ECE569
Module 33

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

- Reduction

What is a reduction computation?

- Summarize a set of input values into one value using a “reduction operation”
  - Max
  - Min
  - Sum
  - Product
- An ideal application offering optimization opportunities
  - All covered by basic efficiency rules
  - Plus bank conflicts

Reduction enables other techniques

- **Reduction is also needed to clean up after some commonly used parallelizing transformations**
- **Privatization**
  - Multiple threads write into an output location
  - Replicate the output location so that each thread has a private output location (privatization)
  - Use a reduction tree to combine the values of private locations into the original output location

Reduction formal definition

- **Inputs:**
  - Set of elements
  - Reduction operator
    - Binary
      - Works on two inputs, generates one output
      - Associative, order of operation does not matter
      - Has a well-defined identity value
  - Which of the following are both binary and associative?
- ❑ Multiply (a*b)
- ❑ Minimum (a min b)
- ❑ Factorial (a!)
- ❑ Logical or (a || b)
- ❑ Bitwise and (a & b)
- ❑ Exponentiation (a^b)
- ❑ Division (a/b)

Sequential Reduction O(N)

- **Initialize the result as an identity value for the reduction operation**
  - Smallest possible value for max reduction
  - Largest possible value for min reduction
  - 0 for sum reduction
  - 1 for product reduction
- **Iterate through the input and perform the reduction operation between the result value and the current input value**

Serial Reduction

```
sum = 0;  //identity
for(i=0; i<n; i++) {
  sum = sum + array[i]:
}
return sum;
```

How many operations does it take to complete?

What is the work complexity in O(?)

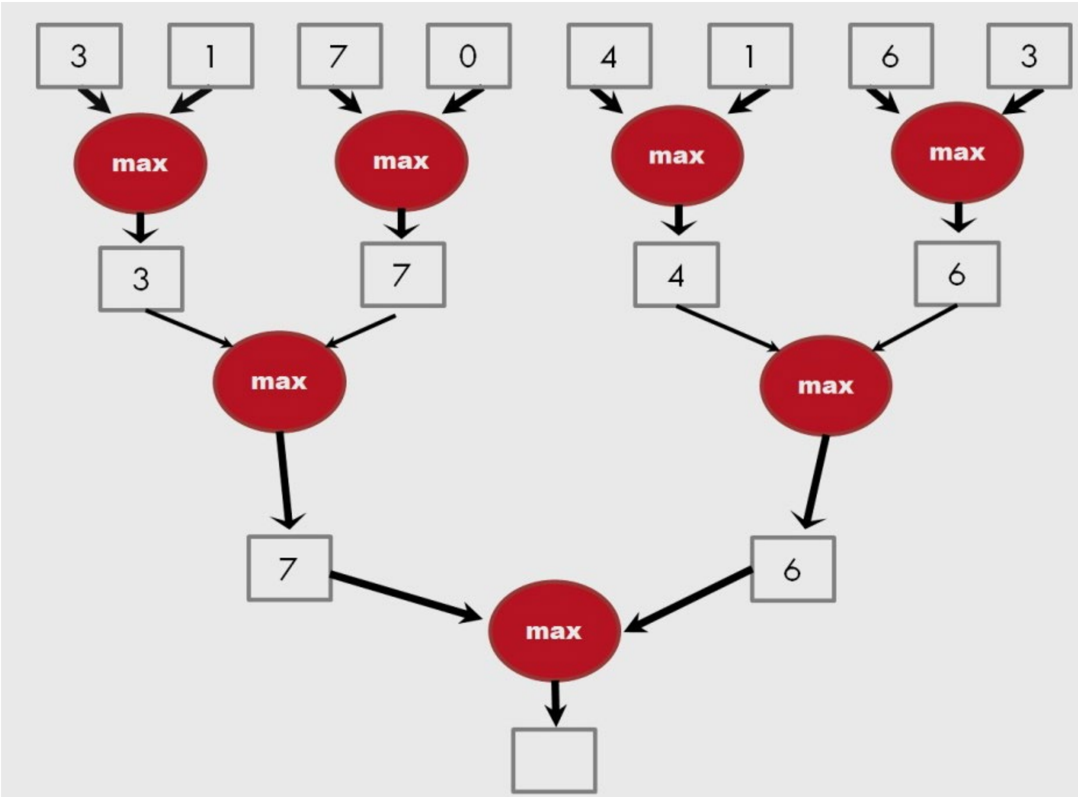
What is the step complexity in O(?)

Reduction - Tesla P100;compute v6.0;

n: 1<<20

Version	Time (ms)
Serial	3.27400

Reduction Steps and Operations



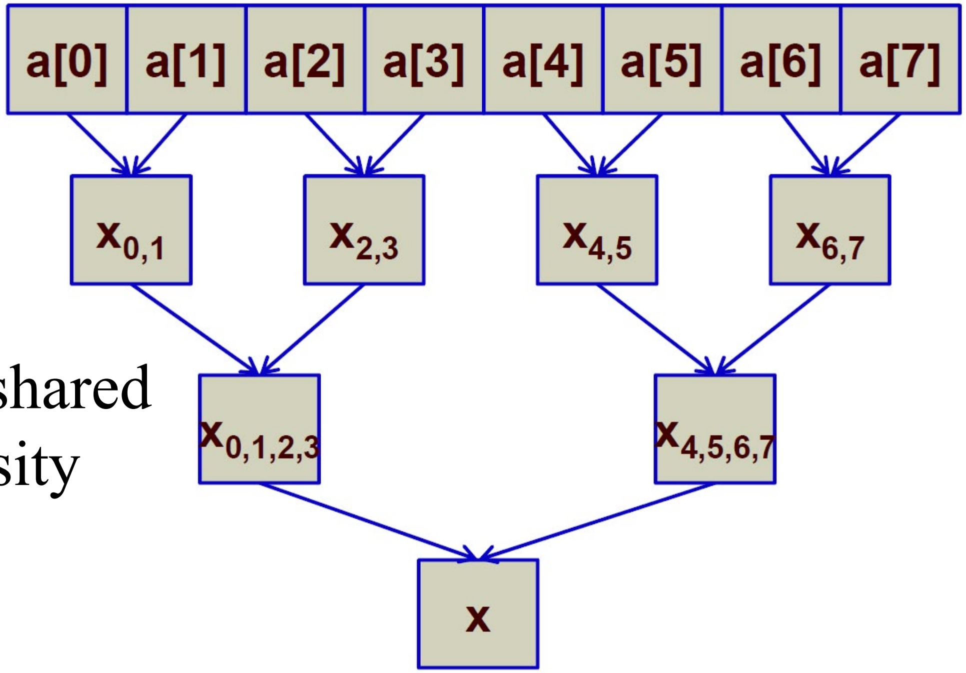
- Many-To-One
- Parameter to Tune
  - Thread Width
  - (total number of threads)

- Step Complexity:
- Work Complexity:
  – Work efficiency: Balance of step and work complexity

Parallel Sum Reduction: Step and Work Complexity

N	Steps
2	1
4	2
8	3
. . .	
1024	10

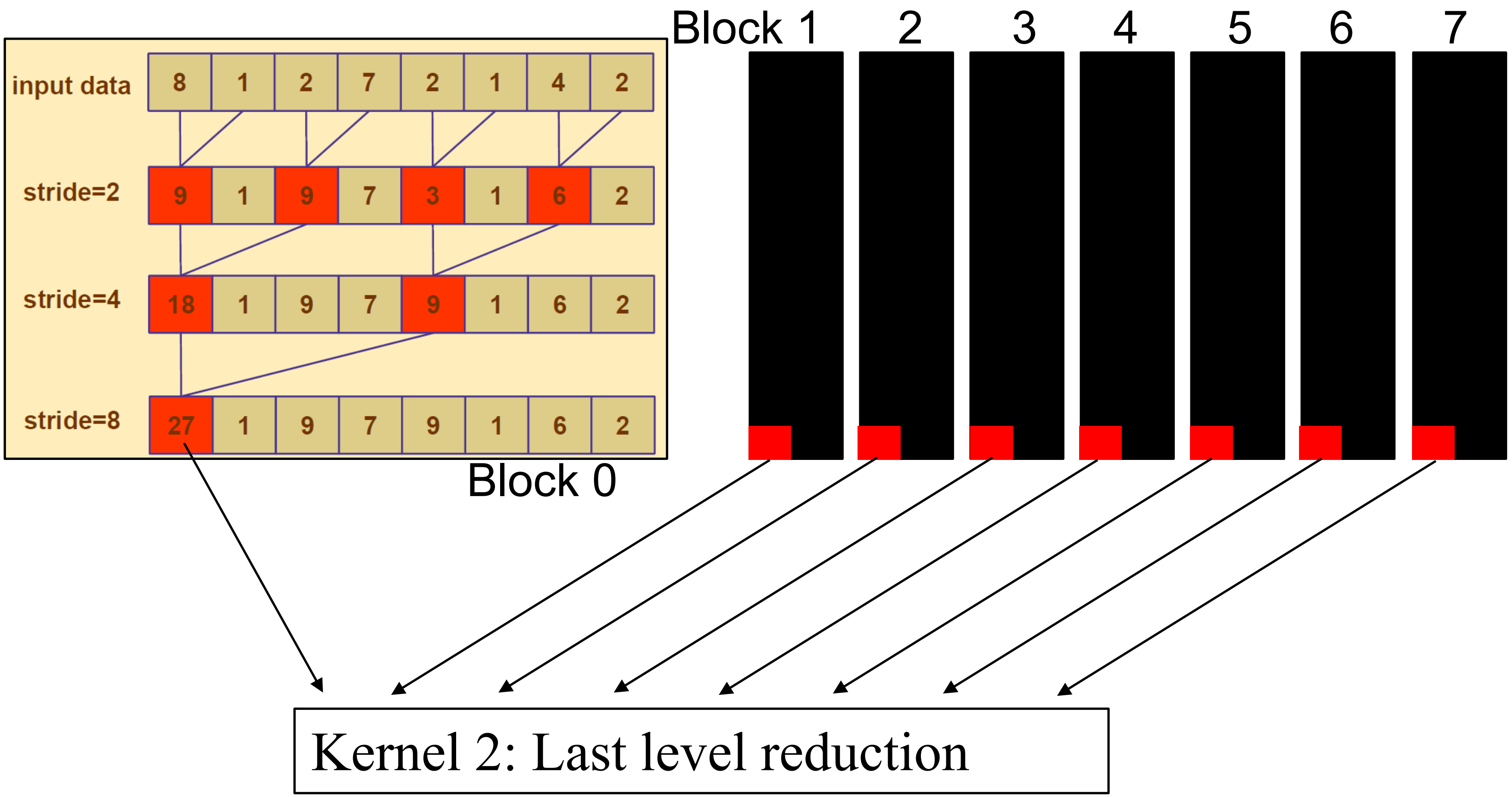
- 1023 steps in serial vs 10 steps in parallel
  - Two orders of magnitude!
- However we need enough processors
  - Stage 1 requires 512 adders!



Problems:
partial results need to be shared
very low arithmetic intensity
Ideally:
global synchronization

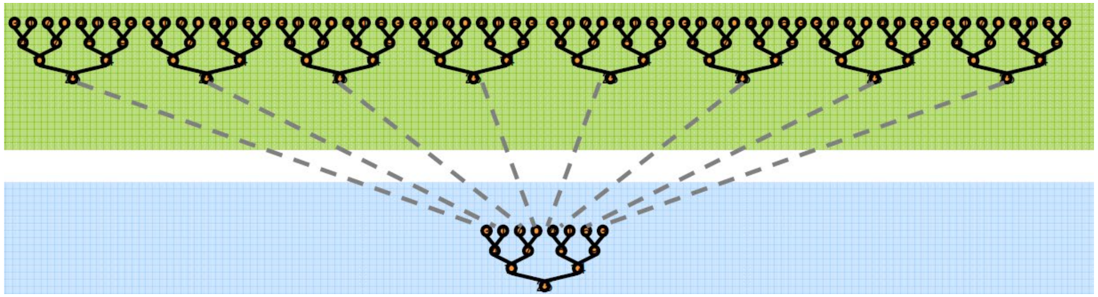
Need for block level synchronization with multiple kernel launches

- **32 elements, 4 threads/block**
  - Kernel 1: Single block parallel reduction



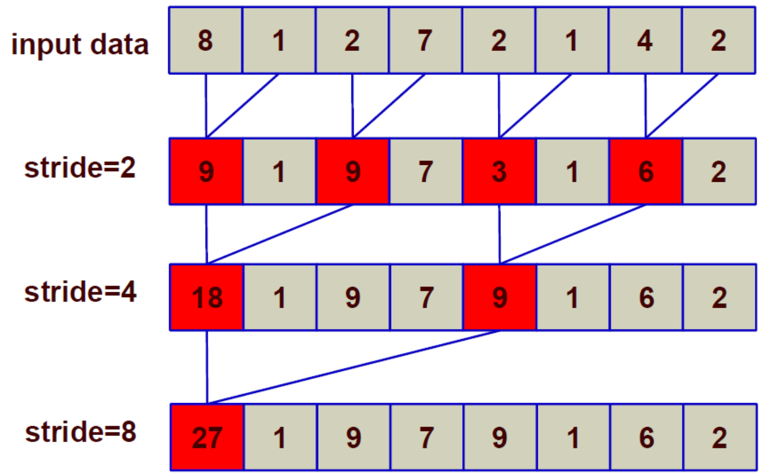
Reducing 2^20 (Million) Elements

- **Assume 1024 blocks and 1024 threads/block**
  - After the execution we have 1024 results to accumulate
- **Requires 2 phases (Kernels)**
  - Phase1: 1024 blocks, 1024threads/block
  - Phase2: 1 Block, 1024 threads
    - Kernel launch serves as a global synchronization point
      - Negligible overhead
  - Code for both phases is the same



Reduction Kernel

- To produce each element of the output, a fragment program reads two or more values and computes a new one using the reduction operator (addition)
- On each pass, the size of the output (the computational range) is reduced by some fraction.
- These passes continue until the output is a single-element buffer, at which point we have our reduced result.



Optimization Goal

- Reduction algorithms take only 1 flop per element loaded.
- They are not *compute bound*,
  - Not limited by flops performance,
- They are *memory bound*,
  - limited by memory bandwidth.
- When judging the performance of code for reduction algorithms, we have to compare to the peak memory bandwidth and not to the theoretical peak flops count.

Main function (sum of array) – Demo

```
:
const int ARRAY_SIZE = 1 << 20;
const int ARRAY_BYTES = ARRAY_SIZE * sizeof(float);
// generate the input array on the host
float h_in[ARRAY_SIZE];
float sum = 0.0f;
for(int i = 0; i < ARRAY_SIZE; i++) {
   // generate random float in [-1.0f, 1.0f]
   h_in[i] = -1.0f + (float)random()/((float)RAND_MAX/2.0f);
   sum += h_in[i];
}
```

For the template code refer to D2L→Content→Demo→7.Reduction

Main function (cntd)

```
// declare GPU memory pointers
float * d_in, * d_intermediate, * d_out;
// allocate GPU memory
cudaMalloc((void **) &d_in, ARRAY_BYTES);
// overallocated for intermediate
cudaMalloc((void **) &d_intermediate, ARRAY_BYTES);
cudaMalloc((void **) &d_out, sizeof(float));
// transfer the input array to the GPU
cudaMemcpy(d_in, h_in, ARRAY_BYTES, cudaMemcpyHostToDevice);
int whichKernel = 0;
cudaEvent_t start, stop;
// whichKernel 4 -> Global, naïve version,
// whichKernel 3 to 0 -> other versions
for(whichKernel=0; whichKernel<5; whichKernel++) {
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    // launch the kernel
```

Main function (cntd)

```
case 0:
:
case 4:
   printf("Running global reduce stride - naive\n");
   cudaEventRecord(start, 0);
   for (int i = 0; i < 100; i++) {
     reduce(d_out, d_intermediate, d_in, ARRAY_SIZE, 4);
   }
   cudaEventRecord(stop, 0);
   break;
}
```

Main function (cntd)

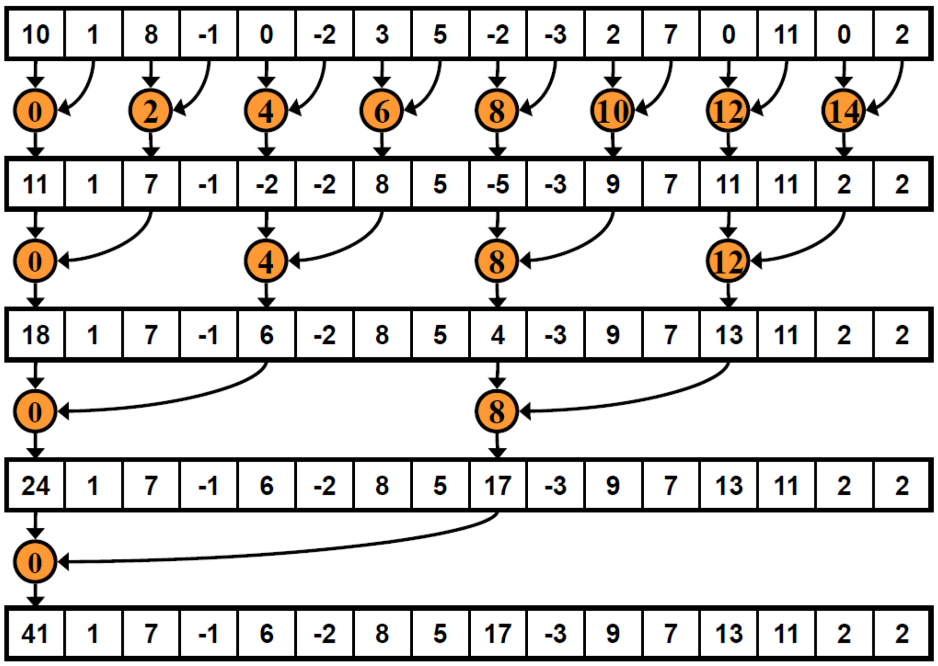
```
:
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
elapsedTime /= 100.0f;      // 100 trials

// copy back the sum from GPU
float h_out;
cudaMemcpy(&h_out, d_out, sizeof(float),
cudaMemcpyDeviceToHost);
printf("serial result = %f, gpu result = %f\n",sum,h_out);
printf("average time elapsed >>>>>>> %f\n", elapsedTime);
    printf("\n");
}
  // end of for loop of kernel versions
```

Kernel Launch

```
void reduce(float * d_out,float * d_intermediate,float * d_in,
            int size, int version){
// assumes that size is not greater than maxThreadsPerBlock^2
// and that size is a multiple of maxThreadsPerBlock
const int maxThreadsPerBlock = 1024;
int threads = maxThreadsPerBlock;
int blocks = size / maxThreadsPerBlock;
if (version == 4){
   global_reduce_stride<<<blocks, threads>>>(d_intermediate,  d_in);
} else if (version == 3) { \\call version 3}
:
:
else if (version == 0) { \\ call version 0 }
// now we're down to one block left, so reduce it
threads = blocks; //launch one thread per block in prev step
blocks = 1;
if (version == 4) {
  global_reduce_stride<<<blocks, threads>>>(d_out, d_intermediate);}
}
```

Next: Global Memory – Stride Pattern-1




```
__global__ void global_reduce_stride(float * d_out, float * d_in)
{


}
```