Distributed and Parallel Computing

Lecture 1

Alan P. Sexton

University of Birmingham

Spring 2018

Task vs Data parallelism

Two types of parallelism

- Task based: do different operations in parallel
  - e.g. Multiply and Add
  - e.g. Gnu Make: supports running sub-tasks in parallel
  - Suitable for standard multi-core CPUs or networks of computers
- Data based: same operations in lock step on different data in parallel
  - e.g. image ops on every pixel of an image simultaneously
  - e.g. dynamical system simulations
  - Suitable for GPUs

Latency vs Throughput

- Latency oriented processors
  - Get each result back with minimum delay
  - Need 100,000 results?
    - Get first back as soon as possible, even if it slows down getting all back
    - Get first of remaining back as soon as possible ...
    - e.g. an operation takes 4 cycles: 10 operations take $4 + 4 + \cdots + 4 = 40$ cycles
- Throughput oriented processors
  - Get all results back with minimum delay
  - Need 100,000 results?
    - Don't care how long it takes to get the first result back, or the second, ...
    - ...so long as the total time to get all back is as small as possible
    - e.g. Implement the operation with deep pipelining on simple ALUs: 10 cycles for the first operation and to fill the pipeline, but 1 cycle for each following operation: $10 + 1 + 1 + \cdots + 1 = 19$ cycles

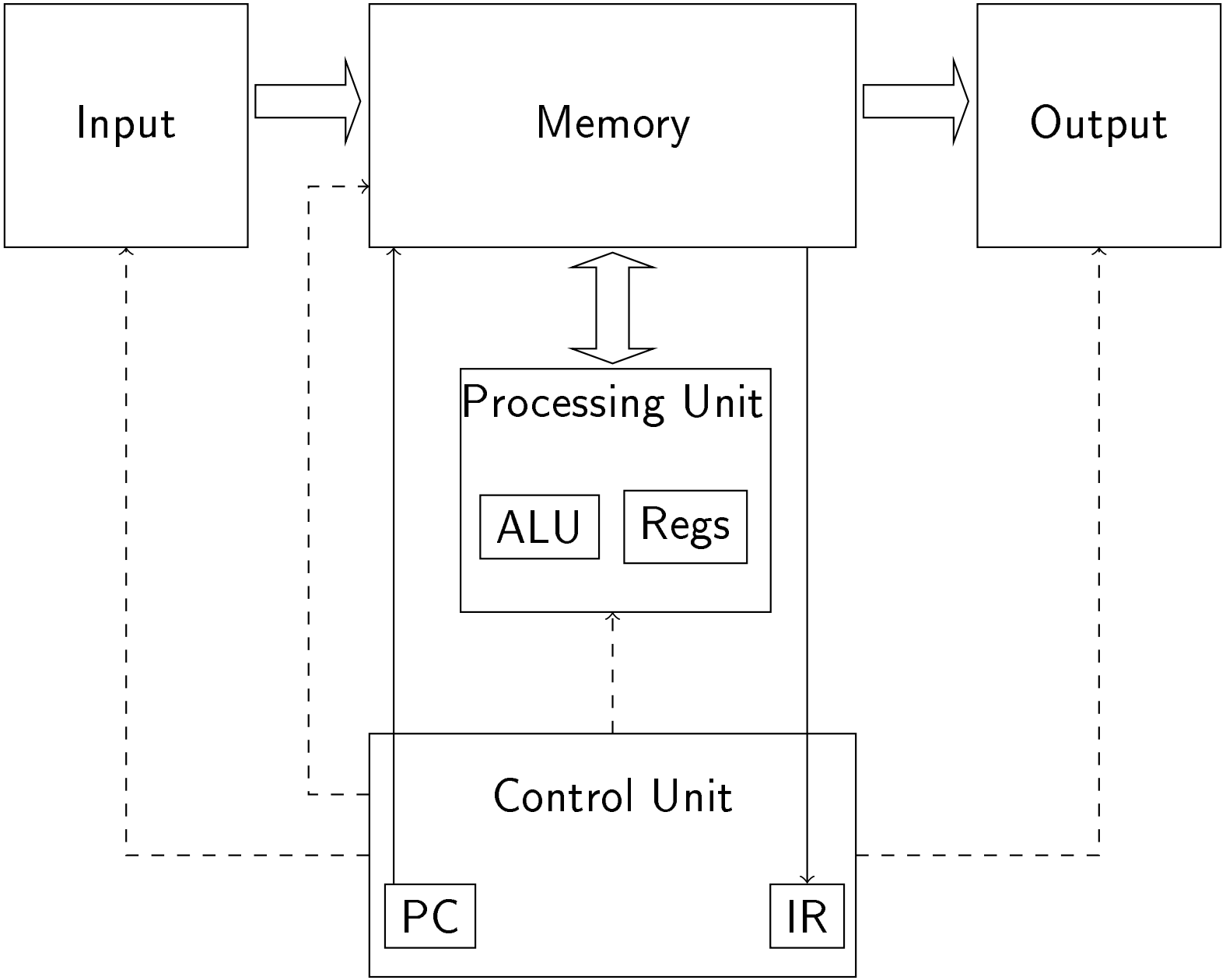
Latency Oriented Processors — standard CPUs

- Large caches to speed up memory access
  - Temporal/Spatial locality, Working sets
    - Try to ensure that each operation has the smallest probability possible of having to fetch from slow memory
- Complex control units
  - Short pipelines, Branch prediction, Data forwarding
    - Jump through hoops to make each instruction finish as quickly as possible with minimal pipeline stalls
- Complex, energy expensive ALUs
  - Large complex transistor arrangements
    - Minimize # Clock Cycles per operation to get result as quickly as possible

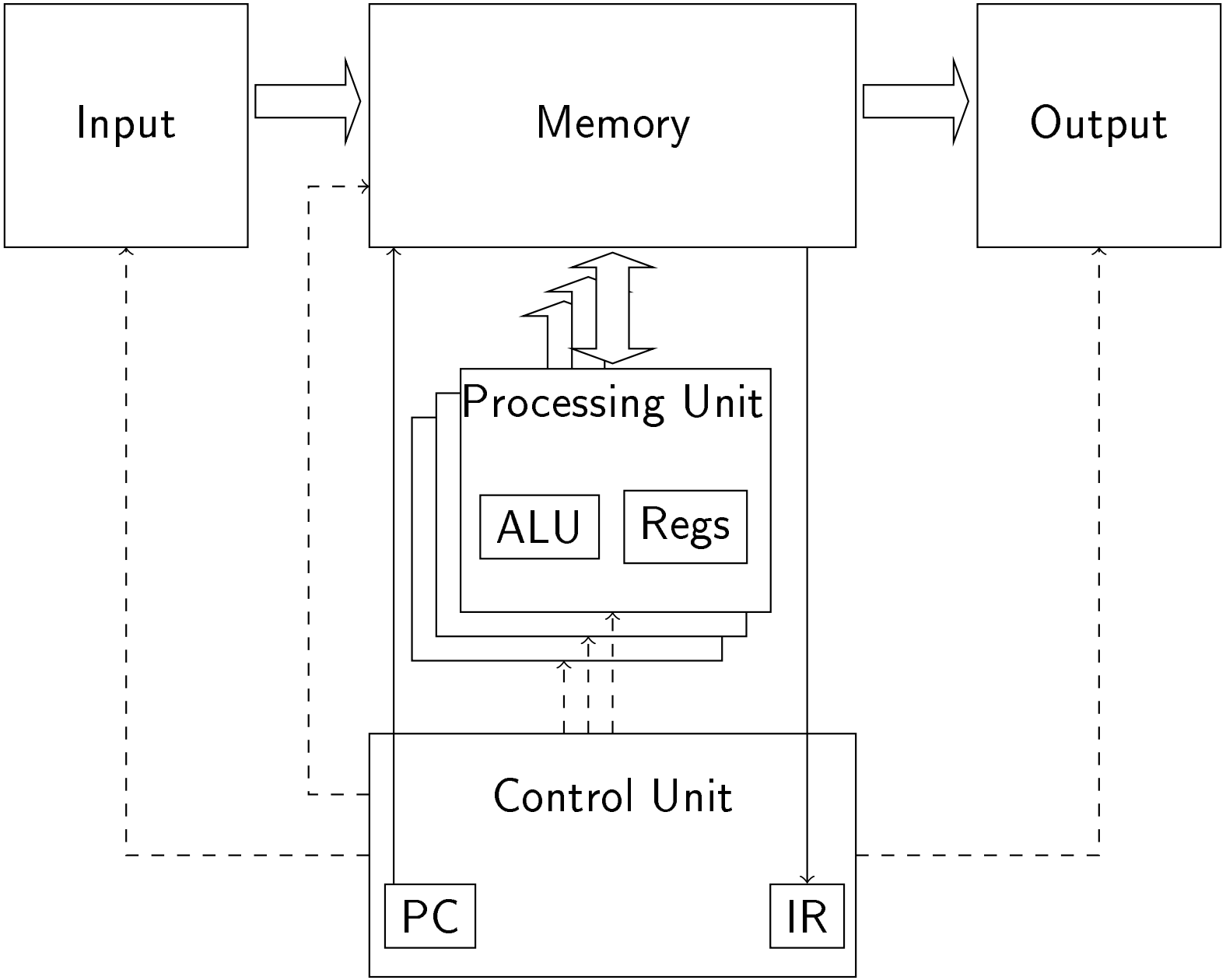
Throughput Oriented Processors — GPUs

- Small caches
  - **NOT** for keeping temporally or spatially located data around
  - For *staging* data
    - Get blocks of data in one go for groups of threads to work on
    - Avoids each thread having to do separate fetches
- Simple control units
  - No branch prediction or data forwarding
  - Control shared across multiple threads operating on different data
- Simple energy efficient ALU
  - Long pipeline
  - Large # cycles per operation but heavily pipelined
    - Long wait for first result (filling the pipeline) but following results come very quickly
  - Requires large numbers of threads to keep the processor occupied

Von Neumann Architecture — standard CPU

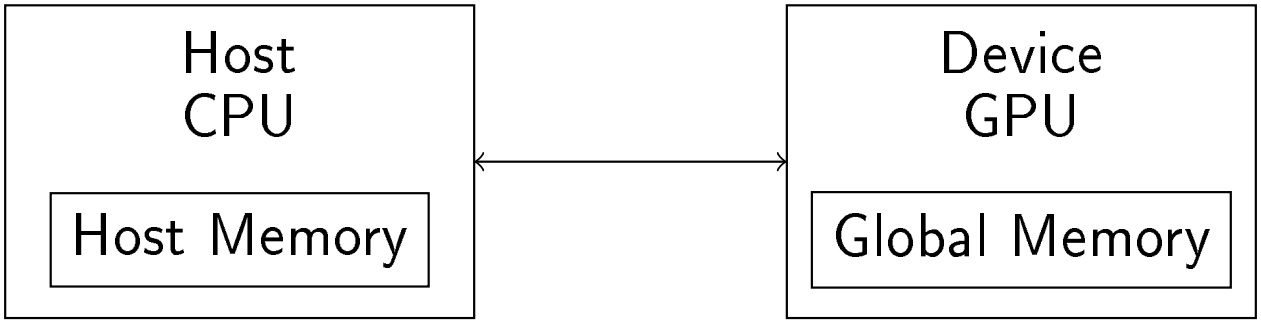


Modified Von Neumann Architecture — GPU



Compiling for CUDA

Conceptually, the host CPU and the GPU are separate devices connected by a communication path:



- Thus we need to generate separate code for each device. The NVidia compiler for CUDA programs is *nvcc*
- nvcc takes a C or C++ program with NVidia extensions, separates out and compiles the GPU Device code itself, and separates out the Host code and passes it to the local host compiler to be compiled.
- The single resulting binary contains both the host and the device binary, which is downloaded to the Device from the Host when the program runs
- While you can compile with `nvcc` just as with `gcc` or `g++`, it is easier to use `nsight`, the NVidia modified, CUDA enabled Eclipse IDE.

VectorAdd

As a first CUDA code example, we will look at adding two vectors:

$$C = A + B$$

In sequential C, the code might look like this:

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    for (i = 0; i < n ; i++)
        h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Declare and set value for N
    // Declare and allocate memory for h_A, h_B and h_C
    // Populate h_A and h_B with data
    vecAdd(h_A, h_B, h_C, N);
    ...
}
```

Programming in CUDA

We have to identify whether a function runs on the host, the device or both, and where it is callable from:

- the host: `__host__ void f(...)`
  - This is the default so can be omitted
  - Callable from the host only
- the device: `__global__ void f(...)`
  - These special functions are called *kernel functions*
  - Callable from the host only, hence this is how the host gets to run code on the GPU
- the device: `__device__ void f(...)`
  - Callable from the device only. Hence they are helper function available to kernel functions and other device functions on the GPU
- both: `__host__ __device__ void f(...)`
  - This generates both a host function and a device (i.e. helper) function so the same code can be run on both. The host can not call the device version or vice versa

GPU computational unit structure

When we call a kernel function, we need to specify how the threads should be organised to execute it. Note:

- Every thread is going to execute the same kernel. For our vector addition example, we will get each thread to add a single element of A to a single element of B to produce a single element of C.
- Different GPU devices can support different numbers of simultaneously executable threads.
- Within a GPU, it would be nice if we could make the thread structure uniform: i.e. if every processing unit had equal access to all the GPU memory, could synchonise equally with any other processing unit and could share all the GPU cache memory. However, we could only do this by sacrificing a huge proportion of potential computational power.

CUDA thread issues

- We don't want the GPU fixed number of threads to dictate the size of the largest vectors we can add
- We don't want to have to change our code to run the program on a different GPU with a different number of threads
- We need to be able to organise our threads to co-locate groups of threads on sets of processing units to take advantage of shared caches and synchronisation facilities.

NVidia GPUs accomplish this by organising threads into a hierarchical structure:

- A *Grid* is a collection of *Blocks*
- A *Block* is a collection of *Threads*
- A *Thread* is the execution of a *kernel* on a single processing unit

CUDA thread organisation

Outside the Grid/Block/Thread hierarchy, there is the concept of a *Warp*

- A *warp* is a set of a number of tightly related threads that must execute fully in lock step with each other.
- Warps are not part of the CUDA specification, but are a feature of all NVidia GPUs, dictated by low level hardware design
- The number of threads in a warp is a feature of a particular GPU, but with current GPUs it is almost always 32
- Warps are the low-level basis of thread scheduling on a GPU. If a thread is scheduled in to execute, all the other threads in that warp are scheduled in too
- As they execute the same instructions in lock step, all threads in a warp will have exactly the same instruction execution timing

Grid/Block/Thread

- A block can have between 1 and the maximum block size number of threads for that GPU device (typically 1024 for our machines) and is the high-level basis for thread scheduling
- Because of the nature of warps, the block size should be a multiple of the warp size. Otherwise blocks will be padded with the remaining threads in the partially used warp and wasted
- Grids can have very large numbers of blocks, many more than can be executed at once

Grid/Block/Thread

- The grid corresponds to the whole problem (vector add of 50,000 elements) divided up into bitesize blocks (e.g. vector add of 1024 elements)
- a GPU that can execute one block at a time (1024 threads) would schedule $\lceil 50,000/1024 \rceil$ blocks, one at a time, hence a minimum of 48 schedulings
- a GPU that can execute three blocks at a time (1024 threads) would schedule $\lceil 50,000/1024 \rceil$ blocks, three at a time, hence a minimum of 17 schedulings

Invoking Kernel Functions

We need to specify the grid/block structure when invoking a kernel function

```
...
int threadsPerBlock = 256;
int blocksPerGrid = 1 + (numElements-1) / threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
     numElements);
...
```

Note that the "`<<<blocksPerGrid, threadsPerBlock>>>`" is not standard C or C++, but is handled by the `nvcc` compiler.

Inside Kernel Functions

Each thread needs to know which part of the data to work on. CUDA provides predefined variables for this purpose:

- `blockIdx.x`: the unique identifier of this block in this grid
- `blockDim.x`: the number of threads in a block for this grid
- `threadIdx.x`: the unique identifier of this thread in this block (between 0 and blockDim.x - 1)

```
__global__ void
vectorAdd(const float *A, const float *B, float *C, int n)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < n)
    {
        C[i] = A[i] + B[i];
    }
}
```

Grid and Block Dimensionalities

- Grids and Blocks can be organised as 1-dimensional (suitable for our vector add example), 2-dimensional (suitable for operating on pixels of an image) or 3-dimensional (suitable for operating 3-dimensional spatial simulations)
- Hence the predefined variables, `blockIdx.x`, `blockDim.x`, `threadIdx.x` have ".y" and ".z" variants as well.
- If you are using one dimensional grids and blocks (i.e. you set the grid and block size of the kernel using simple integers), then you can ignore the ".y" and ".z" variants

Device Global Memory

Memory on a GPU is not (currently) shared with the Host machine. Hence the host has to copy data to the device and copy it back when the kernel finishes

Before doing so, the host must allocate global memory on the device and, afterwards, free it again, just like `malloc` and `free` in C

Note that, unlike `malloc`, the return value of `cudaMalloc` is an error number

```
float *h_A = (float *)malloc(size);
float *h_C = (float *)malloc(size);
float *d_A = NULL;
err = cudaMalloc((void **)&d_A, size);
float *d_C = NULL;
err = cudaMalloc((void **)&d_C, size);
...
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
... // invoke kernel ...
err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
...

err = cudaFree(d_A);
err = cudaFree(d_C);
```

CUDA Error Handling

The only way to check that things are working correctly on the GPU is to check the error return values. *ALWAYS* check them *EVERY* time.
The standard code to check error numbers is:

```
if (err != cudaSuccess)
{
    fprintf(stderr,
            "SUITABLE ERROR MESSAGE (error code %s)!\n",
            cudaGetErrorString(err));
    exit(EXIT_FAILURE);
}
```

CUDA Error Handling

- Kernel functions don't return error numbers. However after it has finished, you can call `err = cudaGetLastError();` to get the error number if an error occurred.
- Since kernel functions can run in parallel with host functions, if you call `cudaGetLastError()` before the kernel function finishes, the error may only occur after you requested the error, leading to a confusing error later on.
- If you really want to be sure it finished correctly (either to check for errors or because you are timing it, or you want to extract the final results from the device), call `cudaDeviceSynchronize()`
- Normally you should not unnecessarily call `cudaDeviceSynchronize()`, because allowing a sequence of kernel calls to work without unnecessary extra synchronisation is more efficient.

Timing Host Code with Host Timers

Typically we will want to time both the sequential version of the code that runs on the Host CPU and the parallel version that runs on the GPU.
The general timing approach for host code is as follows:

```
#include <cuda_runtime.h>
#include <helper_cuda.h>
#include <helper_functions.h>

StopWatchInterface *timer = NULL;
sdkCreateTimer(&timer);                    // create a timer
sdkStartTimer(&timer);                      // start the timer

/* The Host code that is to be timed */

sdkStopTimer(&timer);
double h_msecs = sdkGetTimerValue(&timer);
sdkDeleteTimer(&timer);
```

Timing Host+GPU Code with Host Timers

If using host timers to time GPU code, recall that GPU code runs asynchronously with host code, so make the host wait until the GPU has finished before stopping the timer:

```
#include <cuda_runtime.h>
#include <helper_cuda.h>
#include <helper_functions.h>

StopWatchInterface *timer = NULL;
sdkCreateTimer(&timer);                    // create a timer
sdkStartTimer(&timer);                      // start the timer

/* The Host+GPU code that is to be timed */

cudaDeviceSynchronize();

sdkStopTimer(&timer);
double h_msecs = sdkGetTimerValue(&timer);
sdkDeleteTimer(&timer);
```

Timing GPU Code with GPU Timers

The best way to time GPU code, is to insert an **event** into the GPU execution stream before and after the code to time and get the elapsed time from them:

```
cudaEvent_t start, stop;
float d_msecs;

cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord( start, 0 );

/* Call GPU kernel(s) */

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

cudaEventElapsedTime( &d_msecs, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Time will be in milliseconds with a resolution of approximately 0.5 milliseconds

Distributed and Parallel Computing

Lecture 02

Alan P. Sexton

University of Birmingham

Spring 2018

Measuring Parallel Speedup

- **Latency**: the time from initiating to completing a task
  - Units of time
- **Work**: a measure of what the CPU/GPU has to do for a particular task
  - number of floating point operations
  - number of images processed
  - number of pixels processed
  - number of simulation steps
- **Throughput**: work done per unit time

Speedup and Efficiency

- **Speedup$_P$, S_P**: The ratio of the latency for solving a problem with 1 hardware unit to the latency of solving it with P units
  - $S_P = \frac{T_1}{T_P}$
  - Perfect linear speedup: $S_P = P$
- **Efficiency, E_P**: The ratio of the latency for solving a problem with 1 hardware unit to P times the latency of solving it on P units
  - This measures how well the individual hardware units are contributing to the overall solution
  - $E_P = \frac{T_1}{P \times T_P} = \frac{S_P}{P}$
  - Perfect linear efficiency: $E_P = 1$

Interpreting speedup and efficiency

- Sub-linear speedup and efficiency is normal
  - Overhead in parallelizing a problem
- Super-linear speedup is possible, but usually due to special conditions
  - e.g. Serial version does not fit in CPU cache but each of the parallel sub-problems do.
- Important to compare the best serial version of the program with the parallel version
  - Serial algorithm A is fast but hard to parallelize
  - Serial algorithm B is slow but gives Parallel algorithm B
  - To measure speedup/efficiency, compare Serial A to Parallel B
  - Both must solve the same problem, but allow for minor differences
    - e.g. small round-off differences (but be aware of differing precision on host and on GPU!)
    - differences due to different execution order

Strong Scalability

Gene Amdahl, in 1967, argued that the time spent executing a program is composed of the time spent doing non-parallelizable work plus the time spent doing parallelizable work:

$$T_1 = T_{\text{ser}} + T_{\text{par}}$$

Therefore, if the speedup on P units of the parallel part only is s

$$T_P = T_{\text{ser}} + \frac{T_{\text{par}}}{s}$$

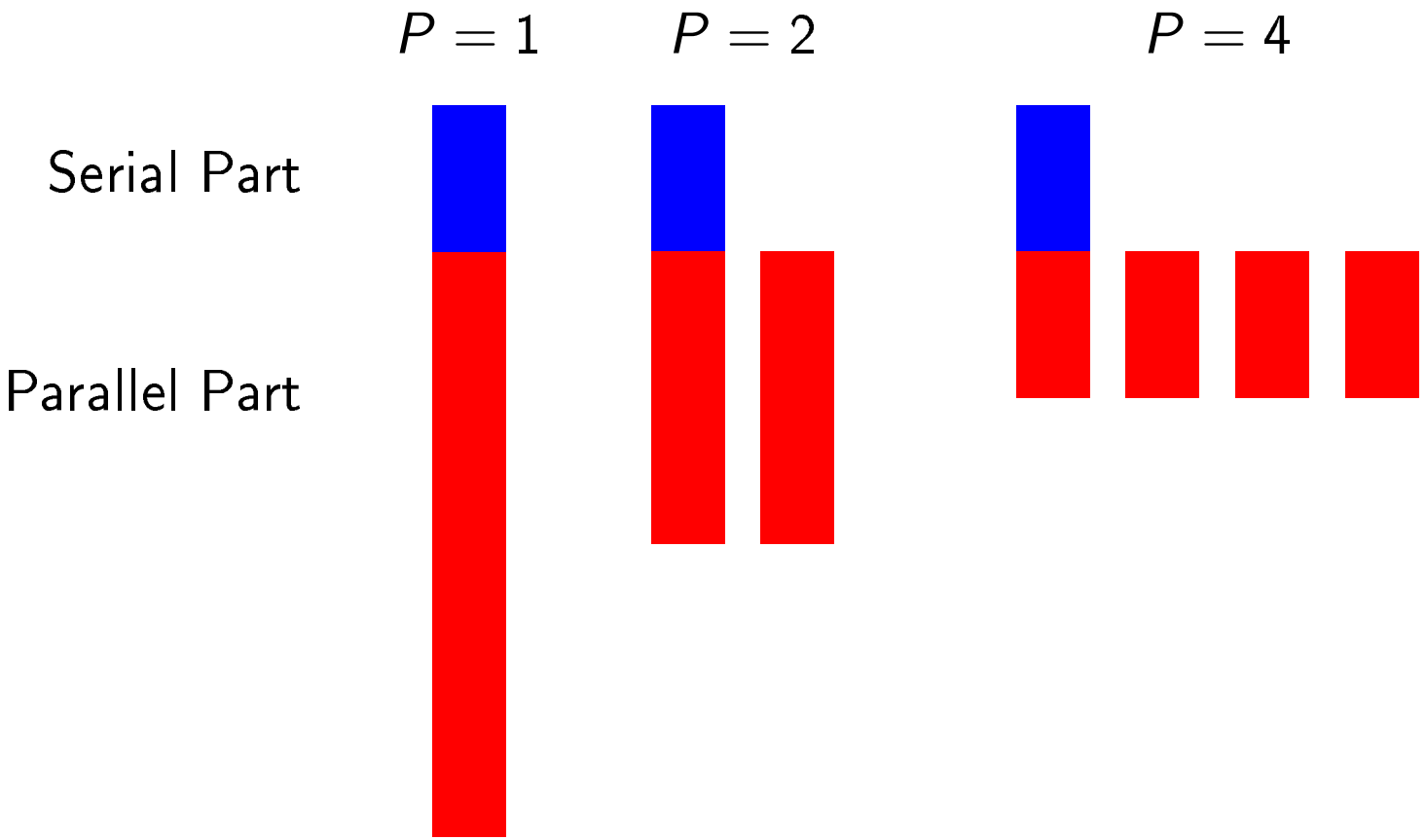
Hence the overall speed up given the speedup of the parallel part is s is:

$$S_P = \frac{T_{\text{ser}} + T_{\text{par}}}{T_{\text{ser}} + \frac{T_{\text{par}}}{s}}$$

If we let f be the fraction of a program that is parallelizable, then $T_{\text{ser}} = (1-f)T_1$ and $T_{\text{par}} = fT_1$. Hence

$$S_P = \frac{(1-f)T_1 + fT_1}{(1-f)T_1 + \frac{fT_1}{s}} = \frac{1}{1-f+\frac{f}{s}} \qquad \text{(Amdahl's Law)}$$

Amdahl's Law Graphically



Interpreting Amdahl's Law

Amdahl's law seems to say that there is a limit to parallel speedup

$$\lim_{s \longrightarrow \infty} S_P = \lim_{s \longrightarrow \infty} \frac{1}{1 - f + \frac{f}{s}} = \frac{1}{1 - f}$$

or, in other words

$$\lim_{s \longrightarrow \infty} \frac{T_1}{T_P} = \frac{1}{1 - f}$$

$$\Rightarrow \lim_{P \longrightarrow \infty} T_P = T_{\text{ser}} \qquad \text{assuming } P \rightarrow \infty \Rightarrow s \rightarrow \infty$$

Weak Scalability

John Gustafson and Edwin Barsis, in 1988, argued that Amdahl's law did not give the full picture

- Amdahl kept the task fixed and considered how much you could shorten the processing time by running in parallel
- Gustafson and Barsis kept the processing time fixed and considered how much larger a task you could handle in that time by running in parallel
- This was motivated by observing that, as computers increase in power, the problems that they are applied to often increase in size

Gustafson-Barsis Law

Assume that W is the workload that can be executed without parallelism in time T. If f is the fraction of the workload that is parallelizable, then

$$W = (1 - f)W + fW$$

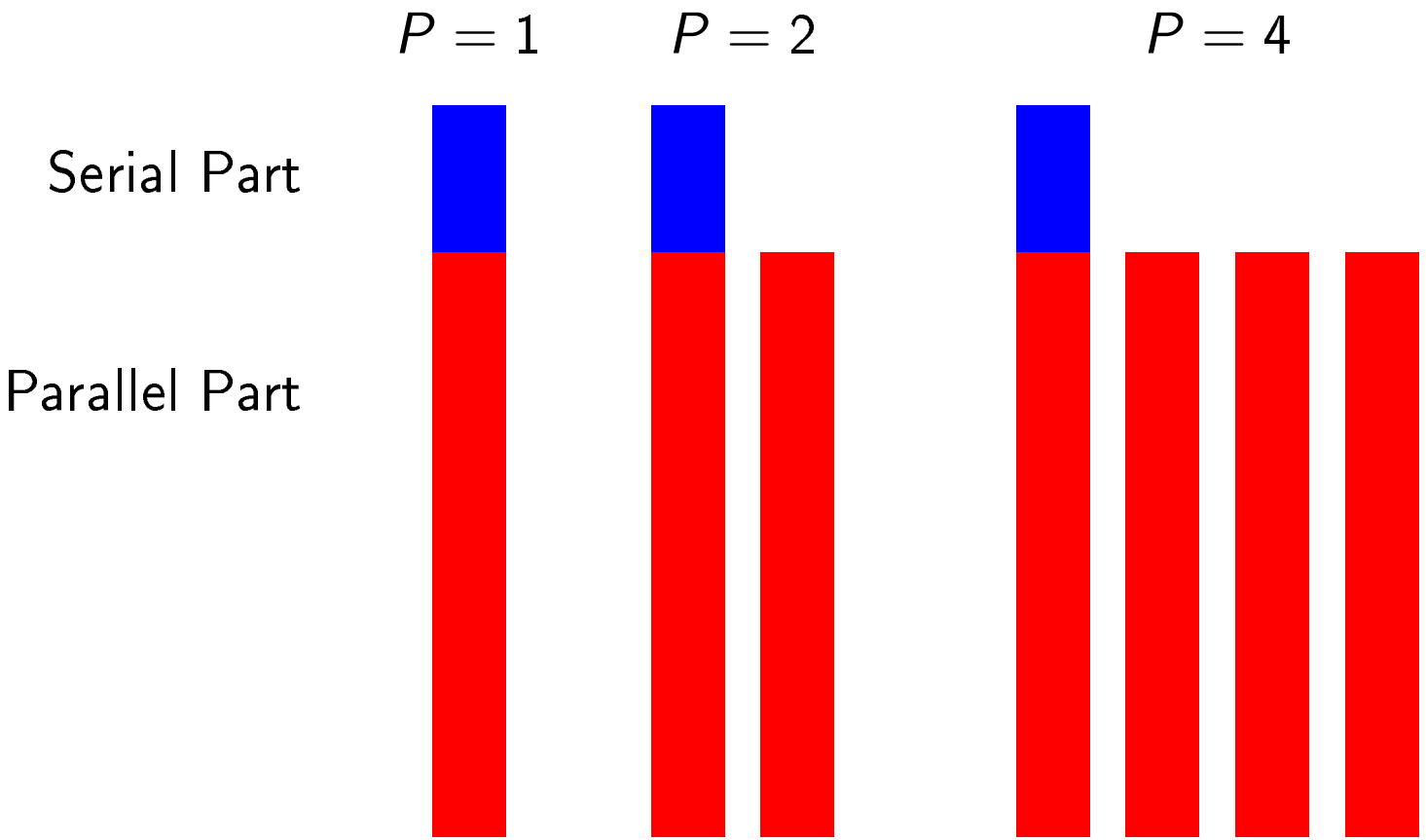
With speedup s, we can run s times the parallelizable part in the same time, although we don't change the amount of work done in the non-parallelizable part:

$$W_s = (1 - f)W + sfW$$

If we do W_s in time T, we are, on average, doing W amount of work in time $\frac{TW}{W_s}$. Hence the total speedup is:

$$S_s = \frac{T}{TW/W_s} = \frac{W_s}{W} = 1 - f + fs \qquad \text{(Gustafson-Barsis)}$$

Gustafson-Barsis Law Graphically



Conclusions

- Both Amdahl (A) and Gustafson-Barsis (GB) are correct
- The two together give guidance on what kinds of problems can benefit from parallelization and how
- You can only go so much faster on a fixed problem by parallelization
- You can avoid Amdahl's limit on speedup if you can increase the size of the parallelizable part of the problem faster than you increase the size of the non-parallelizable part

Distributed and Parallel Computing

Lecture 03

Alan P. Sexton

University of Birmingham

Spring 2018

SMs, Cores and Warps

A GPU has a number of *Streaming Multiprocessors (SMs)*, which have a number of *Cores*. Threads are scheduled in units of *Warps*. Each SM has a number of resources. For example, each SM of a Fermi architecture GPU (GeForce 400 and 500 series) might have 2 instruction dispatch units and

- 3 banks of 16 cores (i.e. 48 cores)
- 1 bank of 16 Load Store Units (for calculating source and destination addresses for 16 threads per clock cycle)
- 1 bank of 4 Special Functional Units (hardware support for calculating sin, cos, reciprocals and square roots - a warp executes over 8 clocks).
- 1 bank of 4 Texture Units

The 2 instruction dispatch units can start, on each clock cycle, processing on any 2 banks at a time. The 3 banks of 16 cores means that 2 sequential instructions from one thread warp can be executing simultaneously if they are not dependent on each other (superscalar instruction parallelism)

Synchronisation

Conceptually, Warps execute in lock step, so synchronisation within a warp is (mostly) automatic but can be tricky (data variables should be marked *volatile*)

- In practice, it is much more complicated.
- We need to be able to synchronise threads in a block
- Also need to be able to synchronise threads in a warp
- Cannot synchronise across different blocks
- **Barrier synchronisation**: `__syncthreads()`
- Must **NEVER** have `__syncthreads()` in a branch of a conditional that some threads in the block will not execute
  - Otherwise deadlock may occur!
- Cannot even fix it by making sure each branch has a `__syncthreads()` call: the different calls are **NOT** necessarily to the same barrier!

More on Warps

For the threads in a block,

- Warp 0 consists of threads 0 to 31
- Warp 1 consists of threads 32 to 63
- ...

For threads in a multi-dimensional block

- Multidimensional threads are linearized in row-major order
- Thus all the threads with z value 0 come first, followed by those with z value 1, etc.
- Within each group of threads with the same z value, the threads with y value 0 come first, followed by those with y value 1 etc.
- Within each group of threads with the same z and y value, the thread with x value 0 comes first followed by that with x value 1 etc.
- Within this linearized order, the first 32 threads belong to the first warp etc.

Warp Execution and Divergence

The whole warp is handled by a single controller. Consider what happens if some threads (A) in a warp take one branch (1) of an `if` statement, and others (B) take the other branch (2):

- All threads in the warp must execute the same instructions
- First execute branch 1: all threads execute the branch 1 instructions but the B threads are disabled (think of de-clutching in a car) so they have no effect.
- Then execute branch 2: all threads execute the branch 2 instructions but now the A threads are disabled.
- This is called a *divergence*
- Whole warp execute same branch $\Rightarrow$ *NO DIVERGENCE*

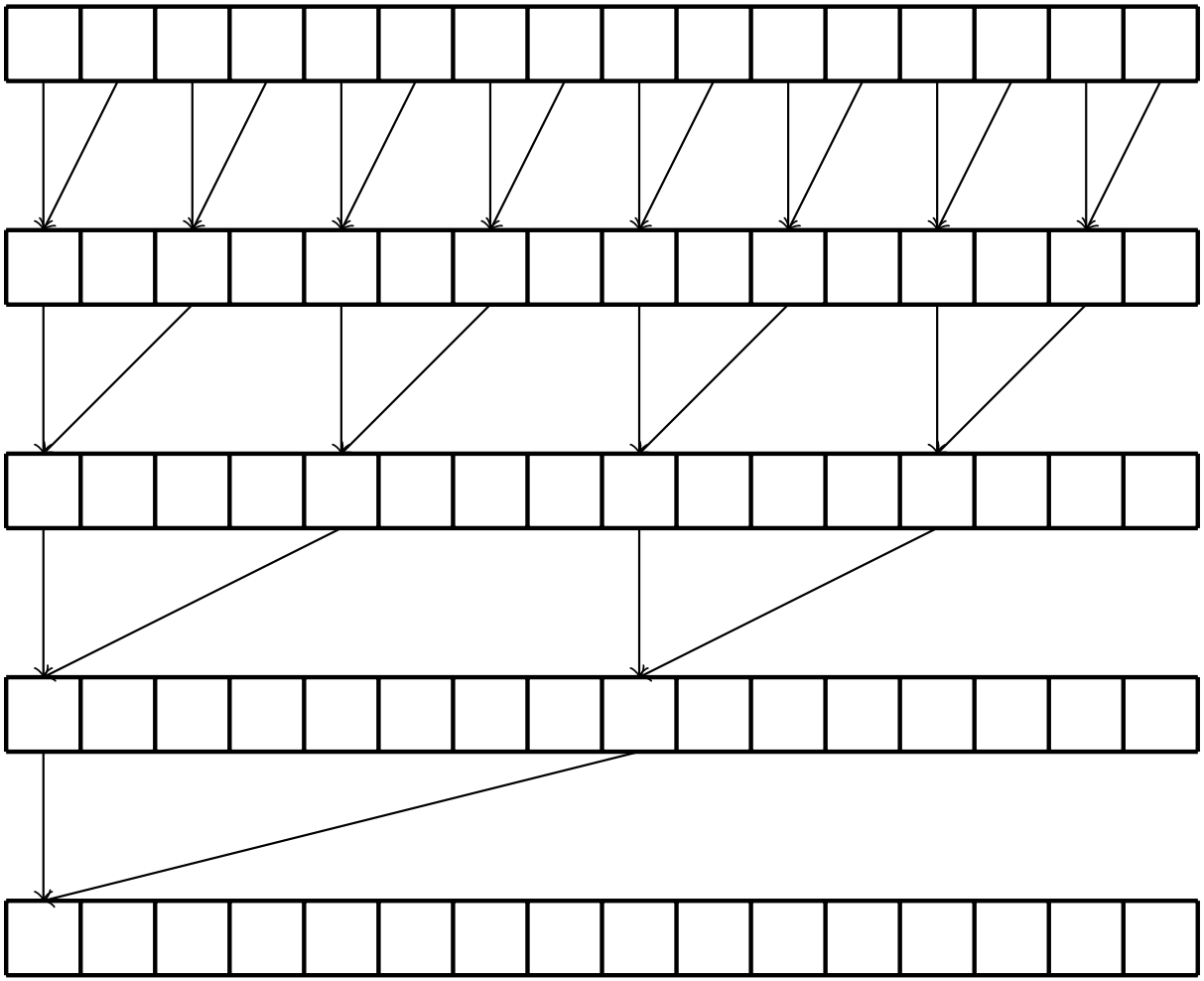
Loops where different threads in the warp execute different numbers of iterations also form divergences: The threads that execute the fewest number of iterations wait on the threads that execute the most number.

Divergence and Reduction

Let's see the consequences of divergence for reduction: the reduction of a set of numbers to one e.g. finding the sum of a vector of length 1024

- Sequentially: iterate through a vector accumulating the sum in a variable (register)
- In parallel, one approach: use a binary tournament:
  - Round 1:
    - thread 0 executes `A[0] += A[1]`
    - thread 2 executes `A[2] += A[3]`
    - ...
    - thread $2i$, where $0 < 2i < 1024$, executes `A[2*i] += A[2*i+1]`
  - Round 2:
    - thread $4i$, where $0 < 4i < 1024$, executes `A[4*i] += A[4*i+2]`
  - ...
  - Round n:
    - thread $2^n i$ executes `A[`2^n`*i] += A[`$2^n * i + 2^{n-1}$`]`

Naive Parallel Reduction



Naive Parallel Reduction Code

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

Naive Parallel Reduction Code

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

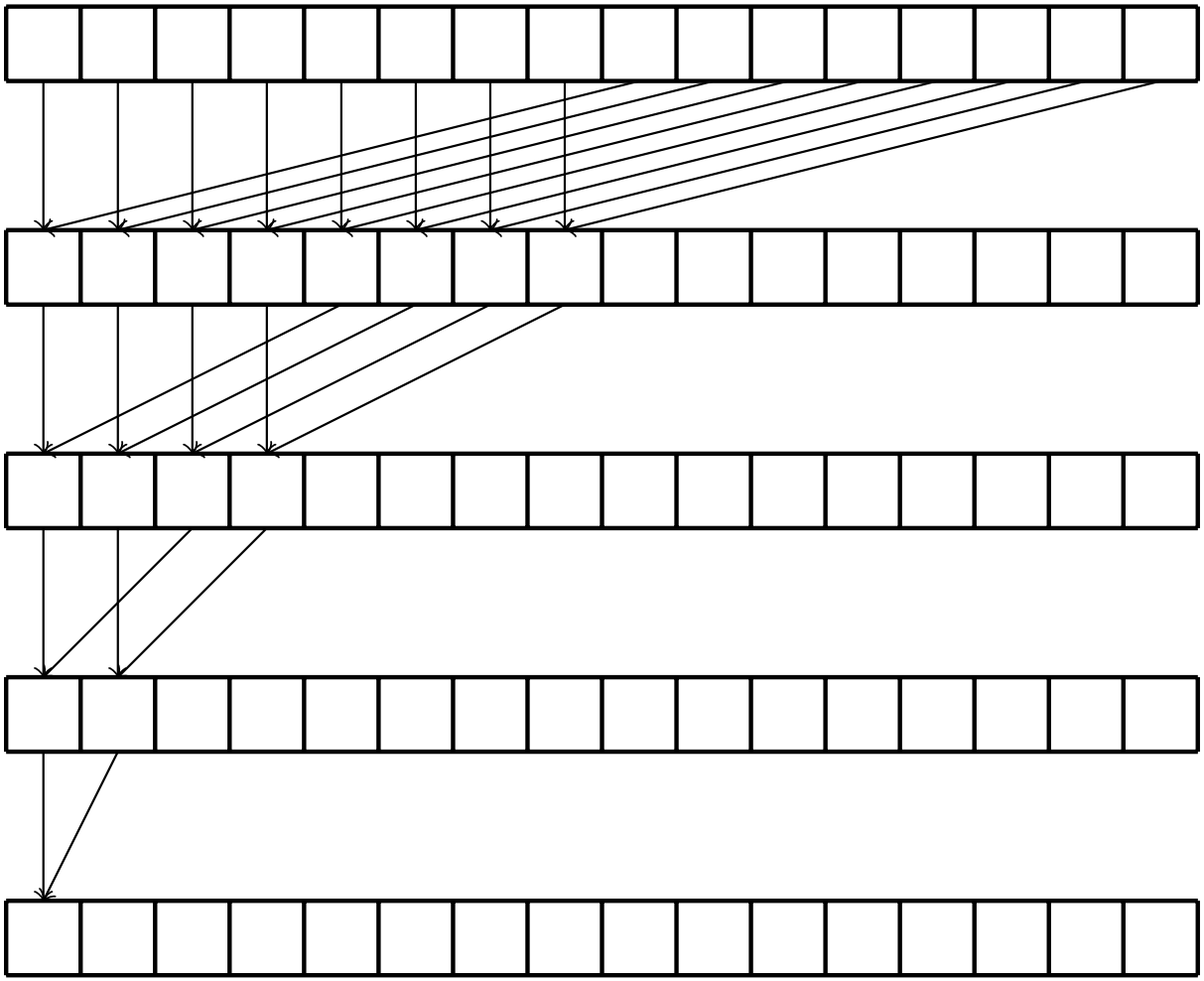
- Note the `__syncthreads()`: necessary to make sure all threads have completed previous stage.

Naive Parallel Reduction Code

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

- Note the `__syncthreads()`: necessary to make sure all threads have completed previous stage.
- Assume `blockDim.x` is 1024
- Iteration 1: Even threads execute add: 1 pass for true branch, 1 pass for false branch, 512 threads = 16 warps all active
- All iterations: 2 passes each iteration
- In progressive iterations, fewer threads doing real work

Parallel Reduction, Alternative Strategy



Alternative Parallel Reduction Code

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

Alternative Parallel Reduction Code

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

- Assume `blockDim.x` is 1024
- Threads 0-511 = warps 0-15 execute true branch, threads 512-1023 = warps 16-31 execute false branch, thus no divergence $\Rightarrow$ 1 pass each iteration

Alternative Parallel Reduction Code

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

- Assume `blockDim.x` is 1024
- Threads 0-511 = warps 0-15 execute true branch, threads 512-1023 = warps 16-31 execute false branch, thus no divergence $\Rightarrow$ 1 pass each iteration
- Continues 1 pass each iteration until less than 32 threads executing

CUDA Memories

Global memory on a GPU is separated from the SMs by a bus and is of DRAM type (i.e. based on capacitors holding charges). This makes data access slow (100s of clock cycles) and limited band width (can't get many words at a time)

GPU	Mem Clock Rate	Mem Bus	Peak Bandwidth
GT 610	535 MHz	64 bits	4.280 GB/s
GTX 960	3600 MHz	128 bits	57.600 GB/s
GTX 1050 Ti	3504 MHz	128 bits	56.064 GB/s
GTX 1080 Ti	5505 MHz	352 bits	242.220 GB/s
Titan V	850 MHz	3072 bits	326.400 GB/s

Memory Bandwidth as a Performance Barrier

- For simple operations (e.g. `vectorAdd`) the *compute to global memory access* ratio (CGMA) is 1/3 (1 flop to 3 memory accesses - 2 reads and a write).
- Assume the global memory access bandwidth is of the order of 200GB/s, and the processor can execute of the order of 1500 GFLOPS, then memory bandwidth is limiting us as follows:
  - 3 read/writes = 12 bytes
  - 12 bytes memory access at 200GB/s for each flop = 200/12 GFLOPS = 17 GFLOPS
  - Thus though the hardware is capable of 1500 GFLOPS, our global memory latencies for this application limits us to 17 GFLOPS.

CUDA Memories

Limits quoted for the GeForce GTX 960:

Memory	Scope	Lifetime	Speed	Limits
Register	Thread	Kernel	Ultra fast	65536/block
Local	Thread	Kernel	Very slow	(part of Global)
Shared	Block	Kernel	Very fast	49152 bytes/block
Global	Grid	Application	Very slow	1996 MBytes
Constant	Grid	Application	Very fast	65536 bytes (in Global but cached)

Using different memory types:

Memory	Variable Declaration
Register	Automatic variables other than arrays
Local	Automatic array variables
Shared	`__device__ __shared__ int var;`
Global	`__device__ int var;`
Constant	`__device__ __constant__ int var;`

Parallel Reduction and Memory

Now reconsider parallel reduction:

- Each reduction iteration reads two words from global memory and writes one word back to global memory per working thread
- If instead the first read copied from global to shared memory, the remainder of the operation would be hugely faster
- In general, many operation can be executed in a *tiled* fashion, where a large problem in global memory can be broken into small tiles, each of which fits in shared memory, the tiles can be solved and then the results recombined.

Distributed and Parallel Computing

Lecture 04

Alan P. Sexton

University of Birmingham

Spring 2018

Prefix Sum, or Scan

Prefix sum, also known as *Scan*, is an operation to that takes a binary associative operator (e.g. addition, multiplication, maximum, etc.) and applies the operator to calculate a cumulative output vector from an input vector.

A sequential version where the operator used is addition might be implemented as follows:

```
void sequential_scan(float *x, float *y, int len)
{
    y[0] = x[0];
    for (int i = 1; i < len ; i++)
        y[i] = y[i-1] + x[i];
}
```

When applied to:

$$\begin{bmatrix}1, & 2, & 3, & 2, & 3, & 1, & 4, & 5\end{bmatrix}$$

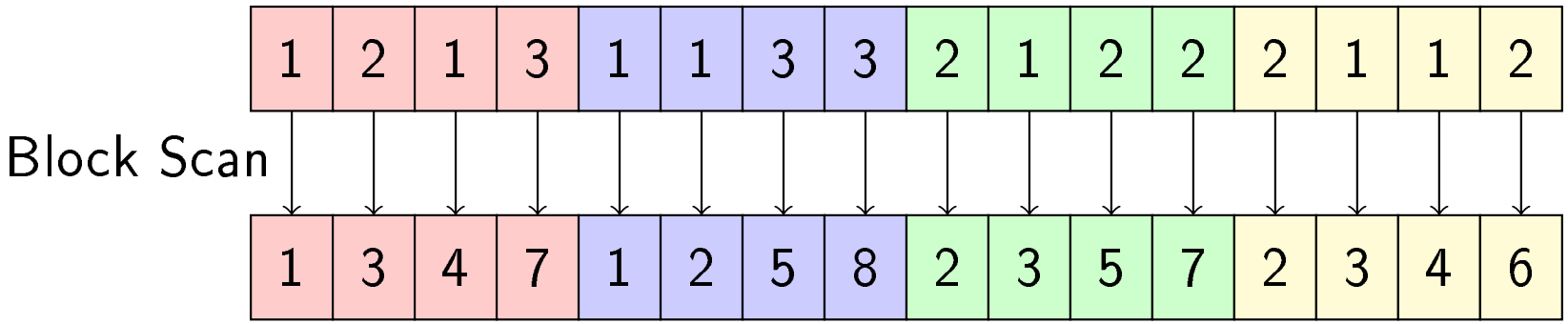
the result would be:

$$\begin{bmatrix}1, & 3, & 6, & 8, & 11, & 12, & 16, & 21\end{bmatrix}$$

Block Prefix Sum, or Block Scan

We will start with algorithms that compute the scan correctly **within each block** but which does not propagate the scan across block boundaries. We shall return later to the problem of how to complete the block scan to a full scan.

An example execution of block scan where the block size is 4 and the length of the vector is 16 is as follows:



Sequential implementation of Block Scan

A sequential version of a block scan where the operator used is addition might be implemented as follows:

```
#define BLOCK_SIZE 1024

void sequential_block_scan(float *x, float *y, int len)
{
    int num_blocks = 1 + (len-1)/BLOCK_SIZE;
    for (blk = 0 ; blk < num_blocks ; blk ++)
    {
        int blk_start = blk * BLOCK_SIZE ;
        int blk_end = blk_start + BLOCK_SIZE ;
        if (blk_end > len)
            blk_end = len;
        y[blk_start] = x[blk_start];
        for (int i = blk_start + 1; i < blk_end; i++)
            y[i] = y[i-1] + x[i];
    }
}
```

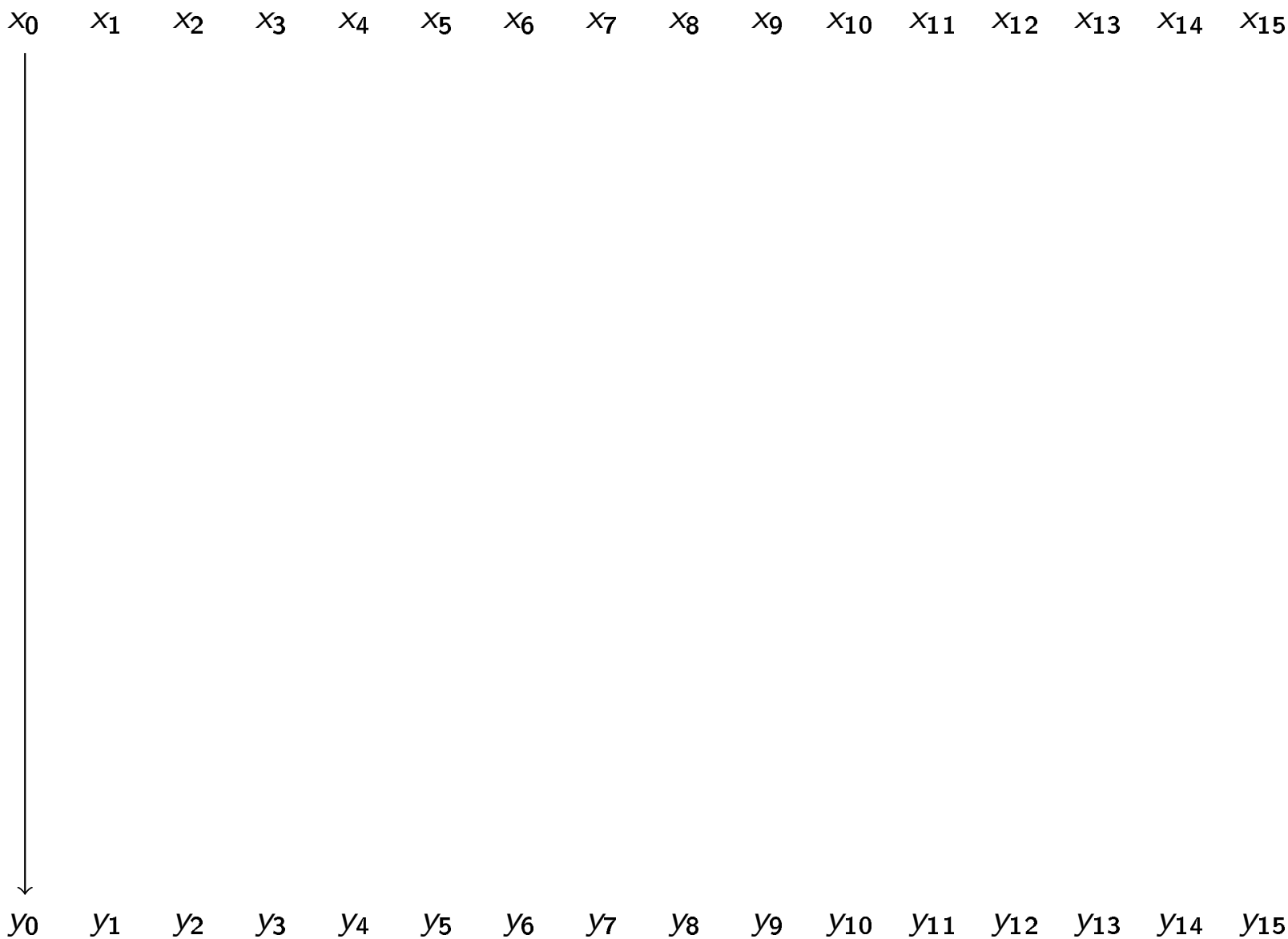
Cost of Sequential Scan

The cost of sequential scan is easy to calculate:

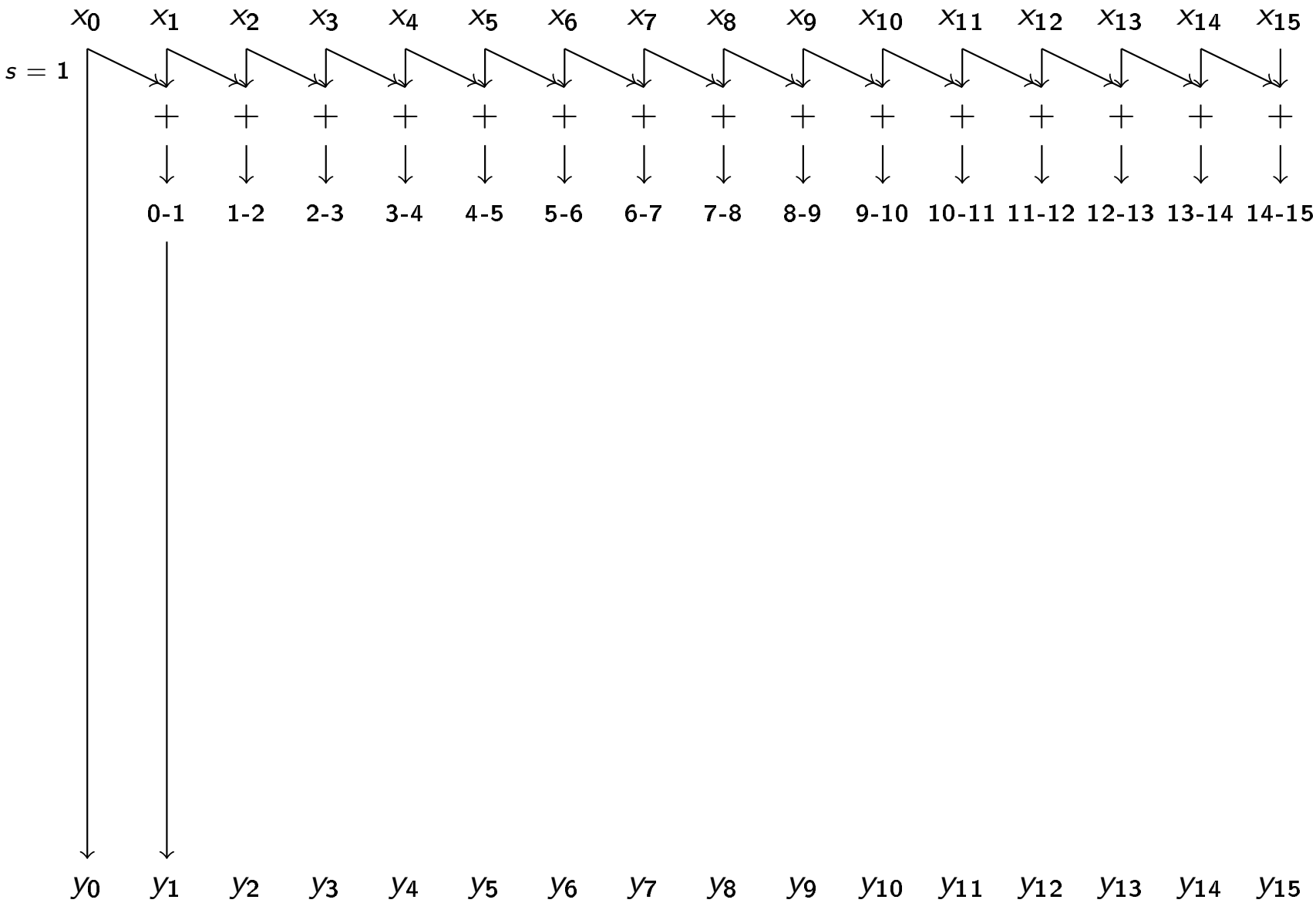
- for N elements, there are $N-1$ floating point additions

$$N = 1024 \Rightarrow \mathrm{Cost}_{\mathrm{SS}} = 1024 - 1 = 1023$$

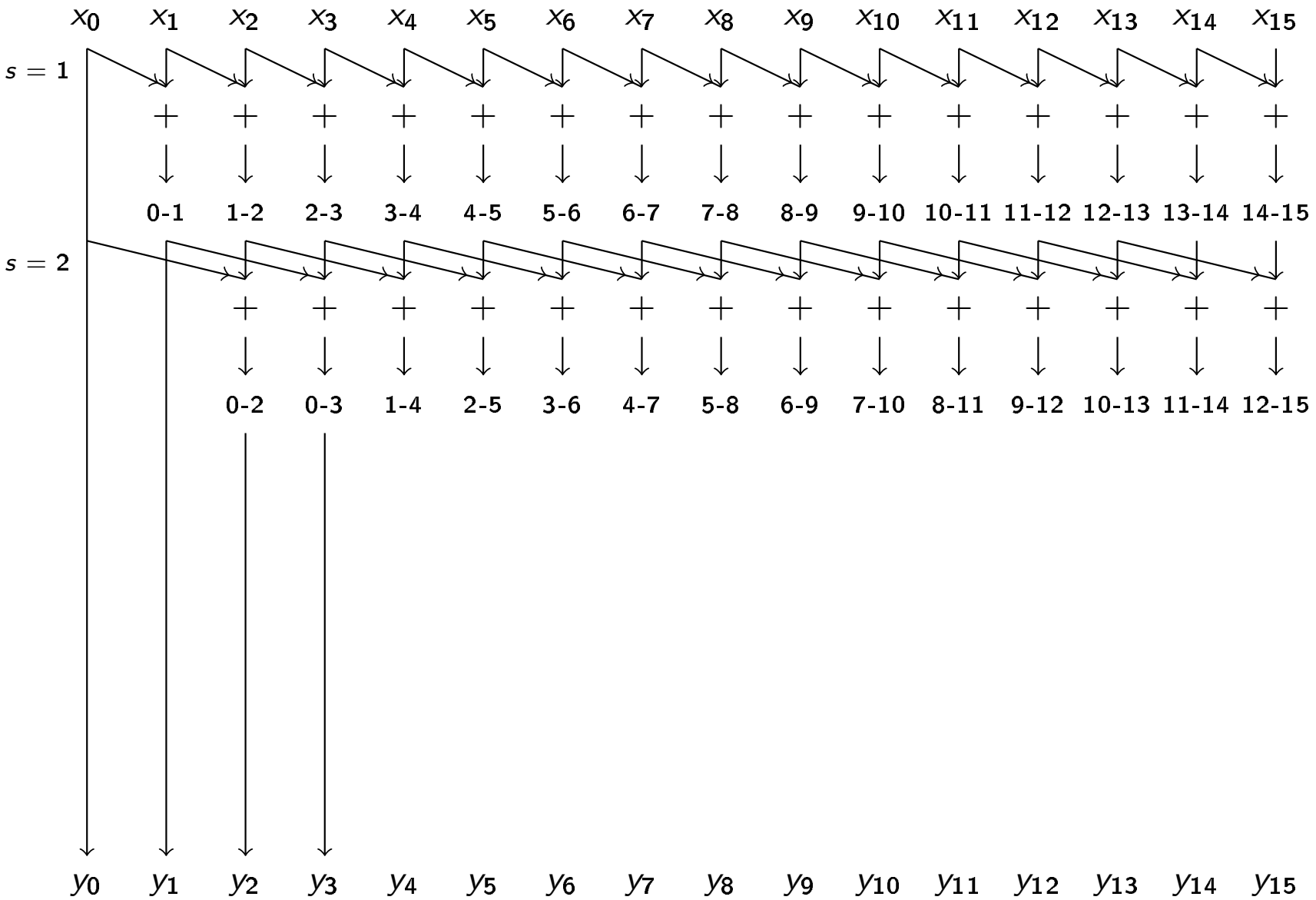
First Attempt at Scan: Hillis Steele Horn (Inefficient)



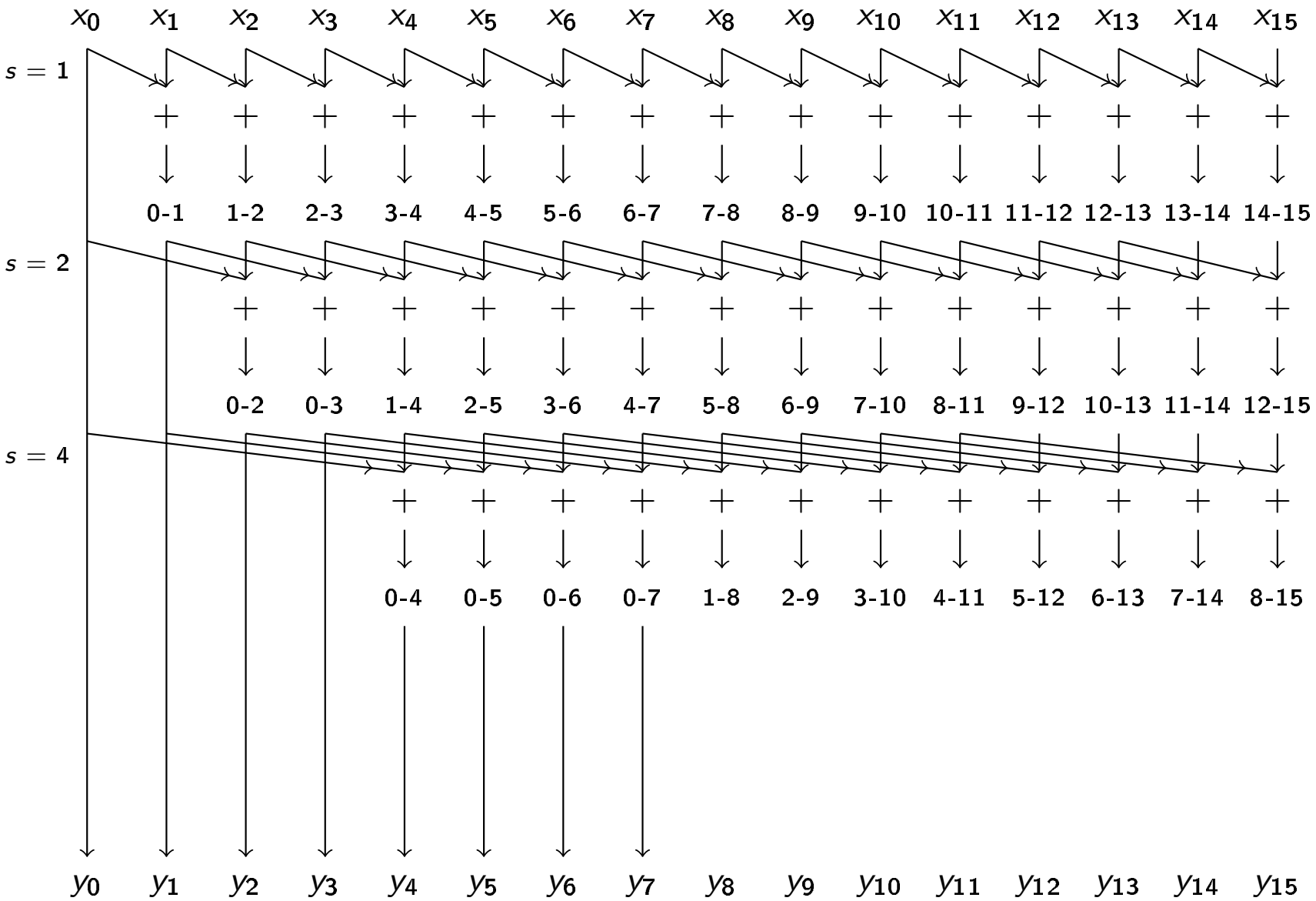
First Attempt at Scan: Hillis Steele Horn (Inefficient)



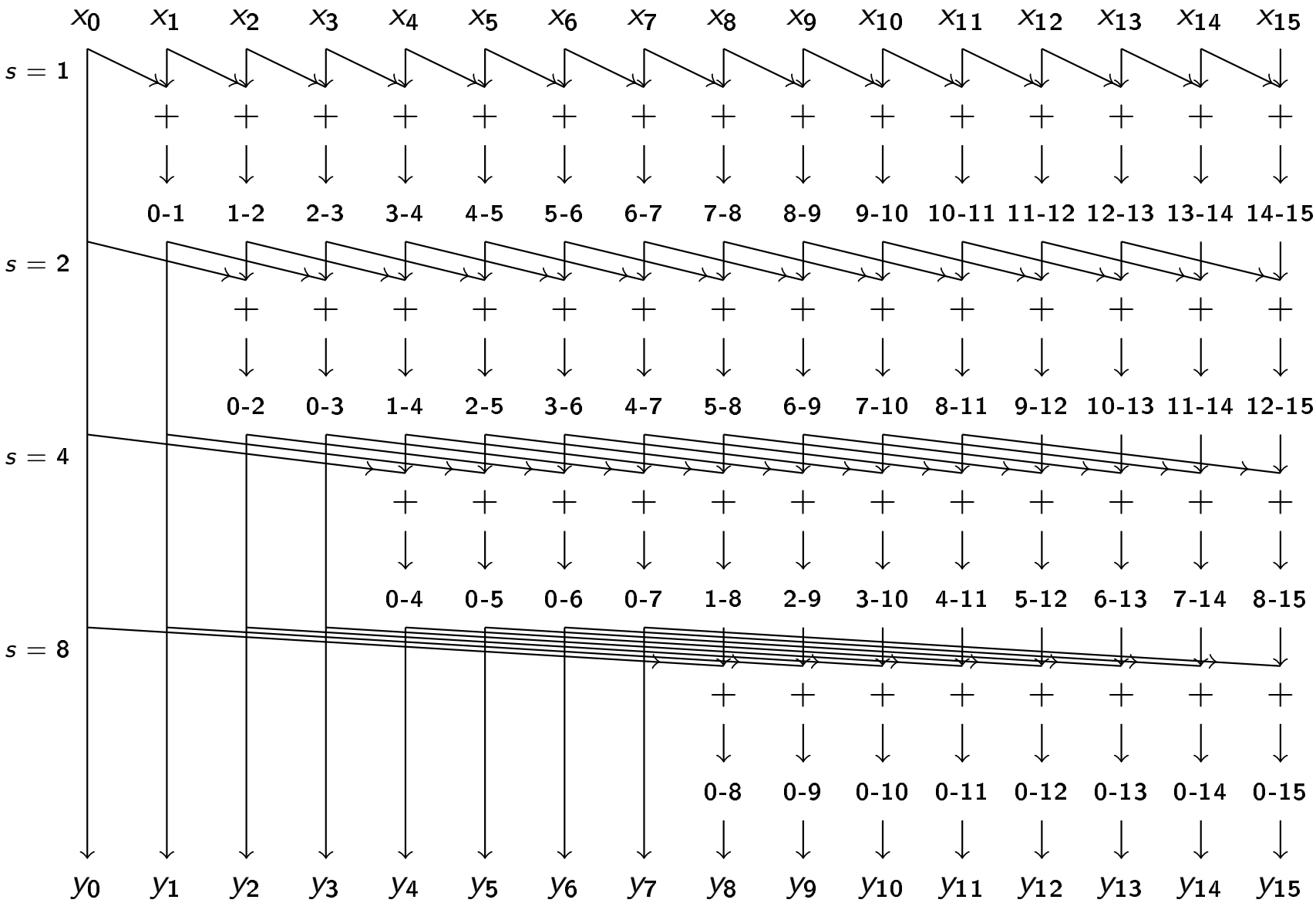
First Attempt at Scan: Hillis Steele Horn (Inefficient)



First Attempt at Scan: Hillis Steele Horn (Inefficient)



First Attempt at Scan: Hillis Steele Horn (Inefficient)



Code for HSH Scan (len $\leq$ block size) - with errors

```
#define BLOCK_SIZE 1024 // the actual configured block size

__global__ void hsh_scan(float *X, float *Y, len)
{
    __shared__ float XY[BLOCK_SIZE];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < len)
        XY[threadIdx.x] = X[i];

    for(uint stride = 1; stride <= threadIdx.x; stride *= 2)
    {
        __syncthreads();
        XY[threadIdx.x] += XY[threadIdx.x-stride];
    }
    if (i < len)
        Y[i] = XY[threadIdx.x];
}
```

Errors in Previous Code for HSH Scan

The previous code had 2 errors. Consider the line:

```
XY[threadIdx.x] += XY[threadIdx.x-stride];
```

- The `__syncthreads` ensures that all threads in one block finish one iteration before any starts the next but . . .
- . . . there is no guaranteed order in which the different threads execute this statement within the same iteration:
- If two threads are from different warps, they can execute in arbitrary sequential orders relative to each other
- Thus we could have a thread write to an element of `XY` either **before** or **after** a different thread reads from it.
- This is called a *read/write race*
- To fix it we need to *double buffer*: double the size of `XY` and, in each iteration, read and write from different halves.

Errors in Previous Code for HSH Scan

The previous code had 2 errors. Consider the line:

```
XY[threadIdx.x] += XY[threadIdx.x-stride];
```

- The `__syncthreads` ensures that all threads in one block finish one iteration before any starts the next but ...
- ...there is no guaranteed order in which the different threads execute this statement within the same iteration:
- If two threads are from different warps, they can execute in arbitrary sequential orders relative to each other
- Thus we could have a thread write to an element of `XY` either **before** or **after** a different thread reads from it.
- This is called a *read/write race*
- To fix it we need to *double buffer*: double the size of `XY` and, in each iteration, read and write from different halves.

Worse: since different threads exit the loop at different times, the barrier synchronise may deadlock!

Corrected Code for HSH Scan

```
#define BLOCK_SIZE 1024 // the actual configured block size
__global__ void hsh_scan(float *X, float *Y, len)
{
    __shared__ float XY[BLOCK_SIZE*2]; // 2 buffers
    int rBuf = 0, wBuf = BLOCK_SIZE ;
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < len)
        XY[wBuf + threadIdx.x] = X[i];

    // note: now All threads execute in ALL iterations
    for(uint s=1; s < BLOCK_SIZE; s *= 2)
    {
        __syncthreads();
        wBuf = BLOCK_SIZE - wBuf; rBuf = BLOCK_SIZE - rBuf;
        if (threadIdx.x >= s)
            XY[wBuf+threadIdx.x] =
              XY[rBuf+threadIdx.x-s] + XY[rBuf+threadIdx.x];
        else // if not adding, thread should copy
            XY[wBuf+threadIdx.x] = XY[rBuf+threadIdx.x];
    }
    if (i < len)
        Y[i] = XY[wBuf + threadIdx.x];
}
```

Cost of HSH Scan

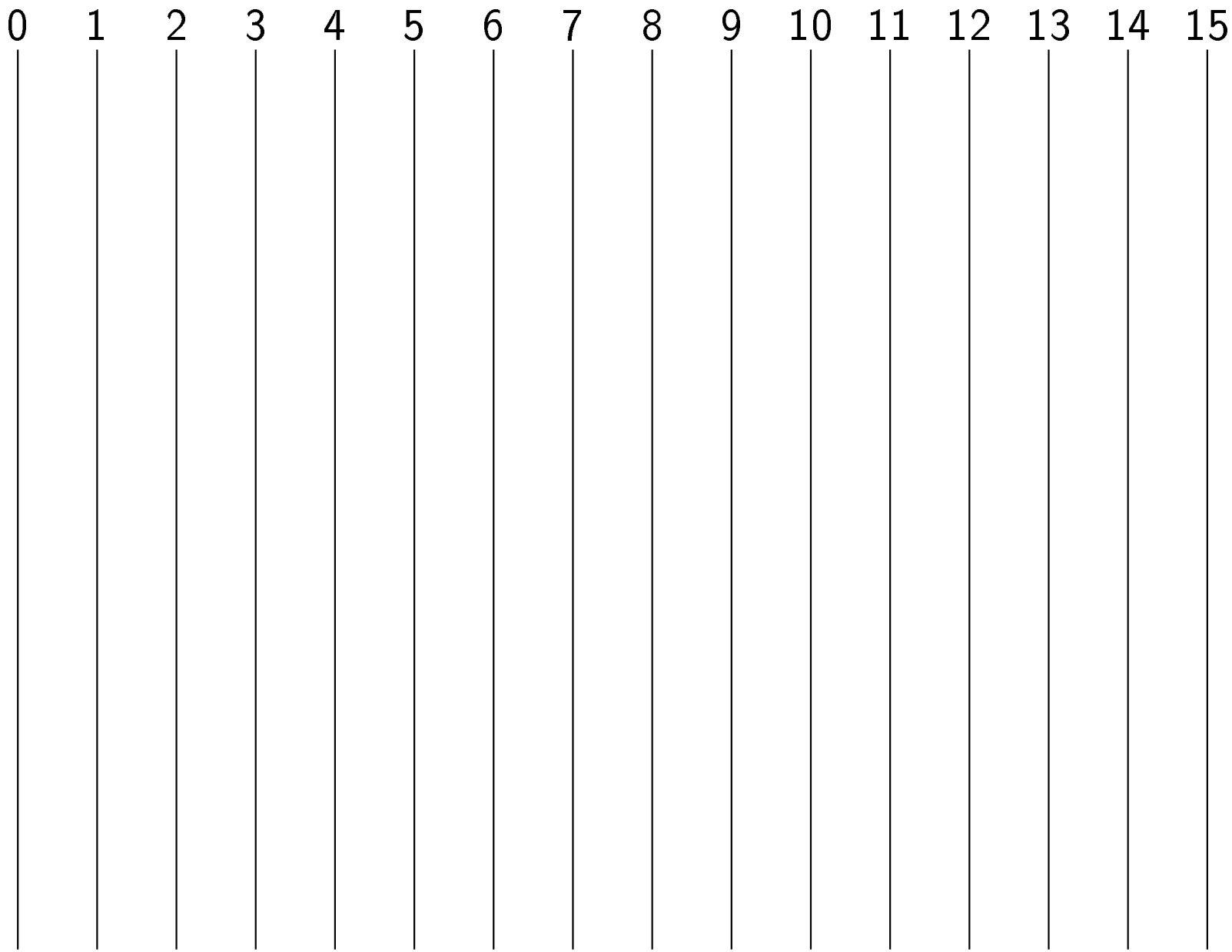
Given N elements:

- There will be $\log_2(N)$ iterations
- In each iteration, all except the *stride* number of threads is doing an addition
- In the first iteration, *stride* is 1
- In the second iteration, *stride* is 2
- In the third iteration, *stride* is 4
- ...

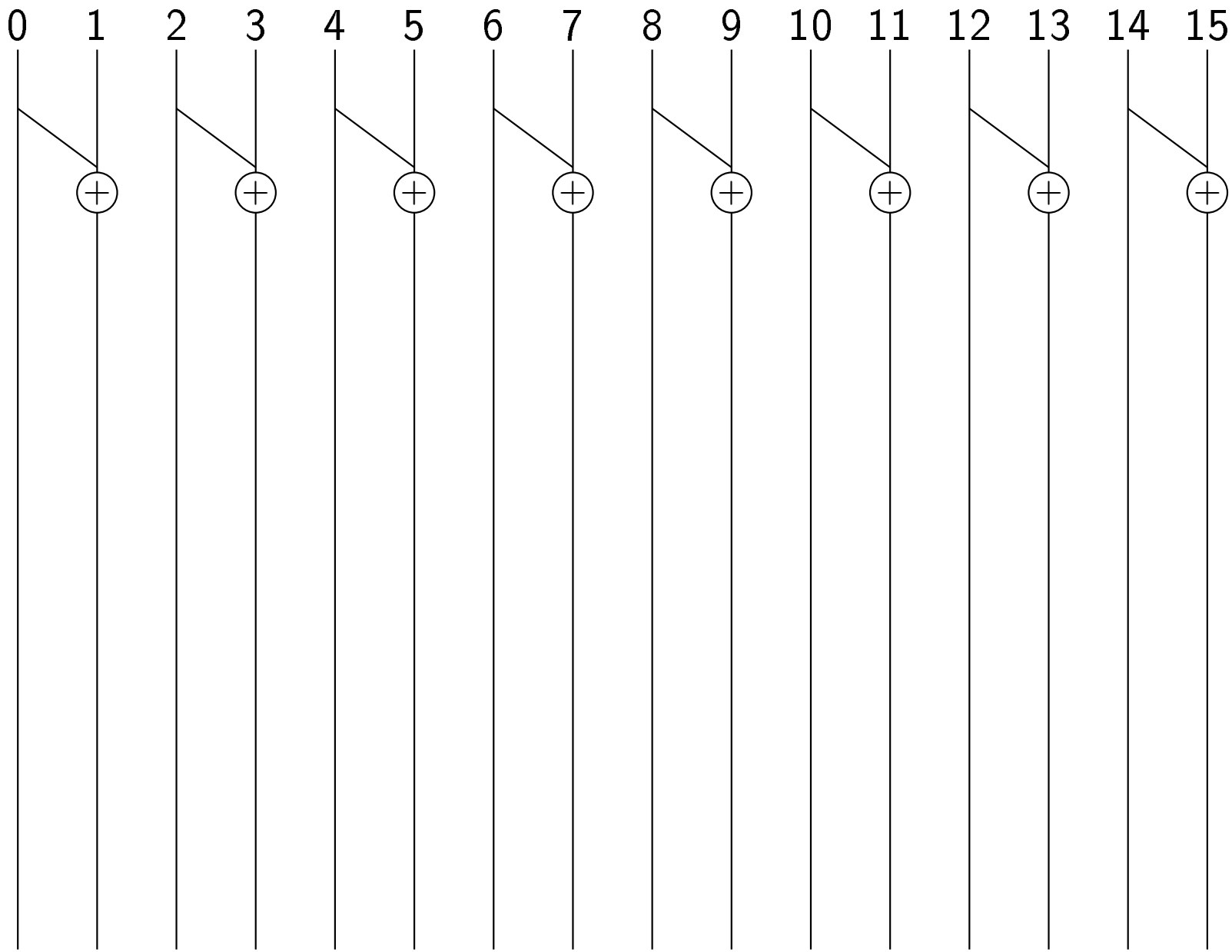
$$\begin{aligned}\mathrm{Cost}_{\mathrm{IS}} &= \sum_{i=1}^{\log_2(N)} (N - 2^{i-1}) = \left(\sum_{i=1}^{\log_2(N)} N\right) - \left(\sum_{i=1}^{\log_2(N)} 2^{i-1}\right) \\ &= N\log_2(N) - (1 + 2 + \cdots + N/2) \\ &= N\log_2(N) - (N - 1)\end{aligned}$$

$$N = 1024 \Rightarrow \begin{cases} \mathrm{Cost}_{\mathrm{HSH}} = 1024 \times 10 - (1024 - 1) & = 9217 \\ \mathrm{Cost}_{\mathrm{SS}} = 1024 - 1 & = 1023 \end{cases}$$

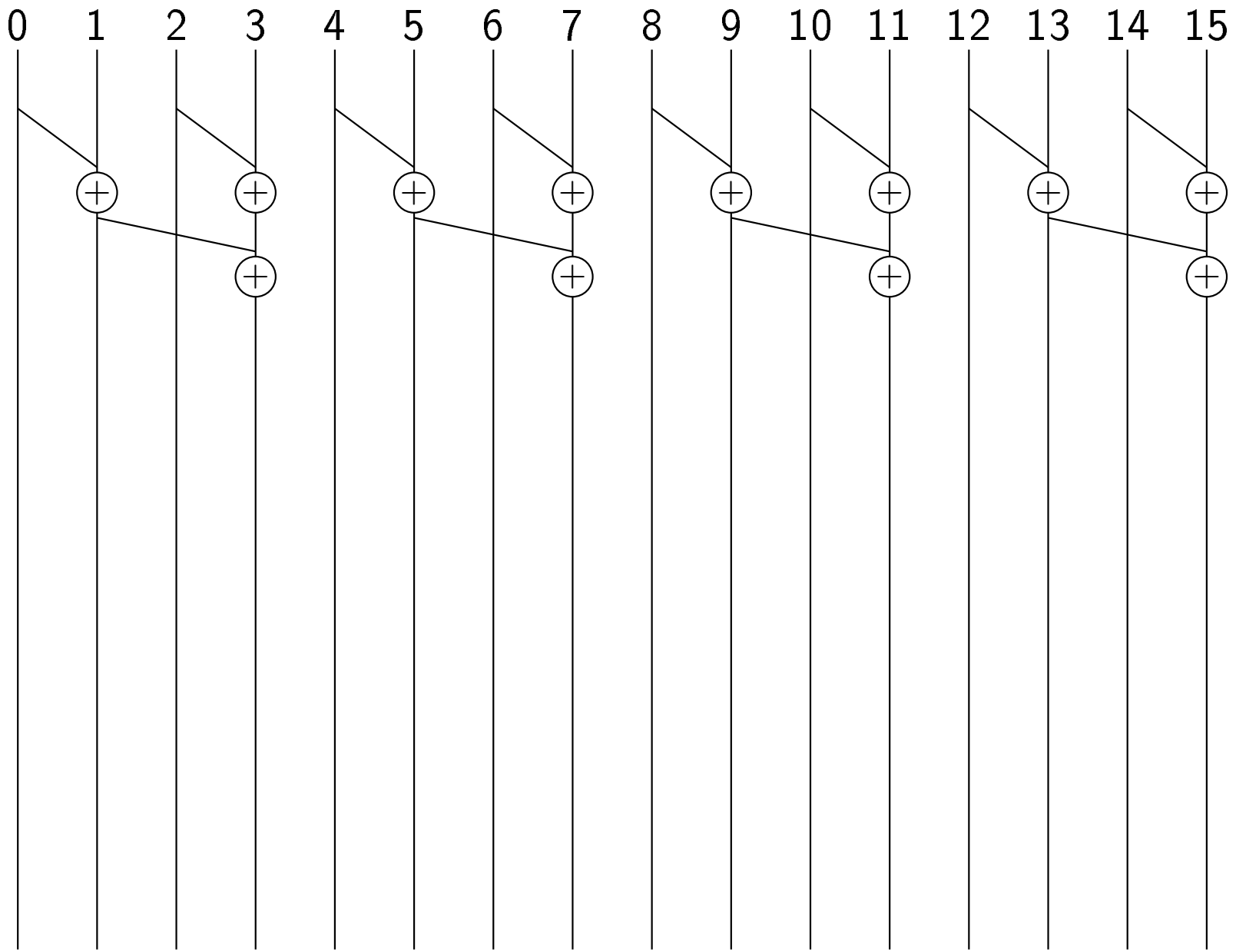
Blelloch Scan [Efficient]



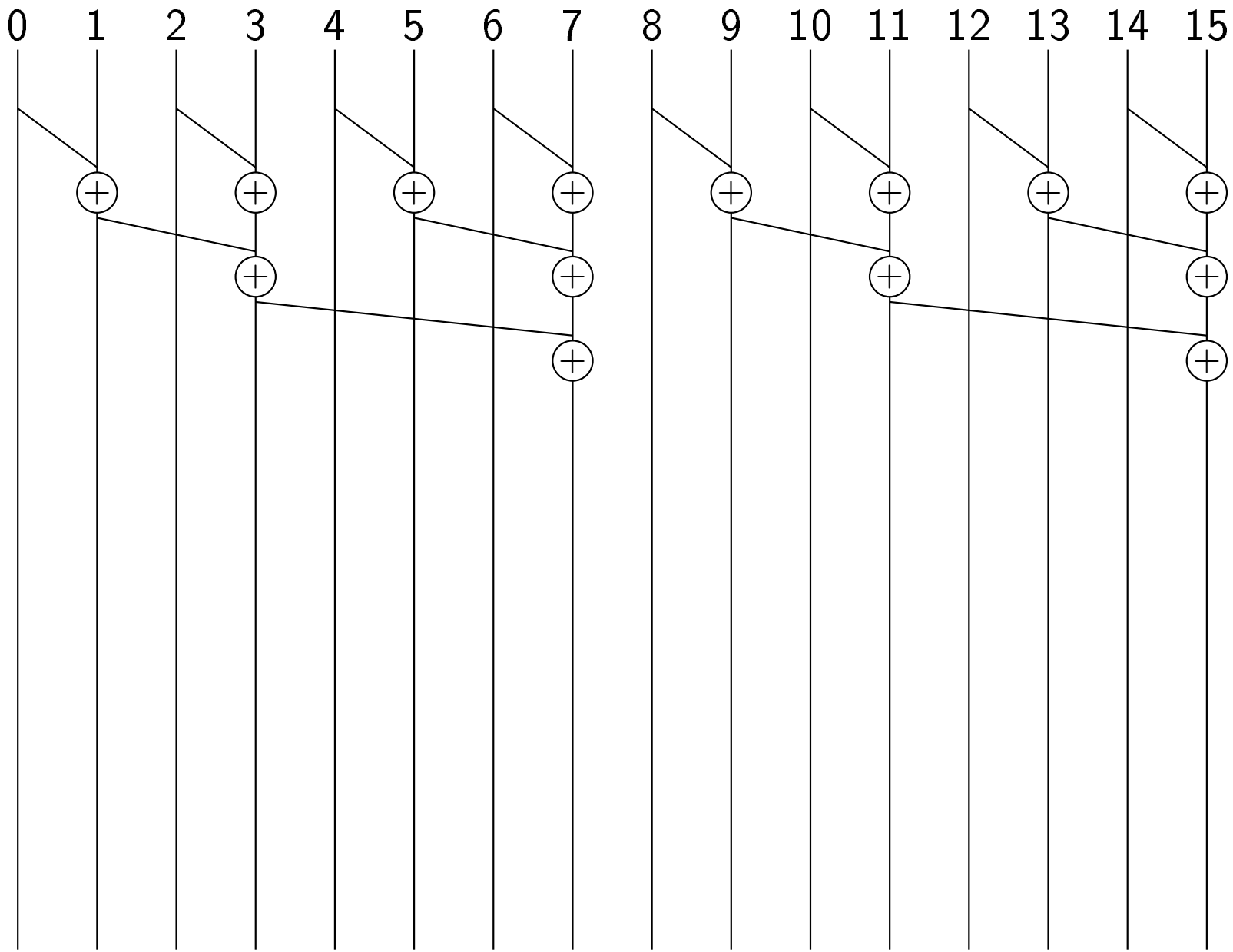
Blelloch Scan [Efficient]



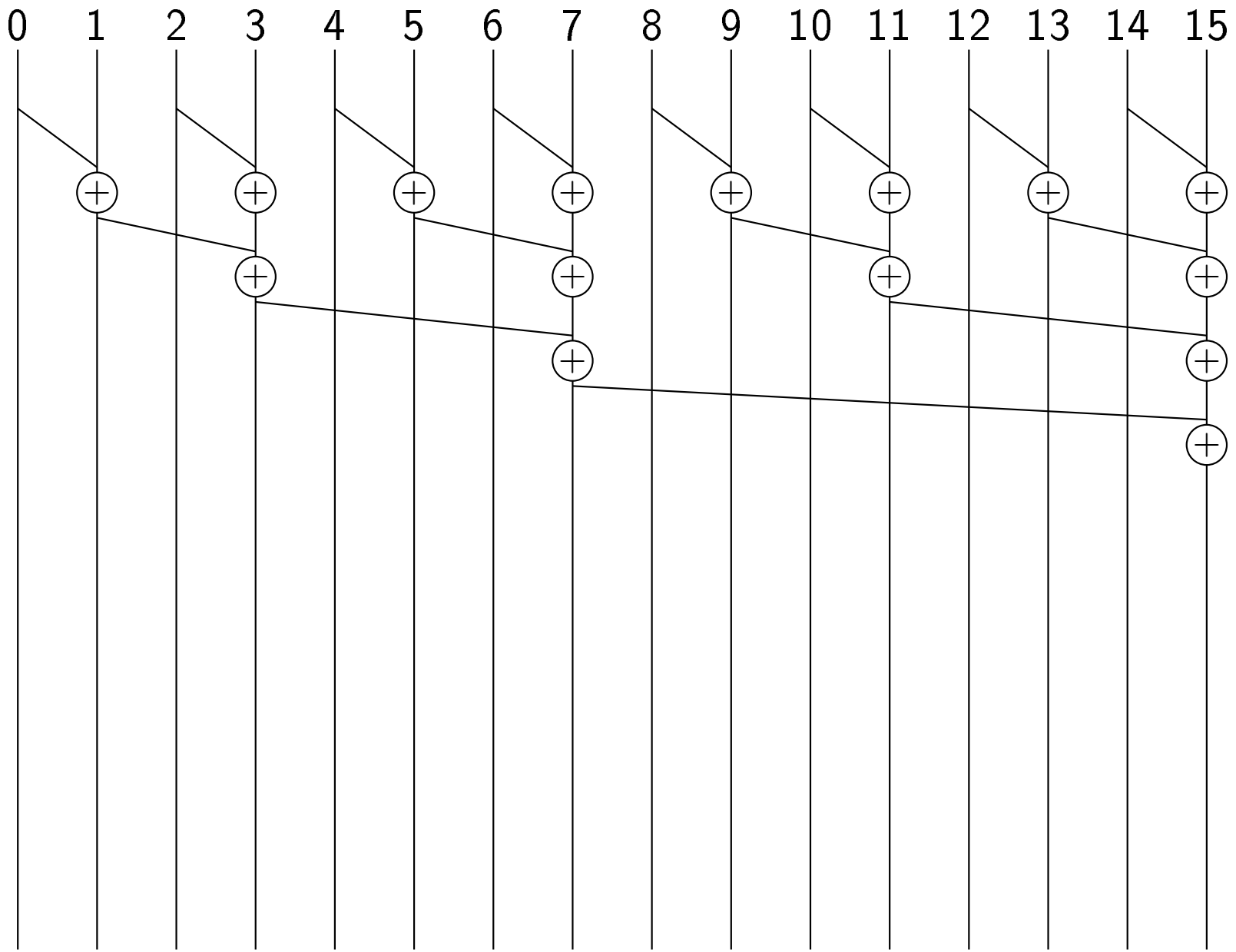
Blelloch Scan [Efficient]



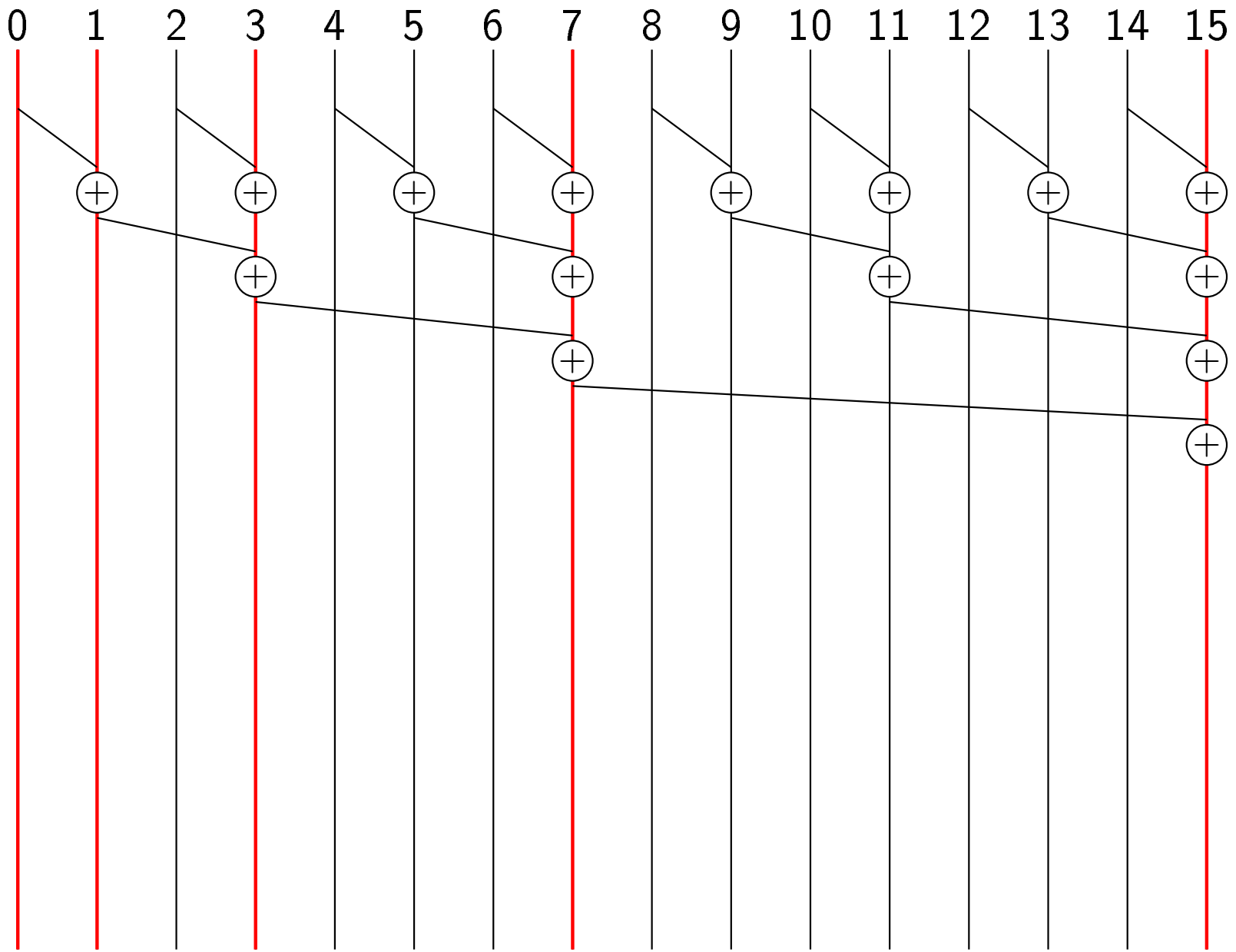
Blelloch Scan [Efficient]



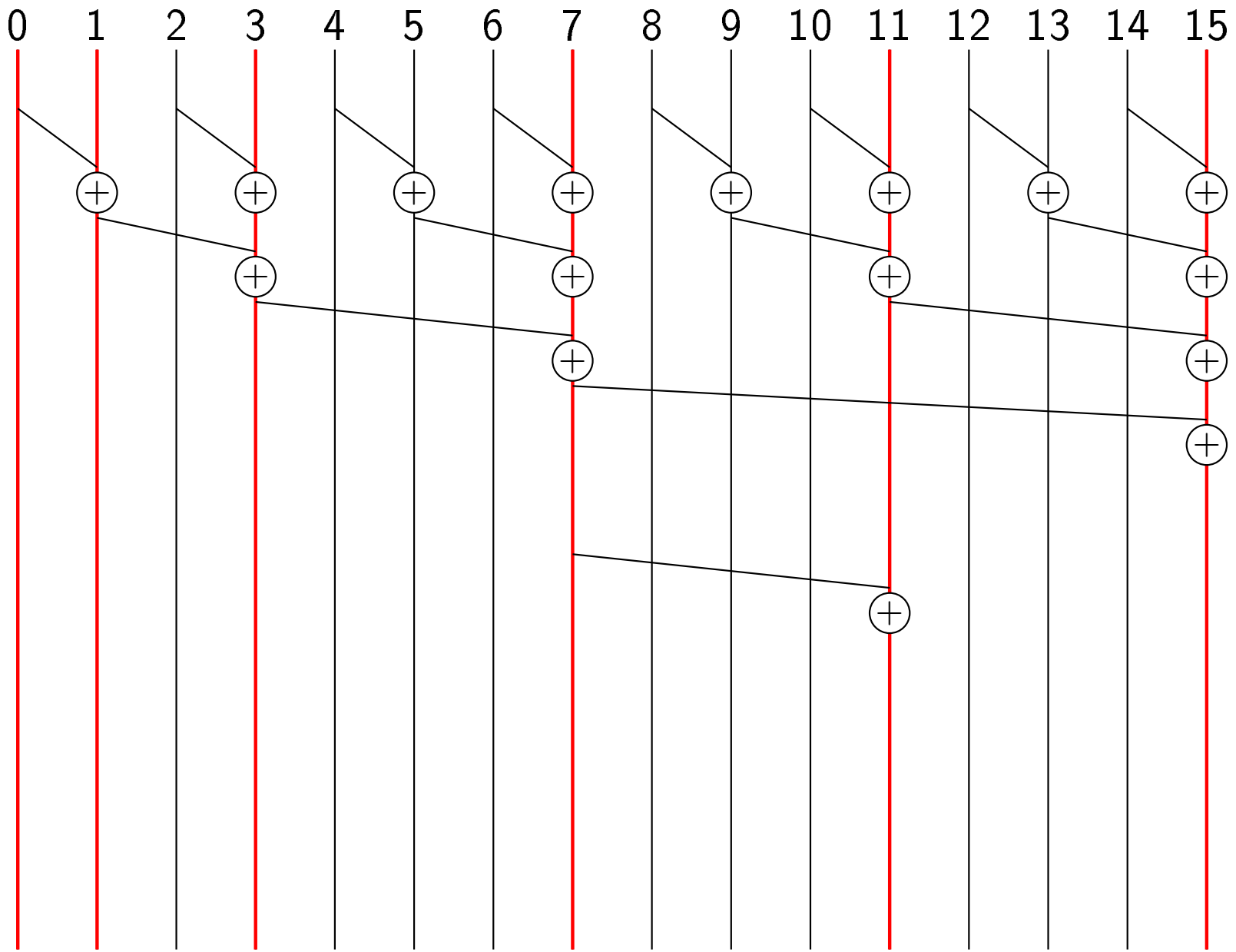
Blelloch Scan [Efficient]



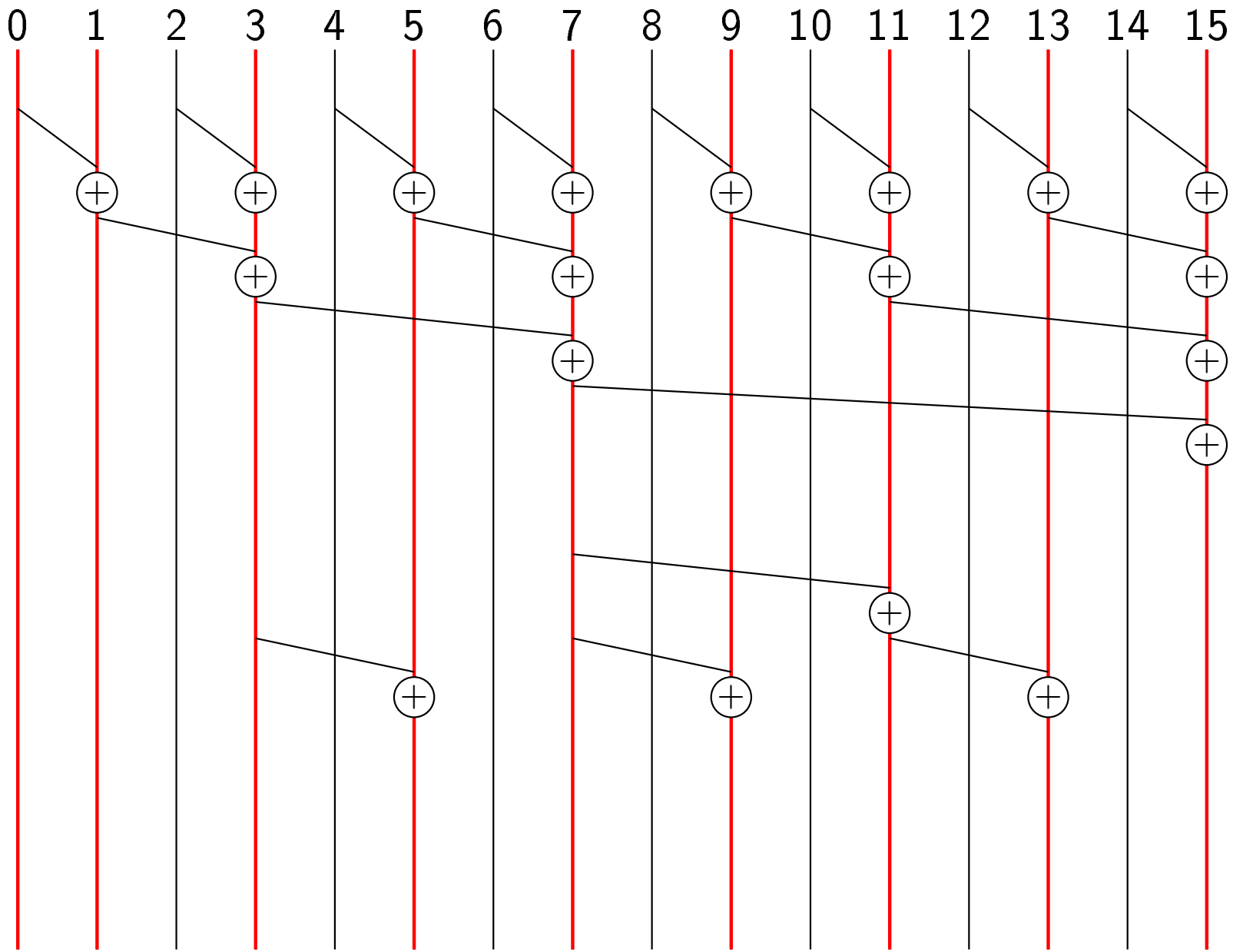
Blelloch Scan [Efficient]



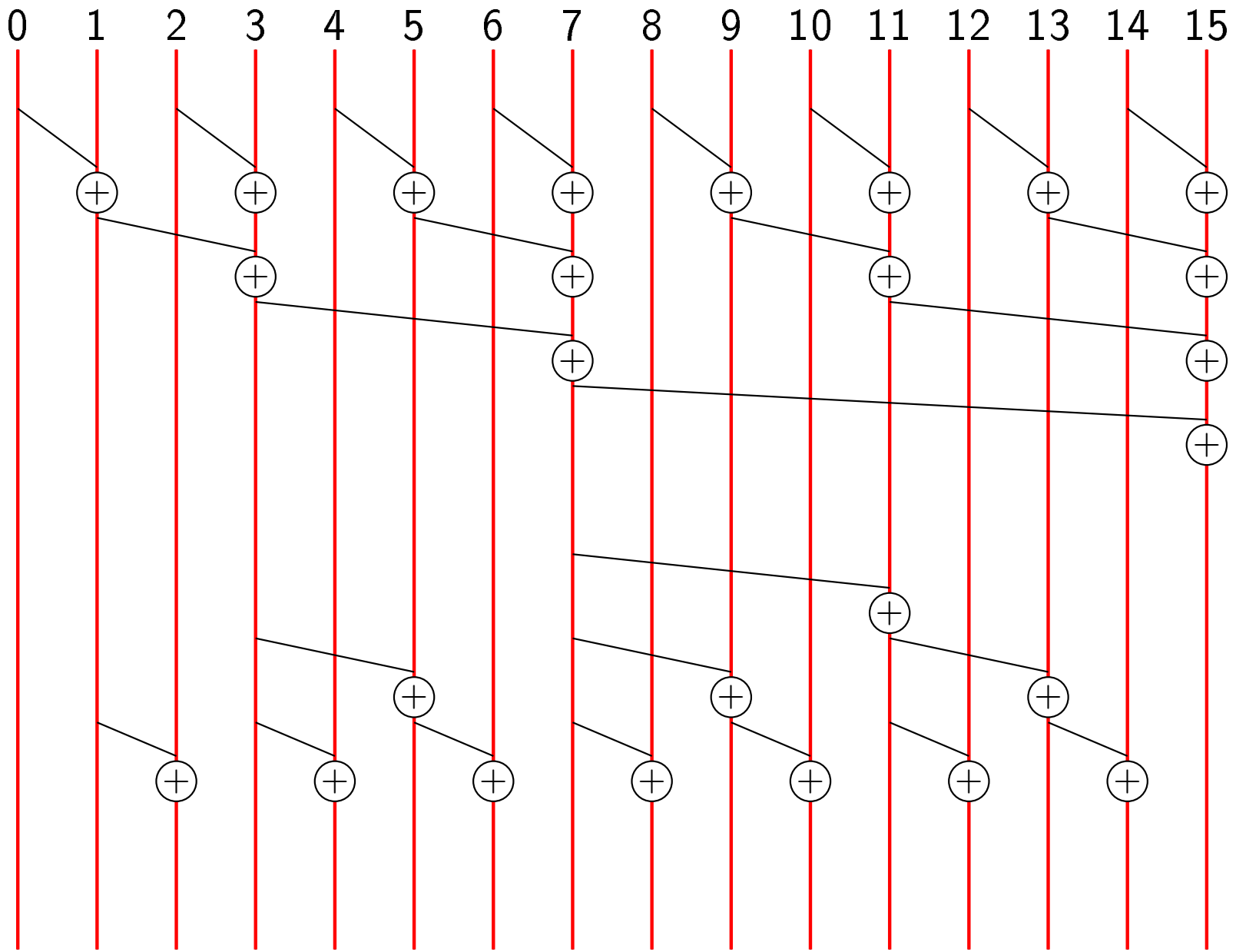
Blelloch Scan [Efficient]



Blelloch Scan [Efficient]



Blelloch Scan [Efficient]



Structure of Blelloch Scan

The basic structure will be:

- Copy `X` from global memory to shared memory `XY`
- Carry out reduction phase
- Carry out distribution phase
- Copy `XY` from shared memory to global memory `Y`

Code for Blelloch Scan: Copying

```
#define BLOCK_SIZE 1024 // the actual configured block size

__global void blelloch_scan(float *X, float *Y, int len)
{
    __shared__ float XY[BLOCK_SIZE];

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < len)
        XY[threadIdx.x] = X[i];

    // Do Reduction phase here

    // Do Distribution phase here

    __syncthreads();

    if (i < len)
        Y[i] = XY[threadIdx.x];
}
```

Code for Blelloch Scan — Reduction

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if ((threadIdx.x + 1) % (2*stride) == 0)
        XY[threadIdx.x] += XY[threadIdx.x - stride];
}
```

- All threads sync every iteration $\Rightarrow$ no deadlock
- No read/write races $\Rightarrow$ no double buffering required
  - No thread writes to a location that another thread reads from within the same iteration

Code for Blelloch Scan — Reduction

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if ((threadIdx.x + 1) % (2*stride) == 0)
        XY[threadIdx.x] += XY[threadIdx.x - stride];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
  - No thread writes to a location that another thread reads from within the same iteration
- Iteration 1: threads 1, 3, 5,. . . do the addition

Code for Blelloch Scan — Reduction

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if ((threadIdx.x + 1) % (2*stride) == 0)
        XY[threadIdx.x] += XY[threadIdx.x - stride];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
  - No thread writes to a location that another thread reads from within the same iteration
- Iteration 1: threads 1, 3, 5,. . . do the addition
- Iteration 2: threads 3, 7, 11, . . . do the addition

Code for Blelloch Scan — Reduction

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if ((threadIdx.x + 1) % (2*stride) == 0)
        XY[threadIdx.x] += XY[threadIdx.x - stride];
}
```

- All threads sync every iteration $\Rightarrow$ no deadlock
- No read/write races $\Rightarrow$ no double buffering required
  - No thread writes to a location that another thread reads from within the same iteration
- Iteration 1: threads 1, 3, 5,. . . do the addition
- Iteration 2: threads 3, 7, 11, . . . do the addition
- **LOTS OF DIVERGENCE**

Code for Blelloch Scan — Reduction, minimising divergence

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index < blockDim.x)
        XY[index] += XY[index - stride];
}
```

Code for Blelloch Scan — Reduction, minimising divergence

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index < blockDim.x)
        XY[index] += XY[index - stride];
}
```

- Iteration 1: thread 0 uses index 1, $1 \rightarrow 3$, $2 \rightarrow 5$,...

Code for Blelloch Scan — Reduction, minimising divergence

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index < blockDim.x)
        XY[index] += XY[index - stride];
}
```

- Iteration 1: thread 0 uses index 1, $1 \rightarrow 3$, $2 \rightarrow 5$,...
- Iteration 2: thread 0 uses index 3, $1 \rightarrow 7$, $2 \rightarrow 11$,

Code for Blelloch Scan — Reduction, minimising divergence

Reduction phase:

```
for(uint stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index < blockDim.x)
        XY[index] += XY[index - stride];
}
```

- Iteration 1: thread 0 uses index 1, $1 \rightarrow 3$, $2 \rightarrow 5$,...
- Iteration 2: thread 0 uses index 3, $1 \rightarrow 7$, $2 \rightarrow 11$,
- Thus the working threads are contiguous, starting at 0 $\Rightarrow$ **minimal divergence**

Code for Blelloch Scan — Distribution

Distribution phase:

```
for(uint stride = BLOCK_SIZE/4; stride > 0; stride /= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index + stride < BLOCK_SIZE)
        XY[index + stride] += XY[index];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
- `stride` divides by 2 from `BLOCK_SIZE/4` each iteration
- Each iteration we *push* the `XY` values:
  - from locations: 1 less than multiples of twice `stride`
  - *to* locations: 1 stride above the *from*

Code for Blelloch Scan — Distribution

Distribution phase:

```
for(uint stride = BLOCK_SIZE/4; stride > 0; stride /= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index + stride < BLOCK_SIZE)
        XY[index + stride] += XY[index];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
- `stride` divides by 2 from `BLOCK_SIZE/` `4` each iteration
- Each iteration we *push* the `XY` values:
  - from locations: 1 less than multiples of twice `stride`
  - *to* locations: 1 stride above the *from*
- Assuming `BLOCK_SIZE = 16`, to match our diagram:
- Iteration 1: thread 0 uses index 7, 1 → 15, 2 → 23,...

Code for Blelloch Scan — Distribution

Distribution phase:

```
for(uint stride = BLOCK_SIZE/4; stride > 0; stride /= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index + stride < BLOCK_SIZE)
        XY[index + stride] += XY[index];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
- `stride` divides by 2 from `BLOCK_SIZE/` `4` each iteration
- Each iteration we *push* the `XY` values:
  - from locations: 1 less than multiples of twice `stride`
  - *to* locations: 1 stride above the *from*
- Assuming `BLOCK_SIZE = 16`, to match our diagram:
- Iteration 1: thread 0 uses index 7, 1 → 15, 2 → 23,...
- Iteration 2: thread 0 uses index 3, 1 → 7, 2 → 11,

Code for Blelloch Scan — Distribution

Distribution phase:

```
for(uint stride = BLOCK_SIZE/4; stride > 0; stride /= 2)
{
    __syncthreads();
    uint index = (threadIdx.x + 1) * stride * 2 - 1 ;
    if (index + stride < BLOCK_SIZE)
        XY[index + stride] += XY[index];
}
```

- All threads sync every iteration ⇒ no deadlock
- No read/write races ⇒ no double buffering required
- `stride` divides by 2 from `BLOCK_SIZE/` `4` each iteration
- Each iteration we *push* the `XY` values:
  - from locations: 1 less than multiples of twice `stride`
  - *to* locations: 1 stride above the *from*
- Assuming `BLOCK_SIZE = 16`, to match our diagram:
- Iteration 1: thread 0 uses index 7, 1 → 15, 2 → 23,...
- Iteration 2: thread 0 uses index 3, 1 → 7, 2 → 11,
- Thus the working threads are contiguous, starting at 0 ⇒ **minimal divergence**

Threads vs Vector Elements

- In the previous code, we dealt with a number of vector elements equal to the block size
- But, at maximum, we only use half that number of threads
- Better to deal with a number of vector elements equal to twice the block size and use all the threads
- Make sure to choose the block size so that `XY` fits within the shared memory per block limit
- Easy modification of code to make this happen (exercise for reader!)

Cost of Blelloch Scan

Given N elements:

- During the Reduction phase:
  - Iteration 1: N/2 floating point additions
  - Iteration 2: N/4 floating point additions
  - ...
  - Last iteration: 1 floating point addition
- During the Distribution phase:
  - Last iteration: N/2 - 1 floating point additions
  - 2^{nd} last iteration: N/4 - 1 floating point operations

$$\begin{aligned}\text{COST}_{\text{ES}} &= 2 \times (N/2 + N/4 + \ldots 1) - \log_2(N/2) \\ &= 2(N-1) - (\log_2(N) - 1) \\ &= 2N - log_2(N) - 1\end{aligned}$$

$$N = 1024 \Rightarrow \begin{cases} \text{Cost}_{\text{BS}} = 2 \times 1024 - 10 - 1 & = 2037 \\ \text{Cost}_{\text{HSH}} = 1024 \times 10 - (1024 - 1) & = 9217 \\ \text{Cost}_{\text{SS}} = 1024 - 1 & = 1023 \end{cases}$$

Why is HSH scan slower?

- The Hillis Steele Horn scan (HSH) uses more additions than the Blelloch scan (B), but takes fewer steps (iterations). Shouldn't it be faster?

Why is HSH scan slower?

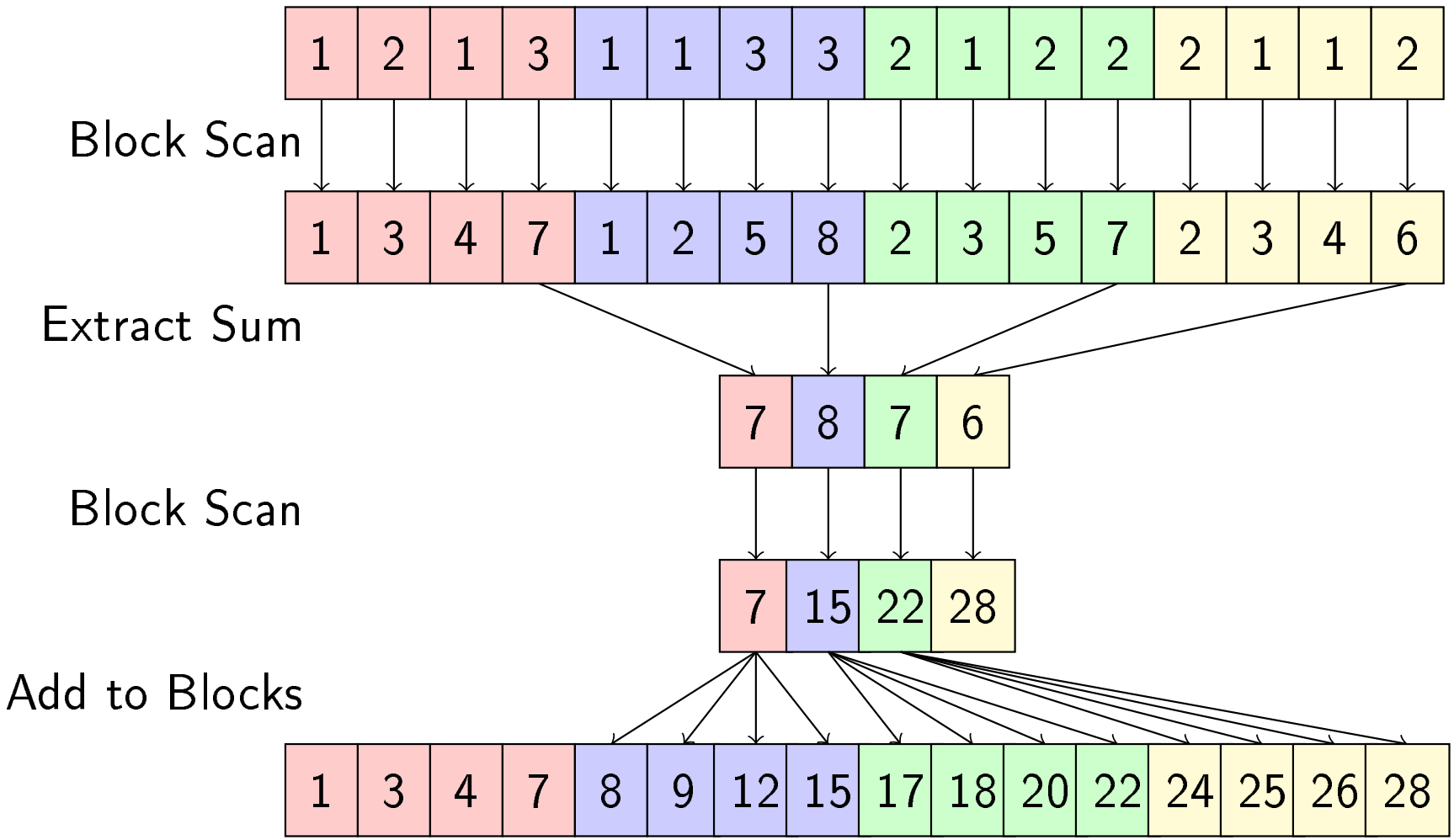
- The Hillis Steele Horn scan (HSH) uses more additions than the Blelloch scan (B), but takes fewer steps (iterations). Shouldn't it be faster?
- Yes IS takes fewer steps but ...
- Remember that the gpu can only execute a small number of warps simultaneously (although it can hold as many warps "*ready to switch in*" as it has cores)
- actual execution of (groups of) warps happens in an interleaved/sequential fashion
- so, in general, more work requires more warps which takes more time

Scaling up to Large Vectors

Until now we have assumed that whole vector fits in a single block. We need to make it work on large vectors that require many blocks.

- Run the scan kernel on the whole vector: each block has been scanned independently into the Y blocks, but the block results have not been propagated between blocks.
- Run a second kernel (this is the same kernel as in the first step, just with different data) that runs a scan on the last elements of each Y block and puts the result into a new vector S
- Run a third kernel that adds each element of S to all elements of the corresponding block of Y.

Large Vector Scan



Maximum Vector Size

Assume the maximum block size is 1024 threads. Then the Blelloch block scan can handle segments of the vector of size 2048 words

- Level 1 & 2 scans only $\Rightarrow$
  - Level 2 scans a single segment = 2K words
  - Level 1 scans 2K segments = 4M words
- Level 1, 2 & 3 scans $\Rightarrow$
  - Level 3 scans a single segment = 2K words
  - Level 2 scans 2K segments = 4M words
  - Level 1 scans 4M segments = 8G words = 32G bytes

But lab machines (GTX 960) have 2G byte global memory $\Rightarrow$ for very large vectors, scans need to be iterated by the host over maximum parallel GPU scans

Code structure for 3 level scan

Pseudocode for a 3 level scan of a vector `d_X` of size `N` to `d_Y`, using a block size of 1024 (= segment size of 2048) might look like this:

```
Allocate globals
    d_X, d_Y both of size N
    d_Sum1, d_Sum1_scanned both of size ceil(N/2048)
    d_Sum2, d_Sum2_scanned both of size 1

Copy host h_X to gpu d_X

block_scan<<<...>>> (d_X, d_Y, N)
extract_sum<<<...>>> (d_Y, d_Sum1)
block_scan<<<...>>> (d_Sum1, d_Sum1_scanned, ceil(N/2048))
extract_sum<<<...>>> (d_Sum1_scanned, d_Sum2)
block_scan<<<...>>> (d_Sum2, d_Sum2_scanned, 1)
block_add<<<...>>> (d_Sum2_scanned, d_Sum1_scanned, ...)
block_add<<<...>>> (d_Sum1_scanned, d_Y, ...)

Copy gpu d_Y to host h_Y
```

- no host/gpu copies between kernels
- `block_scan` and `extract_sum` can be merged into one kernel (with extra bool parameter to request it)

Distributed and Parallel Computing

Lecture 05

Alan P. Sexton

University of Birmingham

Spring 2018

Running Remotely

The following assumes you want to compile on one machine and run on one of the GPU machines in the School lab. This only works on Linux distributions (I assume Ubuntu 16.04)

- From another school machine, do not install anything
- From your own machine, must have the same version of the CUDA toolkit as on the School Lab machines: 8.0 GA2
  - `https://developer.nvidia.com/cuda-toolkit-archive`
  - I advise installing the *network* installer
  - If you have an NVidia gpu in your machine, you can follow the instructions given:

```
sudo dpkg -i cuda-repo-1604_8.0.61-1_amd64.deb
sudo apt-get update
sudo apt-get install cuda
```

  - If you do NOT have an NVidia gpu, or if you do not want to override your video driver, replace the last line above with:

```
sudo apt-get install cuda-toolkit-8-0
```

    This installs everything except the video drivers

Running Remotely

- To run nsight, you need to be running the Oracle (not OpenJDK) Java version 1.7 or 1.8. It does not run with version 9. Note that there is no problem having multiple versions of the Java JDK on your system and switching between them as needed

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread
- Read/Write race conditions e.g. a kernel that is run with multiple threads:

```
__global__ kern(int *A, int len)
{
     int i = blockDim.x * blockIdx.x + threadIdx.x;
     if (i > 1 && i < len)
          A[i] += A[i-1];
}
```

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread
- Read/Write race conditions e.g. a kernel that is run with multiple threads:

```
__global__ kern(int *A, int len)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i > 1 && i < len)
        A[i] += A[i-1];
}
```

- Accessing past the end (or before the beginning) of an allocated array

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread
- Read/Write race conditions e.g. a kernel that is run with multiple threads:

```
__global__ kern(int *A, int len)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i > 1 && i < len)
        A[i] += A[i-1];
}
```

- Accessing past the end (or before the beginning) of an allocated array
- Omitting necessary barrier synchronisations

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread
- Read/Write race conditions e.g. a kernel that is run with multiple threads:

```
__global__ kern(int *A, int len)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i > 1 && i < len)
        A[i] += A[i-1];
}
```

- Accessing past the end (or before the beginning) of an allocated array
- Omitting necessary barrier synchronisations
- Confusion in cudaMemcpy: always copy `h_A` to or from `d_A`, never to or from `d_B`

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread
- Read/Write race conditions e.g. a kernel that is run with multiple threads:

```
__global__ kern(int *A, int len)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i > 1 && i < len)
        A[i] += A[i-1];
}
```

- Accessing past the end (or before the beginning) of an allocated array
- Omitting necessary barrier synchronisations
- Confusion in cudaMemcpy: always copy `h_A` to or from `d_A`, never to or from `d_B`
- Round-off errors

Coalesced Global Memory Access

Global memory accesses occur in *memory transactions* or *bursts* of size 32, 64 or 128 bytes.

- Each memory transaction takes nearly the same amount of time
- Thus reading or writing 8, 16 or 32 words, assuming those reads or writes are appropriately aligned, take approximately the same amount of time as reading a single word.
- So long as the consecutive threads in a warp read consecutive words, only 1 memory transaction is required.
- If consecutive threads read non-consecutive words, then each read requires a separate memory transaction $\Rightarrow$ strided access is much worse than consecutive access
- Array of Structs (AoS) vs Struct of Arrays (SoA)
- Specially important for 2- or 3- dimensional arrays

```
struct {int a; int b} X[LEN] ;        // X[0].a = X[0].b
struct {int a[LEN]; int b[LEN]} X ; // X.a[0] = X.b[0]
```

Shared Memory Banks

Shared memory accesses are approximately 2 orders of magnitude faster than global memory accesses

- Shared Memory in GPUs of compute capability 2.0 or better is divided into 32 equally sized banks
- Shared memory is organised so that 32 consecutive memory word accesses are spread over all 32 banks, one word from each
- On devices of compute capability 3.0 or higher, the banks can be configured to be organised by double, instead of single word.
- Simultaneous access (by different threads in the same warp) to different banks can be serviced simultaneously (4 cycles for a read or write)
- Simultaneous access to the same bank must be serialised
- **Exception:** simultaneous read of the same *address* by all threads in the warp can be serviced simultaneously (broadcast)
- **Exception:** simultaneous read of the same address by some number of threads in the warp can be serviced simultaneously (compute capability 2.0+ multicast)

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

- Restructure our code using `__syncthreads()` to enforce serialised access to the memory location

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

- Restructure our code using `__syncthreads()` to enforce serialised access to the memory location
- Restructure our code so that different threads collect updates locally until complete and then a single thread collates the results and updates the target memory location.

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

- Restructure our code using `__syncthreads()` to enforce serialised access to the memory location
- Restructure our code so that different threads collect updates locally until complete and then a single thread collates the results and updates the target memory location.
- We use an *atomic operation* `atomicAdd(&(A[index]),1)`

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

- Restructure our code using `__syncthreads()` to enforce serialised access to the memory location
- Restructure our code so that different threads collect updates locally until complete and then a single thread collates the results and updates the target memory location.
- We use an *atomic operation* `atomicAdd(&(A[index]),1)`
- Atomic operations *serialise* access to the memory location $\Rightarrow$ limits parallelism

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

- Restructure our code using `__syncthreads()` to enforce serialised access to the memory location
- Restructure our code so that different threads collect updates locally until complete and then a single thread collates the results and updates the target memory location.
- We use an *atomic operation* `atomicAdd(&(A[index]),1)`
- Atomic operations *serialise* access to the memory location $\Rightarrow$ limits parallelism
- `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomic-functions`

Distributed and Parallel Computing

Lecture 06

Alan P. Sexton

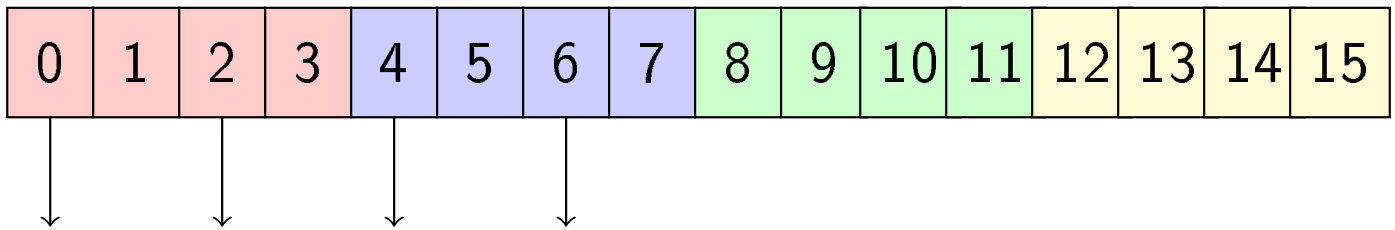
University of Birmingham

Spring 2018

Global Memory Coalescing — Revisited

Global memory is partitioned in **burst sections**

- Whenever a location in global memory is accessed, all other locations in the same section are also delivered
- Burst sections can be 128 bytes or more
- When a warp executes a load or store, the number of dram requests issued (and serialised) is the number of different burst sections addressed
- For example: warp size 4, burst size 16 bytes (4 words), stride=2: 2 memory transactions required



- Order of access doesn't matter

Shared Memory Bank Conflicts — Revisited

Shared memory is stuctured into **banks**

- Modern GPUs have 32 4-byte word banks but can be configured as 32 8-byte double word banks
- The bank used for a word address is the remainder when you divide the word address by the number of banks
- Shared memory can deliver/accept 1 word simultaneously from each bank in a single read/write transaction
- Multiple accesses to the same bank are serialized
- For example: warp size 4, 4 banks, 32 words:
  - Warp accesses (00, 01, 02, 03) or (00, 05, 10, 15) in 1 op
  - Warp accesses (00, 02, 04, 06) in 2 ops, (00, 04, 08, 12) in 4

00	01	02	03
04	05	06	07
08	09	10	11
12	13	14	15

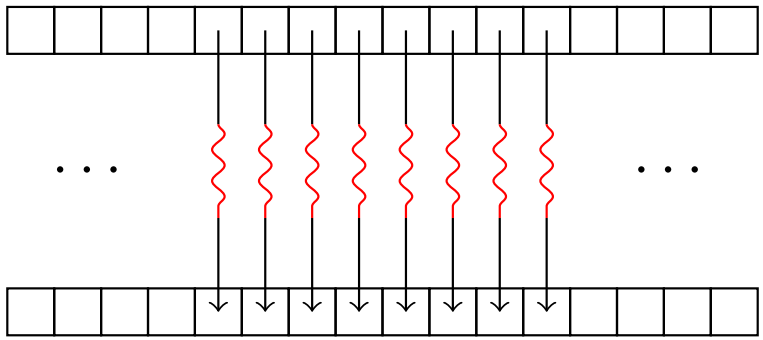
General Parallel Programming Patterns

There are a number of standard parallel programming patterns that form the basic building blocks of most GPU programs:

- Map
- Gather
- Scatter
- Stencil
- Transpose
- Reduce
- Scan/Sort
- Histogram

Map

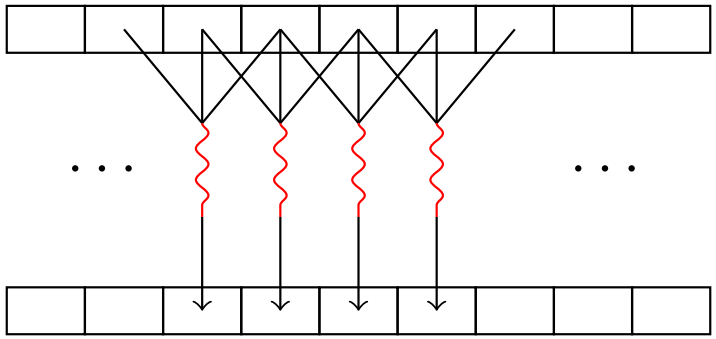
Map uses each thread to take an input value from the location corresponding to the block/thread id and write an output value to the location corresponding to the block/thread id with no two threads reading from or writing to the same location.



- One-to-one pattern
- No read/write race issues if source and destination are different
- Easy to ensure coalesced global memory accesses
- Easy to avoid shared memory bank collisions

Gather

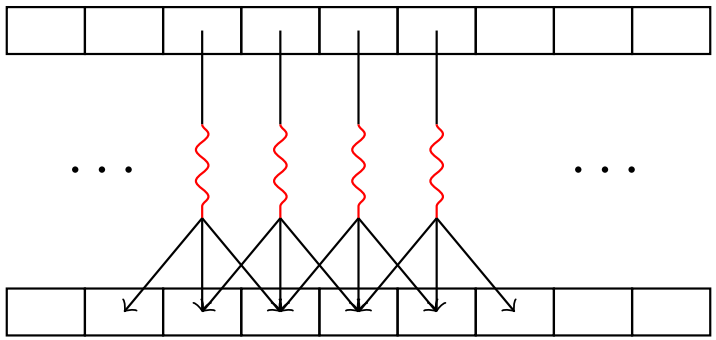
Gather uses each thread to take one or more input value from some locations (not just one from the direct location indicated by the thread and block id) and write an output value to one location with no two threads writing to the same location. Multiple different threads may share some read locations.



- Gather locations may have different patterns for each thread
- Many-to-one pattern
- No read/write race issues if source and destination are different
- Easy to ensure coalesced global memory accesses on writes
- Easy to avoid shared memory bank collisions on writes
- Care needed on read access patterns

Scatter

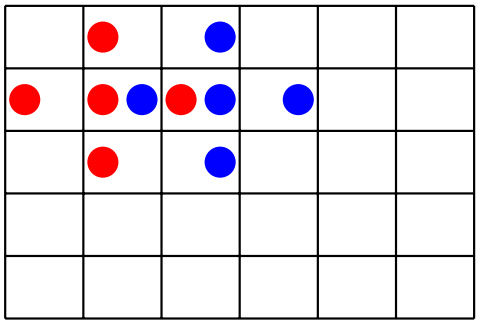
Scatter uses each thread to read the value from the location corresponding to the block/thread id and write output values to a number of locations Multiple different threads may share some write locations (e.g. using read/write operations such as increment)

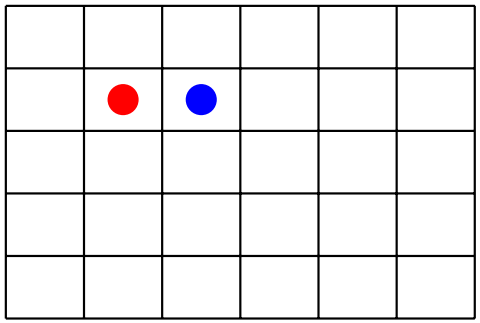


- Write locations may have different pattern for each thread
- One-to-many pattern
- Potential overwrite race issues
- Easy to ensure coalesced global memory accesses on reads
- Easy to avoid shared memory bank collisions on reads
- Care needed on write access patterns

Stencil

Stencil is a special case of Gather where the pattern of reads is constant across the threads.



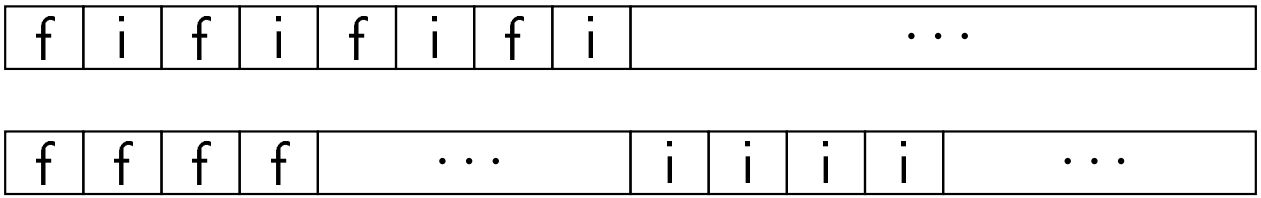


- Read locations have the same pattern for each thread
- Several-to-one pattern
- No read/write race issues if source and destination are different
- Easy to ensure coalesced global memory accesses on writes
- Easy to avoid shared memory bank collisions on writes
- Care needed on read access patterns

Transpose

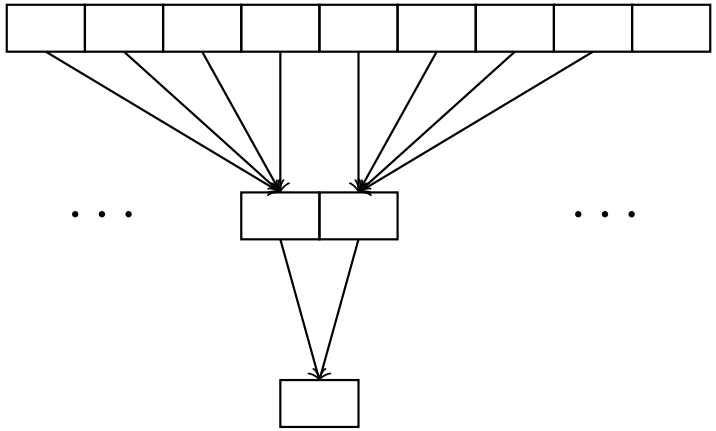
Transpose is most commonly the standard transpose operation on matrices, but is also used for restructuring datastructures for efficient memory accesses.

```
struct {
    float f;
    int i;
} X[LEN];
```



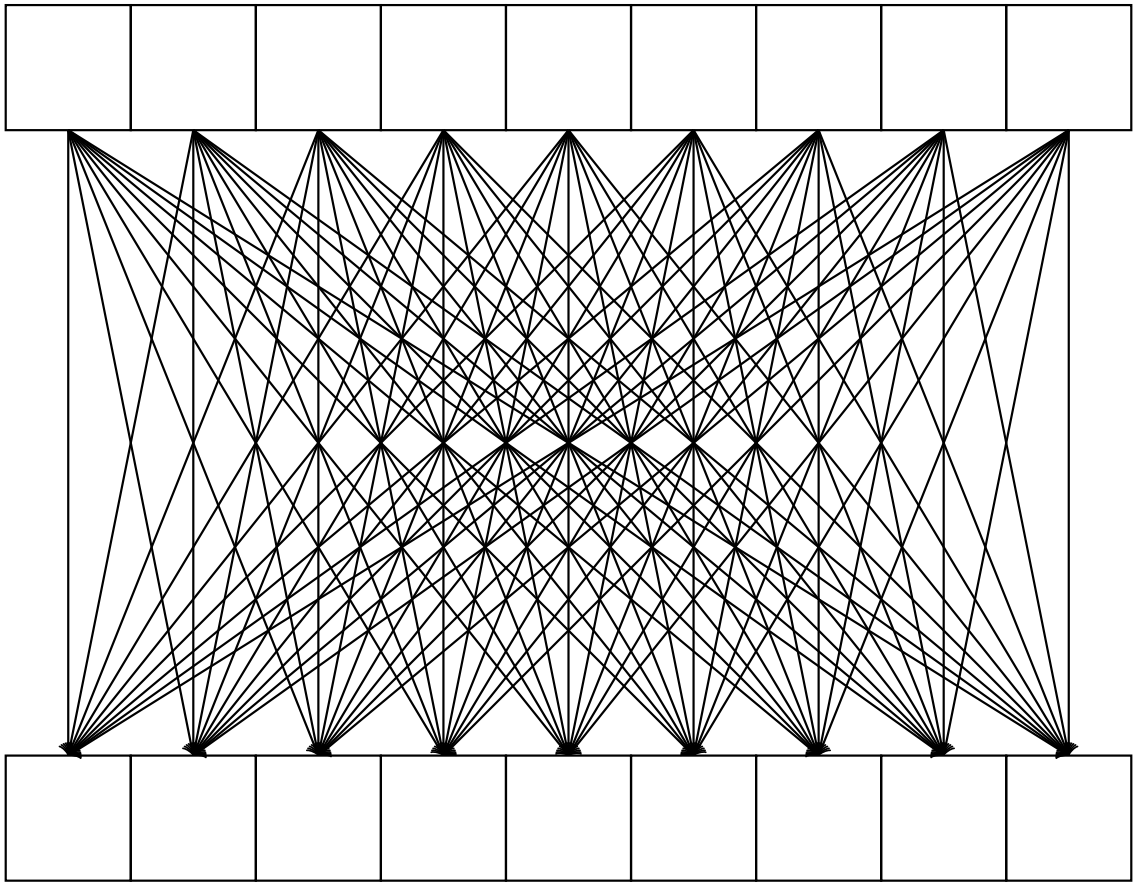
- Can be implemented with a *Scatter* or with a *Gather*

Reduce



- Many-to-one

Scan/Sort



- All-to-All (or at least Many-to-All)

Histogram — Serial Version

```
for (i=0; i < NUM_BINS; i++)
    bin[i] = 0;
for (i=0; i < NUM_VALS; i++)
    bin[computeBin(vals[i])] ++;
```

Histogram — Parallel Version (with error)

```
__global__ void hist(int *d_bins, const int *d_vals,
                     const int NUM_BINS)
{
    int id =  blockDim.x * blockIdx.x + threadIdx.x ;
    int val = d_vals[id] ;
    int bin = val % NUM_BINS; // computeBin function

    d_bins[bin]++ ;
}
```

Histogram — Parallel Version (corrected)

```
__global__ void hist(int *d_bins, const int *d_vals,
                     const int NUM_BINS)
{
    int id =  blockDim.x * blockIdx.x + threadIdx.x ;
    int val = d_vals[id] ;
    int bin = val % NUM_BINS; // computeBin function

    atomic_add(&d_bins[bin], 1) ;
}
```

Histogram — Parallel Version (corrected)

```
__global__ void hist(int *d_bins, const int *d_vals,
                     const int NUM_BINS)
{
    int id =  blockDim.x * blockIdx.x + threadIdx.x ;
    int val = d_vals[id] ;
    int bin = val % NUM_BINS; // computeBin function

    atomic_add(&d_bins[bin], 1) ;
}
```

- Scalability?

Histogram — Parallel Version (corrected)

```
__global__ void hist(int *d_bins, const int *d_vals,
                     const int NUM_BINS)
{
    int id =  blockDim.x * blockIdx.x + threadIdx.x ;
    int val = d_vals[id] ;
    int bin = val % NUM_BINS; // computeBin function

    atomic_add(&d_bins[bin], 1) ;
}
```

- Scalability?
- 1M values $\Rightarrow$ 1M atomic adds

Improving Parallel Histogram

- Have each thread calculate its own histogram on a subset of the data and then add to final result

Improving Parallel Histogram

- Have each thread calculate its own histogram on a subset of the data and then add to final result
  - No atomic adds necessary to build each thread's histogram
  - 1M values, 1024 threads 16 bins $\Rightarrow$ 1024 values per thread

Improving Parallel Histogram

- Have each thread calculate its own histogram on a subset of the data and then add to final result
  - No atomic adds necessary to build each thread's histogram
  - 1M values, 1024 threads 16 bins $\Rightarrow$ 1024 values per thread
  - 1024 histograms of 16 bins $\Rightarrow$ 16 * 1024 = 16284 atomic adds

Improving Parallel Histogram

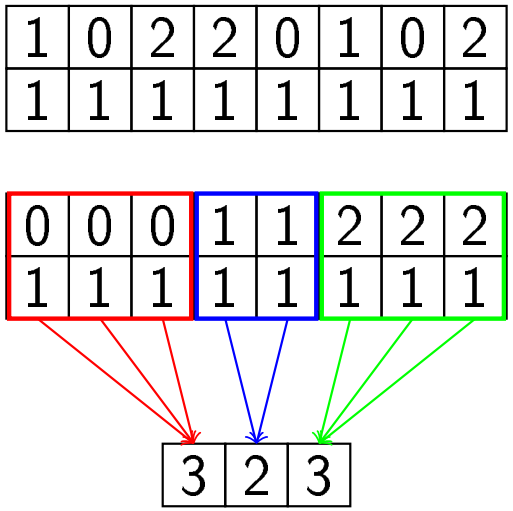
- Have each thread calculate its own histogram on a subset of the data and then add to final result
  - No atomic adds necessary to build each thread's histogram
  - 1M values, 1024 threads 16 bins $\Rightarrow$ 1024 values per thread
  - 1024 histograms of 16 bins $\Rightarrow$ 16 * 1024 = 16284 atomic adds
- Better: Have each thread calculate its own histogram on a subset of the data and then **REDUCE** to final result

Improving Parallel Histogram

- Have each thread calculate its own histogram on a subset of the data and then add to final result
  - No atomic adds necessary to build each thread's histogram
  - 1M values, 1024 threads 16 bins $\Rightarrow$ 1024 values per thread
  - 1024 histograms of 16 bins $\Rightarrow$ 16 * 1024 = 16284 atomic adds
- Better: Have each thread calculate its own histogram on a subset of the data and then **REDUCE** to final result
- Alternative: Sort keys (key is bin number), then Reduce-by-key (value is 1)

Improving Parallel Histogram

- Have each thread calculate its own histogram on a subset of the data and then add to final result
  - No atomic adds necessary to build each thread's histogram
  - 1M values, 1024 threads 16 bins $\Rightarrow$ 1024 values per thread
  - 1024 histograms of 16 bins $\Rightarrow$ 16 * 1024 = 16284 atomic adds
- Better: Have each thread calculate its own histogram on a subset of the data and then **REDUCE** to final result
- Alternative: Sort keys (key is bin number), then Reduce-by-key (value is 1)



Distributed and Parallel Computing

Lecture 07

Alan P. Sexton

University of Birmingham

Spring 2018

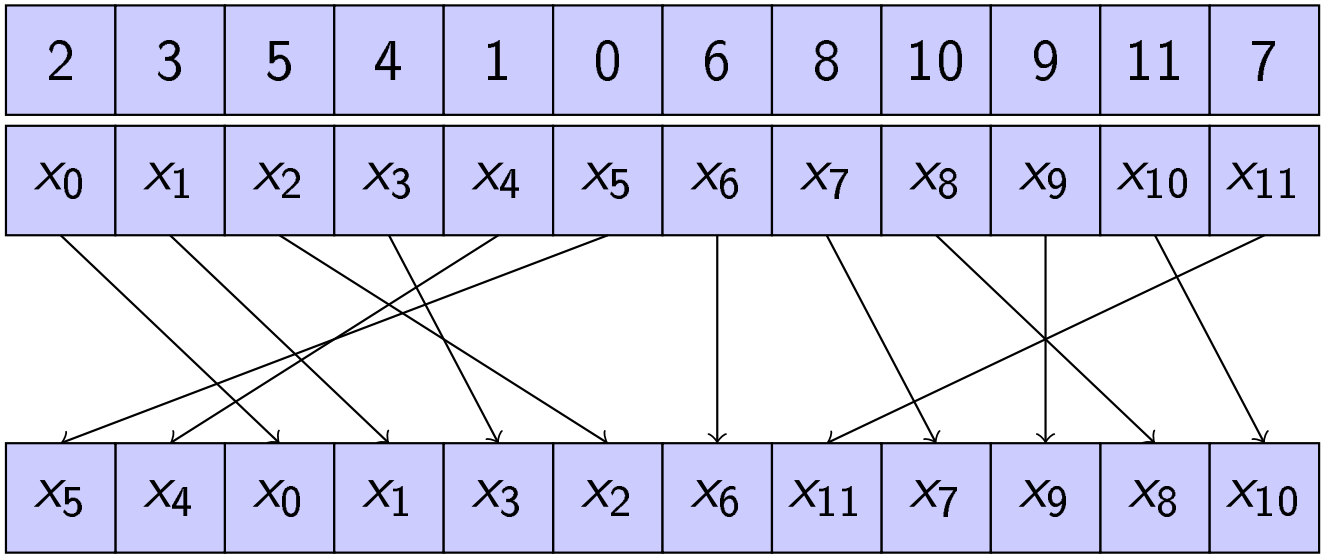
Basic Operations

Operations:

- `map(X, Y, op())`
  - Map memory access pattern
  - e.g. square every element of an array
  - e.g. vectorAdd
- `reduce(X, y, bin_assoc_op())`
  - Many-to-one memory pattern
  - e.g. sum the elements of an array
- `scan(X, Y, bin_assoc_op())`
  - Many-to-many memory pattern
  - e.g. calculate the cumulative sum of an array

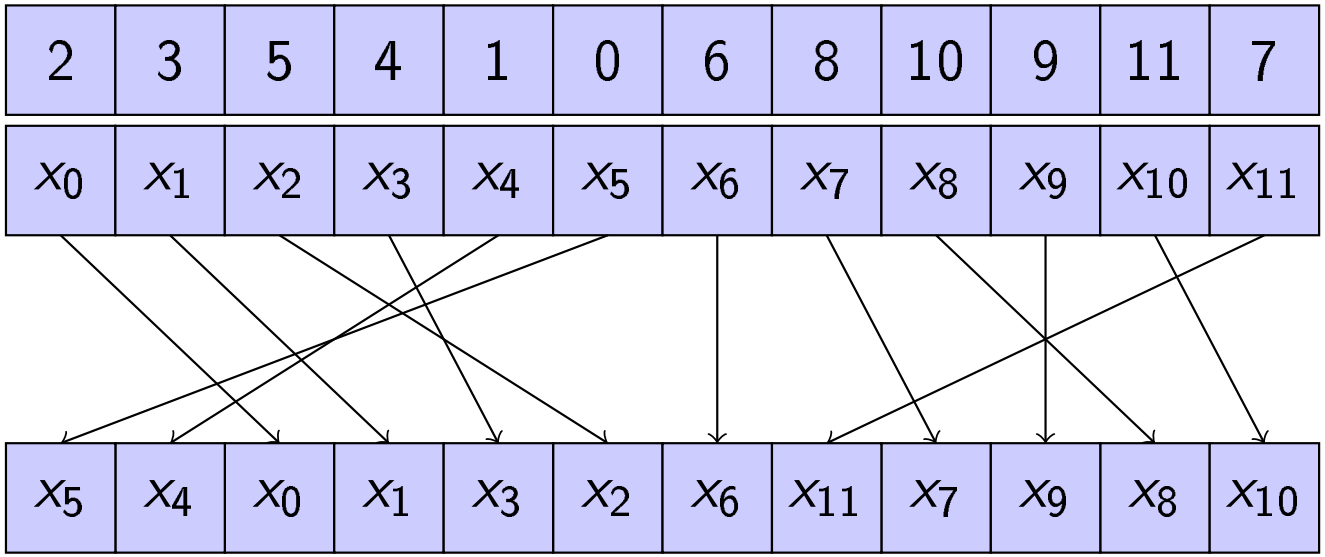
Scatter

The Scatter operation takes an input array I and a target address array T and sends each input element to the target address location in the output array: $O[T[i]] = I[i]$



Scatter

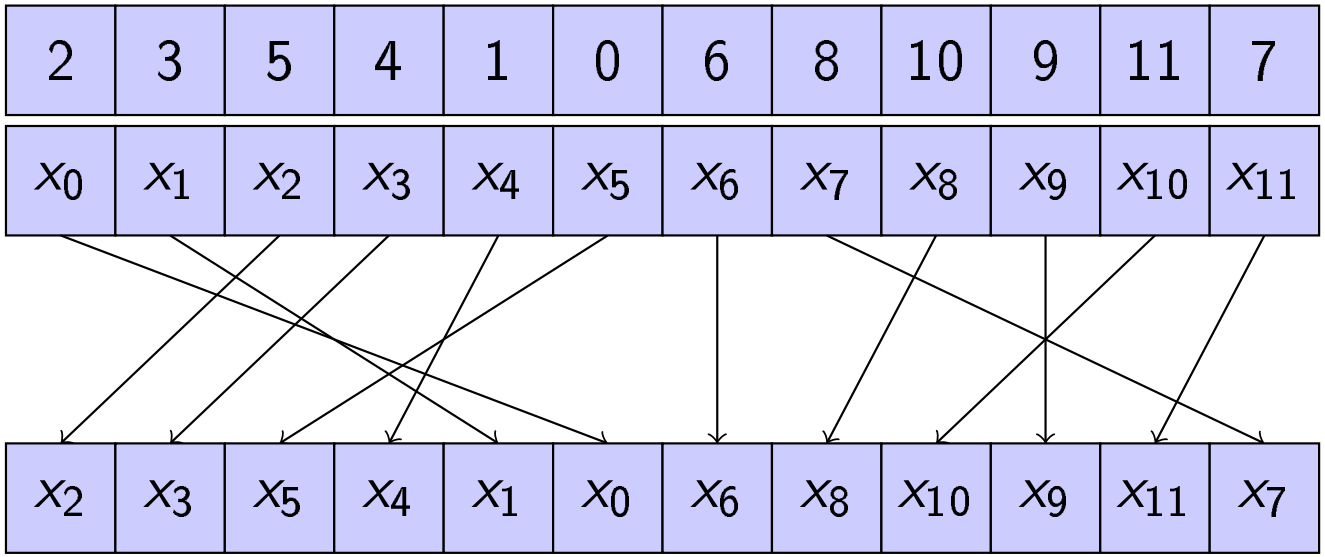
The Scatter operation takes an input array I and a target address array T and sends each input element to the target address location in the output array: $O[T[i]] = I[i]$



- There can be filtered variants where the assignment of an input element to the target is not carried out if either:
  - the target address is negative, or
  - a supplied extra predicate function applied to the input element evaluates to `TRUE`, or
  - a supplied extra array of the same size as the input array has 0 in the corresponding position

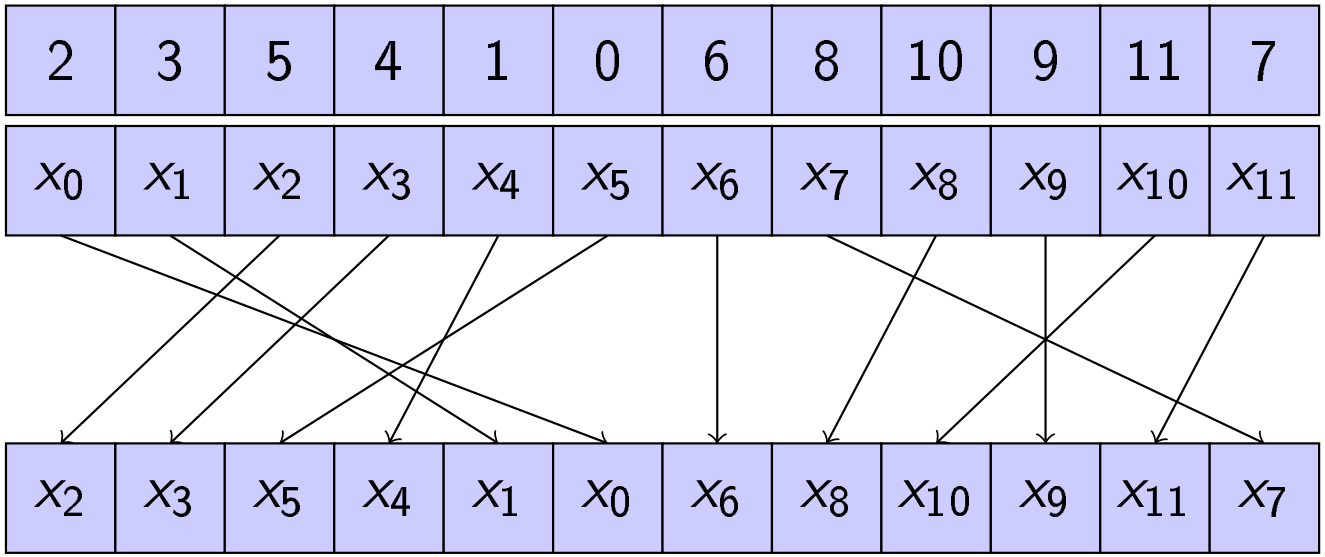
Gather

The Gather operation takes an input array I and a source address array S and writes to each output array (O) cell the value read from the location in the input array described by the corresponding element of the source address array: $O[i] = I[S[i]]$



Gather

The Gather operation takes an input array I and a source address array S and writes to each output array (O) cell the value read from the location in the input array described by the corresponding element of the source address array: $O[i] = I[S[i]]$



- As in the case of Scatter, there are similar filtered variants

Exclusive Scan

The scan we have studied is *inclusive*: it includes the current input element in its current cumulative result across the full input array. There is an exclusive variant:

Input	1	2	1	3	1	1	3	3	2	1	2	2
Inclusive Scan	1	3	4	7	8	9	12	15	17	18	20	22
Exclusive Scan	0	1	3	4	7	8	9	12	15	17	18	20

Exclusive Scan is a small modification to the scan algorithms (Hills-Steele-Horn and Blelloch) we have already studied. It does **NOT** just do an inclusive scan and shift the words on by one.

Segmented Scan

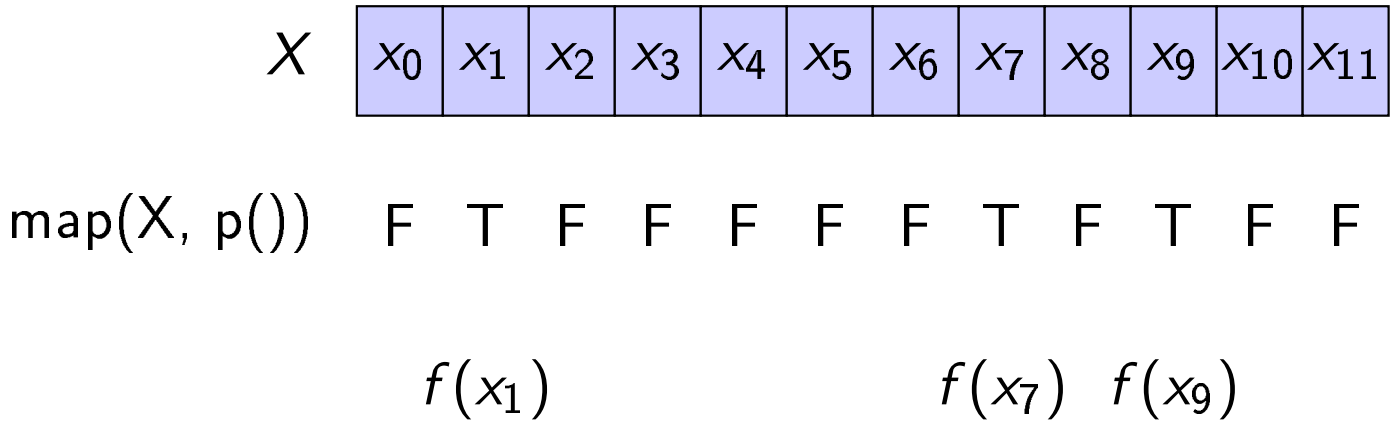
Often we want to do different scans on separate parts of the input array. It would be inefficient to do so by running multiple scan kernels one after the other. Instead we use a **segmented scan**:

Input	1	2	1	3	1	1	3	3	2	1	2	2
Headers	1	0	0	1	0	0	0	0	0	1	0	0
Alternative Headers	0	3	9									
Segmented Inclusive Scan	1	3	4	3	4	5	8	11	13	1	3	5
Segmented Exclusive Scan	0	1	3	0	3	4	5	8	11	0	1	3

This is again a small modification to our previous algorithms and, in spite of the extra work of dealing with the header array, has the same step and work complexity, though is a bit slower.

Filtering an Array: 1

Let's say we want to apply a function, $f()$, to all elements of an array that satisfy predicate $p()$:

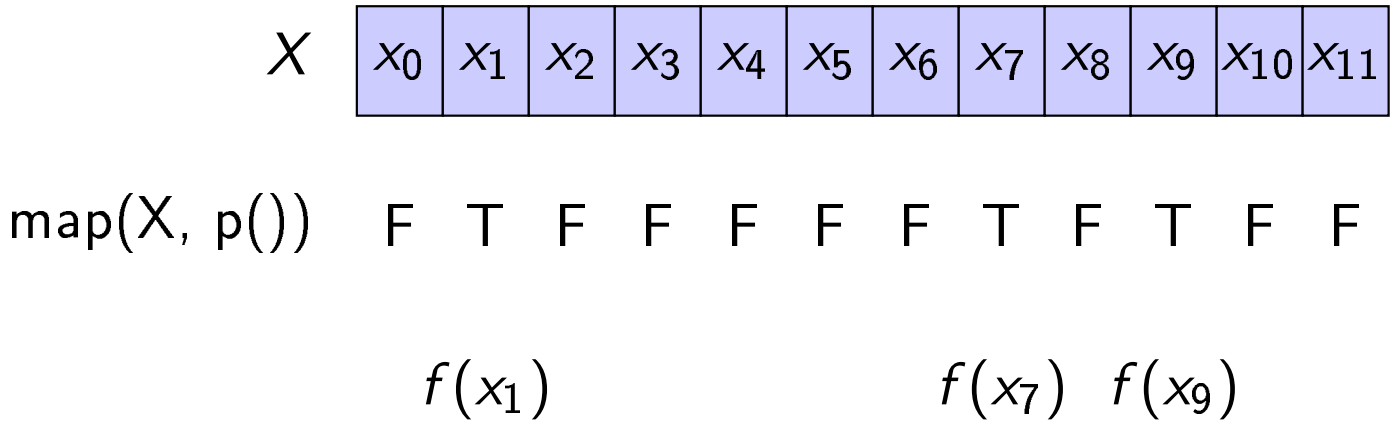


- We could do it by having each thread execute the following:

```
if (p(X(i))
    f(X(i));
```

Filtering an Array: 1

Let's say we want to apply a function, $f()$, to all elements of an array that satisfy predicate $p()$:



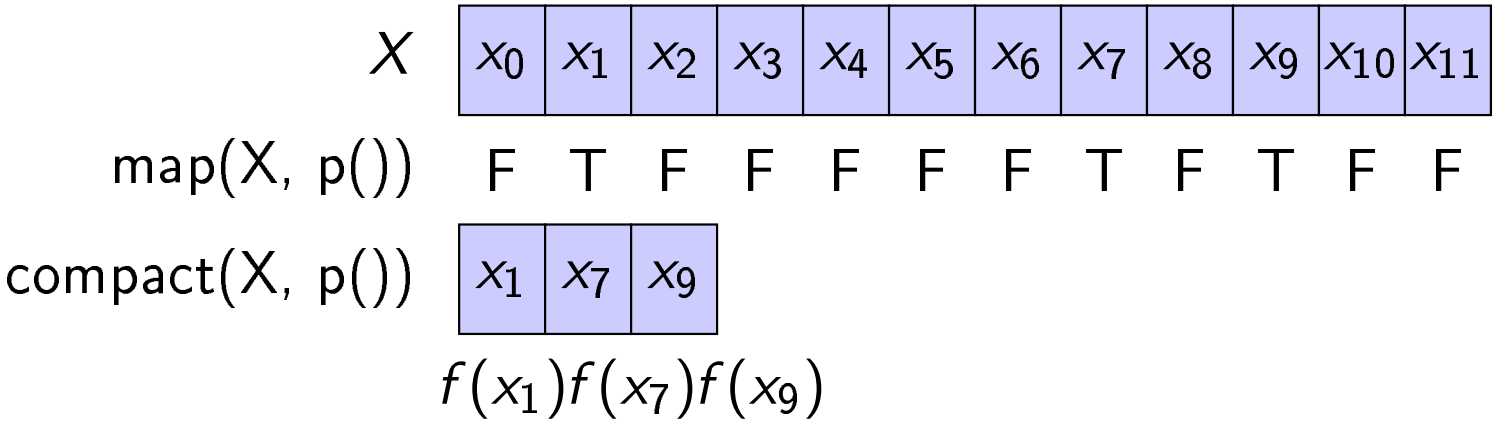
- We could do it by having each thread execute the following:

```
if (p(X(i))
    f(X(i));
```

- But: all threads execute $p()$, only some threads execute $f()$, and those threads are not compacted, so lots of divergence
  - e.g. Array of 1,000,000 entries, 1 in 20 satisfy the predicate, so 19 in 20 execute for the full length of $f()$ doing nothing while the remaining threads in the warp do something useful

Filtering an Array: 2

An alternative approach:

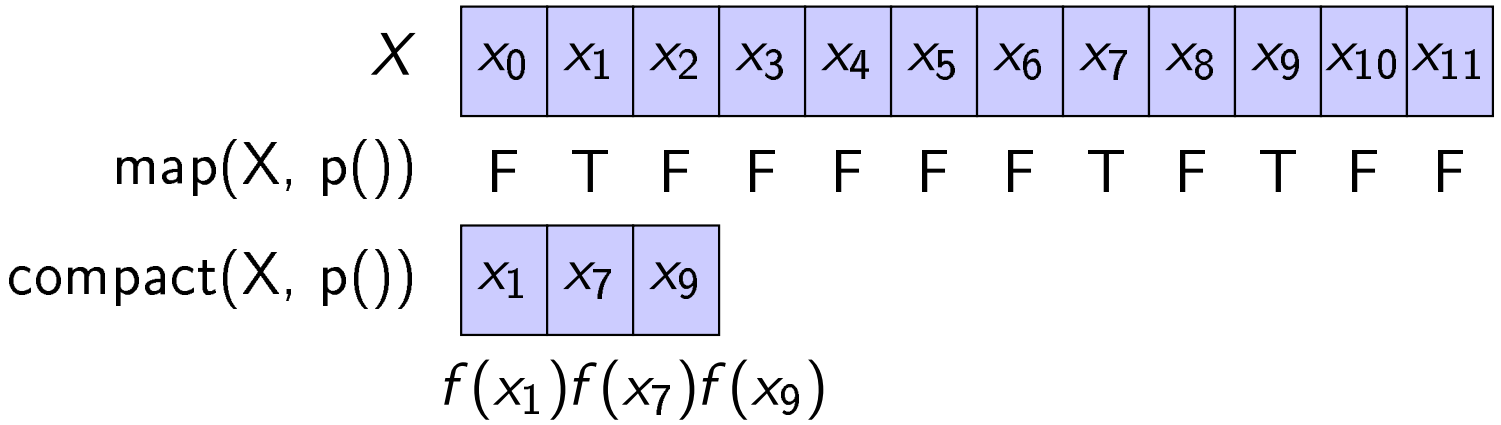


- The code for this would look like (each line in a different kernel)

```
compact(X, Y, p());
map(Y, f());
```

Filtering an Array: 2

An alternative approach:

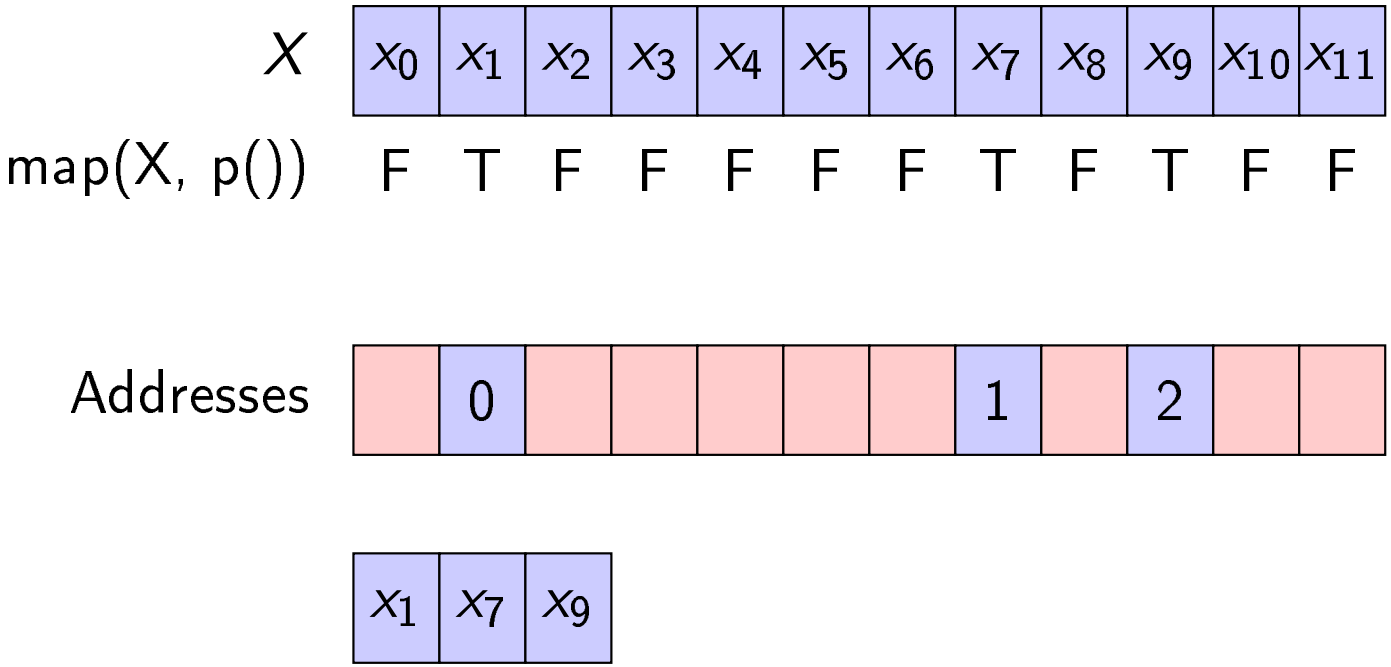


- The code for this would look like (each line in a different kernel)

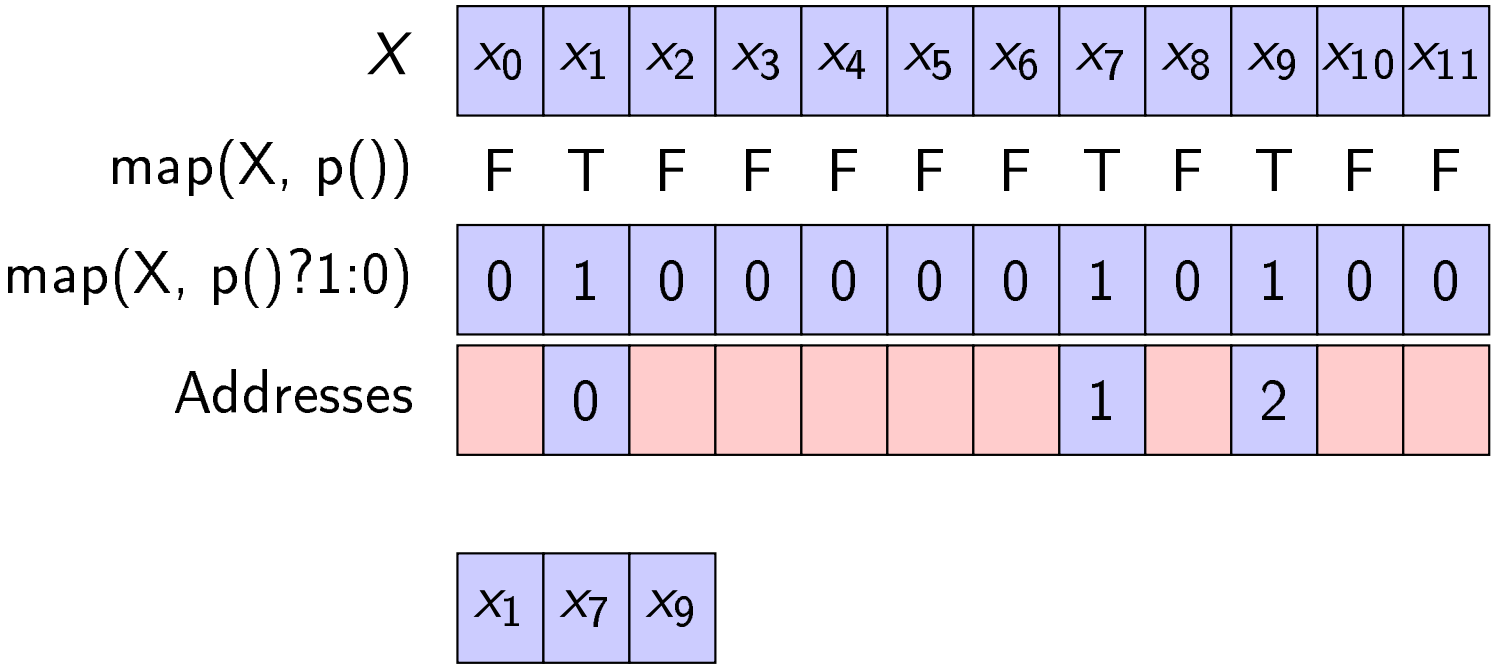
```
compact(X, Y, p());
map(Y, f());
```

- all threads are used to execute the first line, but only as many threads as the length of the filtered array execute $f()$, and those threads are compact, so no divergence
  - e.g. Array of 1,000,000 entries, 1,000,000 threads executing the compact, 1 in 20 satisfy the predicate, so only 50,000 threads execute $f()$

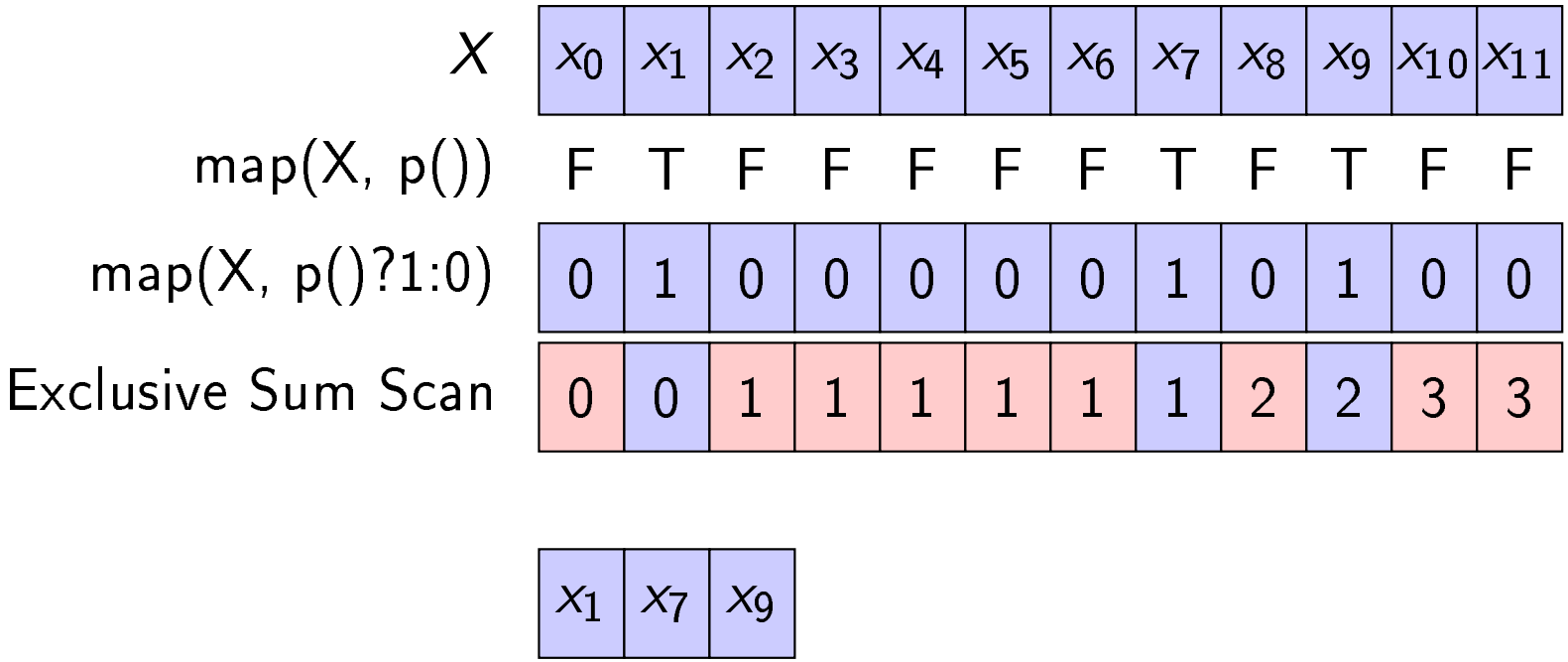
Compact


X
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11}
map(X, p())
F T F F F F F T F T F F
Addresses
0 1 2
x_1 x_7 x_9

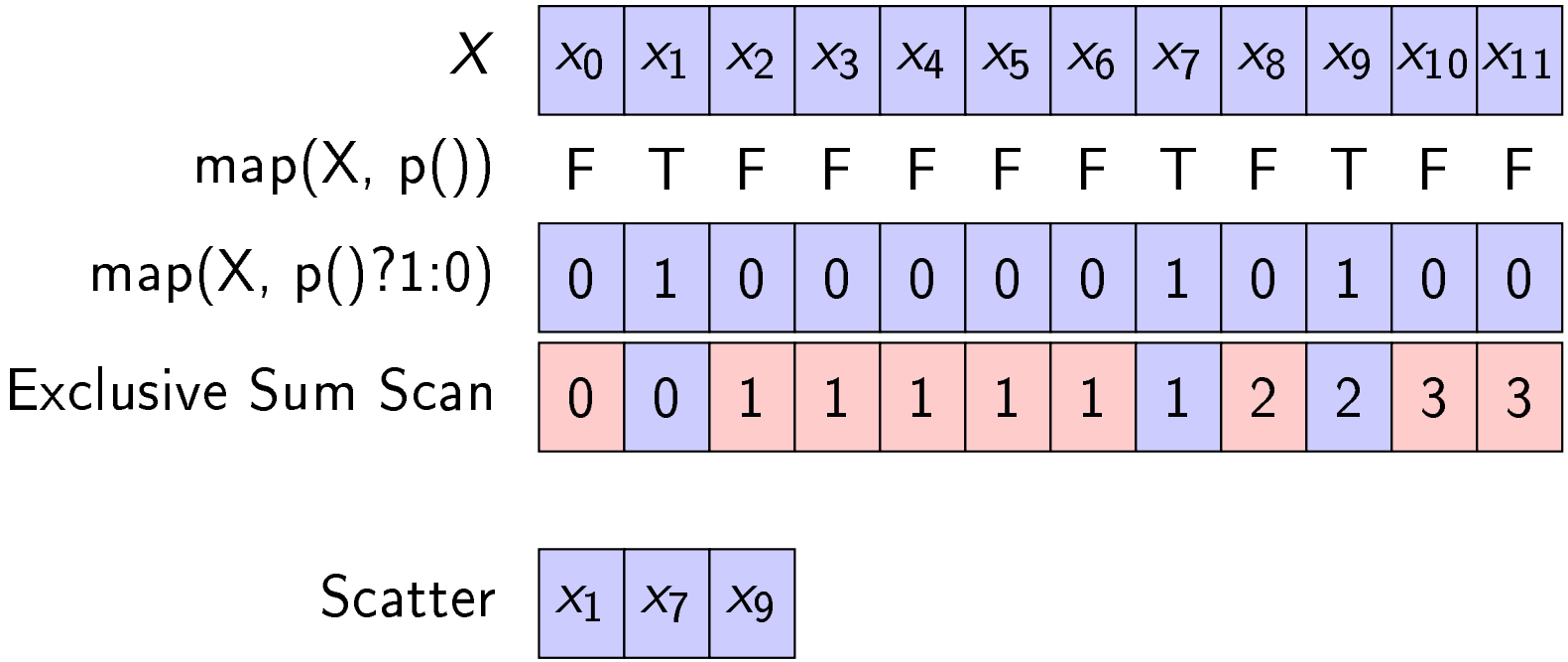
Compact


X
x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11
map(X, p())
F T F F F F F T F T F F
map(X, p()?1:0)
0 1 0 0 0 0 0 1 0 1 0 0
Addresses
0 1 2
x1 x7 x9

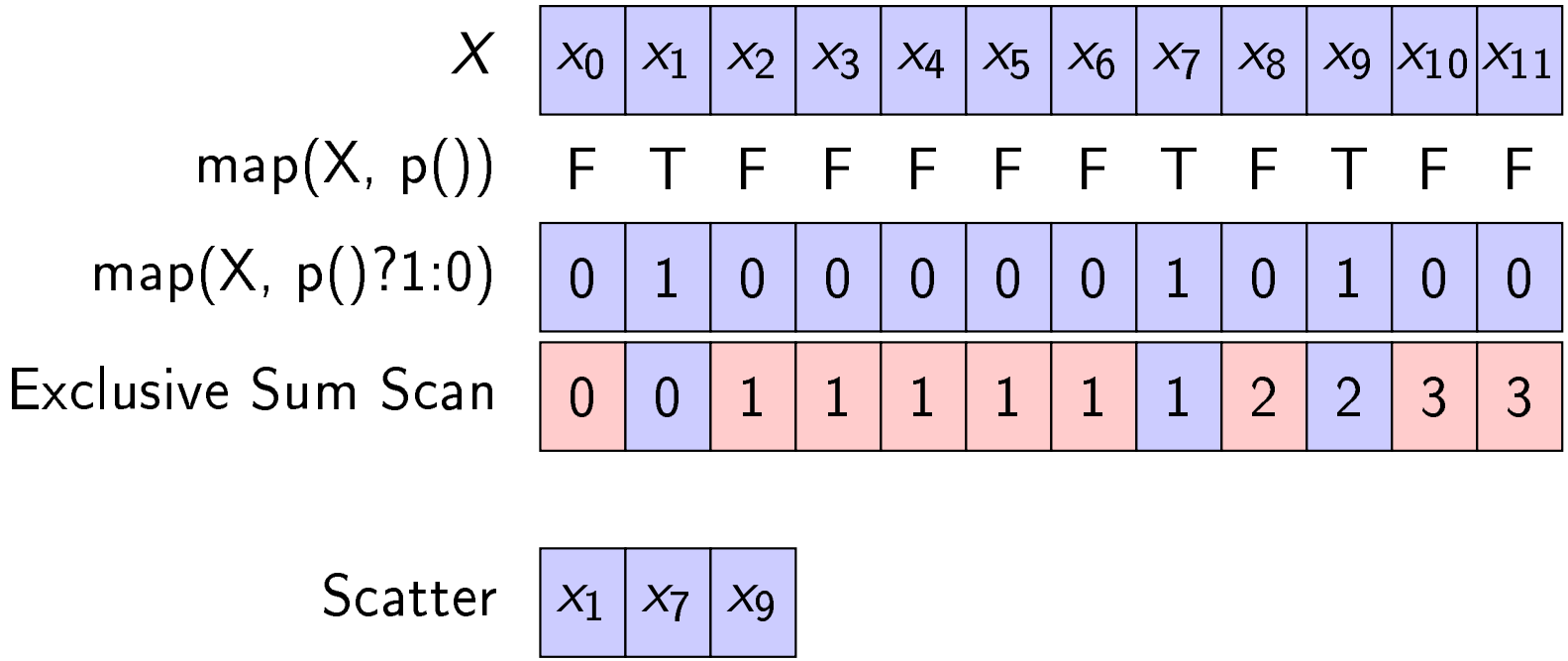
Compact


X
x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11
map(X, p())
F T F F F F F T F T F F
map(X, p()?1:0)
0 1 0 0 0 0 0 1 0 1 0 0
Exclusive Sum Scan
0 0 1 1 1 1 1 1 2 2 3 3
x1 x7 x9

Compact


X
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11}
map(X, p())
F T F F F F F T F T F F
map(X, p()?1:0)
0 1 0 0 0 0 0 1 0 1 0 0
Exclusive Sum Scan
0 0 1 1 1 1 1 1 2 2 3 3
Scatter
x_1 x_7 x_9

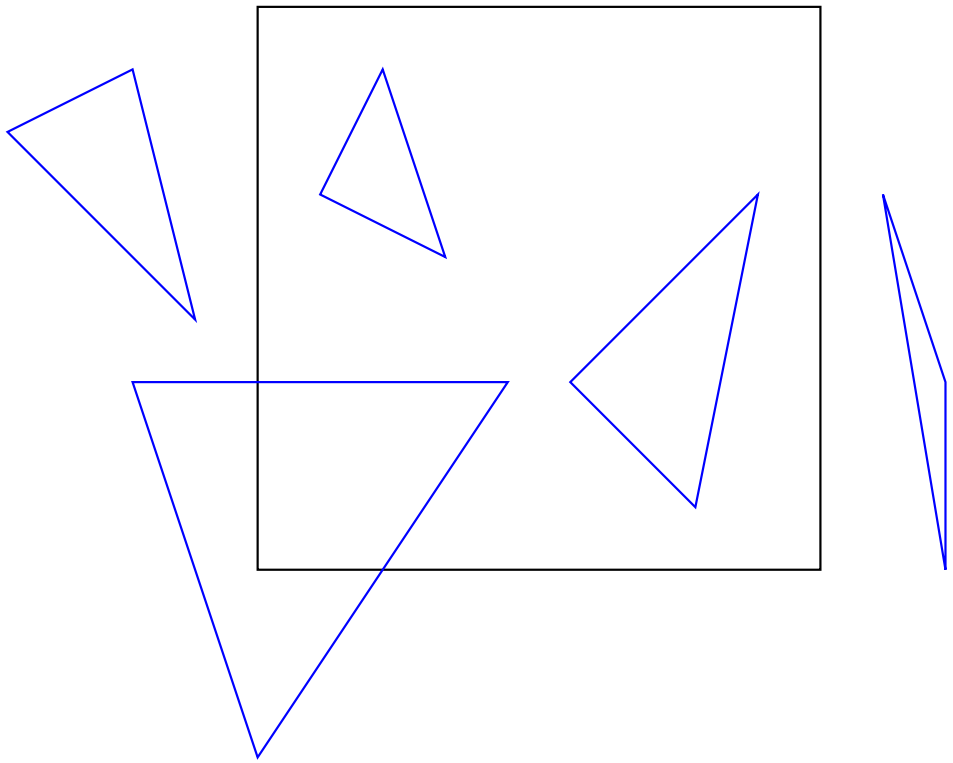
Compact



1. Map Predicate (1 for True, 0 for False) of Input into P
2. Exclusive Sum Scan of P into S
3. Scatter input into output using addresses from S if corresponding P value is greater than 0

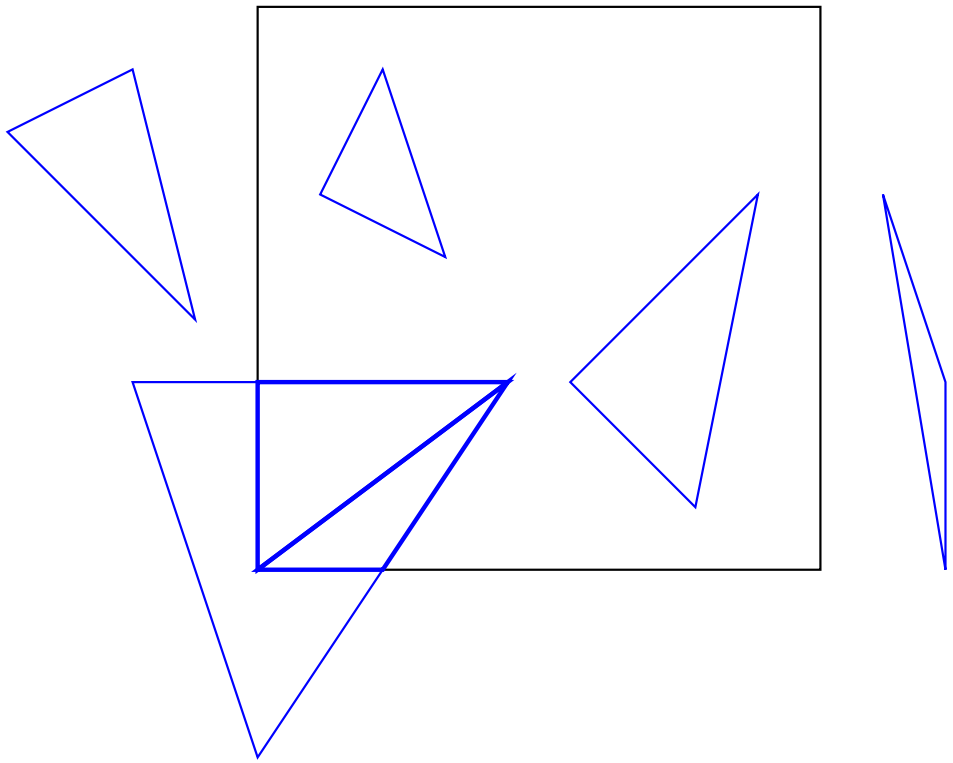
Generalised Compact

We can generalize compact: Compact reserves 1 output cell for each input cell that matches the predicate. Sometimes you may want more than one. Consider clipping triangles to a view port:



Generalised Compact

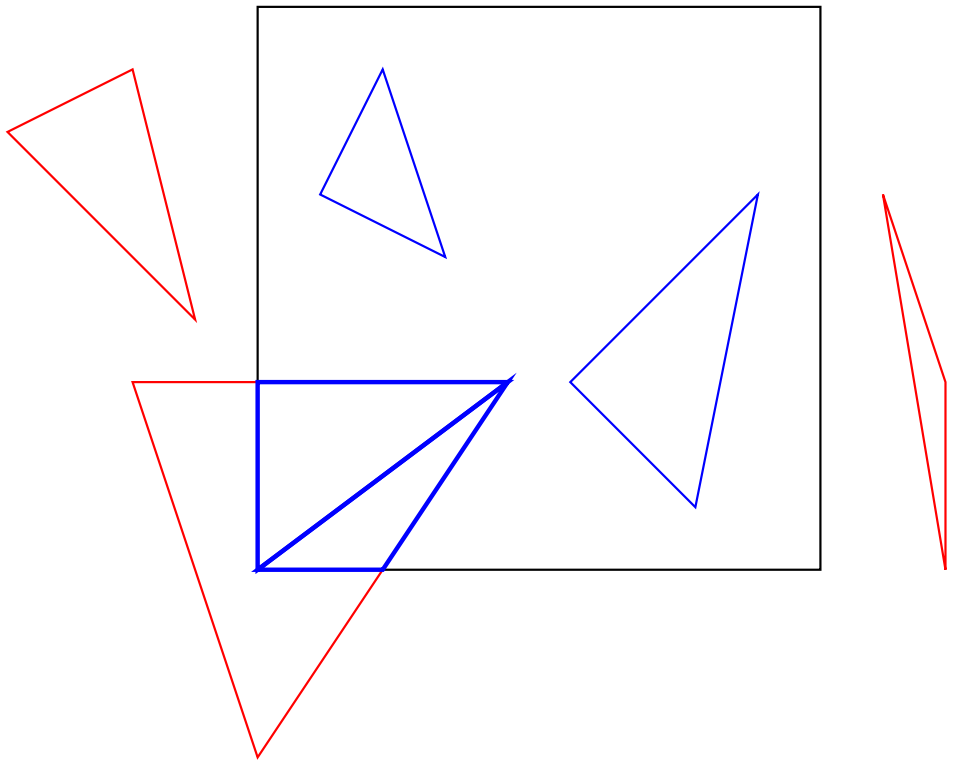
We can generalize compact: Compact reserves 1 output cell for each input cell that matches the predicate. Sometimes you may want more than one. Consider clipping triangles to a view port:



- Can have up to 5 triangles from clipping a single triangle

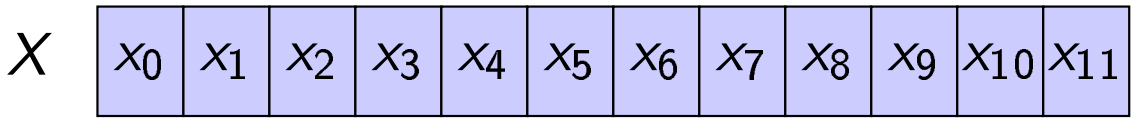
Generalised Compact

We can generalize compact: Compact reserves 1 output cell for each input cell that matches the predicate. Sometimes you may want more than one. Consider clipping triangles to a view port:



- Can have up to 5 triangles from clipping a single triangle
- This is a generalised compact operation

Generalised Compact



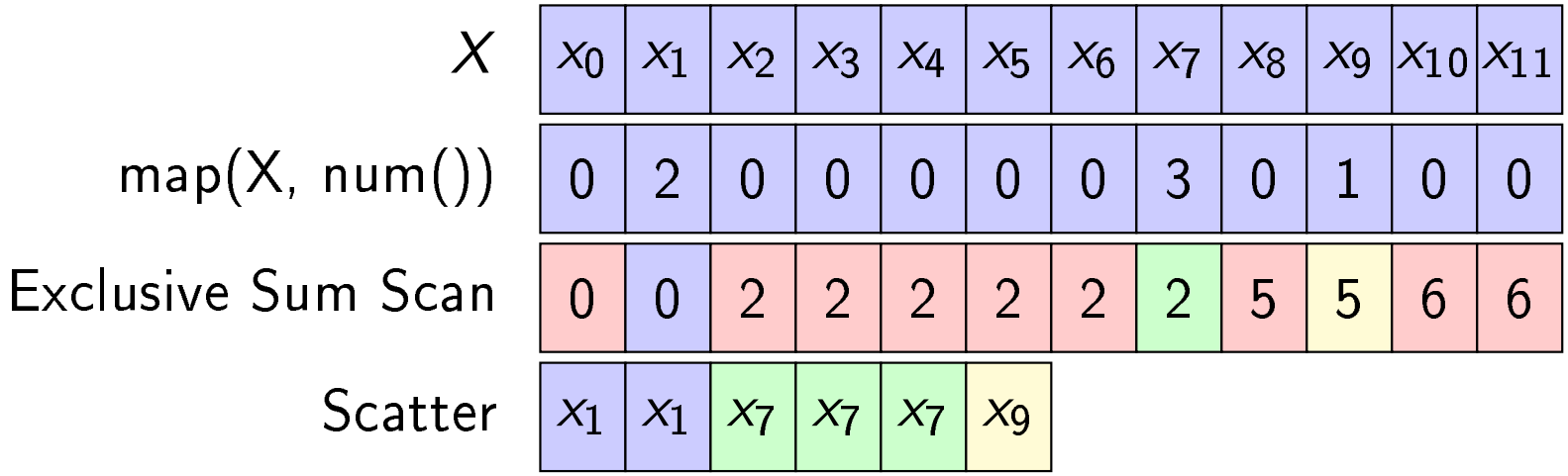
Generalised Compact

X	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
map(X, num())	0	2	0	0	0	0	0	3	0	1	0	0

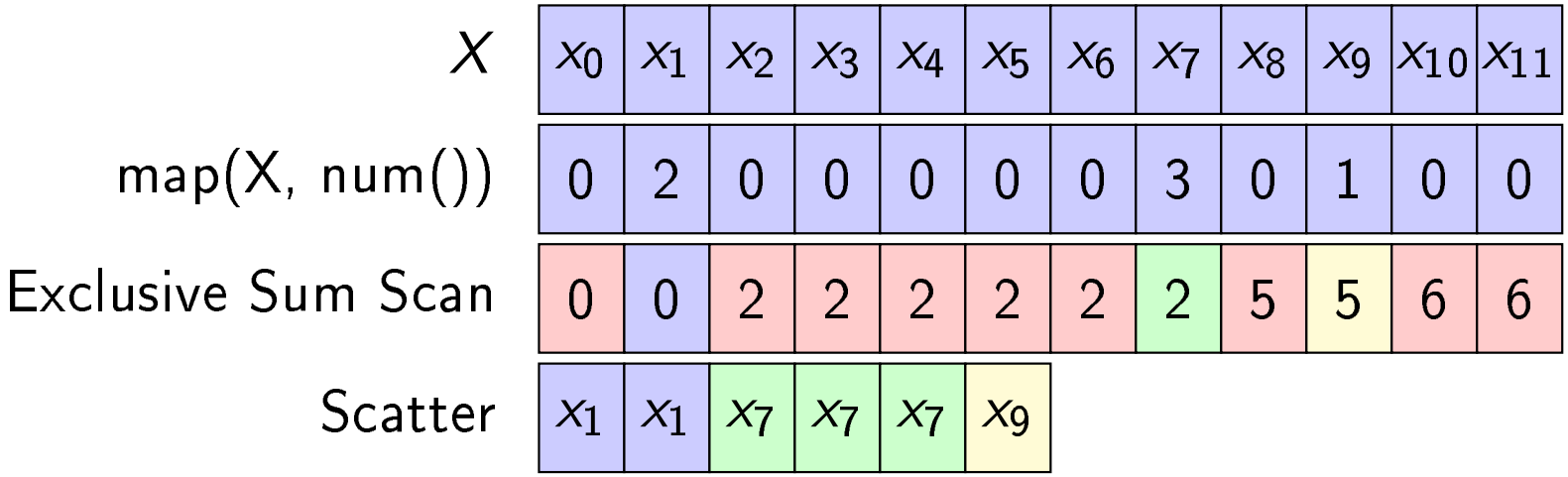
Generalised Compact

X	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
map(X, num())	0	2	0	0	0	0	0	3	0	1	0	0
Exclusive Sum Scan	0	0	2	2	2	2	2	2	5	5	6	6

Generalised Compact



Generalised Compact



- The same Exclusive Sum Scan and Scatter as before, but now
  - the num() function returns the number of output cells needed for each input element and
  - each thread i now generates all num(i) output elements corresponding to a single input element x_i

Application: Sparse Matrix Dense Vector Mult (SpMv)

Recall matrix vector multiplication:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

Sparse matrices have many zeros

- Occur *VERY* frequently
- Traditional 2-dimensional array represention:
  - Very wasteful of space
  - Lots of processing power wasted on multiplications by zero

Representation: Compressed Sparse Row

$$\begin{bmatrix} 0 & b & c \\ d & e & f \\ 0 & 0 & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} by + cz \\ dx + ey + fz \\ iz \end{bmatrix}$$

- V Value array: non-zero values in T to B, L to R order:

$$\begin{bmatrix} b & c & d & e & f & i \end{bmatrix}$$

- C Column array: the column of the corresponding value:

$$\begin{bmatrix} 1 & 2 & 0 & 1 & 2 & 2 \end{bmatrix}$$

- R Row pointer array: the index in V of each element that starts a row:

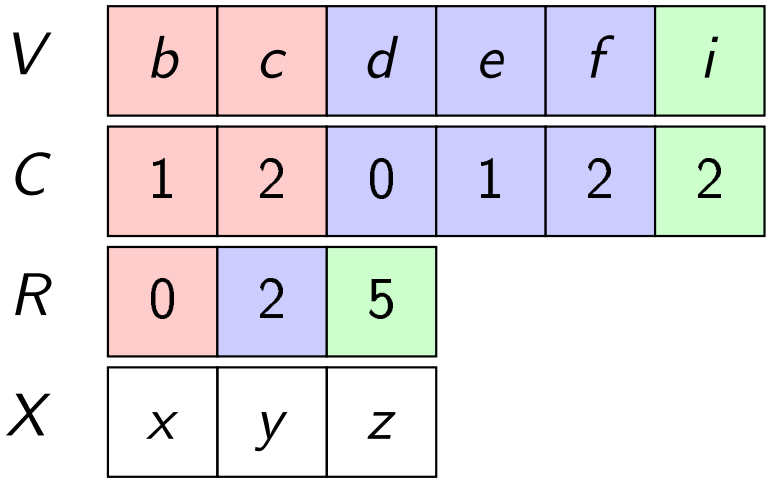
$$\begin{bmatrix} 0 & 2 & 5 \end{bmatrix}$$

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

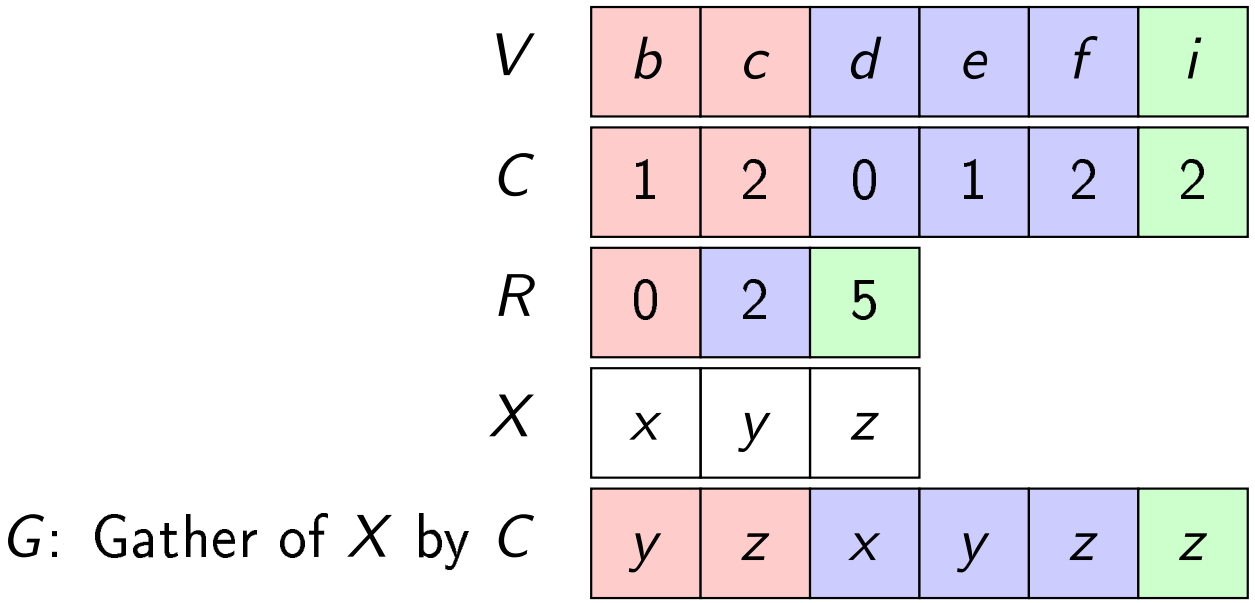
```
V: [b c | d e f | i]
C: [1 2 | 0 1 2 | 2]
R: [0 2 5]
```

- Note that the segments in V and C indicated, are defined by the header indices in R
- A gather using C to choose elements from the vector, will line up the elements of the vector with the correct elements of V that they have to be multiplied with
- The multiplication can be carried out with a map
- Now the segments of the results can be added with a segmented inclusive sum scan

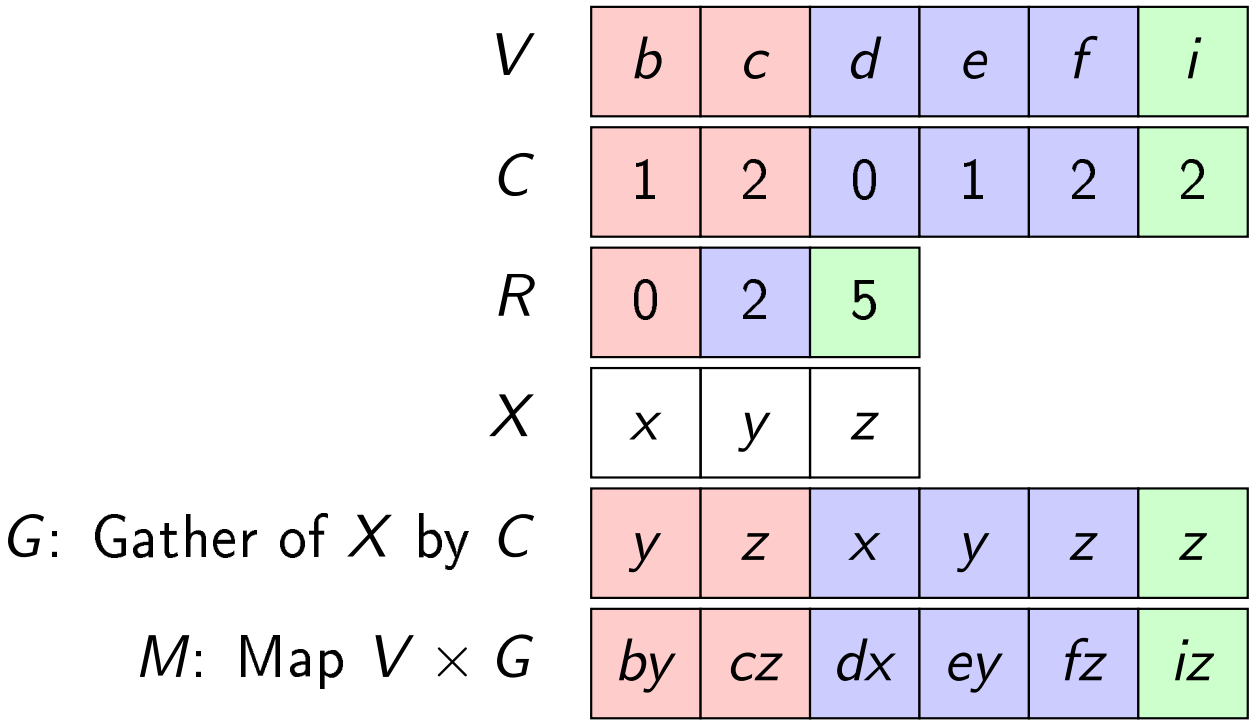
SpMv Execution



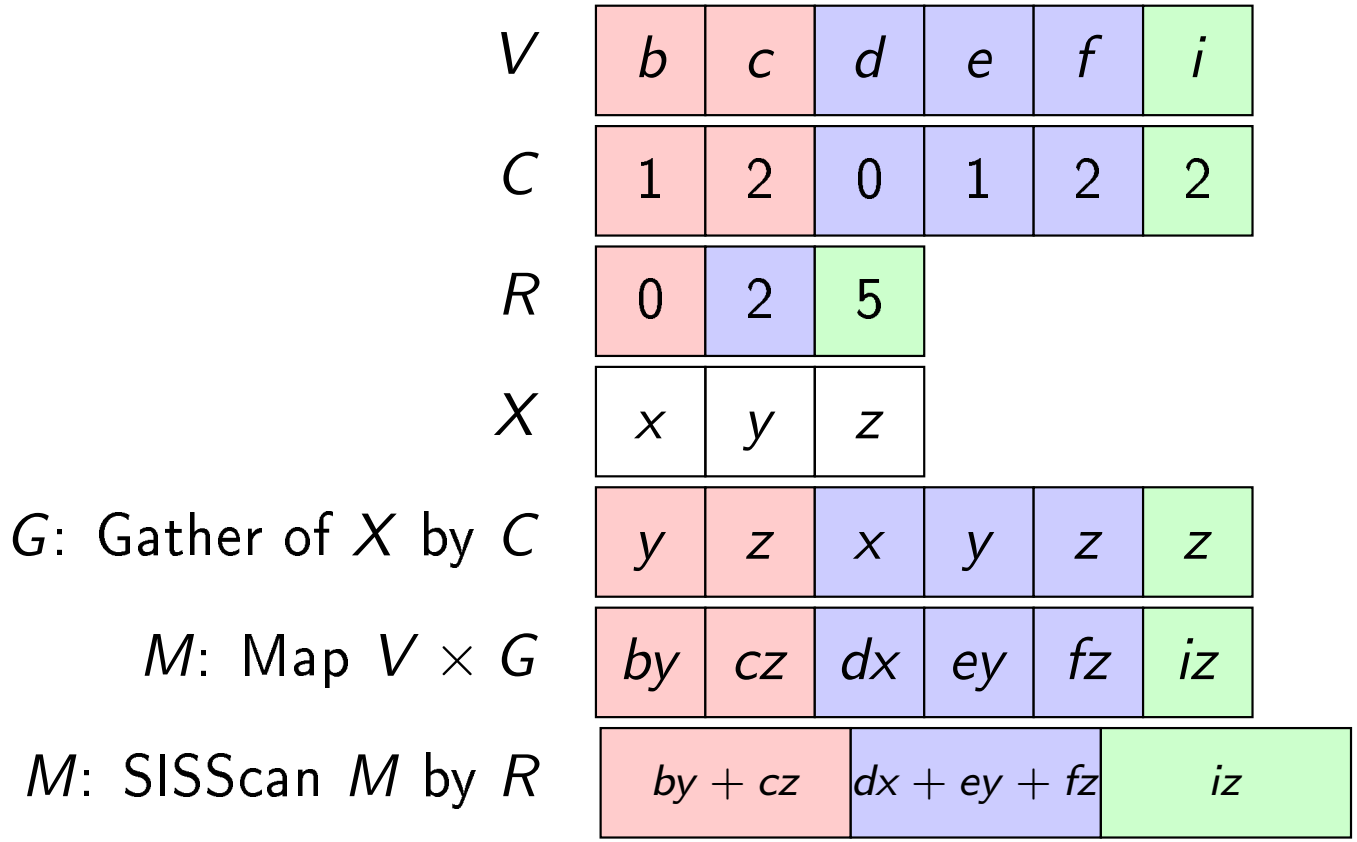
SpMv Execution



SpMv Execution

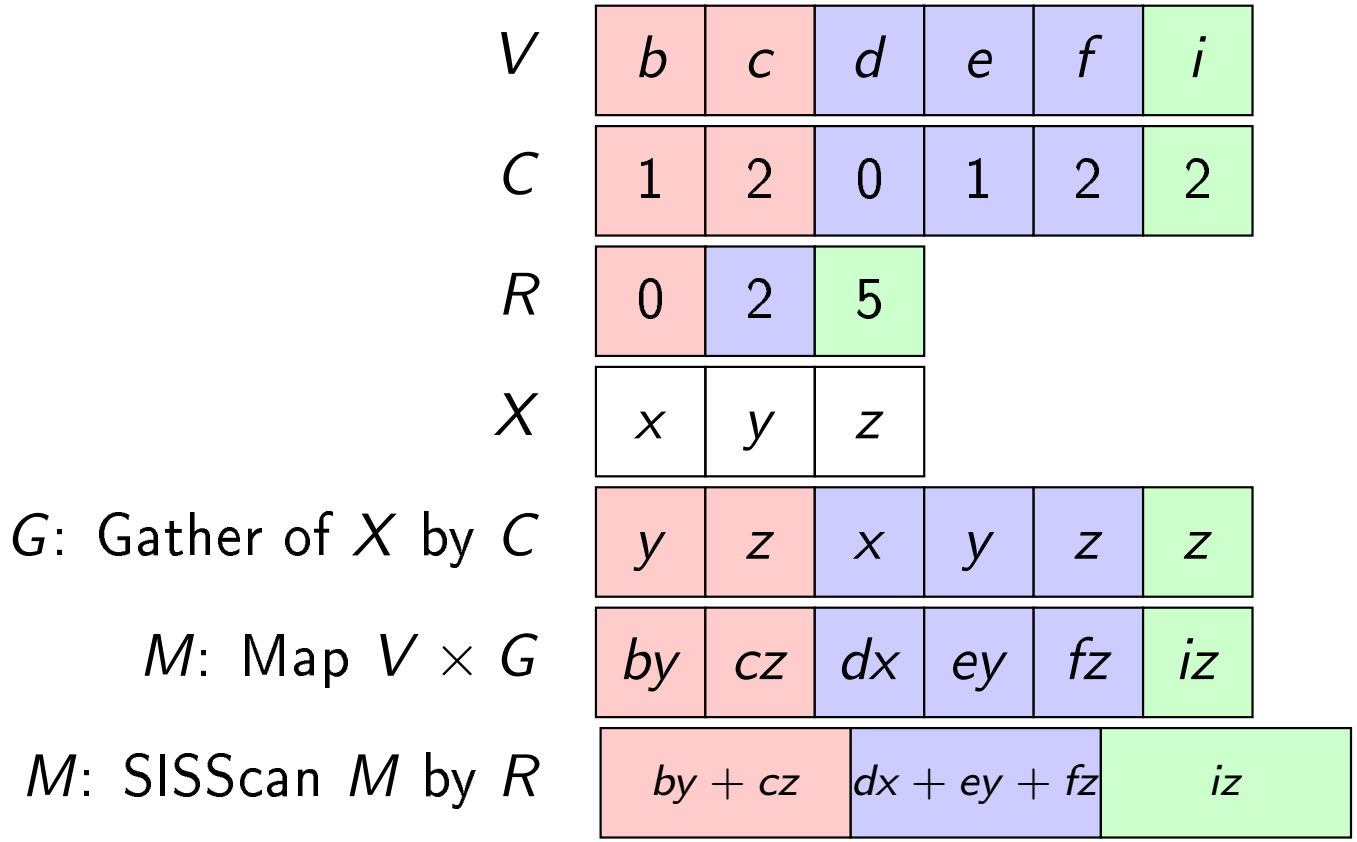


SpMv Execution



- Where SISScan is the Segmented Inclusive Sum Scan

SpMv Execution



- Where SISScan is the Segmented Inclusive Sum Scan
  - No zeros: lots of space saved
  - No zero multiplications: lots of processing saved

Distributed and Parallel Computing

Lecture 08

Alan P. Sexton

University of Birmingham

Spring 2018

Parallel Sorting

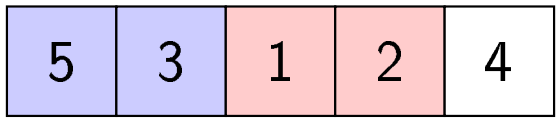
There are many serial algorithms that do not parallelize well. We want algorithms with:

- All many threads to work together on the problem
  - Serial algorithms are often inherently sequential
- Minimize branch divergence
  - Serial algorithms tend to do a lot of branching
- Coalesce memory access
  - Serial algorithms tend to access memory very randomly

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,...) with their neighbours to the right (1,3,5,...) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order



Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,...) with their neighbours to the right (1,3,5,...) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4
3	5	1	2	4

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,...) with their neighbours to the right (1,3,5,...) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4
3	5	1	2	4
3	1	5	2	4

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,...) with their neighbours to the right (1,3,5,...) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4
3	5	1	2	4
3	1	5	2	4
1	3	2	5	4

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,...) with their neighbours to the right (1,3,5,...) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4
3	5	1	2	4
3	1	5	2	4
1	3	2	5	4
1	2	3	4	5

Odd-Even Sort

The algorithm proceeds in a sequence of steps:

- In every even step, compare the elements in the even locations (0,2,4,...) with their neighbours to the right (1,3,5,...) and swap if out of order
- In every odd step, compare the elements in the odd locations with their neighbours to the right and swap if out of order

5	3	1	2	4
3	5	1	2	4
3	1	5	2	4
1	3	2	5	4
1	2	3	4	5

- n inputs, steps: $O(n)$, work: $O(n^2)$

Parallel Merge Sort

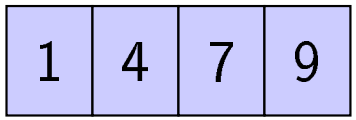
In the simplest form, Parallel Merge Sort works as follows:

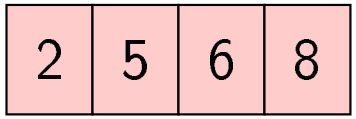
- Start with a set of (trivially sorted) sequences of length 1
  - i.e. single elements
- In each step, merge independent pairs of sequences from the set of sorted sequences together to make a set of half the number of longer sorted sequences
- Finish when the last pair of sequences is merged into one final sorted sequence

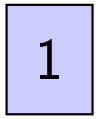
Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



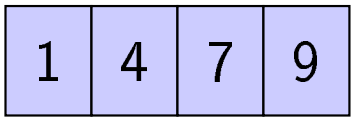


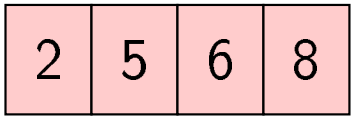


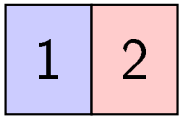
Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



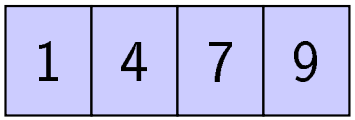


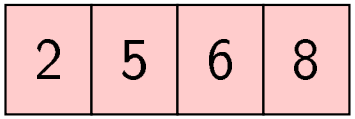


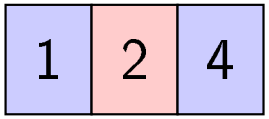
Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



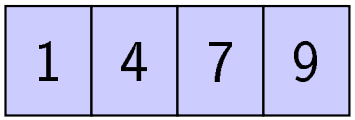


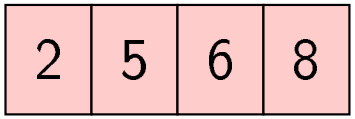


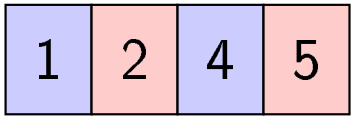
Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



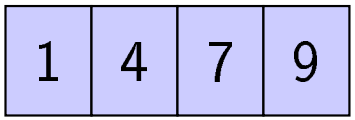


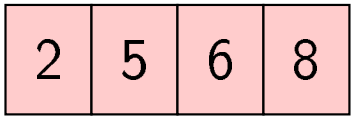


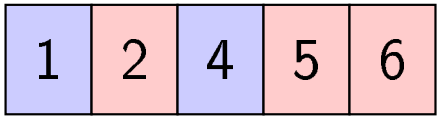
Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



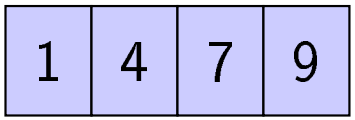


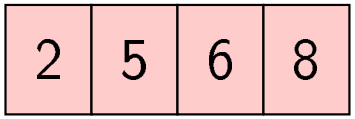


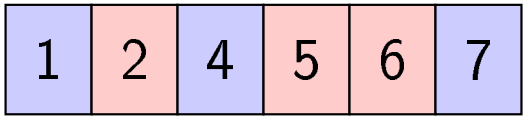
Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



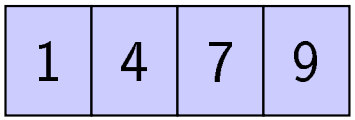


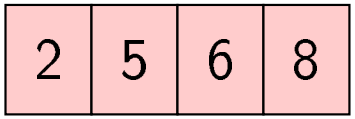


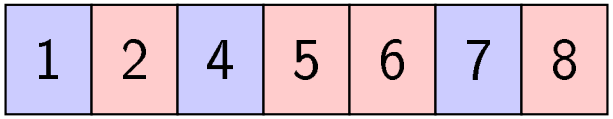
Merging 2 sequences

Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



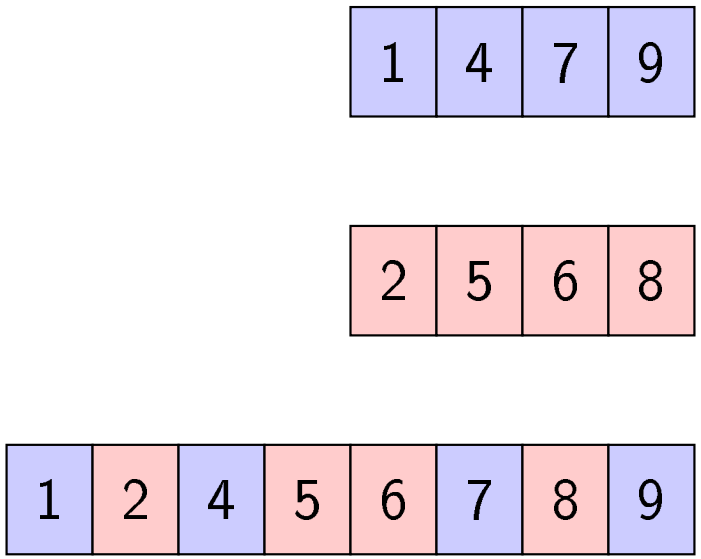




Merging 2 sequences

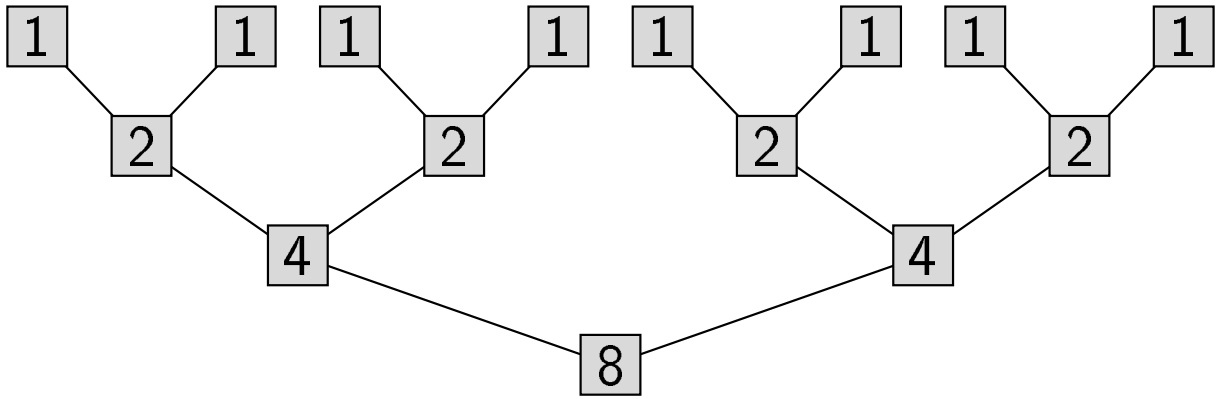
Sequentially merging 2 sequences:

while neither sequence is empty
    Compare the elements at the head of the 2 sequences
    Pop smaller and append to the output sequence
append the elements of the non-empty sequence to the output



Parallel Merge Sort

Progressive sequence sizes shown:



- n inputs
- steps: $O(\log n)$
- work: $O(n \log n)$
  - In each step we are generating n elements.
  - Each element generated (except the last in each merge) is the result of one comparison
  - $n(1 - \frac{1}{2}) + n(1 - \frac{1}{4}) + n(1 - \frac{1}{8}) + \ldots$ with $\log n$ terms
  - $= n \log n - \log n$
  - $= O(n \log n)$

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
   - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
   - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge
     - 1 block of threads: 1 merge

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
   - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge
     - 1 block of threads: 1 merge
3. Very few large sequences — each much larger than block size

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
   - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge
     - 1 block of threads: 1 merge
3. Very few large sequences — each much larger than block size
   - If each merge is done by one block of threads, then not enough merges to occupy all SMs

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
   - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge
     - 1 block of threads: 1 merge
3. Very few large sequences — each much larger than block size
   - If each merge is done by one block of threads, then not enough merges to occupy all SMs
   - Use multiple blocks to handle each merge

Merge Sort on NVidia GPUs

When implementing Merge Sort on NVidia GPUs, in order to make good use of the hardware resources, we consider 3 stages:

1. Many small sequences — each less than block size
   - Here there are many small merges to do: each thread can do one merge, each block handles many merges
     - 1 thread: 1 merge
   - Memory Coalescing: better to copy block to shared memory and use a different sort within the block to make medium sequences (c.f. later)
2. Medium number of medium sequences
   - 1 thread to 1 merge $\Rightarrow$ not enough merges to utilize all SMs
   - If as many merges as the number of blocks that can fit on the GPU: use each block of threads to do one merge
     - 1 block of threads: 1 merge
3. Very few large sequences — each much larger than block size
   - If each merge is done by one block of threads, then not enough merges to occupy all SMs
   - Use multiple blocks to handle each merge
     - Multiple blocks of threads: 1 merge

Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

- Assign one thread to each element

Merge: 1 Block of Threads to 1 Merge

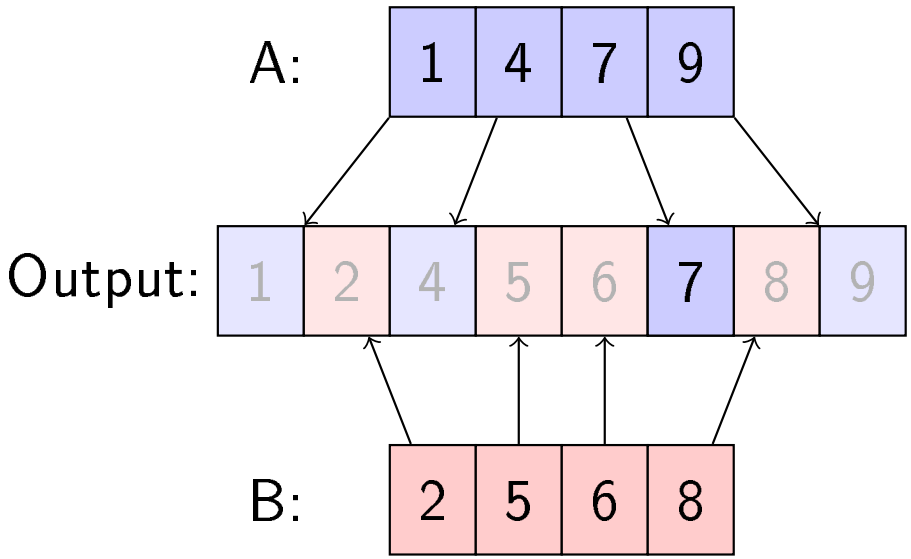
Trick: identify scatter addresses:

- Assign one thread to each element
- Thread calculates scatter address for its element and copies element there

Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

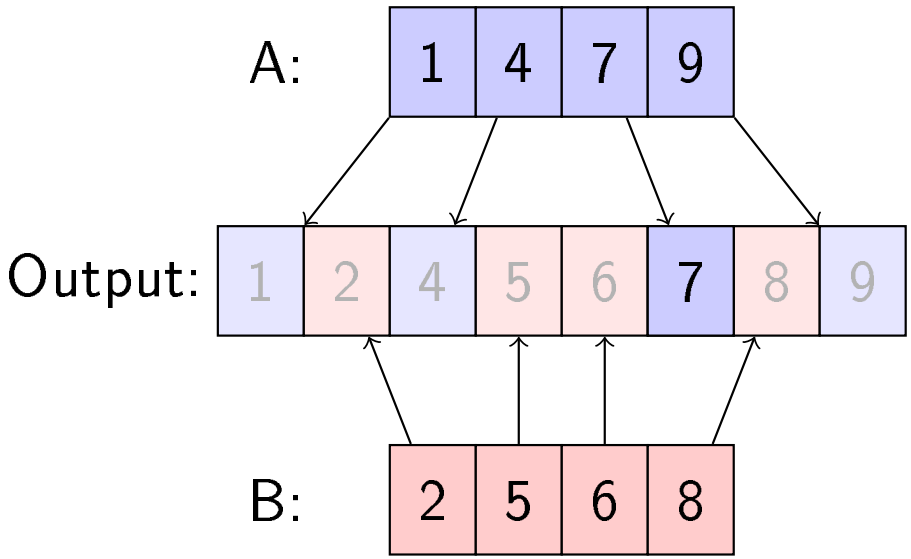
- Assign one thread to each element
- Thread calculates scatter address for its element and copies element there
- Consider $A[2]$ which contains 7:



Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

- Assign one thread to each element
- Thread calculates scatter address for its element and copies element there
- Consider $A[2]$ which contains 7:

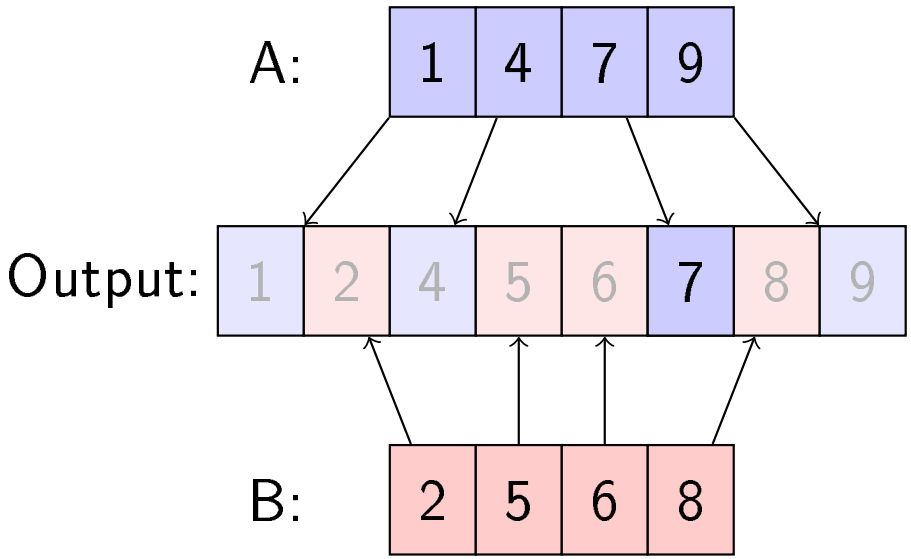


- Thread for $A[2]$ knows location in A is 2

Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

- Assign one thread to each element
- Thread calculates scatter address for its element and copies element there
- Consider $A[2]$ which contains 7:

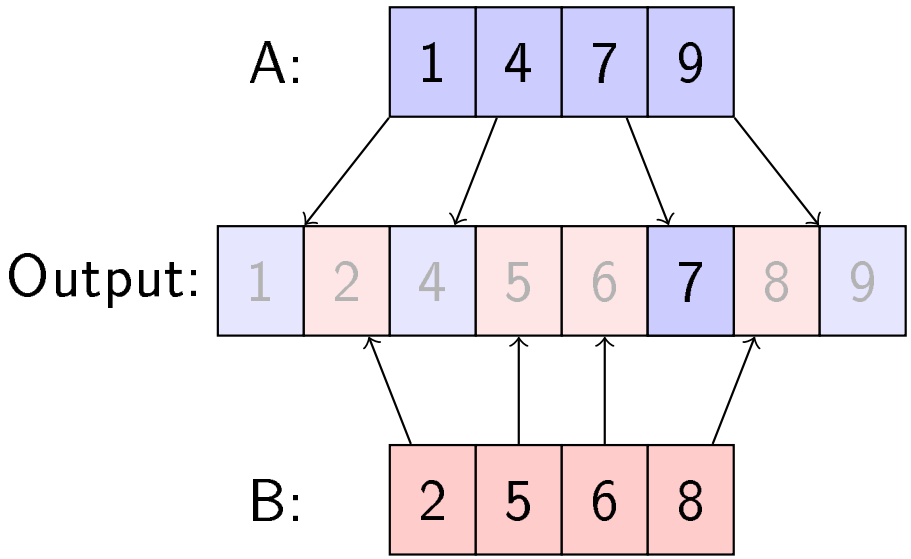


- Thread for $A[2]$ knows location in A is 2
- Thread does binary search to find insertion location in B is 3

Merge: 1 Block of Threads to 1 Merge

Trick: identify scatter addresses:

- Assign one thread to each element
- Thread calculates scatter address for its element and copies element there
- Consider $A[2]$ which contains 7:



- Thread for $A[2]$ knows location in A is 2
- Thread does binary search to find insertion location in B is 3
- Hence location in output is $2 + 3 = 5$

Merge: 1 Block of Threads to 1 Merge

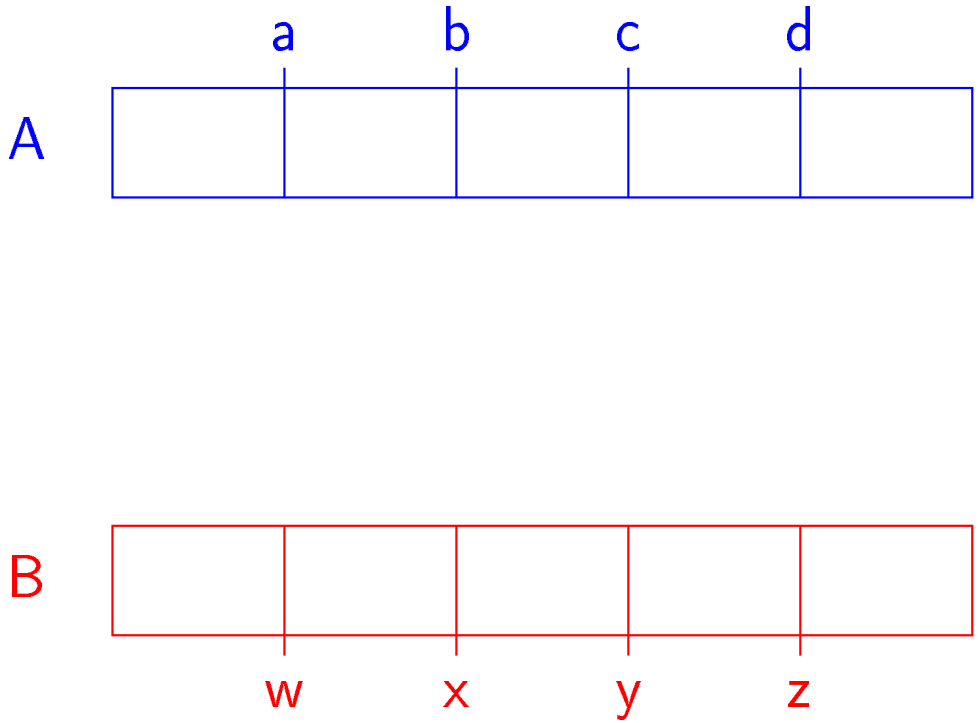
- Merge in a sequence of kernels
- Blocks per Grid is the number of merges to execute
- Threads per block is the number of elements that the merge will produce
- Copy sequences from global to shared memory, merge and copy back
- Thus (on GTX960s) suitable for merges that produce sequences of length 64 to 1024
  - GTX960 allows up to 32 blocks per SM, but can manage 2048 threads per SM. So less than 64 threads per block and the SM will not be fully occupied
- Can handle merges larger than 1024 elements:
  - Read chunks of sequences from global to shared memory, merge chunks and copy back
  - Slightly tricky to handle the *streams* of chunks

Merge: Multiple Blocks of Threads to 1 Merge

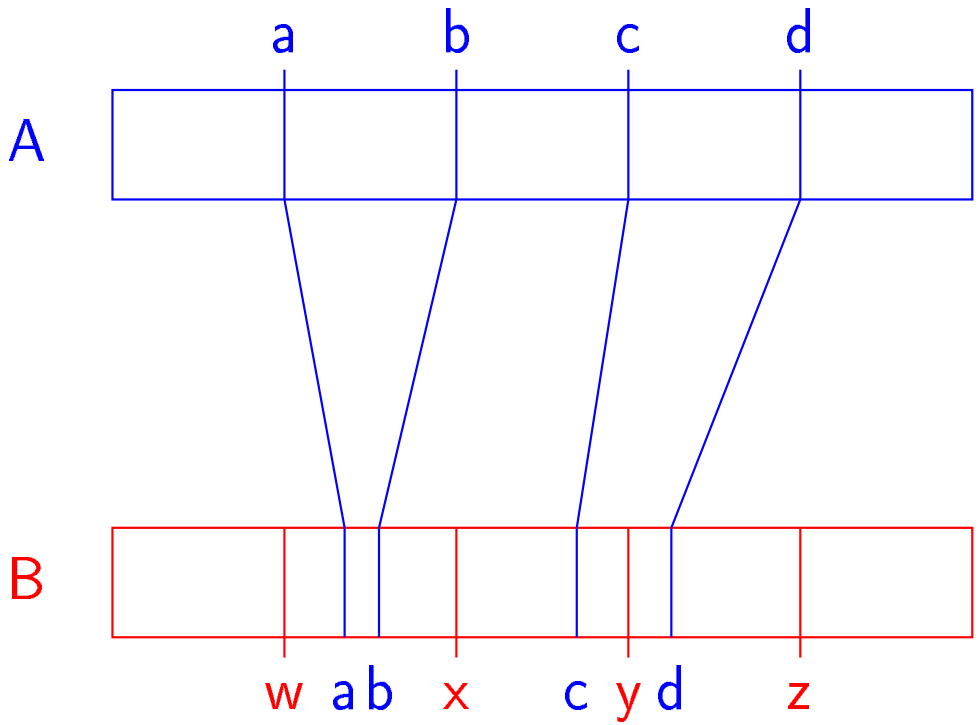
Problem is to break up a large merge so that different blocks can work on different parts of the merge independently

- Choose *splitters*, max K elements apart, from both sequences
- Merge the splitters into a single sorted list, remembering their locations in their home sequences
  - our previous medium merge method can do this
- Find the insertion location of each splitter in its *foreign* sequence (binary search)
  - Each splitter now has locations for both sequences
- Each consecutive pair of splitters thus defines a section of both sequences that can be merged independently of any other sections
- None of these sections can merge into more than $2K$ elements
- Choose K to be maximum 512 and each merge section can be handled by 1 block of 1024 threads

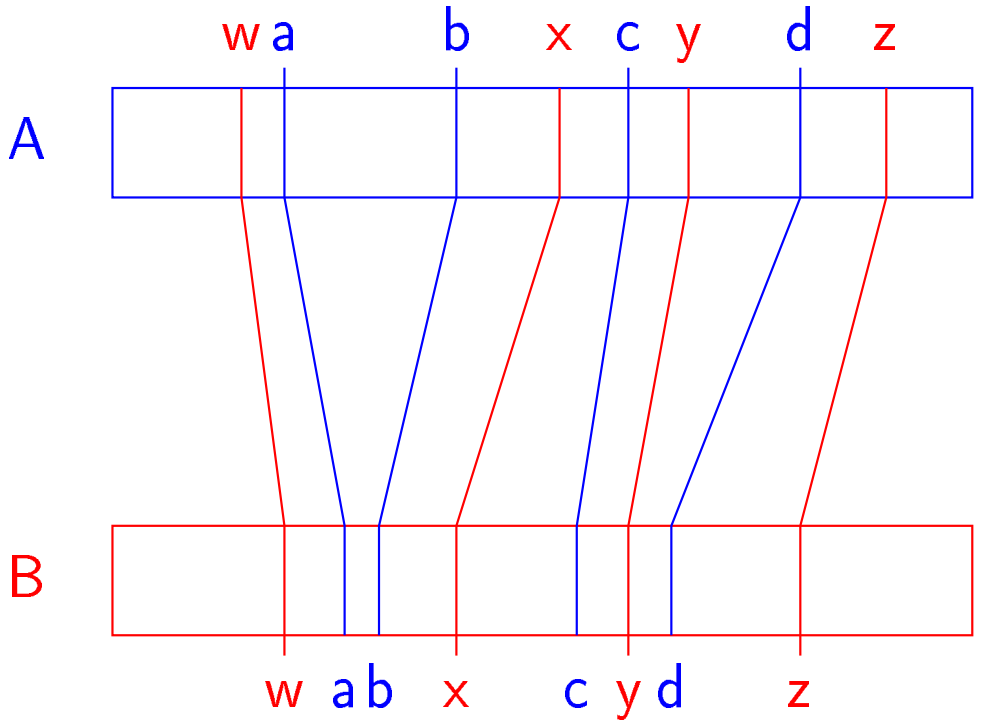
Merge: Multiple Blocks of Threads to 1 Merge



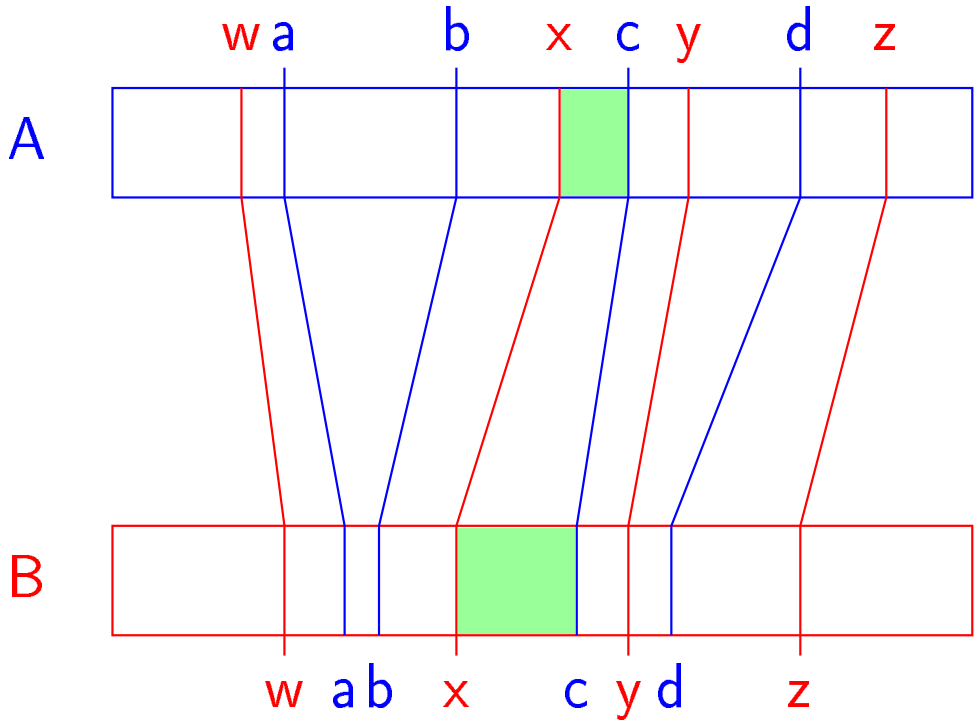
Merge: Multiple Blocks of Threads to 1 Merge



Merge: Multiple Blocks of Threads to 1 Merge



Merge: Multiple Blocks of Threads to 1 Merge

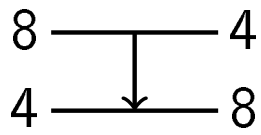


- $|[b, c]|$ in A is $K \Rightarrow |[x, c]| \leq K$ in A
- Similarly $|[x, c]| \leq K$ in B
- Hence the merge of the $[x, c]$ segments is no more than $2K$
- Similarly for all other segment pairs

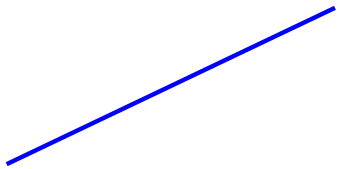
Bitonic Sort

Some definitions:

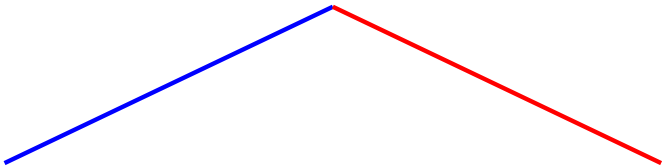
- A comparator is a function that swaps two elements if they are in the wrong order



- A monotonic increasing/decreasing sequence is one where every element is equal to or greater/less than every preceding element in the sequence
  - 1, 4, 8, 16, 16, 18, 19, 22



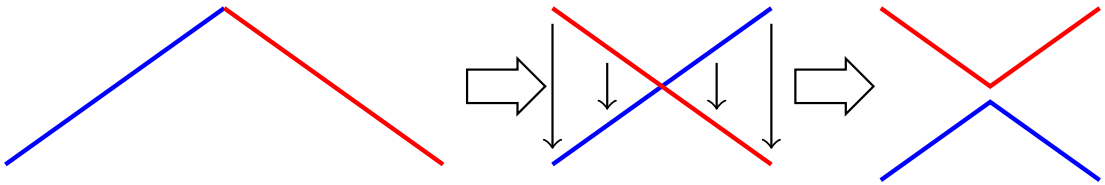
- A bitonic sequence is a sequence which changes order direction at most once, or a circular shift of such a sequence
  - 15, 12, 5, 2, 1, 4, 8, 16
  - 1, 4, 8, 16, 15, 12, 5, 2



Bitonic Split

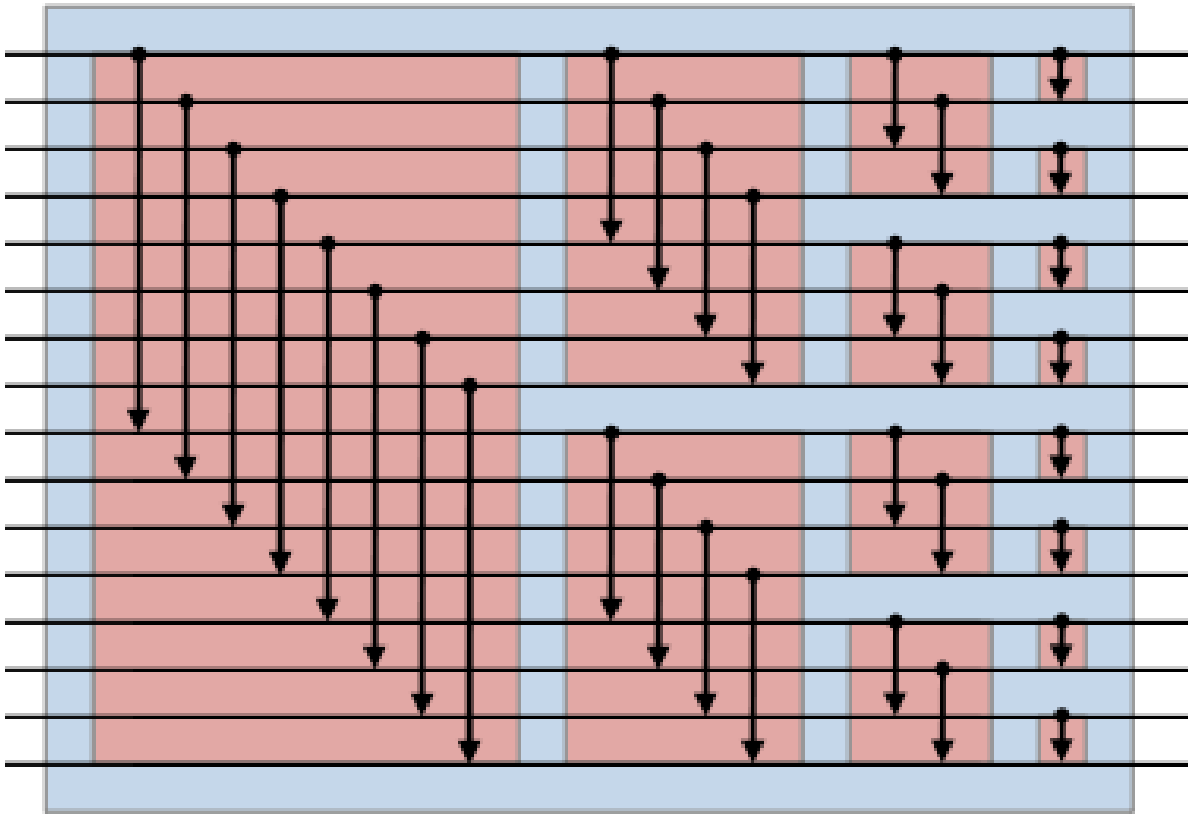
The central idea in Bitonic sort is that:

- A simple parallel arrangement of comparators can split a bitonic sequence into two bitonic sequences, where all elements of the first are less than all elements of the second:



$$1 \quad 4 \quad 8 \quad 16 \quad 15 \quad 12 \quad 5 \quad 2 \Rightarrow \begin{matrix} 15 & 12 & 5 & 2 \\ 1 & 4 & 8 & 16 \end{matrix} \Rightarrow \begin{matrix} 1 & 4 & 5 & 2 \\ 15 & 12 & 8 & 16 \end{matrix}$$

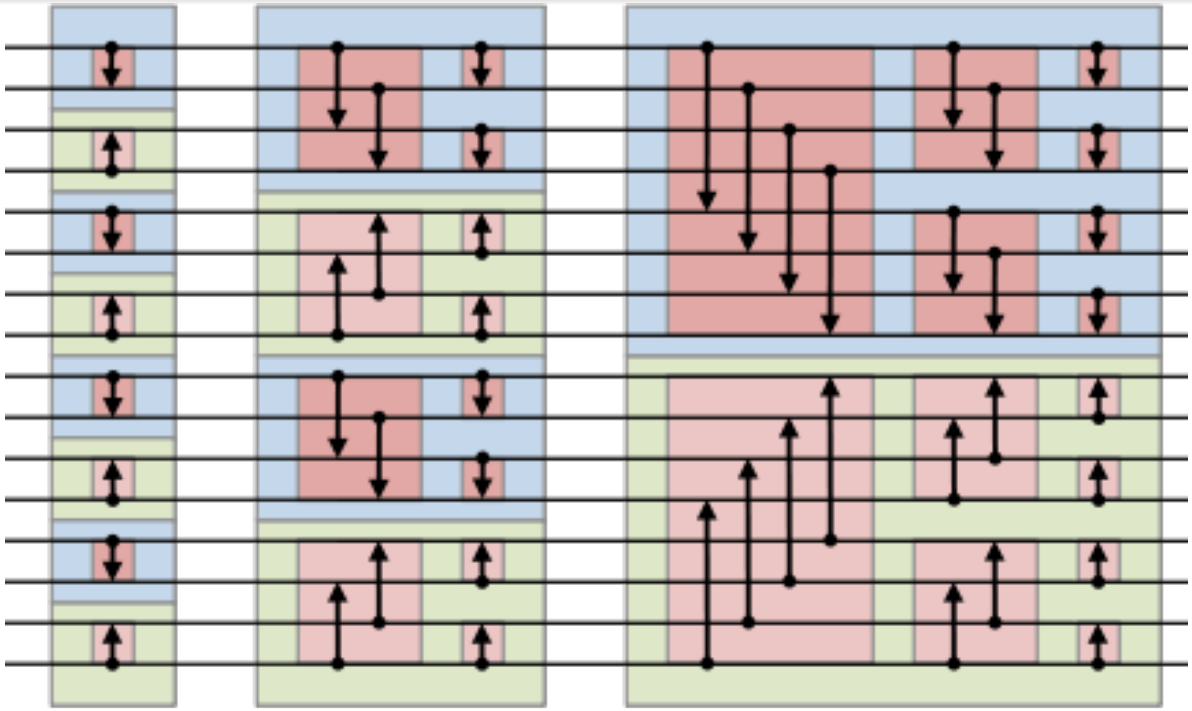
Bitonic Sort: Second Phase



- If the inputs along the left are a bitonic sequence:
  - First red block splits it into two bitonic sequences, where all upper half elements are less than all lower half ones
  - The next two red blocks splits these 2 into 4 similarly, etc.
  - Final output is sorted

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

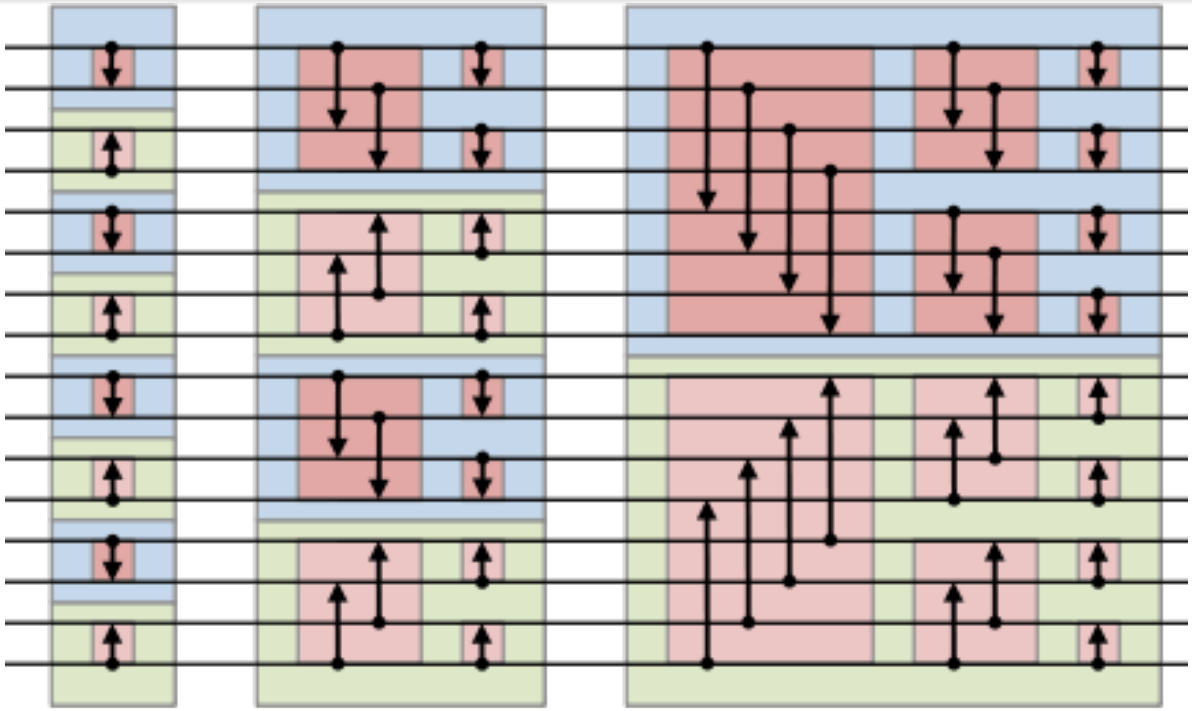
Bitonic Sort: First Phase



- Need to turn a random sequence into a bitonic sequence

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

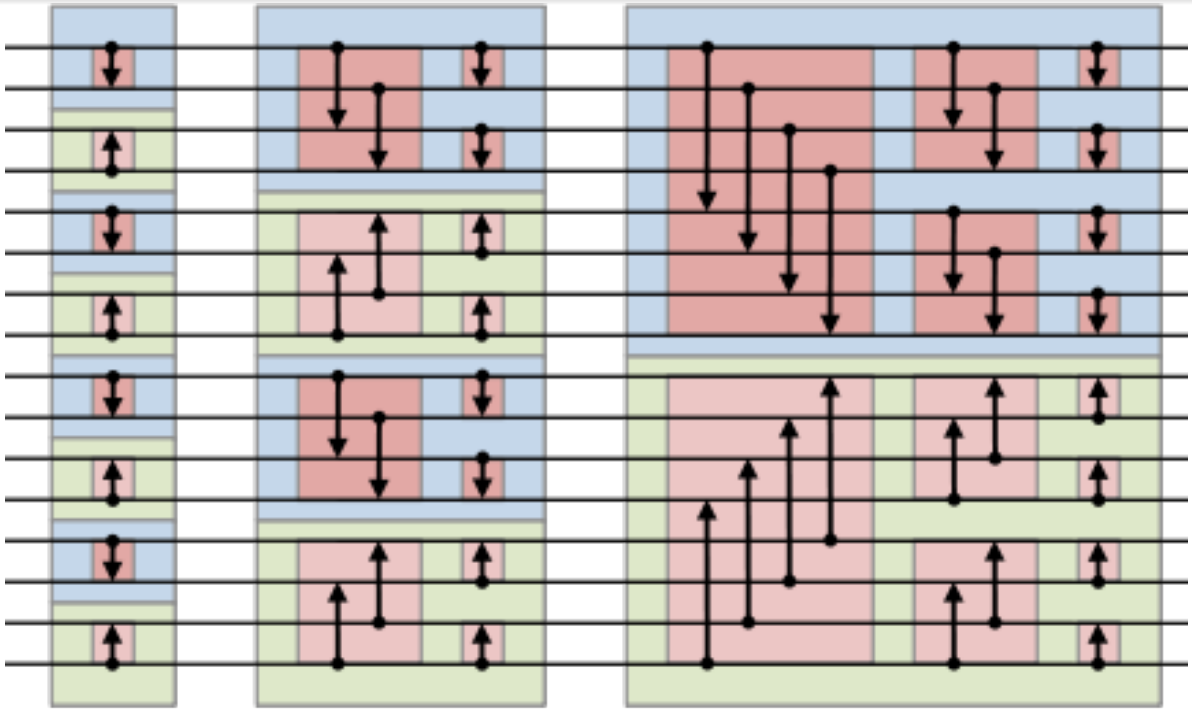
Bitonic Sort: First Phase



- Need to turn a random sequence into a bitonic sequence
  - Top right box sorts bitonic sequence into ascending order, bottom right into descending

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

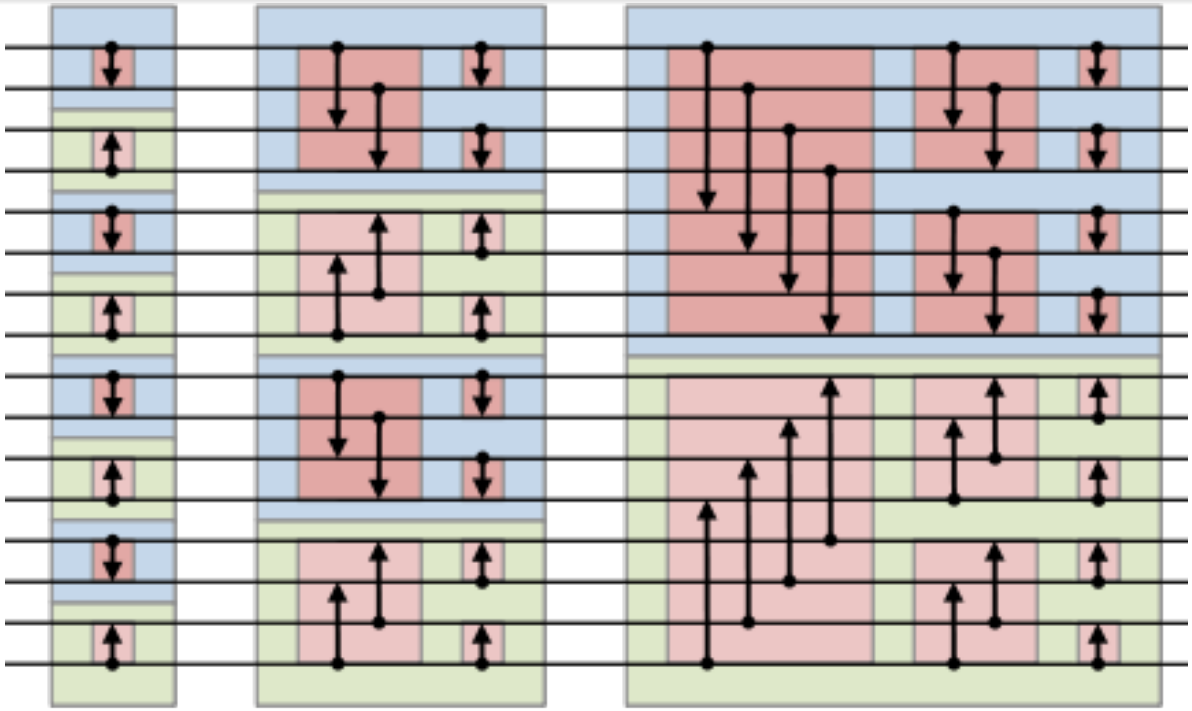
Bitonic Sort: First Phase



- Need to turn a random sequence into a bitonic sequence
  - Top right box sorts bitonic sequence into ascending order, bottom right into descending
  - Right column: 2 bitonic sequences of len 8 $\rightarrow$ 1 of len 16

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

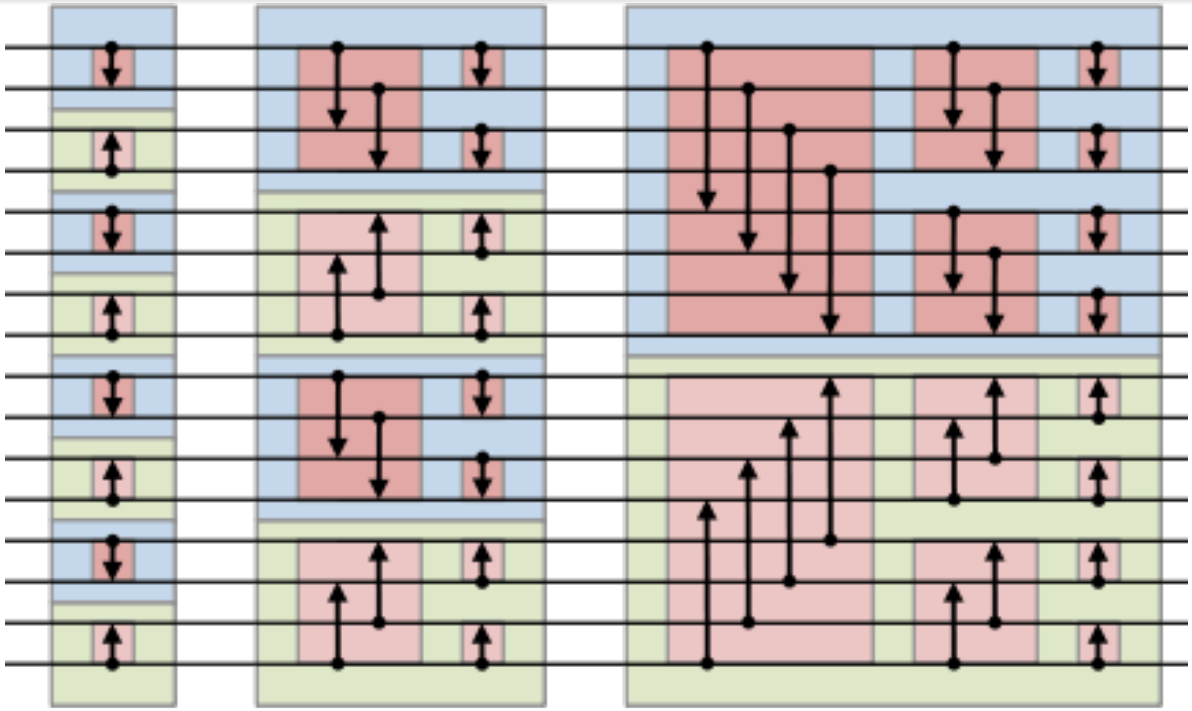
Bitonic Sort: First Phase



- Need to turn a random sequence into a bitonic sequence
  - Top right box sorts bitonic sequence into ascending order, bottom right into descending
  - Right column: 2 bitonic sequences of len 8 $\rightarrow$ 1 of len 16
  - Middle column turns 4 of len 4 $\rightarrow$ 2 of len 8

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

Bitonic Sort: First Phase

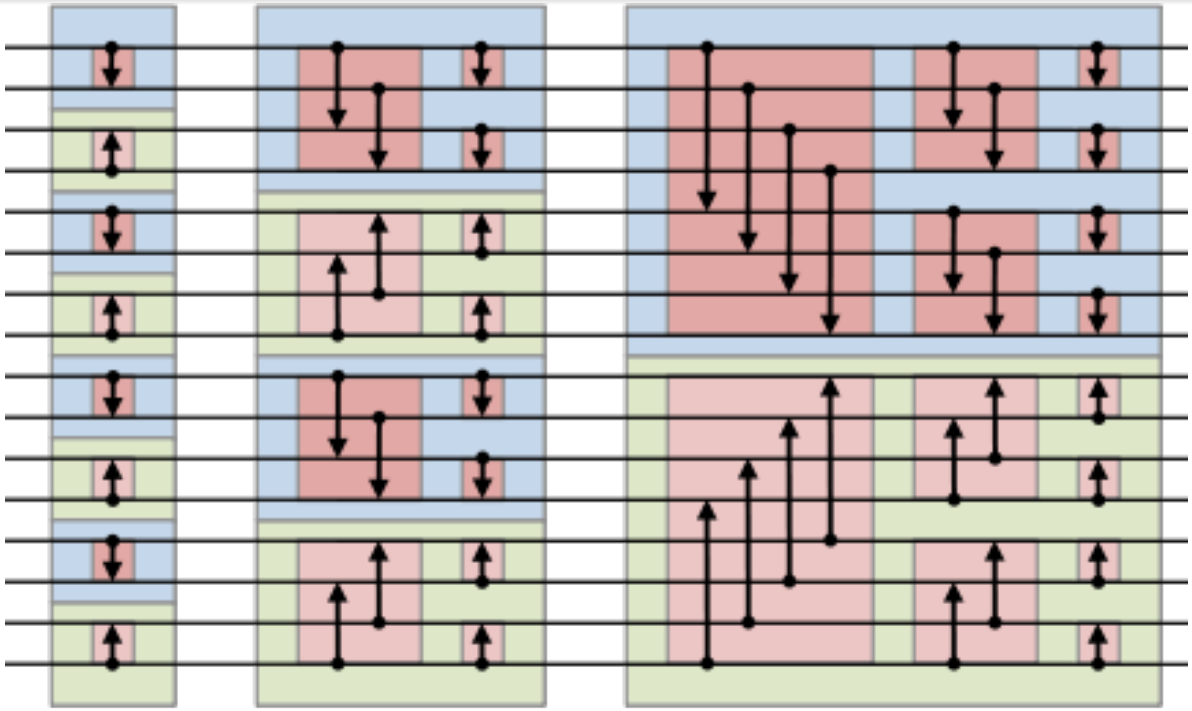


- Need to turn a random sequence into a bitonic sequence
  - Top right box sorts bitonic sequence into ascending order, bottom right into descending
  - Right column: 2 bitonic sequences of len 8 $\rightarrow$ 1 of len 16
  - Middle column turns 4 of len 4 $\rightarrow$ 2 of len 8
  - Left column turns 8 of len 2 $\rightarrow$ 4 of len 4

---

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

Bitonic Sort: First Phase

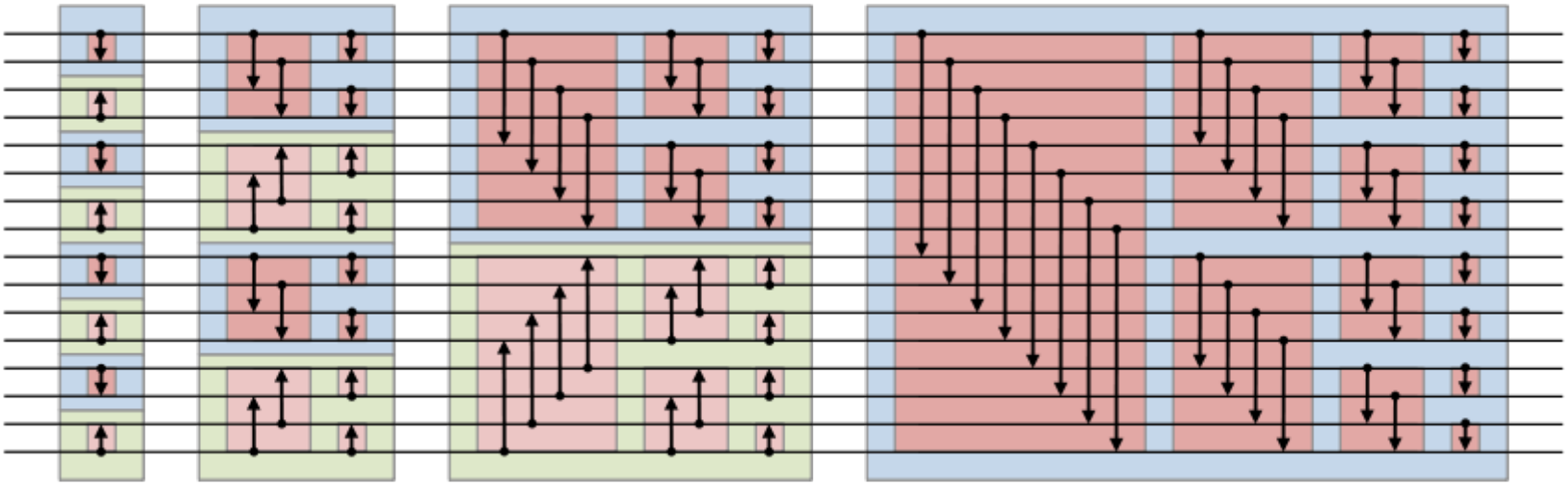


- Need to turn a random sequence into a bitonic sequence
  - Top right box sorts bitonic sequence into ascending order, bottom right into descending
  - Right column: 2 bitonic sequences of len 8 $\rightarrow$ 1 of len 16
  - Middle column turns 4 of len 4 $\rightarrow$ 2 of len 8
  - Left column turns 8 of len 2 $\rightarrow$ 4 of len 4
  - But all sequences of length 2 are trivially bitonic!

---

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

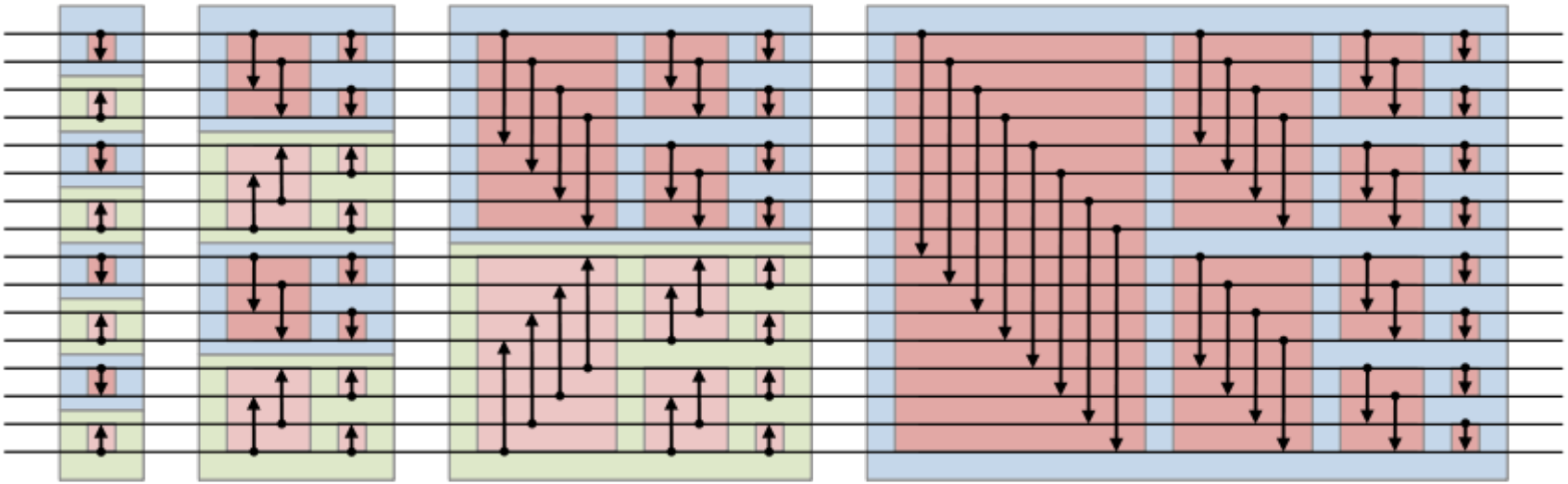
Bitonic Sort



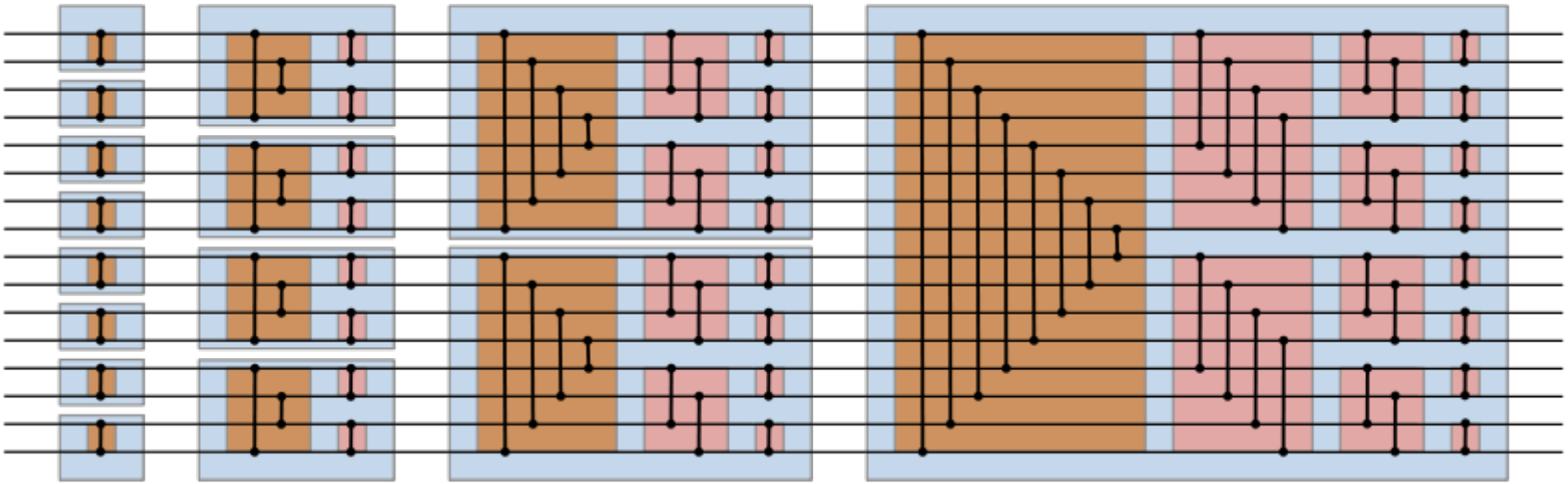
- Each column of red blocks runs in parallel with no races
- Assign each thread to one data element (Some implementations: 1 thread to one comparison)
- Each comparison executed twice:
  - At lower end, thread stores the smaller of the two values
  - At Upper end, thread stores the larger of the two values
- Complexity: $O(n \log^2 n)$ steps: but fastest sort for small sets
- Excellent for first stage of merge sort

image from `https://en.wikipedia.org/wiki/Bitonic_sorter`

Bitonic Sort



Can be rearranged with all arrows down:



images from `https://en.wikipedia.org/wiki/Bitonic_sorter`

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix} 0 \\ 5 \\ 2 \\ 7 \\ 1 \\ 3 \\ 6 \\ 4 \end{bmatrix} = \begin{bmatrix} 000 \\ 101 \\ 010 \\ 111 \\ 001 \\ 011 \\ 110 \\ 100 \end{bmatrix}$$

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \begin{bmatrix}000\\010\\110\\100\\101\\111\\001\\011\end{bmatrix}$$

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \begin{bmatrix}000\\010\\110\\100\\101\\111\\001\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\100\\101\\001\\010\\110\\111\\011\end{bmatrix}$$

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \begin{bmatrix}000\\010\\110\\100\\101\\111\\001\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\100\\101\\001\\010\\110\\111\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\001\\010\\011\\100\\101\\110\\111\end{bmatrix} = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \begin{bmatrix}000\\010\\110\\100\\101\\111\\001\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\100\\101\\001\\010\\110\\111\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\001\\010\\011\\100\\101\\110\\111\end{bmatrix} = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

- Complexity is $O(kn)$, where k is the number of bits, n the number of input elements

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \begin{bmatrix}000\\010\\110\\100\\101\\111\\001\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\100\\101\\001\\010\\110\\111\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\001\\010\\011\\100\\101\\110\\111\end{bmatrix} = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

- Complexity is $O(kn)$, where k is the number of bits, n the number of input elements
- Reduce k: do 2^m splits by splitting on m bits at a time, e.g. 4

Radix Sort

Radix sort works by doing a series of **stable** splits based on ascending significance bits of the input values.

- A stable split preserves the relative orginal order of the elements in each part of the split

$$\begin{bmatrix}0\\5\\2\\7\\1\\3\\6\\4\end{bmatrix} = \begin{bmatrix}000\\101\\010\\111\\001\\011\\110\\100\end{bmatrix} \rightarrow \begin{bmatrix}000\\010\\110\\100\\101\\001\\011\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\100\\101\\001\\010\\110\\111\\011\end{bmatrix} \rightarrow \begin{bmatrix}000\\001\\010\\011\\100\\101\\110\\111\end{bmatrix} = \begin{bmatrix}0\\1\\2\\3\\4\\5\\6\\7\end{bmatrix}$$

- Complexity is $O(kn)$, where k is the number of bits, n the number of input elements
- Reduce k: do 2^m splits by splitting on m bits at a time, e.g. 4
- Fastest CUDA GPU sort for medium to large inputs

Radix Sort Implementation

- Each split section can be generated with a compact operation:
  - Map on LSB = 0, followed by an exclusive sum scan to calculate the first section scatter addresses
  - Use the last scatter address calculated as an offset to the scatter addresses for the second section
  - If using multi-bit radix steps, run a histogram to calculate the number in each section and hence the offsets

Distributed and Parallel Computing

Lecture 09

Alan P. Sexton

University of Birmingham

Spring 2017

Floating Point Number Representation

Single Precision IEEE-754 floating point numbers has:

- 1 sign bit (S)
- an 8 bit biased exponent field with a bias of 127 (E)
  - 0 and 255 reserved for special numbers
  - E=127 corresponds to an exponent of $E - 127 = 0$
- a 23 bit mantissa field (M)
- for normal numbers, an implicit (hidden) initial 1 bit in the mantissa is assumed (24 binary bits $\approx$ 7 decimal digits)
- a number of special bit patterns in S, E and M for positive and negative zeros, positive and negative infinities, NaN and sub-normal numbers

For normal numbers the interpretation is:

$$(-1)^S \left(1 + 2^{-23} M\right) 2^{E-127}$$

Subnormal Numbers

Normal numbers use the implicit 1 bit in the mantissa. What if only normal numbers are represented?

- The smallest representable number greater than 0 would be

$$\left(1 + 2^{-23} \times 0\right) 2^{-127} = 2^{-127} \approx 6 \times 10^{-39}$$

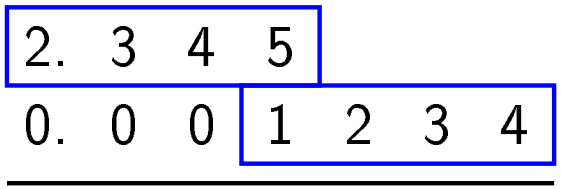
- The next smallest would be

$$\left(1 + 2^{-23} \times 1\right) 2^{-127} = 2^{-127} + 2^{-150}$$

- Thus much larger distance from 0 to succ(0) than from succ(0) to succ(succ(0))
- Sub-normal numbers: if exponent is -127 (i.e. E=0), don't use implicit 1 in mantissa

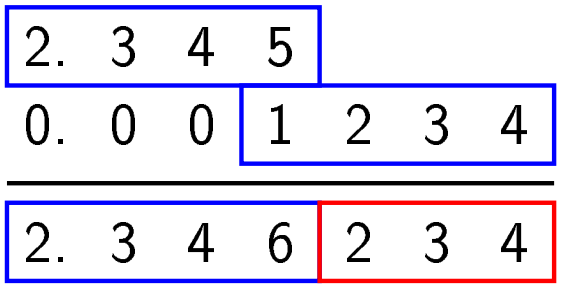
Rounding

- Unit in Last Place ($ULP(x)$): distance between the two floating point numbers that are the closest pair (a, b) that straddle x: $a \leq x \leq b$
- IEEE-754 requires that operations round to nearest representable floating point number: that the computed result be within 0.5 ULPs of the mathematically correct result.
- Consider $a + b$, where $a \gg b$, e.g. in a 4 digit decimal notation:



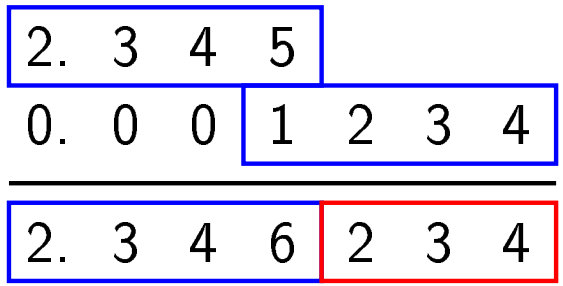
Rounding

- Unit in Last Place ($ULP(x)$): distance between the two floating point numbers that are the closest pair (a, b) that straddle x: $a \leq x \leq b$
- IEEE-754 requires that operations round to nearest representable floating point number: that the computed result be within 0.5 ULPs of the mathematically correct result.
- Consider $a + b$, where $a \gg b$, e.g. in a 4 digit decimal notation:



Rounding

- Unit in Last Place ($ULP(x)$): distance between the two floating point numbers that are the closest pair (a, b) that straddle x: $a \leq x \leq b$
- IEEE-754 requires that operations round to nearest representable floating point number: that the computed result be within 0.5 ULPs of the mathematically correct result.
- Consider $a + b$, where $a \gg b$, e.g. in a 4 digit decimal notation:

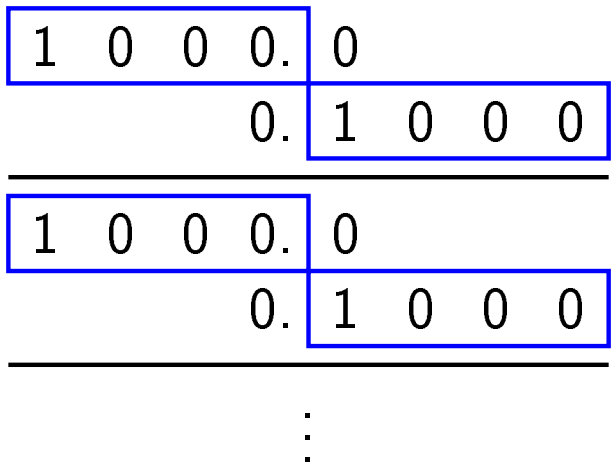


- Thus, in this system, $(2.345 + 0.001234) - 2.345 = 0.001$

Sequence of Additions

Again in 4 digit decimal notation, consider summing a vector containing 1000.0 in the first element and then 10,000 elements of value 0.1.

- Mathematically, result should be 2000.0
- Incrementally adding from first returns 1000.0



Worst case error in summing N numbers: $O(N)$

Fixing Sequence of Additions

To fix this, we can try adding in increasing order:

- Sort vector first
- Now incremental additions should add the smaller values together first then combine accumulated larger values with larger values from later in the vector

Problem:

- Accumulated small values may get significantly larger than later values in the vector
- Thus may help in some cases, but not a full solution

Kahan Sumation

Idea: Accumulate the sum, but calculate a correction term and add it to the next number at each step:

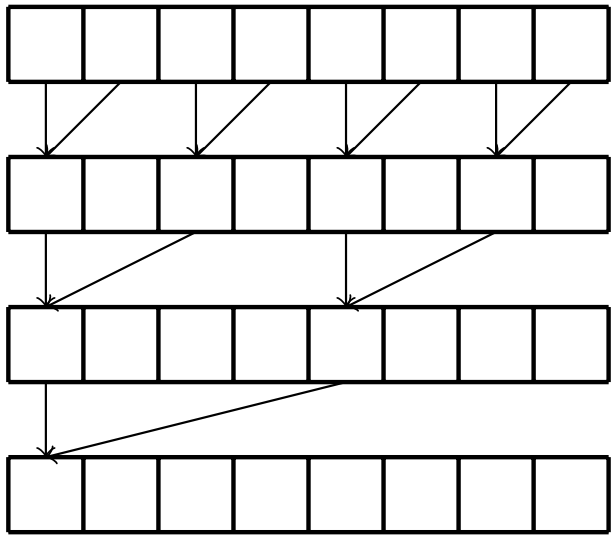
```
float fk_add(float * flt_arr)
{
  long i;
  float sum, correction, corrected_next_term, new_sum;

  sum = flt_arr[0];
  correction = 0.0;
  for (i = 1; i < ARR_SIZE; i++)
  {
    corrected_next_term = flt_arr[i] - correction;
    new_sum = sum + corrected_next_term;
    correction = (new_sum - sum) - corrected_next_term;
    sum = new_sum;
  }
  return sum;
}
```

- Worst case error in summing N numbers: $O(1)$, i.e. dependent only on the precision of the number representation
- But, not easy to parallelise

Fixing Sequence of Additions

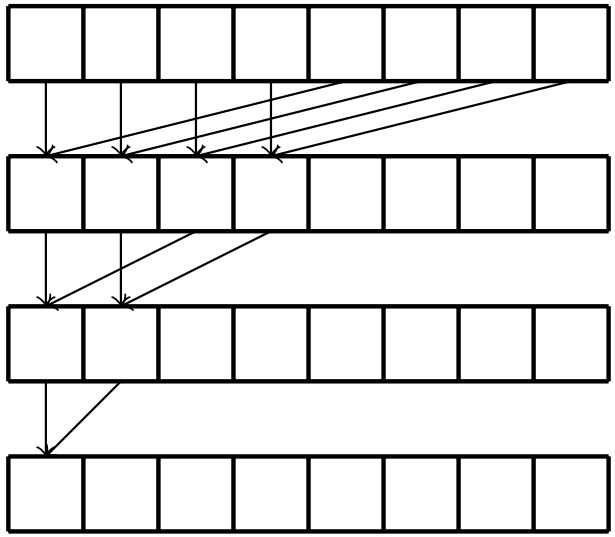
Try sorting, then reducing



- adds similarly sized pairs in early steps
- but later steps add increasingly different sized values
- Worst case error in summing N numbers: O(log(N))
- But: poor thread blocking, memory access patterns

Fixing Sequence of Additions

Can use more efficient compressed thread reduction pattern:



- but this adds values widely separated in vector
- Careful ordering helps
- Possible to use reduction for first few steps and switch to (unrolled) Kahan Summation to finish off

Absolute and Relative Errors

Given a value v, and its computed approximation $\hat{v}$:

- The *absolute error* in $\hat{v}$ is $|v - \hat{v}|$
- The *relative error*, where $v \neq 0$, in $\hat{v}$ is $\frac{|v-\hat{v}|}{|v|}$

The relative error is usually the more useful quantity.

- $|a| \gg |b| \Rightarrow a + b$ has a large absolute error
- $|b| \ll 1 \Rightarrow \frac{a}{b}$ has large relative and absolute errors
- $a \approx b \Rightarrow a - b$ has a large relative error (cancellation errors)

Example

We can compute $e = 2.7182818\ldots$, the base of natural logarithms, with the formula:

$$e = \lim_{n \to \infty} \left(1 + \frac{1}{n}\right)^n$$

n	val
10^1	2.593742
10^2	2.704814
10^3	2.716924
10^4	2.718146
10^5	2.718268
10^6	2.718281
10^7	2.718282
10^8	2.718282
10^9	2.718282
10^{10}	2.718282
10^{11}	2.718282
10^{12}	2.718524
10^{13}	2.716110
10^{14}	2.716110
10^{15}	3.035035
10^{16}	1.000000
10^{17}	1.000000

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

When x is large, denominator has large error: possibly rounded to 0 resulting in divide by 0.

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

When x is large, denominator has large error: possibly rounded to 0 resulting in divide by 0.

$$\frac{1}{\sqrt{x^2+1}-x} = \frac{\sqrt{x^2+1}+x}{\left(\sqrt{x^2+1}-x\right)\left(\sqrt{x^2+1}+x\right)}$$

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

When x is large, denominator has large error: possibly rounded to 0 resulting in divide by 0.

$$\frac{1}{\sqrt{x^2+1}-x} = \frac{\sqrt{x^2+1}+x}{\left(\sqrt{x^2+1}-x\right)\left(\sqrt{x^2+1}+x\right)}$$
$$= \frac{\sqrt{x^2+1}+x}{x^2+1-x^2}$$

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

When x is large, denominator has large error: possibly rounded to 0 resulting in divide by 0.

$$\begin{aligned}\frac{1}{\sqrt{x^2+1}-x} &= \frac{\sqrt{x^2+1}+x}{\left(\sqrt{x^2+1}-x\right)\left(\sqrt{x^2+1}+x\right)} \\ &= \frac{\sqrt{x^2+1}+x}{x^2+1-x^2} \\ &= \sqrt{x^2+1}+x\end{aligned}$$

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\sqrt{x^2+1}-x}$$

When x is large, denominator has large error: possibly rounded to 0 resulting in divide by 0.

$$\begin{aligned}\frac{1}{\sqrt{x^2+1}-x} &= \frac{\sqrt{x^2+1}+x}{\left(\sqrt{x^2+1}-x\right)\left(\sqrt{x^2+1}+x\right)} \\ &= \frac{\sqrt{x^2+1}+x}{x^2+1-x^2} \\ &= \sqrt{x^2+1}+x\end{aligned}$$

Now tiny roundoff error

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\cos^2 x - \sin^2 x}$$

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\cos^2 x - \sin^2 x}$$

Normally fine when denominator is not 0, but when x is near $\pi/4$, we get cancellation error

Dealing with Roundoff errors

Rewrite formulae:

$$\frac{1}{\cos^2 x - \sin^2 x}$$

Normally fine when denominator is not 0, but when x is near $\pi/4$, we get cancellation error

$$\frac{1}{\cos^2 x - \sin^2 x} = \frac{1}{\cos 2x}$$

Now no problem where denominator is close to, but not 0

Distributed and Parallel Computing

Lecture 10

Alan P. Sexton

University of Birmingham

Spring 2018

Multi-Dimensional Kernels

Many problems are inherently multi-dimensional:

- 2 Dim: Image analysis, Matrix algebra etc.
- 3 Dim: Spatial sims, Fluid- or Thermo-dynamics, Weather etc.

CUDA provides support in the form of 2 and 3 dimensional kernels:

```
dim3 dimGrid2(GRID_WIDTH, GRID_HEIGHT) ;
dim3 dimBlock2(BLOCK_WIDTH, BLOCK_HEIGHT) ;
my2d_kernel<<<dimGrid, dimBlock>>>(...) ;
...
dim3 dimGrid3(GRID_WIDTH, GRID_HEIGHT, GRID_DEPTH) ;
dim3 dimBlock3(BLOCK_WIDTH, BLOCK_HEIGHT, BLOCK_DEPTH) ;
my3d_kernel<<<dimGrid, dimBlock>>>(...) ;
```

Multi-Dimensional Kernels

Many problems are inherently multi-dimensional:

- 2 Dim: Image analysis, Matrix algebra etc.
- 3 Dim: Spatial sims, Fluid- or Thermo-dynamics, Weather etc.

CUDA provides support in the form of 2 and 3 dimensional kernels:

```
dim3 dimGrid2(GRID_WIDTH, GRID_HEIGHT);
dim3 dimBlock2(BLOCK_WIDTH, BLOCK_HEIGHT);
my2d_kernel<<<dimGrid, dimBlock>>>(...);
...
dim3 dimGrid3(GRID_WIDTH, GRID_HEIGHT, GRID_DEPTH);
dim3 dimBlock3(BLOCK_WIDTH, BLOCK_HEIGHT, BLOCK_DEPTH);
my3d_kernel<<<dimGrid, dimBlock>>>(...);
```

- `dim3` is a `struct` with `x`, `y` and `z` fields and can take 1, 2 or 3 integer parameters in its constructor (missing parameters are initialised to 1).

Multi-Dimensional Kernels

Many problems are inherently multi-dimensional:

- 2 Dim: Image analysis, Matrix algebra etc.
- 3 Dim: Spatial sims, Fluid- or Thermo-dynamics, Weather etc.

CUDA provides support in the form of 2 and 3 dimensional kernels:

```
dim3 dimGrid2(GRID_WIDTH, GRID_HEIGHT);
dim3 dimBlock2(BLOCK_WIDTH, BLOCK_HEIGHT);
my2d_kernel<<<dimGrid, dimBlock>>>(...);
...
dim3 dimGrid3(GRID_WIDTH, GRID_HEIGHT, GRID_DEPTH);
dim3 dimBlock3(BLOCK_WIDTH, BLOCK_HEIGHT, BLOCK_DEPTH);
my3d_kernel<<<dimGrid, dimBlock>>>(...);
```

- `dim3` is a `struct` with `x`, `y` and `z` fields and can take 1, 2 or 3 integer parameters in its constructor (missing parameters are initialised to 1).
- The grid and block can have different dimensionalities

Multi-Dimensional Kernels

Many problems are inherently multi-dimensional:

- 2 Dim: Image analysis, Matrix algebra etc.
- 3 Dim: Spatial sims, Fluid- or Thermo-dynamics, Weather etc.

CUDA provides support in the form of 2 and 3 dimensional kernels:

```
dim3 dimGrid2(GRID_WIDTH, GRID_HEIGHT);
dim3 dimBlock2(BLOCK_WIDTH, BLOCK_HEIGHT);
my2d_kernel<<<dimGrid, dimBlock>>>(...);
...
dim3 dimGrid3(GRID_WIDTH, GRID_HEIGHT, GRID_DEPTH);
dim3 dimBlock3(BLOCK_WIDTH, BLOCK_HEIGHT, BLOCK_DEPTH);
my3d_kernel<<<dimGrid, dimBlock>>>(...);
```

- `dim3` is a `struct` with `x`, `y` and `z` fields and can take 1, 2 or 3 integer parameters in its constructor (missing parameters are initialised to 1).
- The grid and block can have different dimensionalities
- Note that the number of threads per block will be the product of the block dimensionalities and the number of blocks in the kernel will be the product of the grid dimensionalities

Built-in Variables available to Threads

Every thread has access to a number of variables: either `dim3` or `uint3` structs (`uint3` is like `dim3` except without constructor support)

- `gridDim`: `dim3`: the dimensions of the grid
- `blockDim`: `dim3`: the dimensions of the block
- `blockIdx`: `uint3`: the block index within the grid
- `threadIdx`: `uint3`: the thread index within the block

Thread Order

Because of issues of Global Memory Coalescing, Cache lines and Shared Memory Bank Conflicts, the ordering of threads, and the layout of vectors and matrices in the C language is important:

Threads are ordered in a block first by their `z` index, then by their `y`, then by their `x` index. Thus, in a $2 \times 2 \times 2$ dimensional block:
thread 0 has `threadIdx.z` $= 0$, `threadIdx.y` $= 0$, `threadIdx.x` $= 0$
thread 1 has `threadIdx.z` $= 0$, `threadIdx.y` $= 0$, `threadIdx.x` $= 1$
thread 2 has `threadIdx.z` $= 0$, `threadIdx.y` $= 1$, `threadIdx.x` $= 0$
thread 3 has `threadIdx.z` $= 0$, `threadIdx.y` $= 1$, `threadIdx.x` $= 1$
thread 4 has `threadIdx.z` $= 1$, `threadIdx.y` $= 0$, `threadIdx.x` $= 0$
thread 5 has `threadIdx.z` $= 1$, `threadIdx.y` $= 0$, `threadIdx.x` $= 1$
thread 6 has `threadIdx.z` $= 1$, `threadIdx.y` $= 1$, `threadIdx.x` $= 0$
thread 7 has `threadIdx.z` $= 1$, `threadIdx.y` $= 1$, `threadIdx.x` $= 1$

Memory Layout

Multi-dimensional arrays, `A[k][j][i]`, are ordered by their inner index (`k`) first, then their middle index (`j`), then by their outer index (`i`)

Thus if `A` is a $2 \times 2 \times 2$ matrix, it would be layed out in consecutive memory locations as:

```
0: A[0][0][0]
1: A[0][0][1]
2: A[0][1][0]
3: A[0][1][1]
4: A[1][0][0]
5: A[1][0][1]
6: A[1][1][0]
7: A[1][1][1]
```

Multi-Dimensional Indexing

Even when working multidimensionally, we often have to explicitly apply threads, which have their grid and block dimensionality, to the 2- or 3-dimensional structures of the dimensionality of our problem domain

For a problem domain data structure D of dimension $N \times N$, and the block dimensionality of size $K \times K$, we may, in our kernel access it as follows:

```
int i = blockIdx.x * K + threadIdx.x;
int j = blockIdx.y * K + threadIdx.y;

// assuming D is just a pointer to a block of memory:
... D[i + j*N] ...

// assuming D has been declared as a 2-dimensional C array:
... D[j][i] ...
```

Aside: global thread number in a 3-D model

Sometimes you need to know the thread number of a thread in the whole grid. The most general case is in a 3-d grid of 3-d blocks:

```
__device__ int getGlobalIdx_3D_3D()
{
  int bId = blockIdx.x
          + gridDim.x * blockIdx.y
          + gridDim.x * gridDim.y * blockIdx.z;
  int tId = bId * blockDim.x * blockDim.y * blockDim.z
          + threadIdx.x
          + blockDim.x * threadIdx.y
          + blockDim.x * blockDim.y * threadIdx.z;
  return tId;
}
```

Note that in a 1 or 2 dimensional block or grid configuration, all the `*Dim.*` values for the unused dimensions will have value 1 and all the `*Idx.*` values for those dimensions will have value 0

Coalesced Global Memory Accesses

```
int i = blockIdx.x * K + threadIdx.x;
int j = blockIdx.y * K + threadIdx.y;

... D[i + j*N] ...
```

- thread 0 of the block has `threadIdx.x` $= 0$ and `threadIdx.y` $= 0$
- thread 1 of the block has `threadIdx.x` $= 1$ and `threadIdx.y` $= 0$
- ...

Here i is the fastest changing index of the threads in the block, and an increment of 1 in i contributes a jump of 1 word location in D, or consecutive threads are accessing consecutive words. Hence the access is coalesced.

Coalesced Global Memory Accesses

```
int i = blockIdx.x * K + threadIdx.x;
int j = blockIdx.y * K + threadIdx.y;

... D[j + i*N] ...
```

- thread 0 of the block has `threadIdx.x` $= 0$ and `threadIdx.y` $= 0$
- thread 1 of the block has `threadIdx.x` $= 1$ and `threadIdx.y` $= 0$
- ...

Here i is the fastest changing index of the threads in the block, and an increment of 1 in i contributes a jump of N word locations in D, that is thread 0, 1, 2... is accessing $D[0], D[N], D[2N], \ldots$. Hence the access is **NOT** coalesced.

Shared Memory Bank Conflicts

Here, to avoid shared memory bank conflicts, each thread in a warp should, as far as possible, access a different memory bank in the same memory access instruction.

```
// Block dimension is K times K
    int x = threadIdx.x;     int y = threadIdx.y;
    __shared__ int tile[K][K];

    ... tile[y][x] ...
    ... tile[x][y] ...
```

- If K is 32, then the address that `tile[y][x]` accesses is `x+K*y` words from the start of `tile`.
- If `tile` starts on a 32 word boundary, this accesses shared memory bank $(x + Ky) \mod 32$
- `tile[y][x]`: T 0, 1, ...: addr 0, 1, 2, ... $\equiv$ bank 0, 1, 2, ...
- `tile[x][y]`: T 0, 1, ...: addr 0, 32, 64, ... $\equiv$ bank 0, 0, 0, ...

Shared Memory Bank Conflicts

Here, to avoid shared memory bank conflicts, each thread in a warp should, as far as possible, access a different memory bank in the same memory access instruction.

```
// Block dimension is K times K
    int x = threadIdx.x;     int y = threadIdx.y;
    __shared__ int tile[K][K];

    ... tile[y][x] ...
    ... tile[x][y] ...
```

- If K is 32, then the address that `tile[y][x]` accesses is `x+K*y` words from the start of `tile`.
- If `tile` starts on a 32 word boundary, this accesses shared memory bank $(x + Ky) \mod 32$
- `tile[y][x]`: T 0, 1, ...: addr 0, 1, 2, ... $\equiv$ bank 0, 1, 2, ...
- `tile[x][y]`: T 0, 1, ...: addr 0, 32, 64, ... $\equiv$ bank 0, 0, 0, ...
- What if K is $32k$ for some positive integer k?

Shared Memory Bank Conflicts

Here, to avoid shared memory bank conflicts, each thread in a warp should, as far as possible, access a different memory bank in the same memory access instruction.

```
// Block dimension is K times K
    int x = threadIdx.x;     int y = threadIdx.y;
    __shared__ int tile[K][K];

    ... tile[y][x] ...
    ... tile[x][y] ...
```

- If K is 32, then the address that `tile[y][x]` accesses is `x+K*y` words from the start of `tile`.
- If `tile` starts on a 32 word boundary, this accesses shared memory bank $(x + Ky) \mod 32$
- `tile[y][x]`: T 0, 1, ...: addr 0, 1, 2, ... $\equiv$ bank 0, 1, 2, ...
- `tile[x][y]`: T 0, 1, ...: addr 0, 32, 64, ... $\equiv$ bank 0, 0, 0, ...
- What if K is $32k$ for some positive integer k?
- What if K is 16?

Shared Memory Bank Conflicts

Here, to avoid shared memory bank conflicts, each thread in a warp should, as far as possible, access a different memory bank in the same memory access instruction.

```
// Block dimension is K times K
    int x = threadIdx.x;     int y = threadIdx.y;
    __shared__ int tile[K][K];

    ... tile[y][x] ...
    ... tile[x][y] ...
```

- If K is 32, then the address that `tile[y][x]` accesses is `x+K*y` words from the start of `tile`.
- If `tile` starts on a 32 word boundary, this accesses shared memory bank $(x + Ky) \mod 32$
- `tile[y][x]`: T 0, 1, ...: addr 0, 1, 2, ... $\equiv$ bank 0, 1, 2, ...
- `tile[x][y]`: T 0, 1, ...: addr 0, 32, 64, ... $\equiv$ bank 0, 0, 0, ...
- What if K is $32k$ for some positive integer k?
- What if K is 16?
- What if K is 32 but declaration is: `tile[K][K+1]`?

Transpose

Transpose is a simple, but important operation on 2-dimensional data types:

- For all `i`, `j`: swap `M[i][j]` with `M[j][i]`

```
// Matrix size (N x N)
#define N 1024

void transpose_HOST(int in[], int out[])
{
    for (int j = 0; j < N; ++j) // Loop over Rows
        for (int i = 0; i < N; ++i) // Loop over Columns
            out[j + i*N] = in[i+j*N];
}
```

Serial: 1 thread

```
#define N 1024

__global__ void transpose_serial
        (int in[], int out[])
{
    for (int j = 0; j < N; ++j) // Loop over Rows
        for (int i = 0; i < N; ++i) // Loop over Columns
            out[j + i * N] = in[i + j * N];
}
...
    transpose_serial<<<1, 1>>>(d_in, d_out);
```

Run the profiler

Run the profiler from within nsight, or from the command line with:

```
nvvp ./Transpose
```

- Make sure you are building a **Release** and not a **Debug** build
- If in nsight, make sure you have chosen the "Profile" perspective (top right of window)
- In the analysis tab at bottom, click on "`Examine Individual Kernels`", select a Kernel in the "`Results`" window, then click on "`Perform Additional Analysis`"
- At the bottom of the Results window, you can select a kernel and see the detailed results in the "`Properties`" window

Results

Some fields only appear when relevant.

Duration:	Time this kernel took
Register/Thread:	# registers allocated to each thread
Shared Memory/Block:	How much shared memory allocated to each block
Global Load Efficiency:	% of loads used (c.f. caches)
Global Store Efficiency:	% of stores used (c.f. caches)
Shared Efficiency:	% of shared accesses used (c.f. 32-word bus)
Warp Execution Efficiency:	100% = no divergence
Non-Predicated Warp Execution Efficiency:	100% = no divergence and no predication
Occupancy Achieved:	% of maximum # of warps active each active cycle on average
Occupancy Theoretical:	Peak achievable with this config
Occupancy Limiter:	Any config issue that limits occupancy

Result: Serial

Factor	GTX 1080 Ti	GTX 960
Duration:	160ms	192ms
Register/Thread:	20	15
Shared Memory/Block:	0 B	0 B
Global Load Efficiency:	12.5%	12.5%
Global Store Efficiency:	12.5%	12.5%
Shared Efficiency:	n/a	n/a
Warp Execution Efficiency:	3.1%	3.1%
Non-Predicated Warp Execution Efficiency:	3.1%	3.1%
Occupancy Achieved:	1.6%	1.6%
Occupancy Theoretical:	50%	50%
Occupancy Limiter:	Grid Size	Grid Size

1 thread per row

```
__global__ void transpose_thread_per_row
        (int in[], int out[])
{
    int i = threadIdx.x;
    for (int j = 0; j < N; ++j) // Loop over Rows
        out[j + i * N] = in[i + j * N];
}
...
    transpose_thread_per_row<<<1, N>>>(d_in, d_out);
```

Result: Thread per Row

Factor	GTX 1080 Ti	GTX 960
Duration:	1.18ms	1.59ms
Register/Thread:	17	15
Shared Memory/Block:	0 B	0 B
Global Load Efficiency:	100%	100%
Global Store Efficiency:	12.5%	12.5%
Shared Efficiency:	n/a	n/a
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	49.9%	49.9%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

1 thread per element

```
#define N 1024
#define K 32

__global__ void transpose_thread_per_element
        (int in[], int out[])
{
    int i = blockIdx.x * K + threadIdx.x;
    int j = blockIdx.y * K + threadIdx.y;

    out[j + i * N] = in[i + j * N];
}
...
    dim3 blocks(N / K, N / K);
    dim3 threads(K, K);
    transpose_thread_per_element<<<blocks, threads>>>(d_in,
        d_out);
```

Result: Thread per Element

Factor	GTX 1080 Ti	GTX 960
Duration:	58μs	241μs
Register/Thread:	8	8
Shared Memory/Block:	0 B	0 B
Global Load Efficiency:	100%	100%
Global Store Efficiency:	12.5%	12.5%
Shared Efficiency:	n/a	n/a
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	67.5%	68.3%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

Copy thread per element, coalesced

```
__global__ void copy_thread_per_element_coalesced
        (int in[], int out[])
{
    int i = blockIdx.x * K + threadIdx.x;
    int j = blockIdx.y * K + threadIdx.y;

    out[i + j * N] = in[i + j * N];
}
...
    copy_thread_per_element_coalesced<<<blocks, threads>>>(
        d_in, d_out);
```

Result: Copy thread per element, coalesced

Factor	GTX 1080 Ti	GTX 960
Duration:	23μs	98μs
Register/Thread:	8	8
Shared Memory/Block:	0 B	0 B
Global Load Efficiency:	100%	100%
Global Store Efficiency:	100%	100%
Shared Efficiency:	n/a	n/a
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	72.4%	78.8%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

Copy — thread per element, non-coalesced

```
__global__ void copy_thread_per_element_non_coalesced
        (int in[], int out[])
{
    int i = blockIdx.x * K + threadIdx.x;
    int j = blockIdx.y * K + threadIdx.y;

    out[j + i * N] = in[j + i * N];
}
...
    copy_thread_per_element_non_coalesced<<<blocks, threads
        >>>(d_in, d_out);
```

Result: Copy thread per element, non-coalesced

Factor	GTX 1080 Ti	GTX 960
Duration:	89μs	341μs
Register/Thread:	8	8
Shared Memory/Block:	0 B	0 B
Global Load Efficiency:	12.5%	12.5%
Global Store Efficiency:	12.5%	12.5%
Shared Efficiency:	n/a:	n/a
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	72.4%	70.3%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

1 thread per element tiled

```
__global__ void transpose_thread_per_element_tiled
        (int in[], int out[])
{
    int in_corner_i = blockIdx.x * K;
    int in_corner_j = blockIdx.y * K;

    int out_corner_i = blockIdx.y * K;
    int out_corner_j = blockIdx.x * K;

    int x = threadIdx.x;
    int y = threadIdx.y;

    __shared__ int tile[K][K];

    tile[y][x] = in[(in_corner_i+x) + (in_corner_j+y)*N];
    __syncthreads();
    out[(out_corner_i+x) + (out_corner_j+y)*N] = tile[x][y];
}
...
    transpose_thread_per_element_tiled<<<blocks, threads>>>(
        d_in, d_out);
```

Result: 1 thread per element tiled

Factor	GTX 1080 Ti	GTX 960
Duration:	38μs	161μs
Register/Thread:	11	12
Shared Memory/Block:	4 KiB	4 KiB
Global Load Efficiency:	100%	100%
Global Store Efficiency:	100%	100%
Shared Efficiency:	6.1%	6.1%
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	84.8%	82.4%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

1 thread per element tiled and padded

```
__global__ void transpose_thread_per_element_tiled_padded
        (int in[], int out[])
{
    int in_corner_i = blockIdx.x * K;
    int in_corner_j = blockIdx.y * K;

    int out_corner_i = blockIdx.y * K;
    int out_corner_j = blockIdx.x * K;

    int x = threadIdx.x;
    int y = threadIdx.y;

    __shared__ int tile[K][K + 1];

    tile[y][x] = in[(in_corner_i+x) + (in_corner_j+y)*N];
    __syncthreads();
    out[(out_corner_i+x) + (out_corner_j+y)*N] = tile[x][y];
}
...
    transpose_thread_per_element_tiled_padded<<<blocks,
        threads>>>(d_in, d_out);
```

Result: 1 thread per element tiled and padded

Factor	GTX 1080 Ti	GTX 960
Duration:	23μs	100μs
Register/Thread:	11	10
Shared Memory/Block:	4.125 KiB	4.125 KiB
Global Load Efficiency:	100%	100%
Global Store Efficiency:	100%	100%
Shared Efficiency:	100%	100%
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	?%	90.4%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

Waits on barrier syncs?

Lots of warps in block, maybe delays while they have to wait for all warps to sync?

- Try reducing block size ($K = 16$)
- More blocks, fewer warps per block: different blocks can run on the same SM without waiting for each other

But: beware of limiting factors:

Factor	GTX 1080 Ti	GTX 960
# Threads/Block	64	32
# Thread/SM	2048	2048
Registers/Block	65536	65536
Shared Mem/Block	49142 bytes	49152 bytes

Result: 1 thread per element tiled and padded, K=16

Factor	GTX 1080 Ti	GTX 960
Duration:	23μs	103μs
Register/Thread:	11	10
Shared Memory/Block:	1.062 KiB	1.062 KiB
Global Load Efficiency:	100%	100%
Global Store Efficiency:	100%	100%
Shared Efficiency:	50%	50%
Warp Execution Efficiency:	100%	100%
Non-Predicated Warp Execution Efficiency:	100%	100%
Occupancy Achieved:	88.7%	92.3%
Occupancy Theoretical:	100%	100%
Occupancy Limiter:	none	none

Reduced waiting on `__syncthreads()` balanced out by reduced shared efficiency

Matrix-Matrix Multiplication

$$\begin{bmatrix} A_{11} & \dots & A_{1n} \\ \vdots & & \vdots \\ A_{m1} & \dots & A_{mn} \end{bmatrix} \begin{bmatrix} B_{11} & \dots & B_{1p} \\ \vdots & & \vdots \\ B_{n1} & \dots & B_{np} \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{n} A_{1k}B_{k1} & \dots & \sum_{k=1}^{n} A_{1k}B_{kp} \\ \vdots & & \vdots \\ \sum_{k=1}^{n} A_{mk}B_{k1} & \dots & \sum_{k=1}^{n} A_{mk}B_{kp} \end{bmatrix}$$

- For simplicity we will restrict ourselves to square matrices ($m = n = p$).

Kernel Invocation

- We use a 2-dimensional layout to match the matrix structure.
- Each thread will calculate a single component of the result.

```
#define BLOCK_WIDTH 16
...
int numBlocks = (n - 1) / BLOCK_WIDTH + 1 ;
dim3 dimGrid(numBlocks, numBlocks) ;
dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH) ;
simpleMMM<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, n) ;
...
```

Simple Matrix-Matrix Multiplication Kernel

For a single component of the result matrix:

$$(AB)_{\text{row,col}} = \sum_{k=1}^{n} A_{\text{row},k} B_{k,\text{col}}$$

$$= \begin{bmatrix} A_{\text{row},1} & \cdots & A_{\text{row},n} \end{bmatrix} \begin{bmatrix} B_{1,\text{col}} \\ \vdots \\ B_{n,\text{col}} \end{bmatrix}$$

```
__global__ void simpleMMM(float *d_A, float *d_B,
                          float *d_C, int n)
{
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;
    if (row < n && col < n)
    {
        float val = 0 ;
        for (int k = 0 ; k < n ; k++)
            val += d_A[row*n + k] * d_B[k*n + col] ;
        d_C[row*n + col] = val ;
    }
}
```

Memory Access Efficiency

- Note: multiple threads reading the same global addresses:
  - thread 0,0 combines row 0 of A with column 0 of B
  - thread 0,1 combines row 0 of A with column 1 of B
  - ...
- Recall the discussion previously (lecture 04) of the *Compute to Global Memory Access (CGMA)* ratio.
- What is the CGMA ratio of the inner loop of `simpleMMM`?

Memory Access Efficiency

- Note: multiple threads reading the same global addresses:
  - thread 0,0 combines row 0 of A with column 0 of B
  - thread 0,1 combines row 0 of A with column 1 of B
  - ...
- Recall the discussion previously (lecture 04) of the *Compute to Global Memory Access (CGMA)* ratio.
- What is the CGMA ratio of the inner loop of `simpleMMM`?
- Each iteration, 2 global word memory accesses, one floating mult and one floating add: hence a CGMA ratio of 1.
- GTX1080 Ti: 5505MHz, 352bits bus $\Rightarrow$ 242.22GB/s
- GTX960: 3600MHz, 128 bits bus $\Rightarrow$ 57.6 GB/s

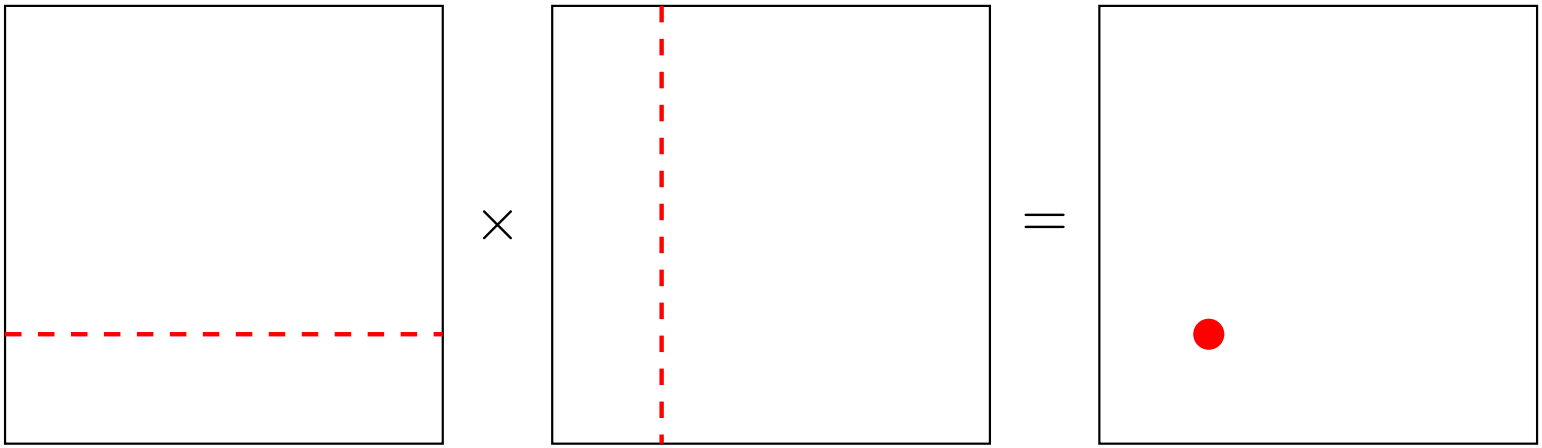
Memory Access Efficiency

- Note: multiple threads reading the same global addresses:
  - thread 0,0 combines row 0 of A with column 0 of B
  - thread 0,1 combines row 0 of A with column 1 of B
  - ...
- Recall the discussion previously (lecture 04) of the *Compute to Global Memory Access (CGMA)* ratio.
- What is the CGMA ratio of the inner loop of `simpleMMM`?
- Each iteration, 2 global word memory accesses, one floating mult and one floating add: hence a CGMA ratio of 1.
- GTX1080 Ti: 5505MHz, 352bits bus $\Rightarrow$ 242.22GB/s
- GTX960: 3600MHz, 128 bits bus $\Rightarrow$ 57.6 GB/s
- If global memory bandwidth is 200GB/s, and 4 bytes/word, then we are limited to 50 Gflops, when the hardware could support maybe 1500 Gflops.

Memory Access Efficiency

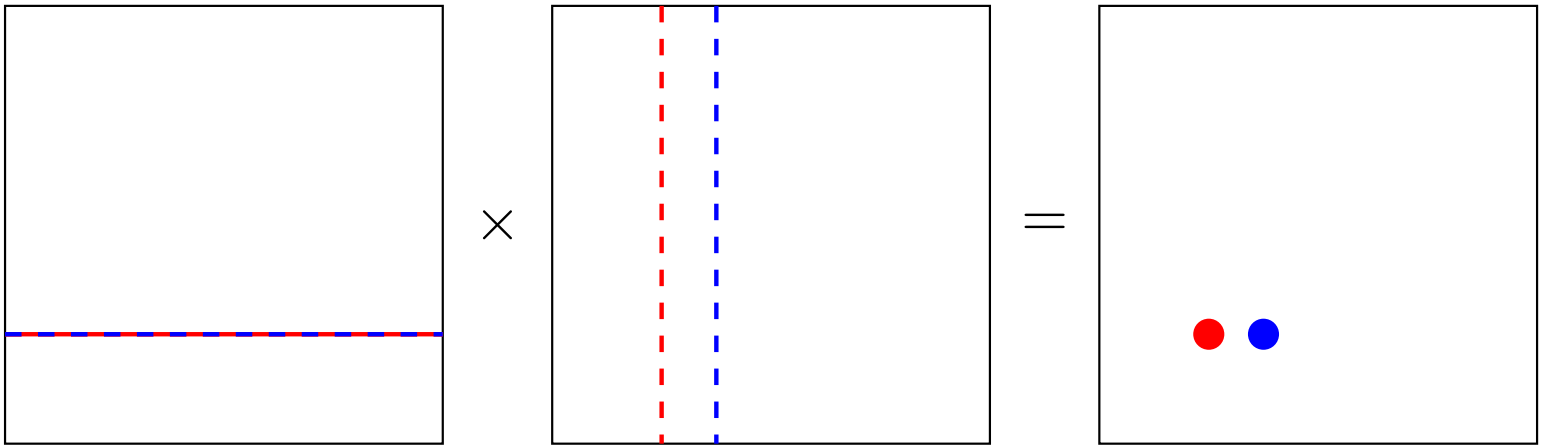
- Note: multiple threads reading the same global addresses:
  - thread 0,0 combines row 0 of A with column 0 of B
  - thread 0,1 combines row 0 of A with column 1 of B
  - ...
- Recall the discussion previously (lecture 04) of the *Compute to Global Memory Access (CGMA)* ratio.
- What is the CGMA ratio of the inner loop of `simpleMMM`?
- Each iteration, 2 global word memory accesses, one floating mult and one floating add: hence a CGMA ratio of 1.
- GTX1080 Ti: 5505MHz, 352bits bus $\Rightarrow$ 242.22GB/s
- GTX960: 3600MHz, 128 bits bus $\Rightarrow$ 57.6 GB/s
- If global memory bandwidth is 200GB/s, and 4 bytes/word, then we are limited to 50 Gflops, when the hardware could support maybe 1500 Gflops.
- Need to work in shared memory.

Matrix Multiplication Graphically



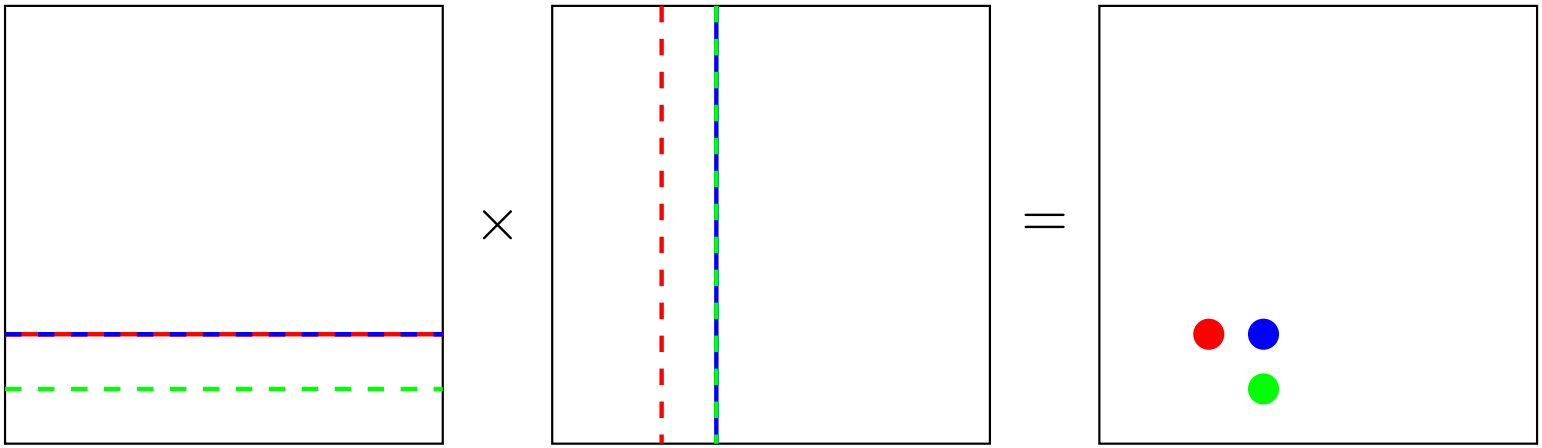
- Each element of C is made up of the dot-product of one row of A and one column of B

Matrix Multiplication Graphically



- Each element of C is made up of the dot-product of one row of A and one column of B
- Each row of A is read once for every column of B, i.e. n times

Matrix Multiplication Graphically



- Each element of C is made up of the dot-product of one row of A and one column of B
- Each row of A is read once for every column of B, i.e. n times
- Each column of B is read once for every row of A, i.e. n times

Tiled Matrix-Matrix Multiplication Idea

Work in tiles:

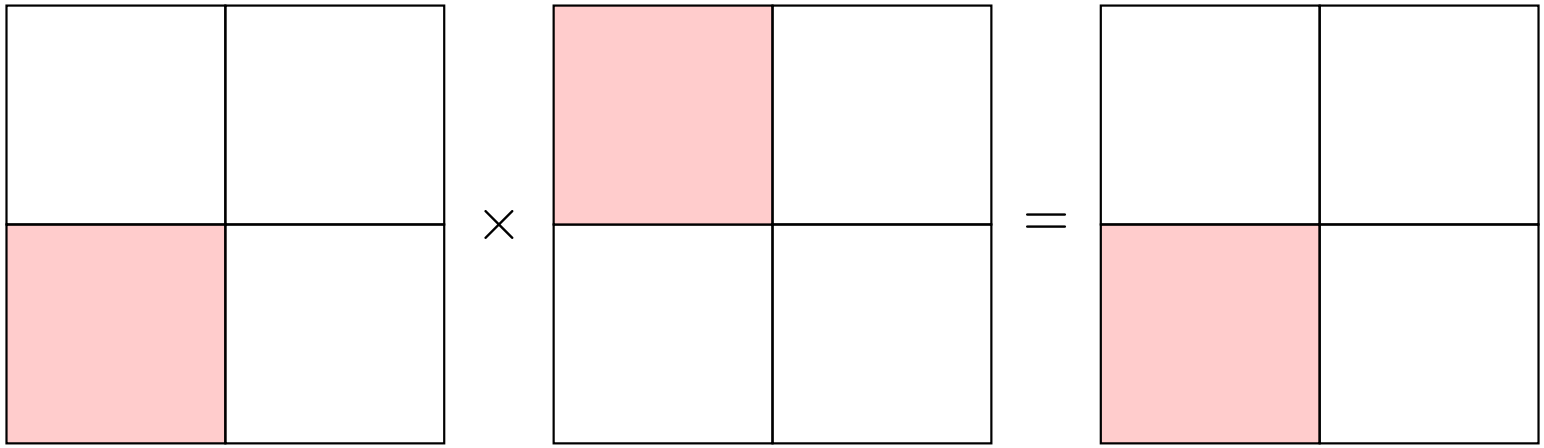
- Each thread calculates same value as before
- But reorganise into nested loop over tiles. Instead of:

```
for each dot product pair in matrix for this location
    accumulate dot product result
```

do the following:

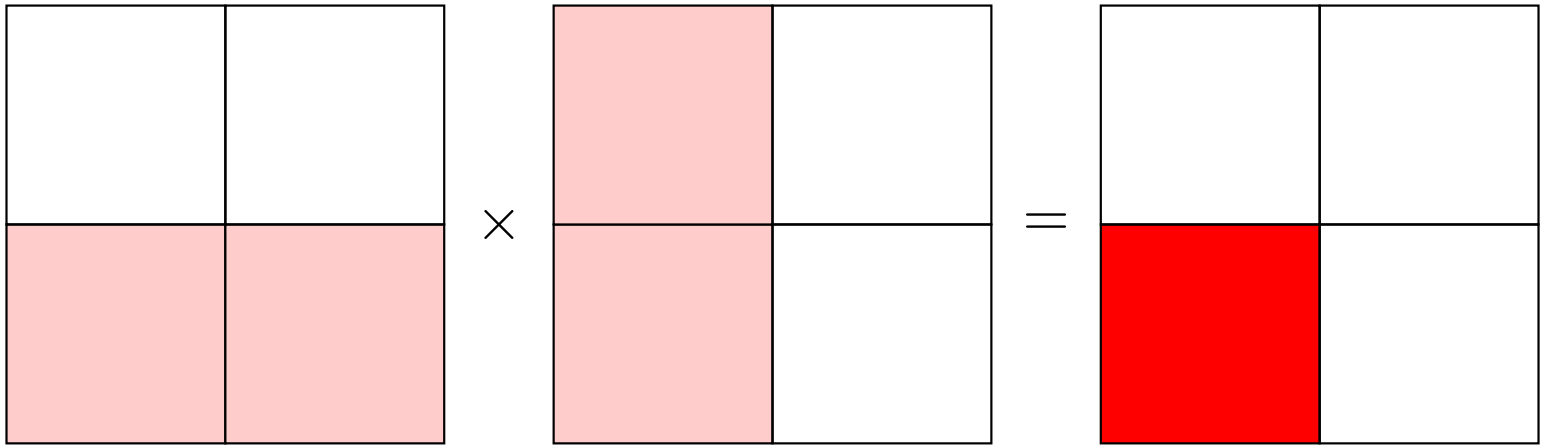
```
for each tile
    copy global tile to shared tile
    for each dot product pair in tile for this location
        accumulate dot product result
```

Tiled Matrix Multiplication Graphically



- Example: assume tile size is 32x32 and matrices are 64x64, i.e. matrices are 2x2 tiles
- When calculating the (row=1, col=0) tile of C, first copy to shared memory the (1,0) tile of A and the (0,0) tile of B and calculate the **PARTIAL** dot products for all cells of the (1,0) tile of C ...

Tiled Matrix Multiplication Graphically



- Example: assume tile size is 32x32 and matrices are 64x64, i.e. matrices are 2x2 tiles
- When calculating the (row=1, col=0) tile of C, first copy to shared memory the (1,0) tile of A and the (0,0) tile of B and calculate the **PARTIAL** dot products for all cells of the (1,0) tile of C ...
- ...then copy the (1,1) tile of A and the (1,0) tile of B and complete calculating the dot products for all cells of the (1,0) tile of C

Tiled Matrix-Matrix Multiplication

```
__global__ void simpleMMM(float *d_A, float *d_B,
                          float *d_C, int n)
{
    __shared__ float As[TILE_DIM][TILE_DIM];
    __shared__ float Bs[TILE_DIM][TILE_DIM];
    int bx = blockIdx.x;     int by = blockIdx.y;
    int tx = threadIdx.x;    int ty = threadIdx.y;
    int row = by * TILE_DIM + ty;
    int col = bx * TILE_DIM + tx;
//  can't do test because of sync: pad matrices instead
//  if (row < n && col < n)
    {   float val = 0 ;
        for (int m = 0 ; m < n/TILE_DIM ; m++)
        {
            As[ty][tx] = d_A[row * n + m * TILE_DIM + tx];
            Bs[ty][tx] = d_B[(m * TILE_DIM + ty) * n + col];
            __syncthreads();
            for (int k = 0 ; k < TILE_DIM ; k++)
                val += As[ty][k] * Bs[k][tx] ;
            __syncthreads();
        }
        d_C[row*n + col] = val ;
    }
}
```

Results

- Each thread is still calculating the same value as before.
- Calculation is now divided into phases: 1 phase per tile
- Each tile copied from global to shared memory allows `TILE_DIM` threads to run without further global memory accesses
- Hence number of global memory accesses is divided by `TILE_DIM`

Results

- Each thread is still calculating the same value as before.
- Calculation is now divided into phases: 1 phase per tile
- Each tile copied from global to shared memory allows `TILE_DIM` threads to run without further global memory accesses
- Hence number of global memory accesses is divided by `TILE_DIM`
- `TILE_DIM` $= 16 \Rightarrow$ 200Gb memory bandwidth can support $(200/4) \times 16 = 800\,GFlops$

Results

- Each thread is still calculating the same value as before.
- Calculation is now divided into phases: 1 phase per tile
- Each tile copied from global to shared memory allows `TILE_DIM` threads to run without further global memory accesses
- Hence number of global memory accesses is divided by `TILE_DIM`
- `TILE_DIM` $= 16 \Rightarrow$ 200Gb memory bandwidth can support $(200/4) \times 16 = 800\,GFlops$
- Modern GPUs can support square tile dimensions of size 32, i.e. 1600GFlops.