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. *Heterogeneous Programming, CUDA, Scan, Brent Kung Scan, Hillis & Steele Scan.*

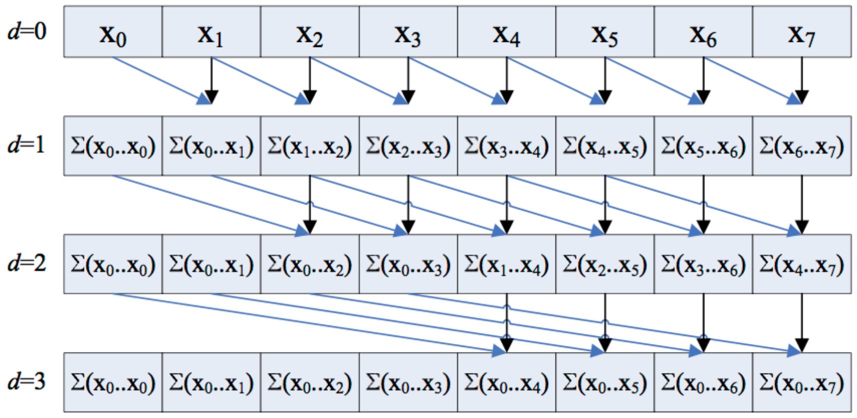
1 Hillis & Steele Algorithm

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

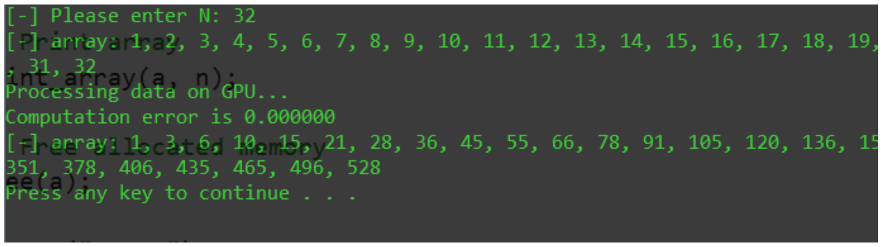
For the first implementation, I've chose the grid size to be 1 in a one dimensional manner. I've also selected the block size as 32 (same as warp size) in the first dimension. Grid size is dedicated to our estimation of number of blocks needed to process the data. The block size should be selected so 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:

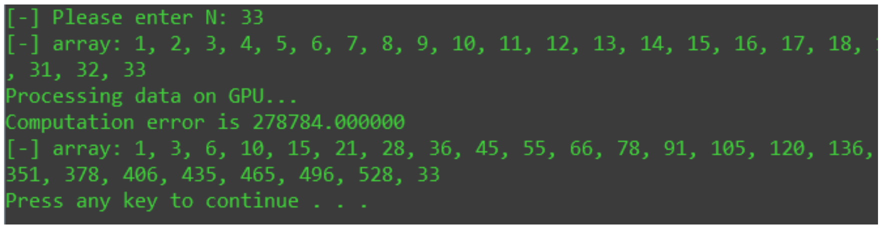
```
	float compute_mse(int *a, int *b, int n) {
		float err;

		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

In this implementation which is under *src/normal-implementation*, we have divided the input array into sub arrays. Thus, we have some blocks on the input array which in each block, there are a big number of threads scanning sub arrays. Each of the threads would do multiple additions. Final element of each sub array is finally loaded into another array which holds the last elements of all sub arrays. Then the scan procedure is performed on this array one more time. The results are added back to the elements of the main array respectively. Note that this implementation handles up to 15000 input sizes which is a massive improvement compared to the initial implementation.

```
__global__ void
prefixSumMap(int *a, int *subArraySumArray, size_t n, size_t k)
{
        int tId = blockIdx.x * blockDim.x + threadIdx.x;

        int startPoint = tId * NUM_OF_ADDS_PER_THREAD;
        int endPoint = startPoint + NUM_OF_ADDS_PER_THREAD;

        // Scan sub arrays
        for (int i = startPoint + 1; i < endPoint; i++) {

                if (i >= n)
                break;

                a[i] += a[i - 1];
        }

        __syncthreads();
        subArraySumArray[tId] = a[endPoint - 1];

        if (tId == 0) {
                for (int i = 1; i < k; i++) {
                        subArraySumArray[i] += subArraySumArray[i - 1];
                }
        }

        __syncthreads();
        printf("");

        int newStartPoint = startPoint + NUM_OF_ADDS_PER_THREAD;
        int newEndPoint = endPoint + NUM_OF_ADDS_PER_THREAD;

        for (int i = newStartPoint; i < newEndPoint; i++) {
                if (i >= n)
                break;

                a[i] += subArraySumArray[tId];
        }
}
```

The greatest bottleneck is when the thread 0 tries to compute the prefix sum of the intermediate array. Also, since *syncthreads* is used to create a synchronization mechanism on the threads of each block, to control other blocks we can buy some time by conducting an I/O instruction. Note that we haven't taken care of the timings in this implementation.

3 Brent Kung Scan with Arbitrary Size Support

This implementation supports input arrays with arbitrary size. It includes 3 main steps(kernels):

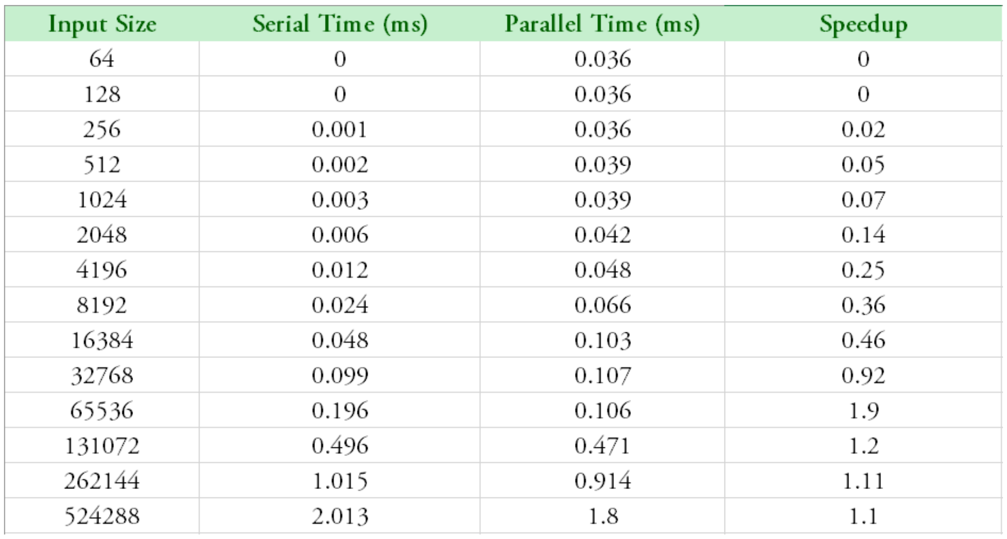
- Scan sub arrays of the input array with normal approach
- Scan the intermediate array with Brent Kung approach
- Reduce the result back to the main array

3.0.1 Source Code

Various implementations are attached in the *src* folder. This implementation is under *final-implementation* subdirectory.

3.0.2 Speedup Analysis

Figure 3.1 illustrates the details of timings of parallel and serial implementations.

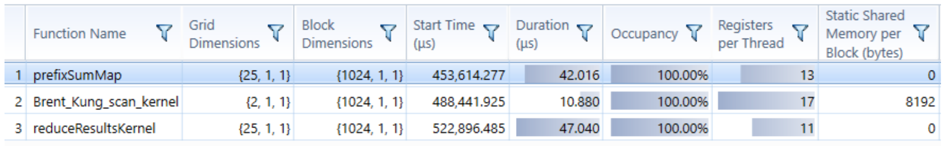


Input Size	Serial Time (ms)	Parallel Time (ms)	Speedup
64	0	0.036	0
128	0	0.036	0
256	0.001	0.036	0.02
512	0.002	0.039	0.05
1024	0.003	0.039	0.07
2048	0.006	0.042	0.14
4196	0.012	0.048	0.25
8192	0.024	0.066	0.36
16384	0.048	0.103	0.46
32768	0.099	0.107	0.92
65536	0.196	0.106	1.9
131072	0.496	0.471	1.2
262144	1.015	0.914	1.11
524288	2.013	1.8	1.1

Figure 3.1: Timings for Serial and Parallel Implementations(GPU).

3.0.3 Grid Size & Block Size Analysis

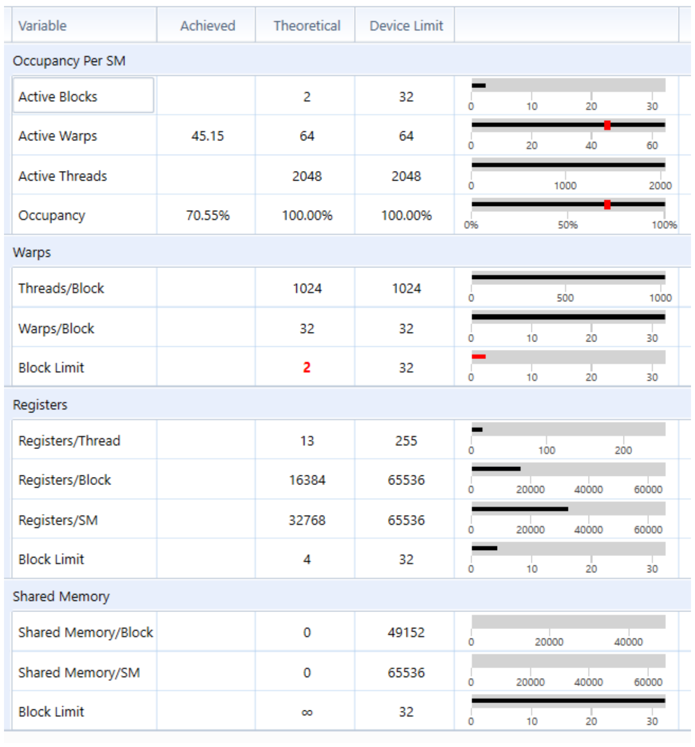
Maximum number of blocks and their sizes is heavily dedicated to the GPU model. The important thing about this implementation is that, whenever a single thread does less computations (smaller sub array size), the performance gets better. For the input of size n, each thread scans a sub array with size *NUM_OF_ADDS_PER_THREAD*. The block size should be a coefficient of the job size each thread is working on. Maximum block size is 1024 for compute capability of 5 in x direction. Thus, the proper grid size (which is the number of blocks) is *ceil((float) n / BLOCK_SIZE)*. For larger input sizes, the value of *NUM_OF_ADDS_PER_THREAD* should be increased for the computation to be error prone. Figure 3.3 illustrates the total occupancy. It also demonstrates that the number of



	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)
1	prefixSumMap	{25, 1, 1}	{1024, 1, 1}	453,614.277	42.016	100.00%	13	0
2	Brent_Kung_scan_kernel	{2, 1, 1}	{1024, 1, 1}	488,441.925	10.880	100.00%	17	8192
3	reduceResultsKernel	{25, 1, 1}	{1024, 1, 1}	522,896.485	47.040	100.00%	11	0

Figure 3.2: Summary of Grid and Block Sizes of 3 Kernels.

blocks, which is 2 in this case, is the cause of occupancy limit.



Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		2	32
Active Warps	45.15	64	64
Active Threads		2048	2048
Occupancy	70.55%	100.00%	100.00%
Warps			
Threads/Block		1024	1024
Warps/Block		32	32
Block Limit		2	32
Registers			
Registers/Thread		13	255
Registers/Block		16384	65536
Registers/SM		32768	65536
Block Limit		4	32
Shared Memory			
Shared Memory/Block		0	49152
Shared Memory/SM		0	65536
Block Limit		∞	32

Figure 3.3: Experimental Results of Occupancy of *PrefixSumMap* Kernel.

Figure 3.4 shows how much time is wasted for global memory accessing. It clearly depicts the issue stall reasons for the *PrefixSumMap* kernel.

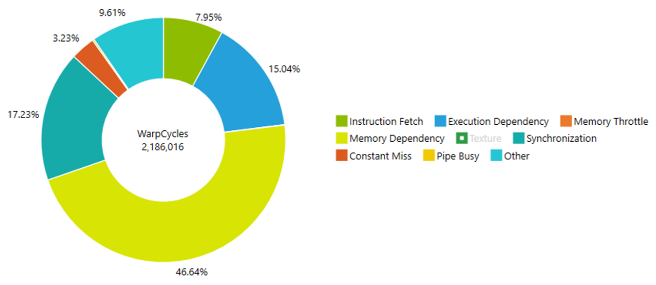


Figure 3.4: Issue Stall Reasons of *PrefixSumMap* Kernel.

3.0.4 SM Activity Analysis

Figure 3.5 shows the percentage of time each multiprocessor was active during the duration of the kernel launch. A multiprocessor is considered to be active if at least one warp is currently assigned for execution. An SM can be inactive - even though the kernel grid is not yet completed - due to high workload imbalances. Such uneven balancing between the SMs can be caused by a few factors:

Different execution times for the kernel blocks, variations between the number of scheduled blocks per SM, or a combination of the two.

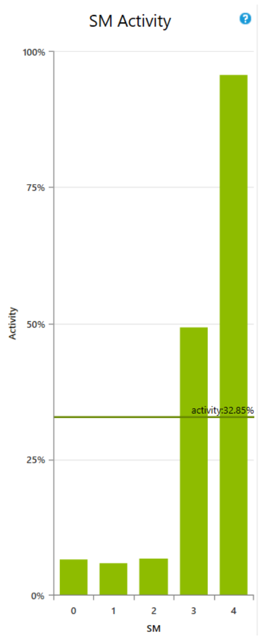
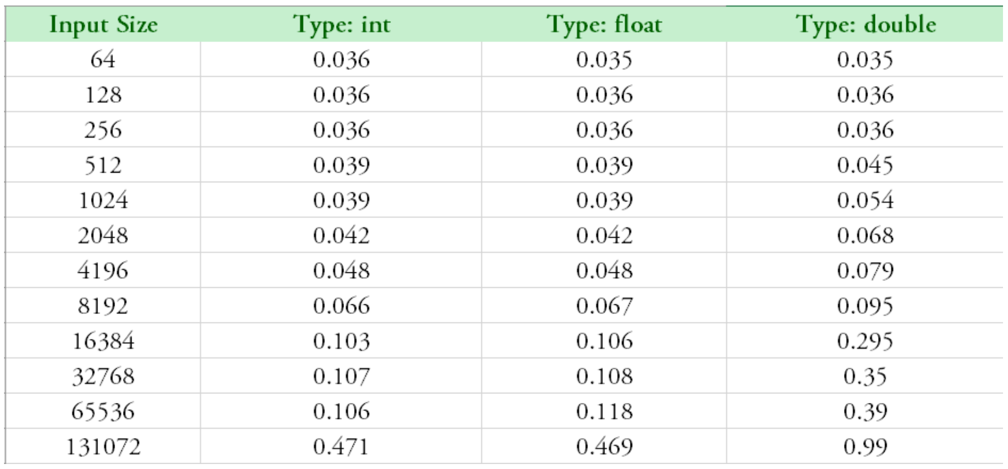


Figure 3.5: SM Activity of *PrefixSumMap* Kernel.

3.0.5 Data Type Analysis

Figure 3.6 illustrates how increasing the data type size can affect performance.



Input Size	Type: int	Type: float	Type: double
64	0.036	0.035	0.035
128	0.036	0.036	0.036
256	0.036	0.036	0.036
512	0.039	0.039	0.045
1024	0.039	0.039	0.054
2048	0.042	0.042	0.068
4196	0.048	0.048	0.079
8192	0.066	0.067	0.095
16384	0.103	0.106	0.295
32768	0.107	0.108	0.35
65536	0.106	0.118	0.39
131072	0.471	0.469	0.99

Figure 3.6: Timings for Various Types, GPU Implementation.