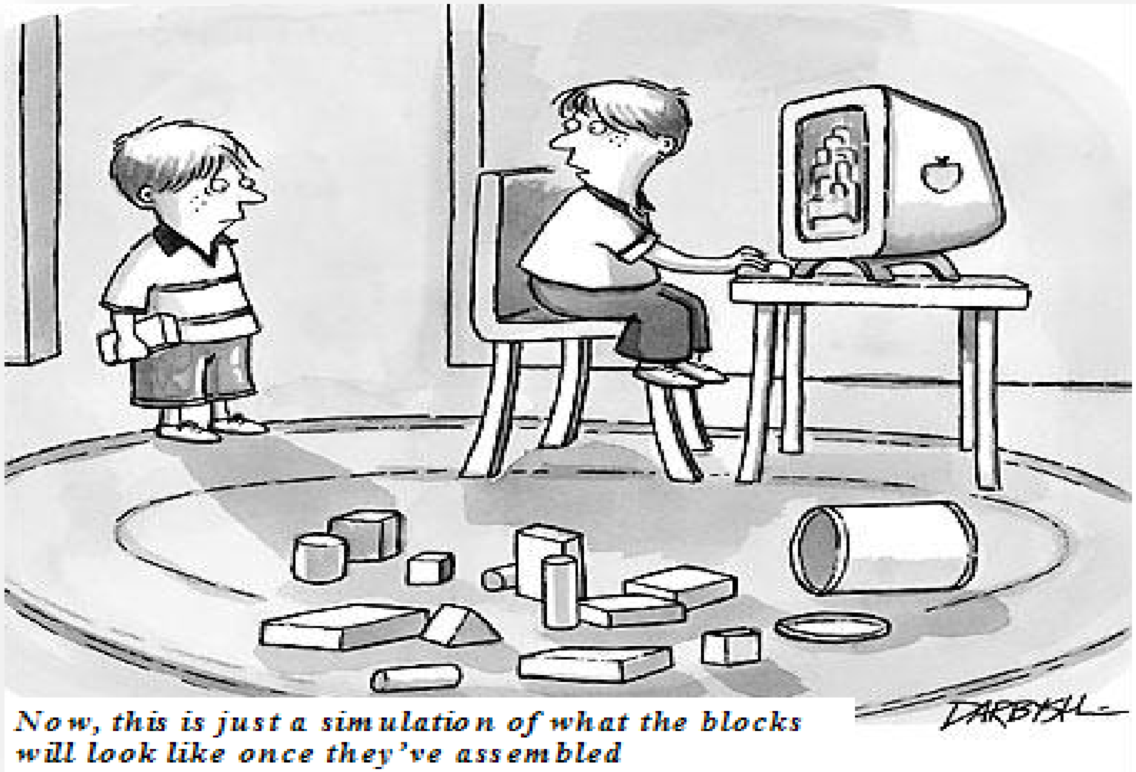
ECE569
Module 40

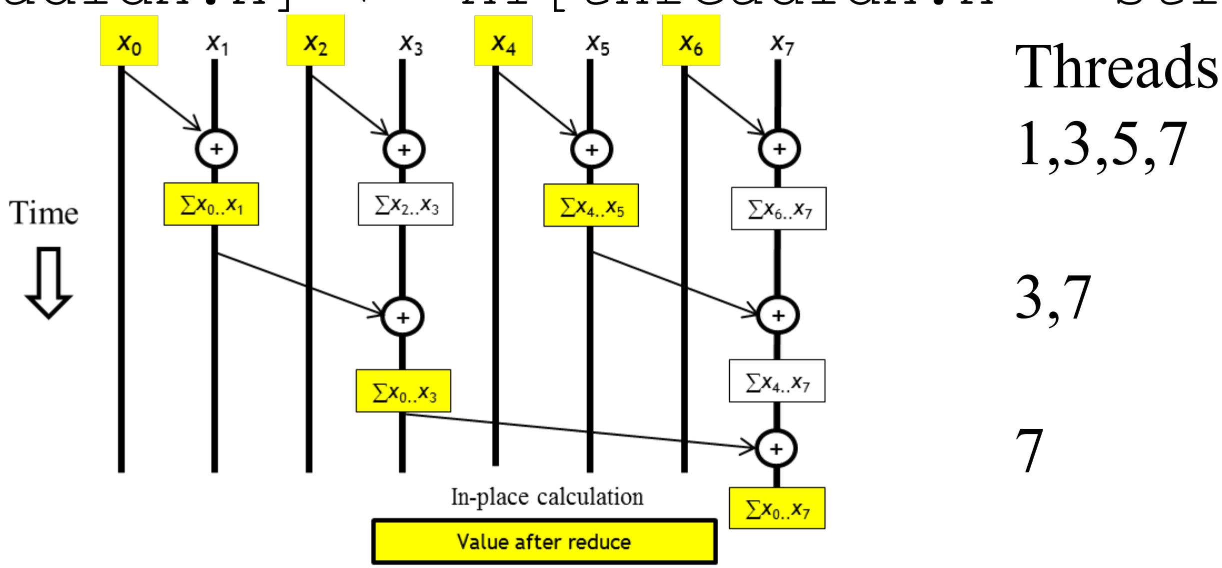


- Scan – Version 3 Implementation with Reduction and Construction

Reduction Phase

- **Stride starts with 1 and doubles in each iteration till we reach block size**
- **For each stride**
  - We need to access shared memory (assume XY) with the thread's current index position and its neighbor on the left side by "stride" distance

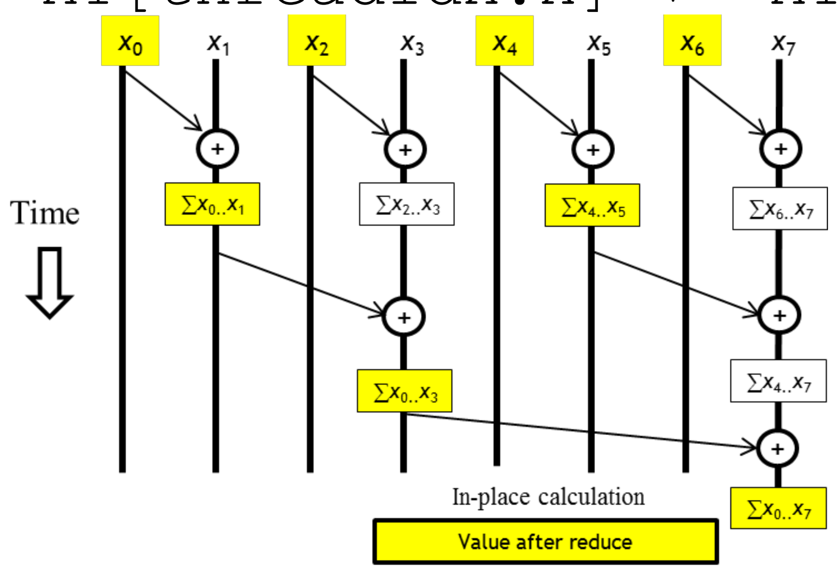
```
if (????)
    XY[threadIdx.x] += XY[threadIdx.x - stride]
```



Reduction Phase

- **Stride starts with 1 and doubles in each iteration till we reach block size**
- **For each stride**
  - We need to access shared memory (assume XY) with the thread's current index position and its neighbor on the left side by "stride" distance

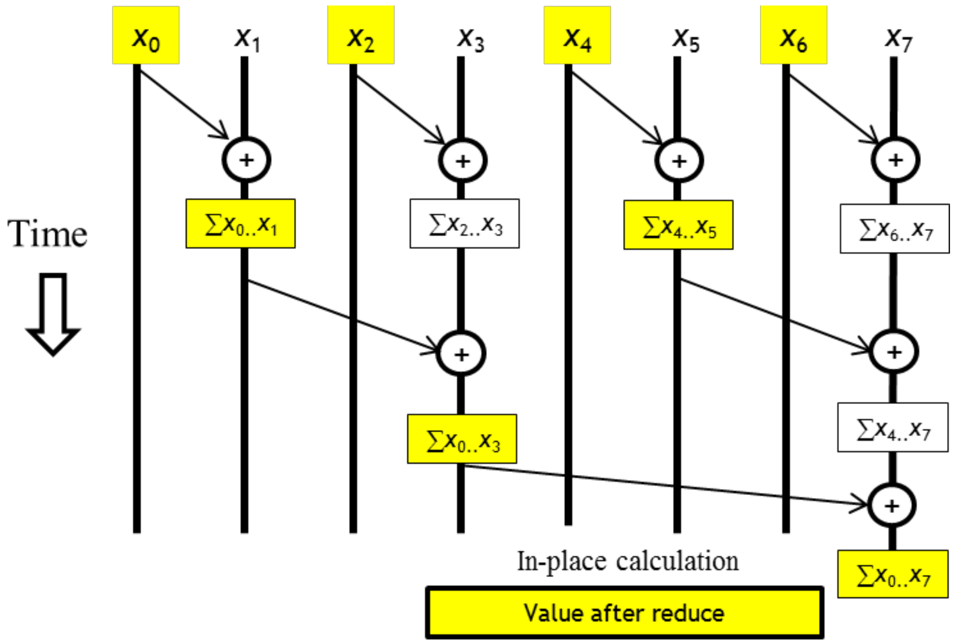
```
if ((threadIdx.x +1)%(2*stride) == 0)
    XY[threadIdx.x] += XY[threadIdx.x - stride]
```



What is the issue with the above if statement?

Reduction Phase

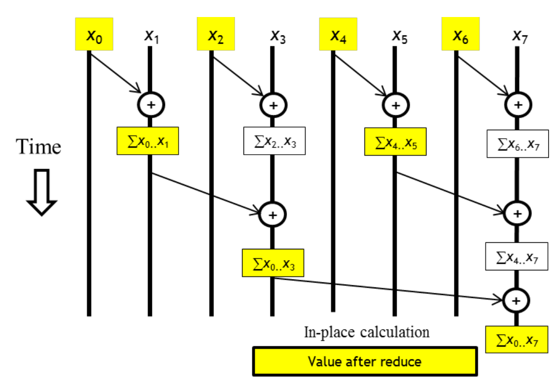
- **What should be the expression for "index"**
  - As a function of "stride" and "threadIdx.x" such that thread divergence is minimized?
    - Subsequent threads should participate in computation
    - Note that for stride=1, we use indexes 1,3,5,7…



index =

Reduction Phase

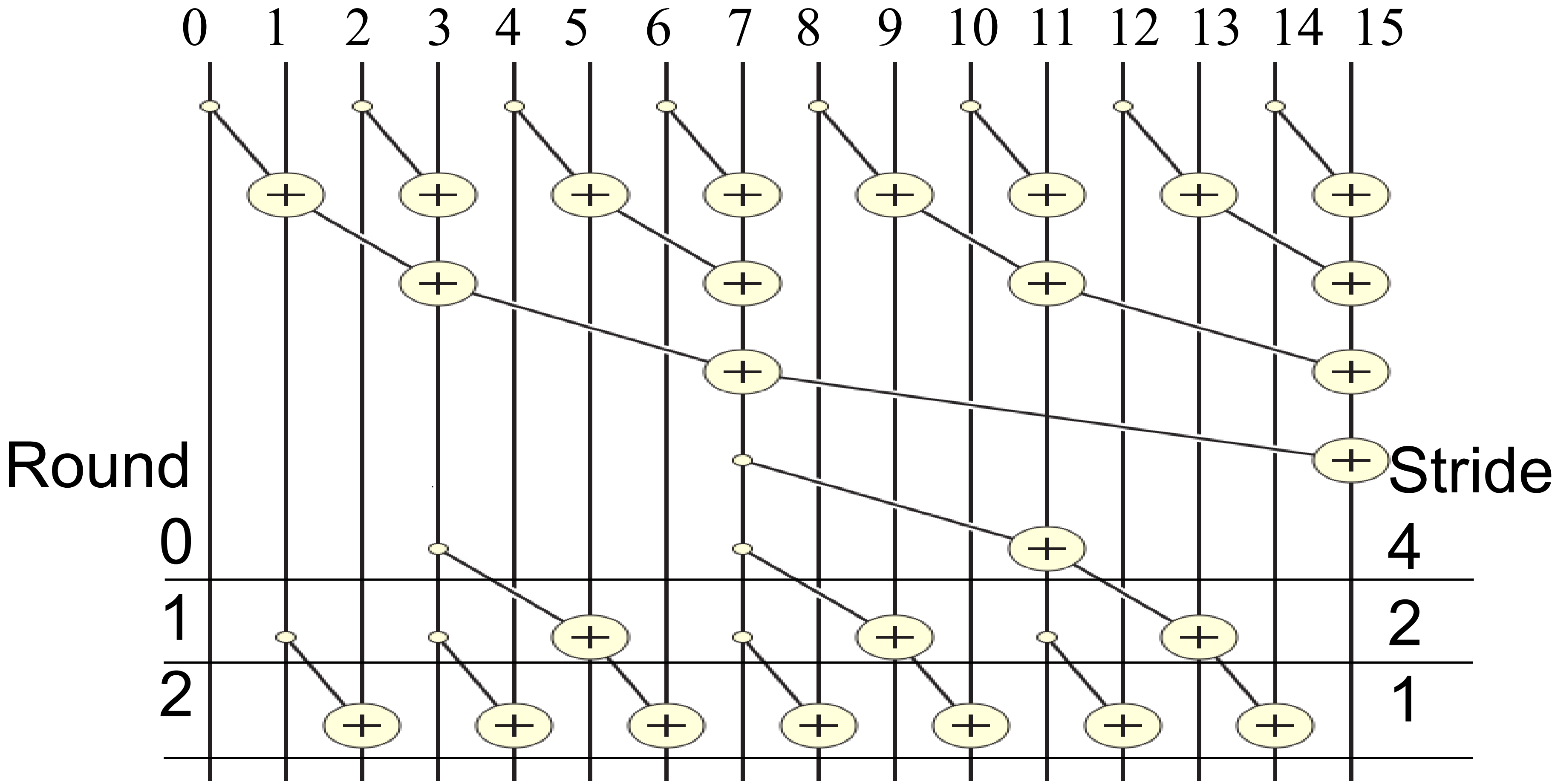
```
// XY[2*BLOCK_SIZE] is in shared memory
// Note that earlier in reduction we reduced the
// block size by half for active threads!
// i=threadIdx.x + blockIdx.x*BlockDim.x;
// tid = threadIdx.x;
int stride;
for (stride = _____;_________________; stride =_______){

   int index = (threadIdx.x+1)*stride*2 - 1;

   if(                             )

      XY[________________] = XY[__________________]

}
// Construction phase starts next
```



Critical Values After Reduction

index = (threadIdx.x+1)*stride*2 - 1;

- **0, 3, 7, 15 has final values after reduction!**

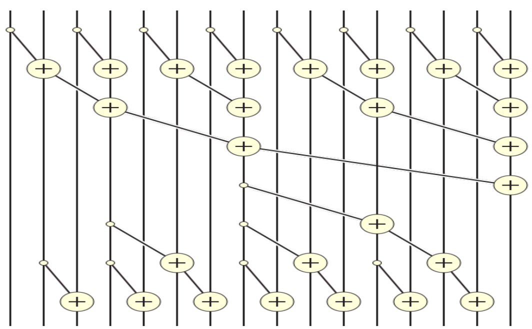


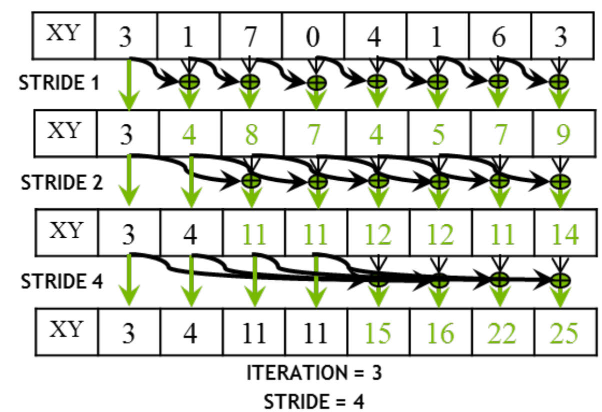
Construction Phase

```
// remember inputs are: X, Y, and InputSize
for(stride =___________; stride________; stride=______){
        __synchthreads();

        index = (tid+1)*stride*2 - 1;

        if(__________________________________) {

           XY[_______________] += XY[__________];
         }
  }
    __synchthreads();
   // update global memory

  if (____________________)

    Y[__________________] = XY[______________];
```

Evaluation

	Step Complexity	Work Complexity
Version 1 (Naïve)	**logN**	**O(N²)**
Version 2	**logN**	**NlogN**
Version 3	**2logN**	**O(n)**





Step count doubled with less work!

One has better step efficiency the other has beater work efficiency.

Scan Papers

- Daniel Horn, Stream Reduction Operations for GPGPU Applications, GPU Gems 2, Chapter 36, pp. 573–589, March 2005.
- Shubhabrata Sengupta, Aaron E. Lefohn, and John D. Owens. A Work-Efficient Step-Efficient Prefix Sum Algorithm. In Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures, pages D–26–27, May 2006
- Mark Harris, Shubhabrata Sengupta, and John D. Owens.Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007.
- Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In Graphics Hardware 2007, pages 97–106, August 2007.
- Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli, “Fast scan algorithms on graphics processors,” in ICS'08: Proceedings of the 22nd Annual International Conference on Supercomputing, 2008, pp. 205–213.
- Shubhabrata Sengupta, Mark Harris, Michael Garland, and John D. Owens. Efficient Parallel Scan Algorithms for many-core GPUs. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, Scientific Computing with Multicore and Accelerators, Chapman & Hall/CRC Computational Science, chapter 19, pages 413–442. Taylor & Francis, January 2011.
- D. Merrill and A. Grimshaw, Parallel Scan for Stream Architectures. Technical Report CS2009-14, Department of Computer Science, University of Virginia, 2009, 54pp.
- Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: fast scan algorithms for GPUs without global barrier synchronization. In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13). ACM, New York, NY, USA, 229-238.

Next

- Applications of Scan
  - Compaction

ECE569
Module 41

Now, this is just a simulation of what the blocks will look like once they've assembled

- Scan Application – Compact

Scan

- Running sum, max, min…
- We know it can be implemented efficiently
- What if we want to process only a subset of an input data?
  - Need a filter so that we throw away the items we don’t care about
  - **Compact** is an interesting application of scan

Compact

- Input: S0 S1 S2 S3 S4
- Predicate: T F T F T (is index even?)
- Output choices:

  S0 _ S2 _ S4 (sparse)

  S0 S2 S4 (dense)

  - We want dense output!
    - Run a compact process and then run fewer number of threads
    - Avoid thread divergence

When do we use compact?

- Compact is most useful when we compact away (Small / Large) number of elements

- And the computation on each surviving element is (Cheap/Expensive)?

Compact Algorithm

- We can do compact in parallel but how do we compute the scatter addresses efficiently in parallel?

Original:

Predicates	T F F T T F T F
	0 _ _ 1 2 _ 3 _

Compact Algorithm

Original:

Predicates T F F T T F T F

0 _ _ 1 2 _ 3 _

Revised:

Predicate array: 1 0 0 1 1 0 1 0

Scan output: **0** 1 1 **1** **2** 3 **3** 4

Scatter output: 0 1 2 3

Steps of Compact

- **Run a predicate on each element**
- **Create a “scan array”**
  - True/False inserting 1 or 0
- **Exclusive sum scan the “scan array”**
  - Create “scan output” array” scatter addresses for compact array
- **Scatter**
  - For each element in the input, if the predicate is true, then **scatter** the input element into the output array at the address in “scan output” array

Analysis of Compact

- **Compact 1M number from 1 to 1M**
  - Operation A: is divisible by 17 (keeps few items)
  - Operation B: is not divisible by 31 (keeps many items)
- **For each phase compare execution time of A and B**
  - Predicate
  - Scan
  - Scatter

Sparse Matrix - Dense Vector Multiplication

- **Pagerank**
  - Web page ranking
  - Largest matrix computation
  - All web pages (N)
    - Form a NxN matrix indicating a link between pairs of web pages (0 or 1)
    - Sparse matrix

Sparse Matrices

a	0	b
c	d	e
0	0	f

X

x
y
z

Compressed Sparse Row Representation
(Helps reconstruct the sparse matrix)

Value (Non zero data):

0	1	2	3	4	5
a	b	c	d	e	f

Column (which column each data from):

a	b	c	d	e	f
0	2	0	1	2	2

Row pointer (for which element index in "Value", each row has a non-zero value):

0	2	5

Sparse Matrices

$$\begin{array}{|c|c|c|}\hline a & 0 & b \\ \hline c & d & e \\ \hline 0 & 0 & f \\ \hline \end{array} \; X \; \begin{array}{|c|}\hline x \\ \hline y \\ \hline z \\ \hline \end{array}$$

Compressed Sparse Row Representation
(Helps reconstruct the sparse matrix)

Value: {a b c d e f}
Column: {0 2 0 1 2 2}
Row pointer: {0 2 5}

1. "Value", "Row pointer" => form a segmented "Value" vector
   {| a b | c d e | f }

Sparse Matrices

$$\begin{bmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Compressed Sparse Row Representation
(Helps reconstruct the sparse matrix)

Value: {a b c d e f}
Column: {0 2 0 1 2 2}
Row pointer: {0 2 5}

1. "Value", "Row pointer" => form a segmented "Value" vector
   {| a b | c d e | f }
2. **Gather** the vector value using "Column" index
   Column index specifies which entry in the dense vector to multiply in each sub-segment
   {| x z | x y z | z}

Sparse Matrices

a	0	b		x
c	d	e	X	y
0	0	f		z

Compressed Sparse Row Representation
(Helps reconstruct the sparse matrix)

Value: {a b c d e f}
Column: {0 2 0 1 2 2}
Row pointer: {0 2 5}

1. "Value", "Row pointer" => form a segmented "Value" vector
   {| a b | c d e | f }
2. **Gather** the vector value using "Column" index
   Column index specifies which entry in the dense vector to multiply in each sub-segment
   {| x z | x y z | z}
3. Pairwise multiply of steps 1 and 2 **(Map operation)**
   a*x b*z | c*x d*y e*z | f*z
4. Accumulate partial products in each segment **(Scan operation)**
   a*x+b*z | c*x + d*y + e*z | f*z

Next

- **CUDA Streams**