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.

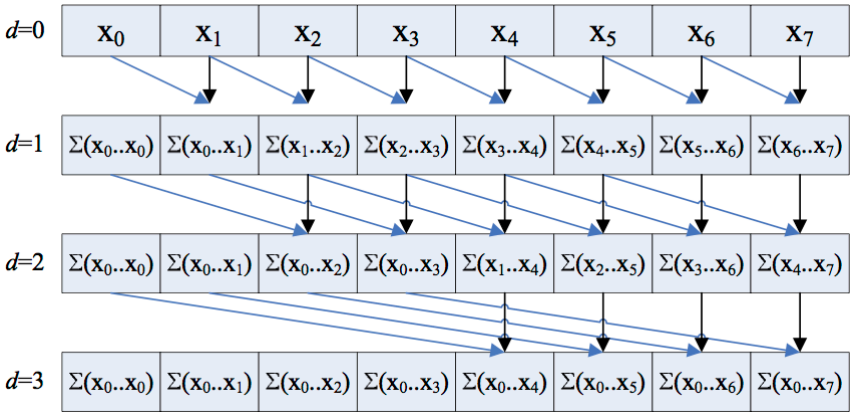

Figure 1.1: Demonstration of *Hillis & Steele* Algorithm.

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.