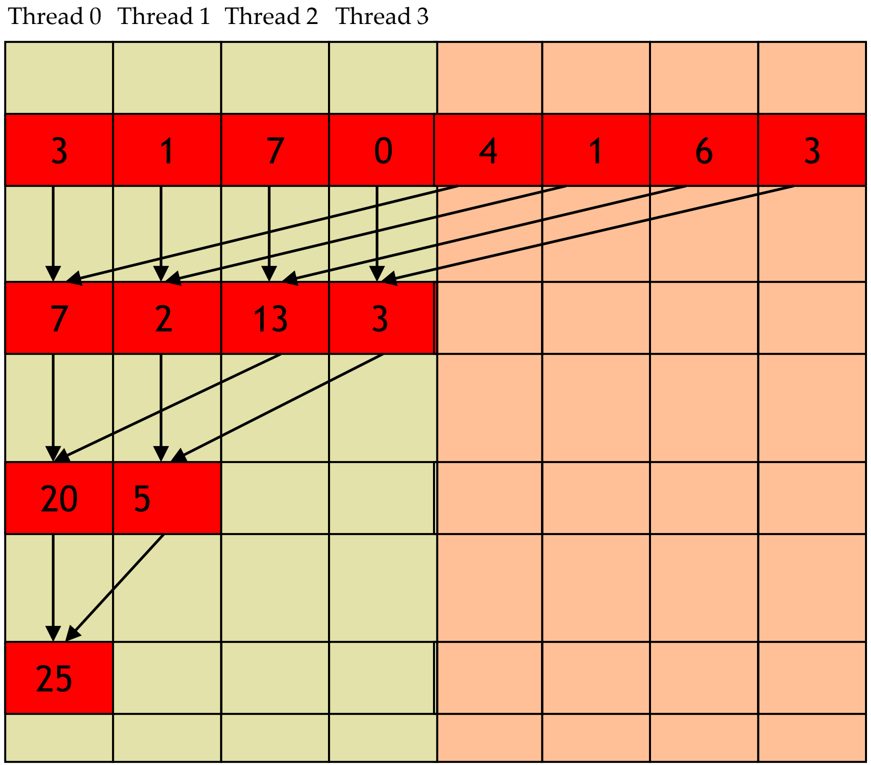


Thread Index Usage Matters

- 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 6 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