Parallel Prefix Scan using CUDA

CS F422: Parallel Computing

Guntaas Singh (2018A7PS0269P)
Harshit Garg (2018A7PS0218P)
K Vignesh (2018A7PS0183P)

April 27, 2021

The Introduction

In the following Document there have been some important assumptions and points which need to be stated here

- The input vector in the project is composed of integer numbers with positive values only
- It should also be known that a CPU core is not equal to a GPU core.Though, the sequential time was done on a CPU while the parallel time was taken with sequential part on CPU and parallel part on the GPU. There is also given a GPU sequential time which was done through some indirect manipulations.
- The code was tested on a machine with a GeForce GTX 1050Ti Mobile with 768 cores and 6 SMs and, Intel i5 8300H 4 cores with 8 threads.

The sequential form of prefix sum scan is given below where '+' can be replaced by any other binary operator $\bigoplus$:

```
1    procedure SCAN(in)
2    begin
3        Initialize all the elements of the output vector out to zero
4        out[1] =  in[1]
5        for i = 2 to n
6           out[i] = in[i]  + out[i-1]
7        return out
8    end SCAN
```

This algorithm has a serial asymptotic time complexity of $T_s = O(n)$ because of the single for loop at line 5

It can be clearly seen that we could parallelize this by not combining the reductions at each index sequentially and move in a more tree based manner with leaves being the elements of input and the internal nodes signifying the sum of it's children.

After this we can do a post scan step of moving in a reverse way, from root to the leaves to propagate the missing sums in the respective reductions at different indices.

The parallel algorithm can be defined as follows in some steps

1. Compute the sums of consecutive pairs of items in which the first item of the pair has an even index: $p_0 = in_0 \bigoplus in_1, p_1 = in_2 \bigoplus in_3$, etc.
2. Recursively compute the prefix sum $q_0, q_1, q_2, ...$ of each such sequence of $p_0, p_1, p_2, ...$till we reach a single element. This is where the reduction steps and we do the post scan step next.

3. Express each term of the final sequence $out_0, out_1, out_2, ...$ as the sum of up to two terms of these intermediate sequences: $y_0 = in_0, out_1 = p_0, out_2 = p_0 \bigoplus in_2, out_3 = q_0 ...$ etc. After the first value, each successive number y_i is either copied from a position half as far through the w sequence, or is the previous value added to one value in the x sequence.

Theoretical Speed-up, Efficiency, Work and Maximum Processor count while being cost optimal

Speedup and Efficiency

We can see that each depth level in the binary tree(starting from the leaves to the root) can be executed concurrently by n/2,n/4...1 processor respectively in the reduction part and same goes for the post scan step in reverse order. Thus the algorithm should take the following asymptotic parallel time with sufficient processors.
We do not consider the message passing overhead because this was implemented in a GPU and the low latency of the memory module available makes the memory access for all parallel elements a constant O(1) operation if no memory bank conflicts occur which is overshadowed by the cost of binary operation and the depth of the tree.
This implies that with sufficient processors we can have all nodes on a depth level being processed concurrently, and as the depth of the tree is $log_2(n)$

$$T_p = O(log_2(n))$$

This means that the speed-up should be -

$$S = \frac{T_s}{T_P} = \frac{O(n)}{O(log_2(n))} = O(\frac{n}{log_2(n)})$$

The efficiency would be

$$E = \frac{S}{p} = \frac{O(\frac{n}{log_2(n)})}{p}$$

For a theoretical constant efficiency

$$p = O(\frac{n}{log_2(n)})$$

This information would be very useful for further analysis.

Work

If we try to find the work done and treat one addition or binary operator to be one unit of work. For the Serial algorithm the Work can be easily defined as

$$W_s = n - 1 = O(n)$$

For this parallel algorithm, it can be broken down into two parts , one of reduction from leaves to root and the other post-scan step to propagate partial sums from the root to leaves.

1. For the first part, of reduction, we move up a binary tree (from leaves to the root) with a depth of $log_2(n)$, with each iteration doing n/2, n/4, n/8.....1 use of the binary operator.This gives a total sum of (n-1) use of the operator.

2. For the second part, of propagating the partial reductions we move down the same binary tree(from the root to leaves) of depth $log_2(n)$, with iterations doing 1,2,4..n/4,n/2 use of the binary operator. This gives a total sum of n-1 use of the operator.

Adding both of these findings we find the total parallel work done to be

$$W_p = (n-1) + (n-1) = 2 * (n-1) = O(n)$$

As the work is equal to the sequential work asymptotically done we can say that the algorithm is cost optimal and work efficient with sufficient processors and is scalable.

Maximum Count of Processors while being cost optimal

We saw that the parallel work was equal to serial work for sufficient processors. To find a bound on the count of such processor we try to find for what bound of processor is the problem cost optimal The problem is cost optimal if

$$Cost = pT_p = W_s$$

So,

$$p = W_s/T_p = O(n)/O(log_2(n))$$

$$p = O(\frac{n}{log_2(n)})$$

For this bound of processors we should not face any asymptotic slowdown while being cost optimal.Any less and we would face a slowdown and anymore and we may not stay cost optimal. Thus. this is the maximum amount of processors p for any input size n for which we can solve the problem cost optimally. This completes the part(e) of the problem.

Experimental data

No.of Elements	CPU Sequential time	GPU sequential time	GPU parallel time	Speed Up	Efficiency
2048	0.008	0.82944	0.0256	32.4	0.0421875
4096	0.017	0.970752	0.027648	35.11111111	0.045717593
8192	0.034	1.292352	0.029696	43.51939655	0.056665881
16384	0.068	1.904064	0.032864	57.93768257	0.075439691
32768	0.141	3.097152	0.035296	87.74796011	0.114255156
65536	0.284	5.894592	0.052256	112.8022045	0.14687787
131072	0.589	10.834368	0.09248	117.1536332	0.152543793
262144	1.189	22.041024	0.224224	98.29912944	0.127993658
524288	2.355	41.565312	0.378208	109.9006684	0.143099829
1048576	4.843	81.302016	0.672768	120.847032	0.157352906
2097152	9.637	160.65024	1.238816	129.6804691	0.168854777
4194304	19.674	318.008832	2.403296	132.3219578	0.172294216
8388608	38.803001	633.230802	4.715584	134.2847041	0.174849875

Table 1: Experimental data for the Implementation on 768 cores

No.of Elements	CPU Sequential time	GPU sequential time	GPU parallel time	Speed Up	Efficiency
256	0.0004	0.667968	0.041432	16.12203128	0.125953369
512	0.001	0.704784	0.042312	16.65683494	0.130131523
1024	0.002	0.759168	0.043992	17.25695581	0.134819967
2048	0.008	0.82944	0.044312	18.71818018	0.146235783

Table 2: Experimental data for the Implementation on 128 cores

No.of Elements	CPU Sequential time	GPU sequential time	GPU parallel time	Speed Up	Efficiency
512	0.001	0.704784	0.037888	18.60177365	0.072663178
1024	0.002	0.759168	0.039132	19.40018399	0.075781969
2048	0.008	0.82944	0.039851	20.8135304	0.081302853
4096	0.017	0.970752	0.04124	23.53908826	0.091949564

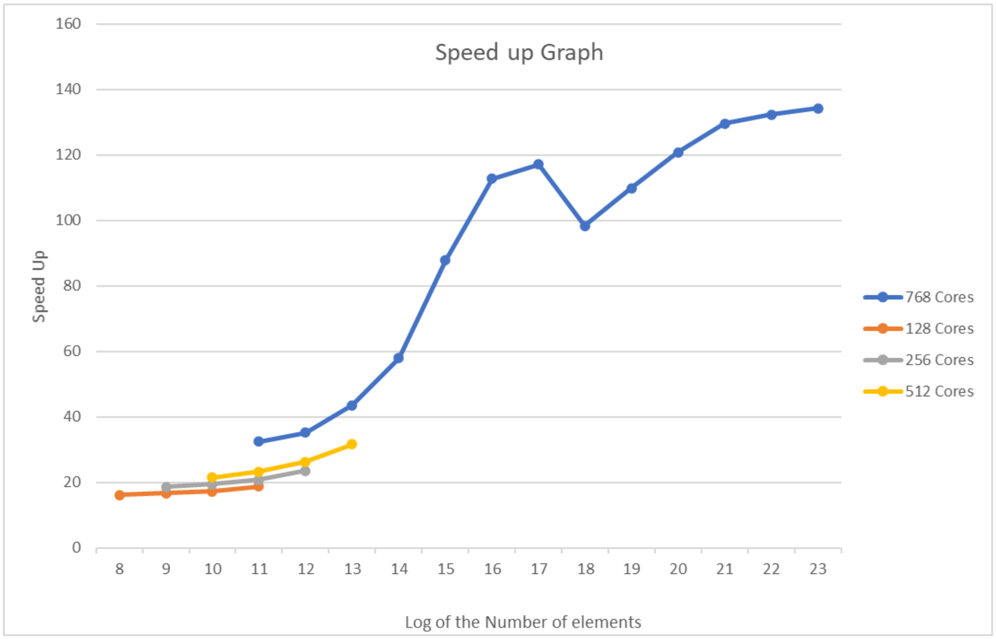
Table 3: Experimental data for the Implementation on 256 cores

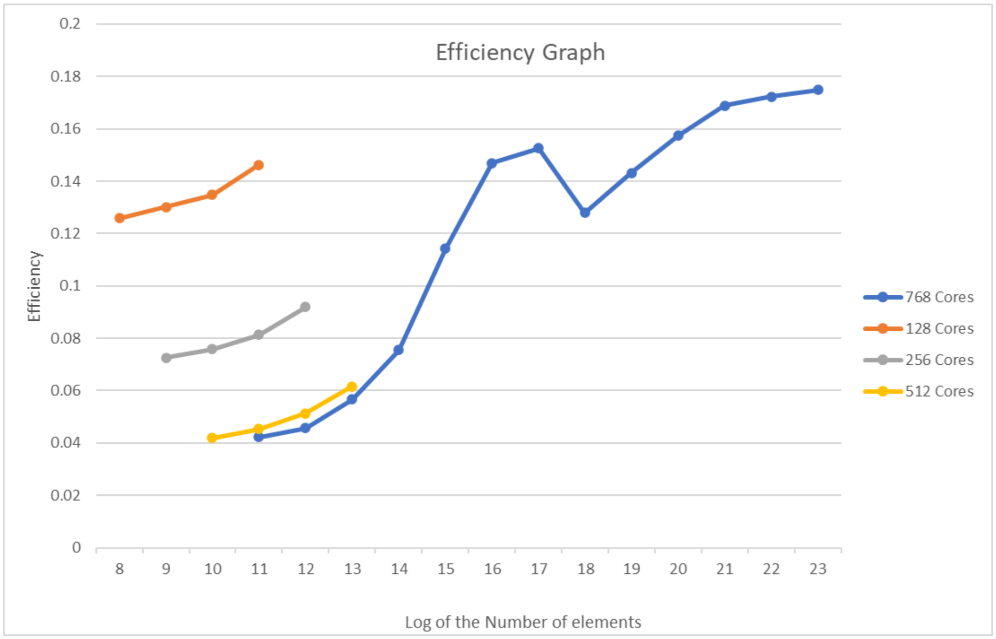
While reading and interpreting the data here and in further analysis we need to keep the following matter in mind

- The sequential GPU data has been obtained by setting the threads per block macro to be 1 and the number of blocks to be sequentially given to the GPU instead of as a grid.
- For Finding the parallelizable code, We have used CUDA events which have been commented to check the performance because these calls are slightly intensive and may distort the parallel execution results.
- The Speed- up and Efficiency have been calculated based on the sequential and parallel runtime on the GPU(Because we need to keep the system constant during performance analysis), the CPU times are given for reference only.

No.of Elements	CPU Sequential time	GPU sequential time	GPU parallel time	Speed Up	Efficiency
1024	0.002	0.759168	0.035424	21.43089431	0.041857215
2048	0.008	0.82944	0.035744	23.20501343	0.045322292
4096	0.017	0.970752	0.036992	26.24221453	0.051254325
8192	0.034	1.292352	0.04096	31.5515625	0.061624146

Table 4: Experimental data for the Implementation on 512 cores





- As CUDA doesn't allow for individual core control, we have tried to indirectly manipulate the cores used by manipulating the THREADS_PER_BLOCK macro, though this has limited strength in terms of control as seen by the data points.

Serial Fraction

The serial fraction is defined as

$$f = \frac{T_{ser}}{W}$$

We have used the CUDA Event calls between all parallelized workload to add all the time taken in the parallel part(T_{par}) of the code to find the parallel part which turned out to be

$$T_{par} = 629.8285ms$$

Using this and the serial time taken for the same input size of $8388608 = 2^{23}$.

$$f_{768} = \frac{W - T_{par}}{W} = \frac{633.2308 - 629.8285}{633.2308} = 0.00538$$

Taking average over all the processors and input size like the following
$f_{512} = 0.005394$
$f_{256} = 0.005374$
$f_{128} = 0.005351$
We arrive at the following serial fraction value after averaging it out.

$$f = \sum_{i=1}^{all} f_i \approx 0.005375$$

This is because at these point we have achieved the maximum parallelism as can be seen from the speedup table which is possible for the GPU without adding more parallel cores, so this should give us the best estimate of serial fraction possible for our algorithm with sufficient parallel processing elements. We would be using the 0.53% serial fraction value for future references in this document.

Experimental Speed up, Efficiency and their comparison to Amdahl's law

Speed up and Efficiency tables have already been shared above which tell us that the algorithm continues to give better speed ups till the input size of $8388608 = 2^{23}$.

The Amdahl's law allows us to give us a upper bound on the $S(p)$ on the GPU given by

$$S(p) = \frac{1}{f + \frac{1-f}{p}} = \frac{1}{0.0053754 + \frac{1-0.0053754}{768}} \approx 149.91$$

This is the upper bound of our speed up for 768 cores and it confirms with our experimental speedup data seen in the table. For other processor count the S(p) bound is

$$S(128) = \frac{1}{0.0053754 + \frac{1-0.0053754}{128}} \approx 76.07$$

$$S(256) = \frac{1}{0.0053754 + \frac{1-0.0053754}{256}} \approx 107.98$$

$$S(512) = \frac{1}{0.0053754 + \frac{1-0.0053754}{512}} \approx 136.65$$

These data points also conform our position in the speed up table, we could not come near this speed up in non-768 cores test cases because we cannot directly choose how many cores are used , but that depends on the input size and can only be indirectly manipulated.

Iso-Efficiency Function of the Algorithm

The Iso-efficiency function of this algorithm could be calculated with the following data and way:-

We have already defined the total work done by the parallel process and the work done by serial process.

$$W_s = n - 1 \text{ and } W_p = 2 * (n-1)$$

The overhead function can thus be formed considering that each work unit composes of a unit time.

$$T_o = pT_p - T_s = 2plog(p)$$

As we know we can make an asymptotic estimate as follows

$$T_o(W,p) = O(plog_2(p))$$

Thus the Iso-efficiency function becomes

$$W = KT_o(W,p) = O(plog_2(p))$$

Thus we can arrive at the Asymptotic Iso-efficiency function of $O(plog_2(p))$

Some Estimate of Maximum Processor for being cost optimal

The maximum processor for being cost optimal is given by the earlier formed bound of

$$p = O(n/log_2(n))$$

This can also be confirmed from the fact that the iso-efficiency function. We already know that the upper bound of processor for cost optimal solution is given by $O(f^-1(W))$ where $W = O(f(p))$ is the Iso-efficiency function. The inverse of this f(p) gives a Lambert function which in the asymptotic analysis yields

$$p = O(n/log_2(n))$$