COL 380 : Lab 1

Sagar Goyal 2015CS10253

1. Prefix Sum:

Algorithm Used :-

1. The first step of the algorithm was to divide the entire vector input into 'p' parts where 'p' is the number of threads. This was done symmetrically by using the index of the vector.
2. The next step was to calculate the prefix sum of the vector by using the linear method → prefix[n]=prefix[n-1] +arr[n]
3. This was done parallely by the 'p' threads and takes n/p time to accomplish. The value of the last elements of each block were stored in a separate array of size 'p'. Now a special algorithm was applied on this array.
4. We divided our work into essentially two parts, "up" traversal and "down" traversal.

Firstly, in "up" traversal. We treated the array like a tree of height log(n) where at each level the parent stored the sum of its children. See the diagram for reference, we move from d=3 towards d=0 i.e. the final value which we will have in our array will be the values given at d=0 state.

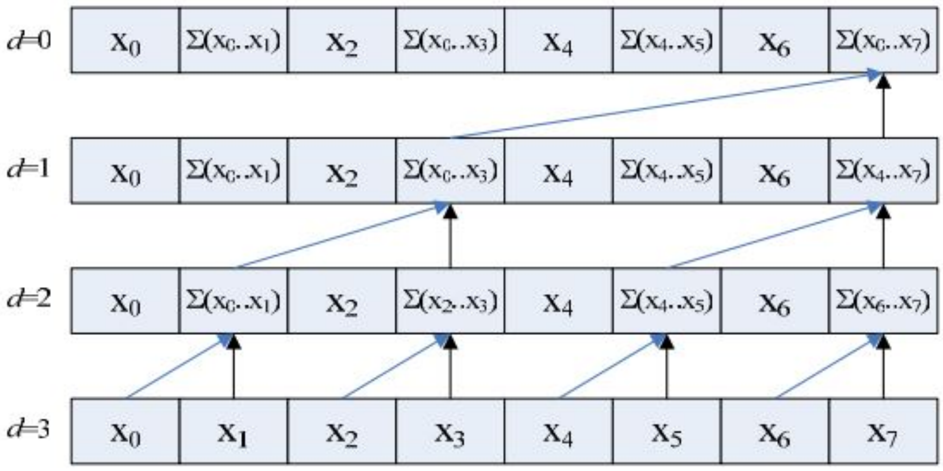
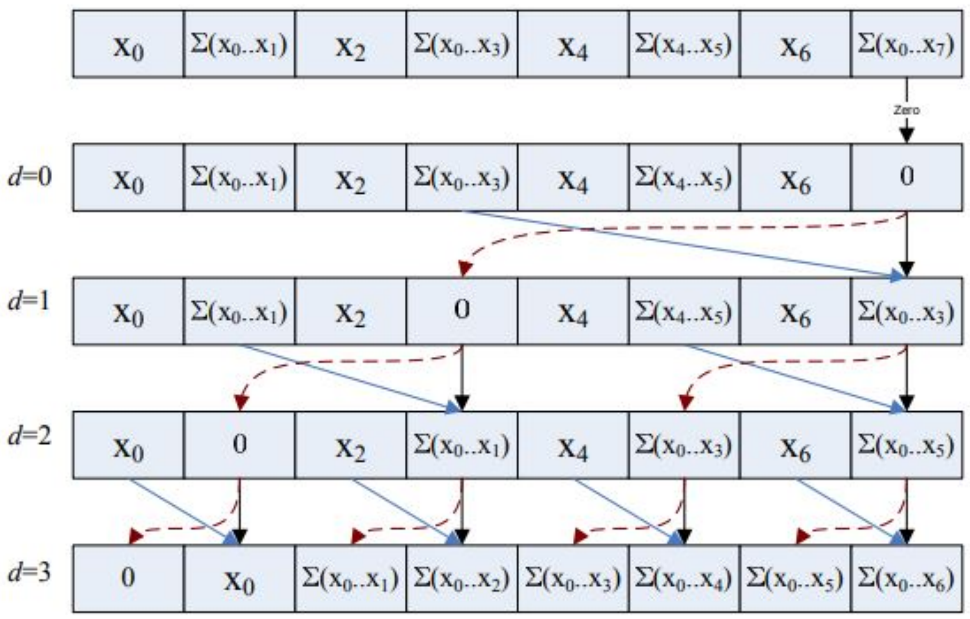


Diagram taken from the document : "Parallel Prefix Sum (Scan) with CUDA" by Mark Harris, NVIDIA

Now in “down” traversal. We will replace the last element of the array obatined by a 0 and store the preexisting one in a temporary variable.
Again using the same division into log(n) levels like a tree, at each level, the left child stored the value of the parent itself while the right child stored the sum of the parent and itself.
See the diagram for reference, we will move from d=0 to d=3 and thus at the end our array will be in the state d=3.



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

Diagram taken from the document : “Parallel Prefix Sum (Scan) with CUDA” by Mark Harris, NVIDIA

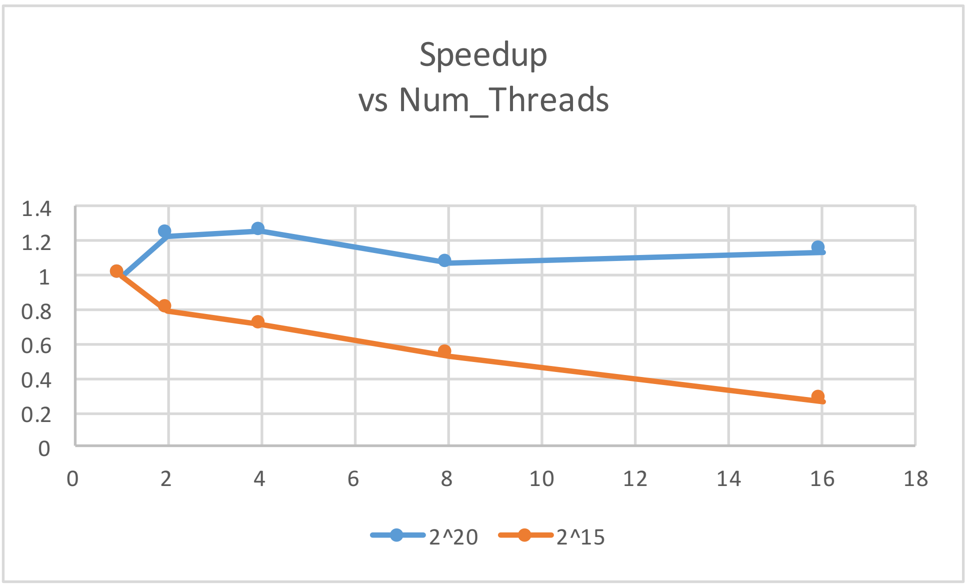
Clearly as can be seen be seen, we have obtained the prefix sum array which is shifted by one element, and last element is stored in a temporary variable.

At each level we find that we dont access any memory location simultaneously, and therefore each level can be run without taking anything in critical using a for loop.

Specifications used-

a. An array of size 2^20 filled with all 1s used as the input array for all the test cases.
b. Average time taken for 20 runs for every number of threads.

Speedup vs Number of Threads :-

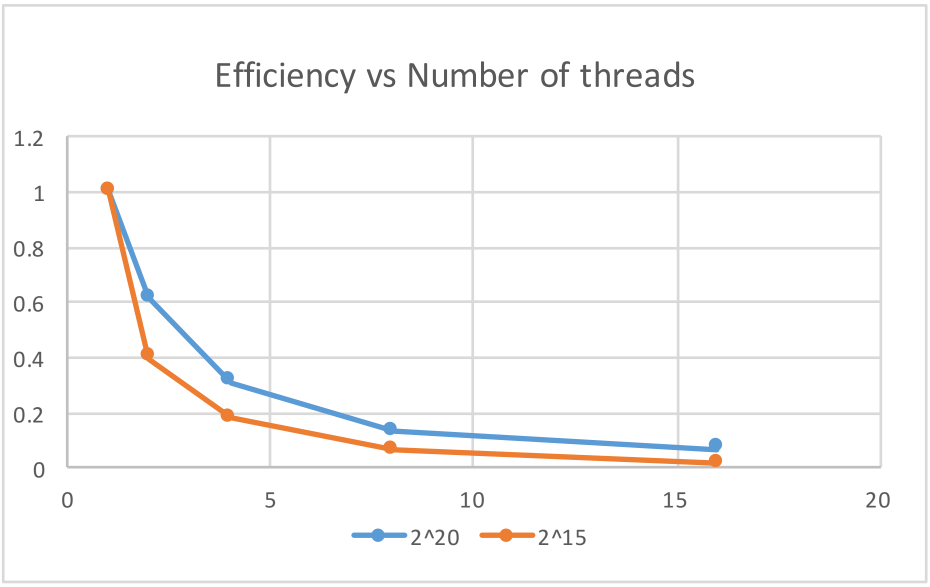


Ideally the speedup should have been equal to ‘p’ for each case but parallel programming involves a lot of overheads in interprocess communication and hence the speedups obtained are considerably less.
The number of threads that the computer can run at a time is limited by the number of cores on the pc and even if we keep increasing the number of threads in the code it doesn’t make much difference and rather increases the overhead without any improvement in the time of execution and hence reducing the speedups on increasing the threads beyond 4.

For smaller amount of data the overhead of parallel processing becomes way larger than the time saved in parallel execution because the total time that a program takes in linear execution is already less due to the small amount of data that is being processed and hence the speed up continuously decreases on increasing the number of threads when the data size is small.

However, since the time is calculated outside the ‘function’ which does the prefix sum and this ‘function’ actually copies the input vector, when the input problem size is so humongous (like 2^25) that this copying overpowers any other computational advantages that our code provides, we can see a drop in the speedup.

Efficiency vs Number of Threads :-



Now let us analyse the efficiency of the algorithm. As we increase the number of threads, the speedup is increase slower than the number of threads, thus the efficiency continuously decreases with the increase in number of threads for both the cases. The drop in case of small problem size is the maximum because of the overheads.