Multi-core Programming
Assignment 5

Ali Gholami

Department of Computer Engineering & Information Technology
Amirkabir University of Technology

https://aligholamee.github.io
aligholami7596@gmail.com

Abstract

In this report, we'll analyze the *prefix-sum* (*scan*) problem. There are many uses for scan, including, but not limited to, sorting, lexical analysis, string comparison, polynomial evaluation, stream compaction, and building histograms and data structures (graphs, trees, and so on) in parallel. There are also multiple solutions and algorithms to compute the prefix sum of an array. Sources for this report are provided in the *src* folder.

Keywords.

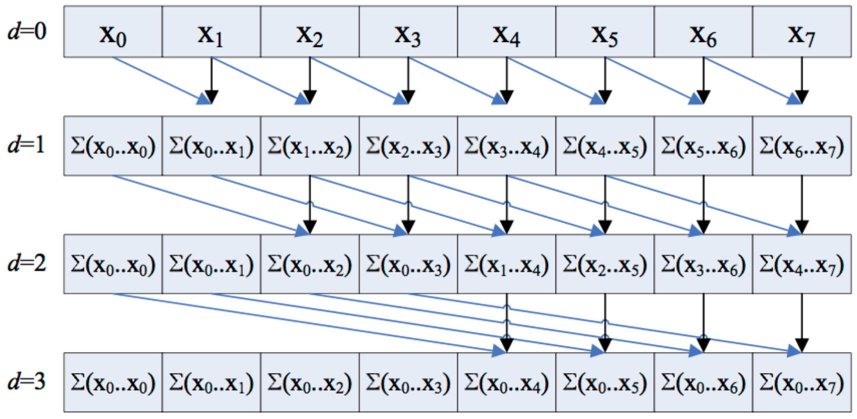
1 Hillis & Steele Algorithm

1.1 How it works?

Recall that in each step, we had to add up the elements of the array until now and replace the current element with the addition result. The sequential implementation takes exactly n operations (*n is the size of the array*) to complete. The *Hillis & Steele* algorithm, provides a simple and intuitive parallelization trick using *addition* as a *binary operator* which takes only two arguments. The idea is parallel in theory but it has some problems in practice. Figure 1.1 demonstrates this algorithm better.

1.2 Assumptions

This algorithm assumes that there are as many as processors as data elements. The programmer must divide the computation among a number of thread blocks that each scans a portion of the array on a single multiprocessor of the GPU. Even still, the number of processors in a multiprocessor is typically much smaller than the number of threads per block, so the hardware automatically partitions the *for all* statement into small parallel batches (called warps) that are executed sequentially on the multiprocessor. Because not all threads run simultaneously for arrays larger than the warp size, this algorithm will not work, because it performs the scan in place on the array. The results of one warp will be overwritten by threads in another warp.



d=0
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7
d=1
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_1..x_2)$ $\Sigma(x_2..x_3)$ $\Sigma(x_3..x_4)$ $\Sigma(x_4..x_5)$ $\Sigma(x_5..x_6)$ $\Sigma(x_6..x_7)$
d=2
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_1..x_4)$ $\Sigma(x_2..x_5)$ $\Sigma(x_3..x_6)$ $\Sigma(x_4..x_7)$
d=3
$\Sigma(x_0..x_0)$ $\Sigma(x_0..x_1)$ $\Sigma(x_0..x_2)$ $\Sigma(x_0..x_3)$ $\Sigma(x_0..x_4)$ $\Sigma(x_0..x_5)$ $\Sigma(x_0..x_6)$ $\Sigma(x_0..x_7)$

Figure 1.1: Demonstration of *Hillis & Steele* Algorithm.

1.3 Work Efficiency

Even if this algorithm works in some cases, it is not *work efficient*. The minimum number of addition operations needed in the sequential algorithm was n. In this algorithm, this number increases to $n \log n$. That's the main reason we call the *Hillis & Steele* algorithm a *work inefficient* algorithm.

1.4 Pseudocode

```
1: for d = 1 to log2 n do
2:     for all k in parallel do
3:          if k >= power(2, d)  then
4:               x[k] = x[k - power(2, d-1)] + x[k]
```

1.5 CUDA Kernel

```
__global__ void
prefixSumCUDA(int *a, size_t n)
{

        int tId = threadIdx.x;

        for (int offset = 1; offset < n; offset *= 2) {
                if (tId >= pow((float)2, offset)) {
                        int temp = tId - pow((float)2, offset - 1);
                        a[tId] += a[temp];
                }
        }
}
```

This implementation can handle arrays not bigger than a *warp* size.

1.6 Grid & Block Size Analysis

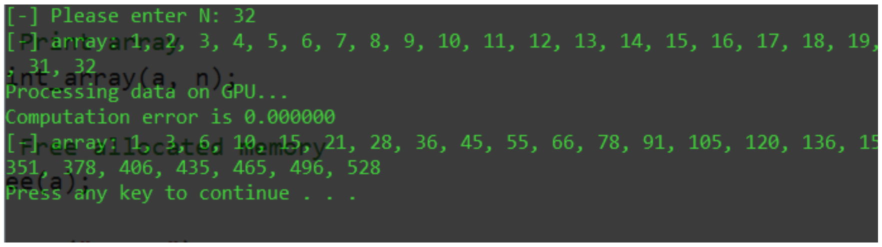
For the first implementation, I've chose the grid size to be 1 in a one dimensional manner. I've also selected the block size as 32 (same as warp size) in the first dimension. Grid size is dedicated to our estimation of number of blocks needed to process the data. The block size should be selected so that computation mean squared error is 0. In this case, the proper size for the blocks is 32 (same as warp size). The reason is there are multiple add operations needed to be written back to the array *a* whenever each thread does its job. The problem does not appear until the number of threads is greater than the warp size(32). Each warp overwrites the values previous warp was written. That yields incorrect results.

1.7 Mean Squared Error Analysis

All of the results are evaluated using *Mean Squared Error* metric. The code is given below:

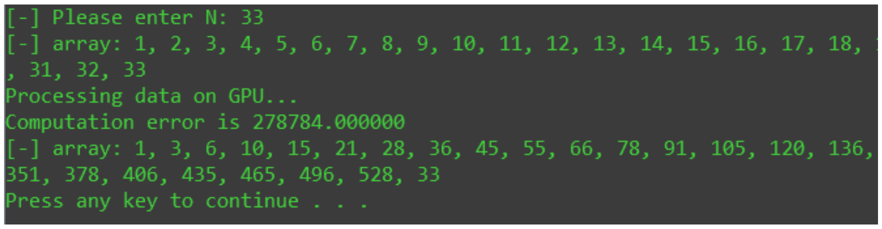
```c
float compute_mse(int *a, int *b, int n) {
        float err;

        for (int i = 0; i < n; i++) {
                err += pow(a[i] - b[i], 2);
        }

        return err;
}
```

The mean squared error for arrays not bigger than the warp size (which is 32 in my case), is 0. For warp sizes greater than 32 the error increases. The result is absolutely stunning! As you can see in the figure 1.2, the mean square error for the input of sizes less than 32 is computed without errors. That's mainly because of the issue with the maximum number of threads running together. Figure 1.3 illustrates this issue as the size of the input array goes higher than 32.



```
[-] Please enter N: 32
[-] array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
, 31, 32
Processing data on GPU...
Computation error is 0.000000
[-] array: 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 15
351, 378, 406, 435, 465, 496, 528
Press any key to continue . . .
```

Figure 1.2: Computed MSE for Arrays with Size Less than 32.



```
[-] Please enter N: 33
[-] array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
, 31, 32, 33
Processing data on GPU...
Computation error is 278784.000000
[-] array: 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136,
351, 378, 406, 435, 465, 496, 528, 33
Press any key to continue . . .
```

Figure 1.3: Computed MSE for Arrays with Size Greater than 32.

1.8 Performance Analysis

Now, let's look at how this implementation performs against the sequential implementation. Table 1.1 illustrates this phenomenon.

Size	5 MB	50 MB	80 MB	
Error	Very High	Very High	Very High	
(grid, block)	(n/1024, 1024)	(n/1024, 1024)	(n/1024, 1024)	
Time(ms)	1.285472	1.431008	1.461440	

Table 1.1: Results of Prefix Sum Sequential Implementation of ***integer*** type.

As can be seen from this table, there is a lot of computation error with the raw algorithm of *Hillis & Steele*. The improvement of the algorithm in next section will solve this problem.

1.9 A Deeper Look at Blocks and Grids

Here we'll suppose the array size to be 5 MB only. In this case, there are 1250000 integer elements in the array. Since each multiprocessor can accept up to 1024 threads, we can set the number of blocks to 1221 (Ceil of 1220.70) and the number of threads to 1024 in each block. This solution will not work since the warp size is 32 and there is dependency in different steps of the algorithm. The array will be overwritten and the result is incorrect.

1.10 Update 1; Minor Improvements

We can simply scale this implementation by increasing the number of threads per block. We can compute the proper dimensions of **grid** and **block** as following:

```
// Kernel launch
dim3 gridDimensions(ceil((float)n / BLOCK_SIZE), 1, 1);
dim3 blockDimensions(BLOCK_SIZE, 1, 1);

prefixSumCUDA << < gridDimensions, blockDimensions >> > (d_A, n);
```

This implementation handles input sizes up to *414* elements. Note that *BlockSize* is held 1024 in this case.

2 Map Reduce Algorithm

We would like to find an algorithm that would approach the efficiency of the sequential algorithm, while still taking advantage of the parallelism in the GPU.

2.1 Main Idea

To do this we will use an algorithmic pattern that arises often in parallel computing: balanced trees. The idea is to build a balanced binary tree on the input data and sweep it to and from the root to compute the prefix sum. A binary tree with n leaves has $d = \log_2 n$ levels, and each level d has 2^d nodes. If we perform one add per node, then we will perform O(n) (work efficient) adds on a single traversal of the tree. Note that *map* phase is the phase that we break the array into pieces to compute the partial sums on shared memory and the *reduce* phase is the phase that we add up all the results to get the final results.

2.1.1 Up Sweep Phase

The demonstration of this phase is illustrated in figure 2.1. Now that partial sums are computed, we

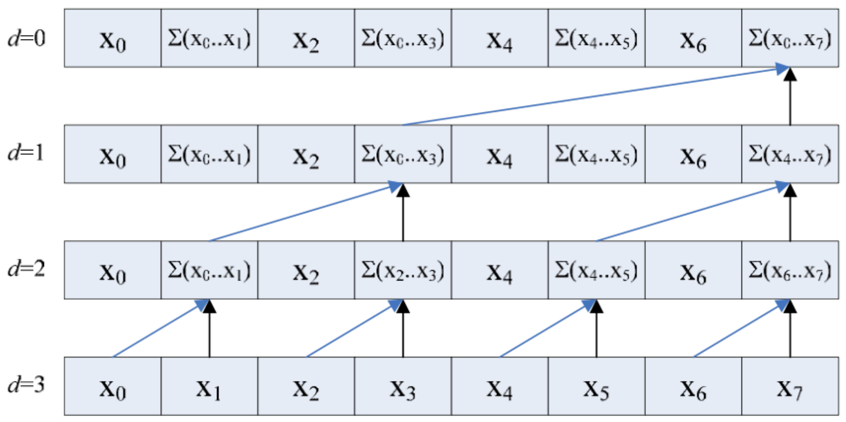


Figure 2.1: Demonstration of *Up Sweep* Phase.

can perform a *down sweep* which yields in the prefix sum result.

2.1.2 Down Sweep Phase

In this phase, each node passes down its value to its left child, and the sum of its value and former value of its left child, to its right child.