AMS 148 Chapter 5: Reduce and Scan

Steven Reeves

In this chapter we investigate the **reduce** type algorithm for GPU applications. Reduction algorithms are a set of algorithms take a set of data (usually large) and produces one output. Key examples: computing the maximum, and computing the sum of the data.

1 Road to Parallel Reduction

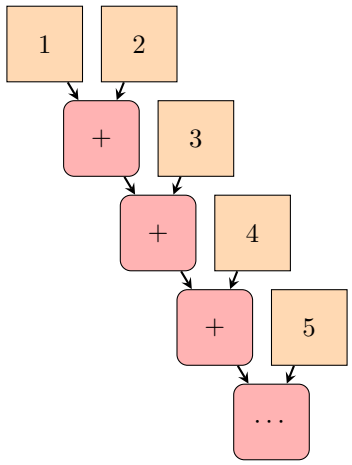
This type of algorithm is more challenging because it requires cooperation between processors.

Suppose we wish to add the numbers 1 through 10 together. Serially this is trivial, we just write the code in a for loop:

```
int summ = 0;
for(int i = 1; i <=10; i++)
        summ += i;
```

However, if we wish to do this in a parallel fashion we should investigate a computational tree.

Figure 1: Computational Tree Add Reduce



1.1 Reduce as Mathematical Function

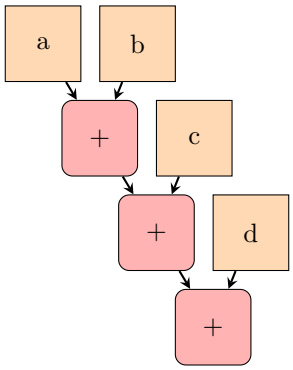
Reduce can be thought of as a mathematical function with two inputs: an array and an operation.

$$\mathcal{R}(\mathbf{x}, \oplus)$$

Reduction will give one output. There are some restrictions on this operator. The operator must be binary, the operator takes two variables and outputs one. And this operator must be **associative**. That is, let $\oplus$ be a binary operator then for it to be associative $(a \oplus b) \oplus c = a \oplus (b \oplus c)$. For example, addition is a binary associative operator. Associativity is necessary to reorganize reduce to be parallel.

Consider the addition of 4 variables a, b, c, d. When we write the reduction algorithm as a tree:

Figure 2: Serial Add



Mathematically:

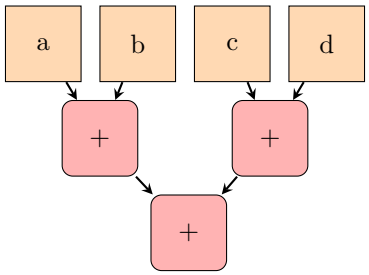
$$((a+b)+c)+d$$

However, if we rearrange using associativity:

$$(a+b)+(c+d)$$

we can construct a new tree:

Figure 3: Parallel Add



Suppose we have two threads. In the above example, thread 0 would add a and b, where as thread 1 would add c and d. On the next row one thread would combine the results for the final value.

1.2 Complexity of Parallel Reduce

By abstracting the example in 3 we can find a relation between the number of data entries and the number of steps to complete the computation.

N	Steps
2	1
4	2
8	3

Table 1: Step complexity for parallel reduce

Observing the pattern in table 1 leads us to believe that the step complexity of a parallelized reduction algorithm is

$$\mathcal{O}(\log_2(N))$$

So if we needed to reduce 1024 elements it would only take 10 steps, instead of 1023 steps in the serial case, this is 2 orders of magnitude! Now we can start to see the potential speed up power of using a parallel algorithm. Note that it only scales this way if we have enough threads to handle these operations simultaneously.

1.3 Brent's Theorem

The computational tree's that we showed above are known as *directed acyclic graphs*(DAG). A DAG is useful to depict algorithms and their dependencies.

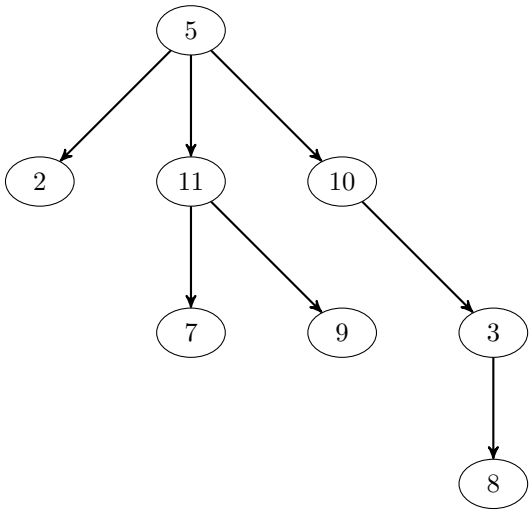


Figure 4: Example directed acyclic graph

Constructing a DAG from an algorithm Each fundamental unit of computation is represented by a node. A directed edge is drawn from node x to node y if computation x is required as an input for computation y. The resulting graph should be connected to represent an algorithm.

Note that operations on different levels **cannot** be computed in parallel together. However, the nodes on the same level illustrates data parallel computations. Essentially by constructing a DAG for your algorithm, you are discovering the inherent concurrency.

On a sequential machine, each node is computed top-down in series. We look for the *leaves* of the tree, asincethese depend on no prior computations. We evaluate all leafs and then continue through the tree until we reach the root node(bottom most).

So how long does it take to complete the graph? The time is proportional to the number of nodes in the graph. So let's define

$$T_1 = \text{number of nodes in DAG}$$

So, T_1 is going to be the time it takes to compute a DAG in a sequential meaching.

Suppose we had an infinite ammount of threads, and all the parallelism of the graph was exposed. Then we define

$$T_\infty = \text{depth of the computational DAG}$$

here,

$$T_\infty$$

represents the time it would run through the levels of a DAG, if all nodes in the level was done simultaneously.

So if we have the number of processors required to access all parallelism of an algorithm, we expect it to complete in T_∞ steps. So in our parallel reduce example

$$T_\infty = \log_2(N)$$

where N was the number of nodes.

However, usually we have more data or nodes than threads. This is especially the case in a big data sence. So how many steps will an algorithm with N nodes and p threads take to execute to completion? This is where Brent's Theorem comes in. Here we define

$$T_p = \text{steps an algorithm takes with p threads}$$

Theorem 1 (Brent's Theorem) *We claim that with T_1, T_p, T_∞, then*

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

The term T_1/p is the optimal steps for embarassingly parallel algorithms using p threads. So here, T_p is bounded by the optimal steps and the optimal steps added with the number of levels. So we see here T_∞ is a measure of the necessary sequentialism, or a measure of how parallel an algorithm is or isnt.

For example, we see that for an algorithm based on the map communication pattern will have $T_\infty = 1$. Where as the reduction algorithm has $T_\infty = \log_2(N)$.

So what is the true run time for the parallel reduction when we have N nodes and only p threads? Using Brent's theorem, we have an upper bound on the steps for a parallel reduce.

$$T_p \leq \frac{N}{p} + \log_2(N).$$

1.4 Parallel Summation in CUDA

A serial implementation of reduce sum algorithm is pretty trivial.

```
float reduce(const float *data, int N)
{
        float summ=0.0f;
        for(int i = 0; i < N; i++)
                summ += data[i];
        return summ;
}
```

This will be slow, if say we have a million points (acutally 2^{20}).

Our parallel code will have two stages. The first stage we will launch 1024 block of 1024 threads each. Each block will reduce its 1024 elements, so each block will produce one single item. So at the end of the first stage we will have 1024 elements reduced from the original 2^{10} elements.

Then the second stage will take the 1024 reduced elements and perform a reduction on them to produce one single element.

Lets write a code to do a parallel summation of 2^{20}.

We will implement this in two ways.

1.4.1 First Way

```
__global__ void global_reduce_kernel(float *d_out, float *d_in)
{
        int myId = threadIdx.x + blockDim.x*blockIdx.x;
        int tid = threadIdx.x;

        //do reduction over global memory
        for (unsigned int s = blockDim.x/2; s > 0; s >>= 1)
        {
                if(tid < s)
                {
                        d_in[myId] += d_in[myId + s];
                }
           __syncthreads();   //Maske sure all adds at one stage
        }

        // only thread 0 writes result for this block back to global mem
        if(tid == 0)
        {
                d_out[blockIdx.x] = d_in[myId];
        }
}
```

Each block is responsible for a 1024 partition of floats. In the first iteration of the interior loop, we start with 1024 element, we will have two 512-element regions. Each of the 512 threads will add its element in the 2nd have to its element in the 1st half. Then we synchronize every thread, to remove the possibility of a race condition. Now, we have 512 elements remaining. So we loop again dividing the 512 elements to two 256-element partions using 256 threads. Then sum the 256 items in the second chunk into the first, and synchronize the threads. The loop repeats this strategy for 10 iterations, where each block has 1 element remaining. Then we write this back into global memory.

Notice that this first try of the code isn't as efficient as it could be. Most notably, we are reading from global memory quite frequently. On each iteration of the inner loop, we read n items in from global memory and write out $n/2$ items. Then in the next iteration we read back in those $n/2$ elements, and write $n/4$ new elements, this continues until we reach the reduced element.

Ideally we could do an original read where all 1024 elements of the thread block are read in, all the reduction is done internally, and then a final value is written back. This would be more efficient and faster because fewer memory transactions would occur overall.

1.4.2 Second Way, Shared Memory

The CUDA feature we use to do this is shared memory. We will write and store all intermediate values in a shared memory array where all threads can access them within the thread block.

```
__global__ void shmem_reduce_kernel(float * d_out, const float *d_in)
{
        // sdata is allocated in the kernel call: via dynamic shared memeory
        extern __shared__ float sdata[];

        int myId = threadIdx.x + blockDim.x*blockId.x;
        int tid = threadIdx.x;

        //load shared mem from global mem
        sdata[tid] = d_in[myId];
        __syncthreads(); // always sync before using sdata

        //do reduction over shared memory
        for(int s = blockDim.x/2; s>0; s >>=1)
        {
                if(tid < s)
                {
                        sdata[tid] += sdata[tid + s];
                }
                __synthreads(); //make sure all additions are finished
        }

        //only tid 0 writes out result!
        if(tid == 0)
        {
                d_out[blockIdx.x] = sdata[0];
        }
}
```

We use the binary operation `>>=1` to divide by two, literally `s >>= 1` means *s shifted by one bit to the right.*

The shared memory version differs slightly from the original version. In this case we use the

```
extern __shared__ float sdata[];
```

pragma to say that shared memory is allocated at run time and is used inside the loop. This saves memory transactions from global memory.

To finish our reduction, the second stage is merely calling the first stage's kernel with only one thread block, using stage 1's `d_out` as stage 2's `d_in`.

1.5 Application of Reduce

A clear application of reduce is numerical integration. We will test our summation code by calculating pi using

$$\int_{-1}^{1} \sqrt{1-x^2}dx = \arcsin(1) = \frac{\pi}{2}$$

The composite Trapezoidal Rule is a good way to do numerical integration. Suppose that an interval, $[a, b]$, is split up into an even number, n, of subintervals. From these intervals we use the sample points

$\{x_j\}_{j=0}^n$. Then, the composite Trapezoidal rule is given by

$$\int_a^b f(x)dx \approx \frac{\delta x}{2}\left[\sum_{j=1}^{n}[f(x_{j-1}) + f(x_j)]\right] = \sum_{j=0}^{n-1} f(x_j)\frac{\delta x}{2} + \sum_{j=1}^{n} f(x_j)\frac{\delta x}{2}$$

where $\delta x = \frac{b-a}{n}$.

In fact, with this example, we can illustrate a map-reduce algorithm. Map-reduce, is a combination of the reduction primitive with the map communication pattern. We can have two kernels then. The first being a map kernel, applying the function f onto the data $\{x_j\}_{j=0}^n$. And the second will be a reduction.

1.5.1 Map Kernel

The map parallel communication pattern is very simple as we have demonstrated in Chapter 3. In this case, we will also separate by index. We also declare a `__device__` function.

```
__device__ float myfun(float x)
{
        float result;
        result = sqrtf(1 - x*x);
        return result;
}
```

The `__device__` directive informs the compiler that this function is meant to be called within the kernel, or from another device routine.

Now let us construct the map kernel.

```
__global__ void map(float xbeg, flat dx, int n, float *f1)
{
        int tid = threadIdx.x + blockDim.x*blockIdx.x;
        if(tid < 1 && tid > n)
                return;

        float2 x;
        x.x = xbeg + tid*dx;
        x.y = xbeg + (tid - 1)*dx;
        float ftemp = myfun(x.x)*dx*0.5f + myfun(x.y)*dx*0.5f;
        f1[tid-1] = ftemp;
}
```

Since we are using the Trapezoidal rule the map stage is very simple. If we were using a numerical integration scheme with a higher order of accuracy, like Simpson's Rule, the map kernel would look a bit different.

1.5.2 Host Code

In order to complete our application we must write intellegent host code. Recall that

$$\int_{-1}^{1} \sqrt{1-x^2}dx = \frac{\pi}{2}$$

So the host will have the reduction returned to it, and then multiply the sum by two. To do this, it will be convenient to create a host function. In our case, let $n = 2^{20}$.

```
float mmmmmm_pi(int n)
{

        //Initialization

        float value;
        float *d_data;
        float *d_reduc, *h_reduc;
        size_t original = n*sizeof(float);
        size_t reduc = n/(1024)*sizeof(float);

        //Allocation
        h_reduc = (float*)malloc(reduc);
        cudaMalloc((void**)&d_data, original);
        cudaMalloc((void**)&d_reduc, reduc);
```

```
        //Kernel Parameters
        dim3 block_dim = (1024,1,1);
        dim3 grid_dim = (n/block_dim.x,1,1);

        //integration parameters
        float xbeg = -1.0f;
        float dx = (float)(1.0f - xbeg)/n;
        dim3 map_grid(grid_dim.x+1);
        //map+stencil kernel Note because of the shift we need more threads.
        map<<<map_grid, block_dim>>>(xbeg, dx, n, d_data);
        cudaDeviceSynchronize(); // Need map to be applied to all data before reduction!

        size_t size = block_dim.x*sizeof(float);

        shmem_reduce_kernel<<<grid_dim, block_dim,size>>>(d_reduc, d_data);
        //Recall that this makes a reduced array of size grid_dim/block_dim.
        cudaDeviceSynchronize();
        //Second Stage of First sum!
        limited_add_reduce<<<1,block_dim, size>>>(d_reduc, d_reduc);
        cudaMemcpy(h_reduc, d_reduc, reduc, cudaMemcpyDeviceToHost);
        value = h_reduc[0];

        //Recall that value now = pi/2
        value *= 2.0f;
        //Free memory
        free(h_reduc);
        cudaFree(d_reduc);
        cudaFree(d_data);
        return value;
}
```

Notice that we could do this in two different ways. By the composite Trapezoidal rule we have

$$\int_b^a f(x)dx \approx \frac{\delta x}{2}\sum_{j=1}^{n}[f(x_{j-1})+f(x_j)] = \sum_{j=0}^{n-1} f(x_j)\frac{\delta x}{2} + \sum_{j=1}^{n} f(x_j)\frac{\delta x}{2}$$

If your problem is data limited, meaning far greater data elements than processors. It will combine a map-stencil kernel as to only perform one sum reduction as is done in this example. However, if your problem has more processors than data elements, we can separate the data into two arrays to be reduced simultaneously. Since kernel calls are asynchronous, we can compute both sums nearly at the same time.

Then lastly the main program will be:

```
#include <stdlib.h>
#include <stdio.h>
#include <cmath>

int main()
{
        int n = pow(2,20);
        float pi = mmmmmm_pi(n);
        std::cout<<" Pi = "<< pi <<std::endl;
}
```

2 Scan

Scan is one of the most important parallel primitives. Scan is generalization of reduce to yield an array, instead of a single value. Any binary mathematical operation can be used to create a scan algorithm. Notable examples are cummulative distribution calculations, or a sorting algorithm.

A short example:

$$\text{Input: } \{1,2,3,4\}$$

$$\text{Operation: } +$$

$$\text{Output: } \{1,3,6,10\}$$

This is called a **Prefix-Sum** primitive, and in this case, each element is the sum of input element along with the previous input elements. At first glance, this algorithm seems difficult to parallelize, because each element in the output depends on the previous element.

Let's discuss a mathematical representation of scan. Like reduce, scan takes an array of elements and a binary operation, but instead of producing one element, it produces another array.

$$\mathcal{S}(\mathbf{x}, \oplus) = \mathbf{y}$$

In this case the binary operation must be associative, but there also needs to be an indentity element. That is,

$$a \oplus e = a$$

For addition, $e = 0$.

Specifically what does scan do? Given an array **a**, an operator $\oplus$, and an identity I:

$$\left[a_0, a_1, a_2, \cdots, a_{n-1}\right] \text{ :input}$$

$$\left[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \cdots, \bigoplus_{j=0}^{n-1} a_j\right] \text{ :output}$$

Note that this is an *inclusive scan* primitive. A good example defininition *exclusive scan* primitive

$$\left[a_0, a_1, a_2, \cdots, a_{n-1}\right] \text{ :input}$$

$$\left[I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \cdots, \bigoplus_{j=0}^{n-2} a_j\right] \text{ :output}$$

2.1 Implementation of Scan

Looking into serial implementations generally give us insights into algorithms. Usually we are interested in how many operations must be done, and how many steps will that take.

2.1.1 Serial

```
int acc = identitiy;
for (int i = 0; i < elements.length(); i++)
        {
                acc = acc op element[i];
                out[i] = acc;
        }
```

For either inclusive and exclusive the results are the same. For each n iterations of the loop, we take 1 step to do 1 operation. Thus, serial scan takes n steps to complete n operations.

2.1.2 Parallelization

Parallelization of the scan primitive is more complicated than with reduce. We can think of each element as the reduction of the previous elements, and then loop over each element, and compute a parallel reduction at each step. We find that this implementation may not be the most efficient. There are two popular implementations of scan: the Hillis and Steele algorithm, and the Blelloch scan. Each has their own strengths and weaknesses. In short, the Hillis and Steele version is more step-efficient, that is it is more parallel. However, the Blelloch scan is more work-efficient. Both are still relevent today in parallel computing.

Hillis and Steele Scan

This implementation is for an inclusive scan, that was popularized by Danny Hillis and Guy Steele in 1986. Danny Hillis founded a company that built a massively parallel computer called the *Thinking Machine*, which had scan as one of its core primitives. It was designed mostly for AI applications, with the goal of creating a computer that was proud of him. At his company, Thinking Machines, he worked with Guy Steele,

who developed the scheme programming language at MIT, and was later one of the core developers of Java. Together they designed this implementation of the scan algorithm.

A simple was to explain their algorithm is to examine a graph of the computation. We'll start with an example, suppose that we have an input array that's the integers 1 to 8.

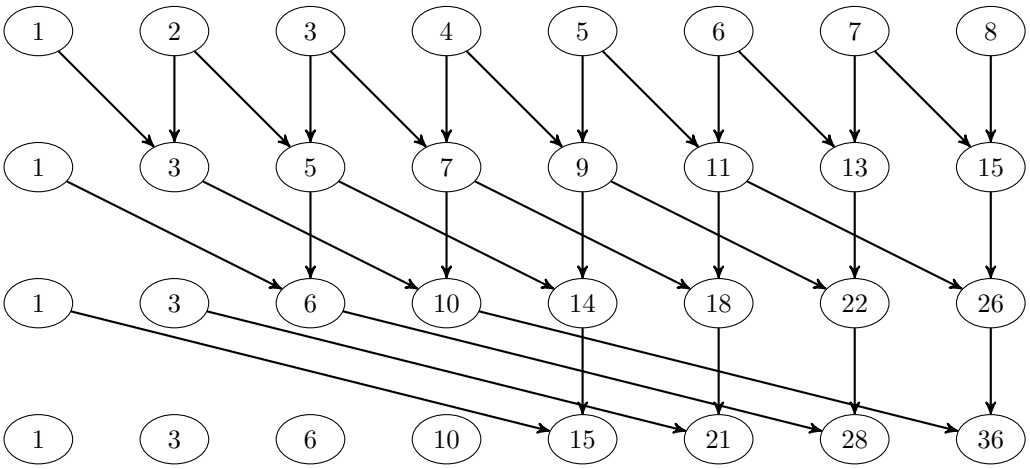


Figure 5: Example Hillis and Steele Scan

The first stage reduces pairs, that is elements with one to the left. The second stage operates on the elements with another 2 to the left. Lastly, the third stage adds element that have a neighbor 4 to the left. So if we number our stages $i = 0, 1, 2$ then each stage operates on elements with neighbors 2^i to the left. We have a total of three stages, which is $\log_2(8)$, in general, this will give us the number of stages in a Hillis and Steele scan. Elements that do not have the prescribed neighbor to the left will just be copied into the next stage. So at completion we have a running sum of the 8 integers.

So let's analyze the complexity of this algorithm. As a function of n, the size of array what is the complexity? In this case we have two different types of complexity, work and steps. Steps as we mentioned before will be $\log_2(n)$, whereas the work complexity will be $n \log_2(n)$.

Blelloch Scan

The Blelloch scan is named after Guy Blelloch, in 1990. This scan prioritizes work-efficiency, but is a bit more complex in terms of the algorithm. Lets look at the another example, using the integers 1 through 8, only this time doing an exclusive addition. This algorithm has two stages, first a reduction, then a downsweep. The first stage will look similar to what we've done before, but the second stage will be new, and will take a different operator than we've seen before. Lets examine the graph for this example.

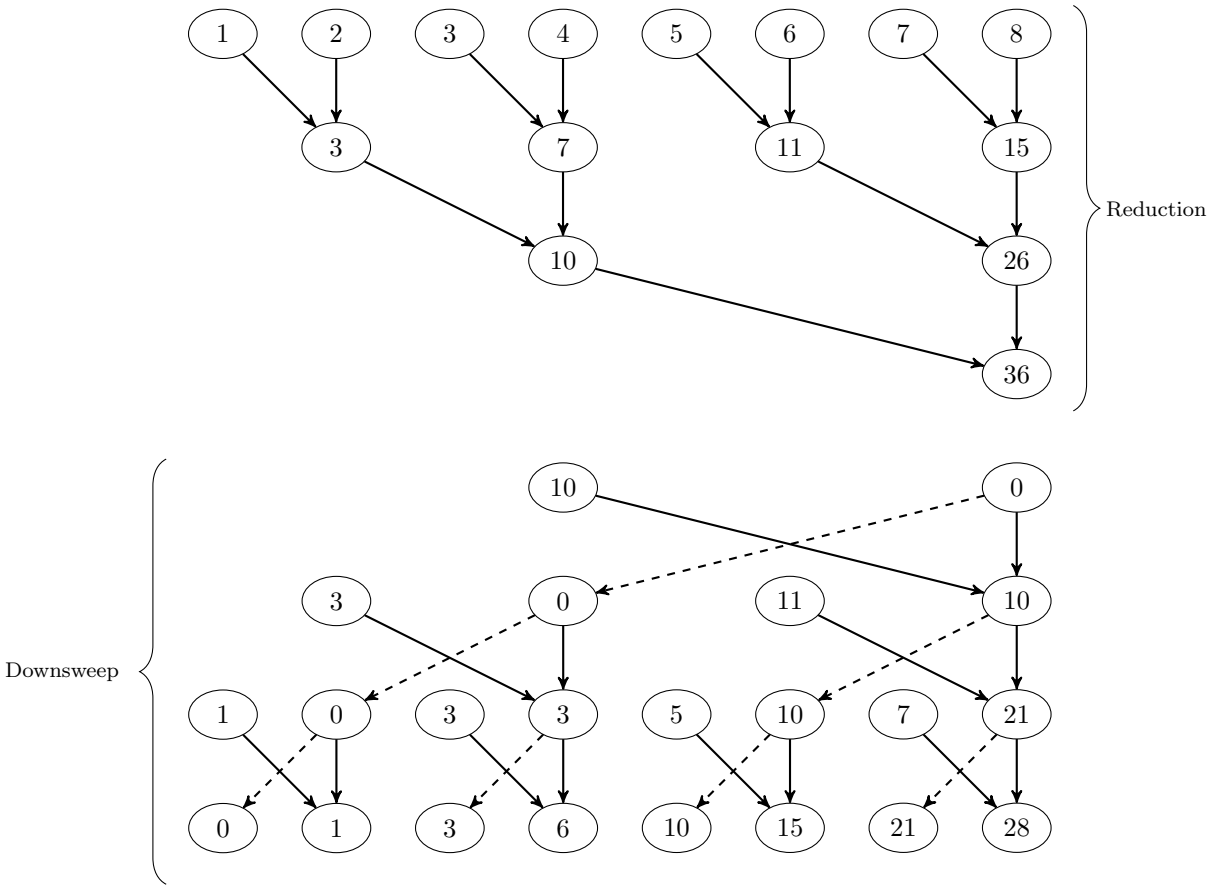
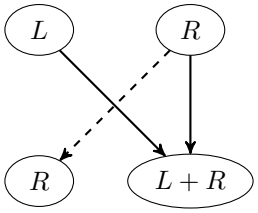


Figure 6: Example Blelloch Scan

The downsweep sequence takes two elements, applies the binary operation to replace the left element, then copys the left element into the right element's space.



This operation is continued on all levels, increasing the number of elements in each stage by 2. Elements untouched by the reduction stage are filled into the downsweep stage. That is for the first step of the downsweep, the identity element and 10 are inserted. On the next step, 11 and 3 are inserted, and so on. At the end of the last step in the downsweep stage, we have the exclusive prefix sum of the 8 integers, where each element in the output array is the sum of the previous elements in the input array.

The Blelloch Scan algorithm is more challenging in terms of steps. We know the complexity of the first stage, it is $\mathcal{O}(\log(n))$ steps for reduction, with $\mathcal{O}(n)$ operations.

Notice that the communication pattern in the downsweep phase is the mirror of the reduce phase. So the complexity analysis will be the same. There will be $\mathcal{O}(\log(n))$ steps and the work will be $\mathcal{O}(n)$. Notice that we regain the work efficiency as in the serial algorithm, but recall that the Hillis and Steele formulation had $\log(n)$ steps. So the Blolloch formulation has $2\log(n)$ steps, double the steps. The advantage of the Blelloch is that there is less work over all.

Which One To Use?

Either formulation can be useful depending on the context. The size of your dataset, the power of your GPU and the optimizations you choose within your program will tell you which implementation is more suited to your problem. Suppose your data size far exceeds the ammount of processors at your disposal. Then you have more work than processors. In this case your implementation is limited by the number of processors you have, because these processors will be busy during the whole run. So as the developer, you would prefer an implementation that is more work efficient, the Blelloch Scan.

On the contrary, suppose you have a smaller data set, and your computing power exceeds the work. In this case, you are limited by the number of steps in your algorithm, so you prefer an algorithm that induces more parallelism, with fewer steps. That is, the Hillis and Steele scan is more worth your time.

2.2 Application of Scan

In general, scan is an primitive that can tackle some recurrence relations. In this example, we'll compute the cummulative distribution function (CDF) of a normal distribution. Recall that the normal distrubution takes the form

$$f(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

And the CDF is

$$\Phi(x|\mu,\sigma^2) = \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right]$$

note that we cannot represent the error function using elementary functions are generally must be computed numerically. Note that the Normal distribution spans all of the real numbers, but is rapidly decreasing, that is as $x \to \pm\infty$ $f(x) \to 0$. This alleviates our problem that computers cannot understand infinity. Thus we can sufficiently estimate the CDF by integrating from 5 standard deviations from the mean. That is

$$F(x|\mu,\sigma^2) \approx \int_{\mu-5\sigma}^{x} f(t)dt$$

Recall that using the Trapezoidal rule we can split our integral to

$$\int_a^b f(x)dx = \sum_{j=0}^{n} \int_{x_j}^{x_{j+1}} f(x)dx$$

This time we will generate the input data for the scan algorithm to be

$$y_j = \big(f(x_j) + f(x_{j+1})\big)\frac{\delta x}{2}$$

where j runs from 0 to $n-1$.

2.2.1 Stencil + Map Kernel

Before we get to the scan algorithm, let us discuss our data generation. This time we will apply our knowledge of stencil as well as map.

```
__global__ void data_gen(float *data, float *x, float xbeg, float dx, float mu, float sig, int n)
{
        //generate x and data
        //use x for plotting later
        tid = threadIdx.x + blockDim.x*blockIdx.x;
        if(tid<1 && tid > n)
        {
                return;
        }
        float xtemp[2];
        xtemp[0] = xbeg + tid*dx;
        xtemp[1] = xbeg + (tid-1)*dx;
        data[tid-1] = (normal(xtemp[0], mu, sig) + normal(xtemp[1], mu, sig))*dx*0.5;
        x[tid-1] = xtemp[0];
}
```

Here we use the variable `xtemp` to construct our stencil. We're also using the `__device__` function `normal` defined as:

```
__device__ float normal(float x, float mu, float sig)
{
        float temp1 = 1.0/(sqrt(2*PI)*sig);
        float temp2 = (x-mu)*(x-mu)/(2*sig*sig);
        float val = temp1*exp(-temp2);
        return val;
}
```

This routine will generate the data needed for our scan algorithm. Notice that in the Trapezoidal rule we have an odd $n+1$ data points. This routine creates an even lengthed data set ready for our scan kernel.

2.2.2 Blelloch Scan Kernel

For this application lets code up a single threadblock's Blelloch scan.

```
__global__ void bl_scan(float *odata, const float *idata, int n)
{
         extern __shared__ float temp[];// allocated within kernel config
         int thid = threadIdx.x;
         int offset = 1;
         temp[2*thid] = idata[2*thid]; // load input into shared memory
         temp[2*thid+1] = idata[2*thid+1];
         for (int d = n>>1; d > 0; d >>= 1) // Do reduction!
         {
                  __syncthreads();
                  if (thid < d)
                  {
                           int ai = offset*(2*thid+1)-1;
                           int bi = offset*(2*thid+2)-1;
                           temp[bi] += temp[ai];
                  }
                  offset *= 2;
         }
         if (thid == 0)
         { temp[n - 1] = 0; } // clear the last element
         for (int d = 1; d < n; d *= 2) // Do downsweep operation
         {
                offset >>= 1;
                __syncthreads();
                if (thid < d)
                {
                           int ai = offset*(2*thid+1)-1;
                           int bi = offset*(2*thid+2)-1;
                           float t = temp[ai];
                           temp[ai] = temp[bi];
                           temp[bi] += t;
                  }
         }
         __syncthreads();
         odata[2*thid] = temp[2*thid]; // write results to device memory
         odata[2*thid+1] = temp[2*thid+1];
}
```

Note that this implementation only works for one thread block. This is an exclusive scan, however, an inclusive scan can be generated from an inclusive scan by shifting the resuling array left, and instering the last element of the scan summed with the last element of the input array at the end. Likewise, an exclusive scan can be generated from an inclusive scan by shift the resulting array to the right by one element and inserting the identity of the binary operator at element 0.

Since calculating a CDF is an inclusive scan, we need to shift the elements and add the last element of the scan with the last element of the input array.

```
__global__ void ex2in(float *scan, float *idata, int n)
{
        extern __shared__ float temp[]; //allocated via kernel config
        thid = threadIdx.x;
        if(thid >= n)
                return;
        temp[thid] = scan[thid]; //load scan data;
        __syncthreads();

        if(thid > 0)
                scan[thid-1] = temp[thid];
```

```
        if(thid == n-1)
                scan[thid] = temp[thid] + idata[thid]; //last element clean up!

}
```

2.2.3 Host Code

For this example, we'll use the standard normal distribution $\mathcal{N}(\mu = 0, \sigma^2 = 1)$, with 1024 data points.

```
void compute_norm_cdf(float *cdf, float *h_pdf, float *x, float mu, float sigma, const int num_data)
{
        //data and parameter set up!
        float *pdf, *d_cdf, *d_x;
        float xbeg = mu - 5*sigma; //Five standard deviations from the mean!
        float xend = mu + 5*sigma;
        float dx = (xend - xbeg)/(float)num_data;

        //data allocation!
        size_t size = num_data*sizeof(float);
        cudaMalloc((void**)&pdf, size);
        cudaMalloc((void**)&d_x, size);
        cudaMalloc((void**)&d_cdf, size);

                //data generation kernel!
        data_gen<<<2, num_data/2+1>>>(pdf, d_x, xbeg, dx, mu, sigma, num_data);
        cudaDeviceSynchronize();

        //Perform scan for cdf calculation!
        bl_scan<<<1, num_data/2, 2*size>>>(d_cdf, pdf, num_data);
        cudaDeviceSynchronize();

        //Shift Correct for CDF
        ex2in<<<1,num_data, size>>>(d_cdf, pdf, num_data);
        cudaDeviceSynchronize();
        //Transfer to Host!
        cudaMemcpy(cdf, d_cdf, size, cudaMemcpyDeviceToHost);
        cudaMemcpy(x, d_x, size, cudaMemcpyDeviceToHost);
        cudaMemcpy(h_pdf, pdf, size, cudaMemcpyDeviceToHost);

        //Free memory!
        cudaFree(pdf);
        cudaFree(cdf);
        cudaFree(d_x);
}
```

Then the main function will call `compute_norm_cdf`.

```
#include <iostream>
#include <fstream>
#include <cmath>

#define PI 3.1415926

//Function declarations

int main()
{
        int num_data = 1024;
        size_t size = num_data*sizeof(float);

        //Memory allocation
        float *cdf, *x;
        cdf = (float*)malloc(size);
        x = (float*)malloc(size);

        //call function
        compute_norm_cdf(cdf, x, 0.0f, 1.0f, num_data);

        //create file object to output data
        std::ofstream myfile_tsN;
        myfile_tsN.open("cdf.dat");
        for(int aa = 0; aa<num_data; aa++){
                t = t1 + aa*h;
                myfile_tsN << x[aa] << cdf[aa] << '\t';
                }
        myfile_tsN << std::endl;

        //destroy file object
        myfile_tsN.close();
```

```
        //free memory!
        free(x);
        free(cdf);
}
```

Note that this implementation has two flaws. The first is that is only designed for a singular thread block. The second is that it sufferes from **shared memory bank conflicts**. These hurt the performance of every access to shared memory, and can affect overall performance. We will address the first issue now, and the second issue in a future chapter on optimization.

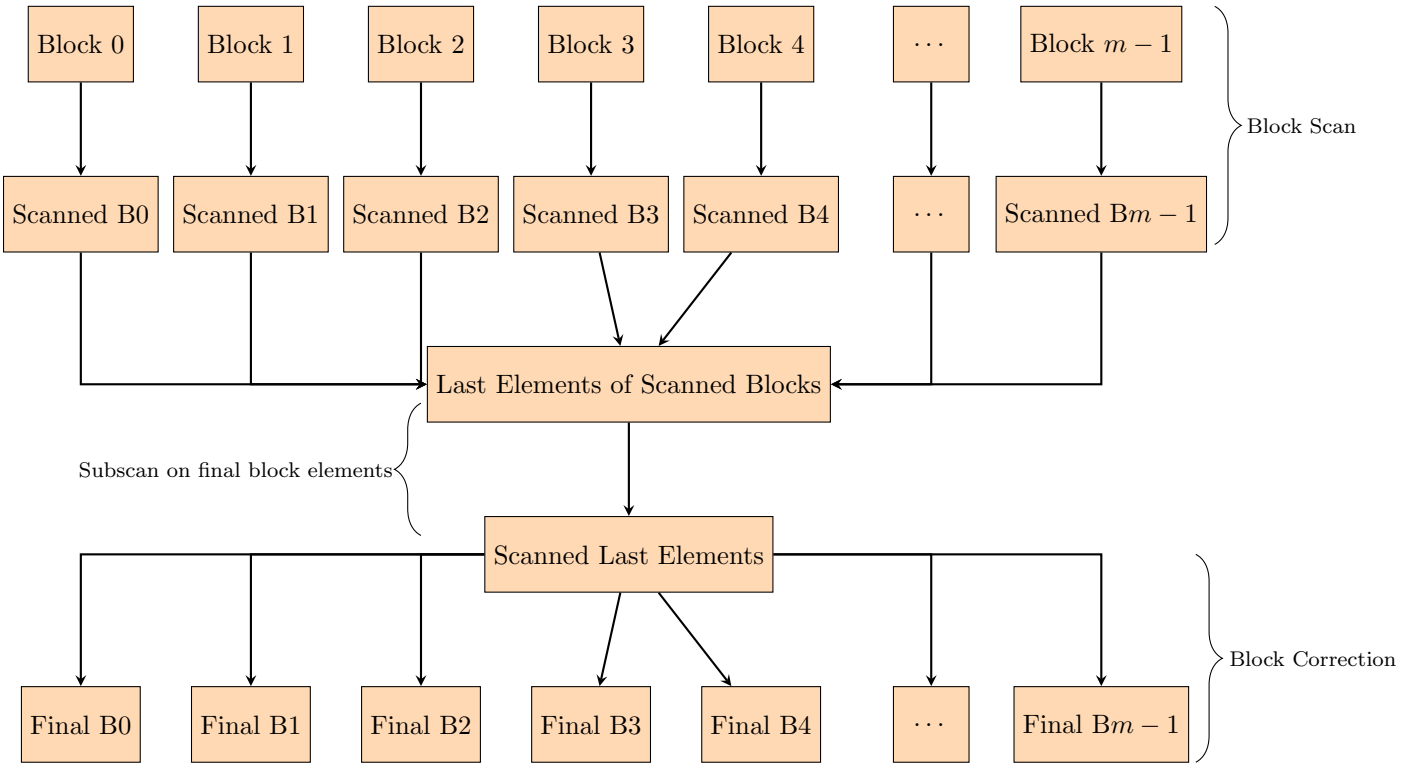
2.3 Generalizing Scan Algorithms to Larger Arrays

When working with arrays that are larger than the possible thread block size, we need to adopt a new strategy. Suppose that we scan an array that is m times larger than our thread block size. Then a solution to this problem is the following:

1. Perform scan on all m thread blocks
2. Create a second array to contain the last element from each scanned thread block
3. Perform a scan on this array
4. Use newly updated second array to correct original array by thread block

This process can be thought of better in terms of a graph. For a sum scan, we do not need to update the values in thread block 0. However, for a max/min scan, we will need to do this.

Figure 7: Large Array Scan



Using this plan we can create a more generalized scan algorithm for GPU. Note that in each scan step, there's also a exclusive to inclusive correction. The implementation is left to the reader.