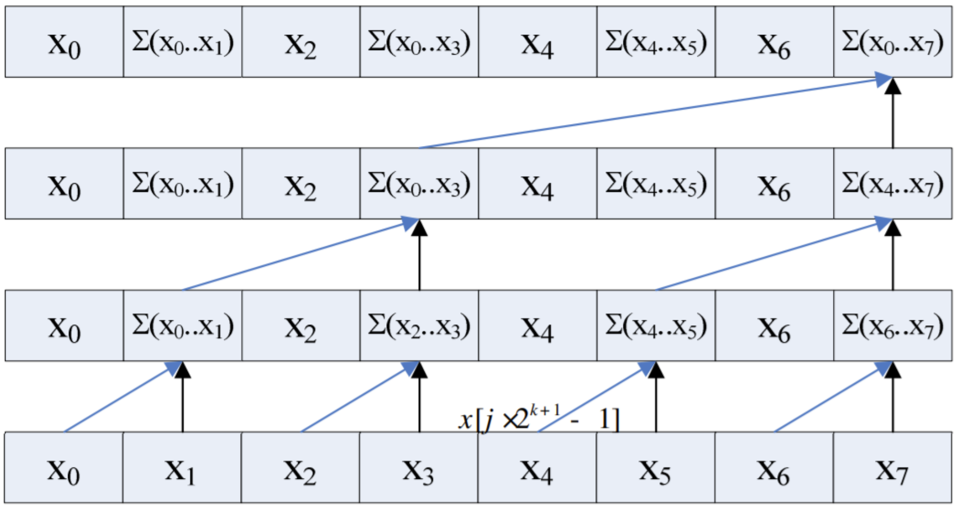
Parallel Prefix-Sum (Scan) Operation using CUDA

(a) Code explanation:

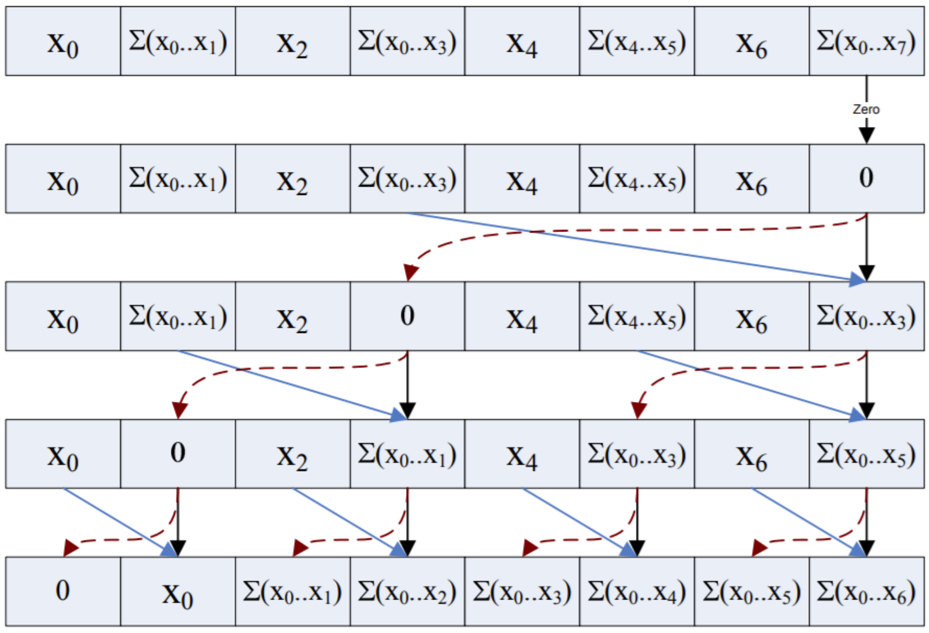
The algorithm that has been implemented for parallel prefix-sum is the Blelloch scan algorithm that uses the up-sweep (reduce) and down-sweep technique. It involves visualizing the input array as a balanced binary tree.

The first phase (up-sweep) involves traversing from the bottom (leaves) and up till the root while building partial sums. At each level all the operations can be done parallelly. Depiction of up-sweep phase is shown below:



x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_7)$
x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_4..x_7)$
x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_2..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_6..x_7)$
$x[j \times 2^{k+1} - 1]$
x_0 | x_1 | x_2 | x_3 | x_4 | x_5 | x_6 | x_7

The second phase (down-sweep) first involves setting the root element to zero and then traversing from the top (root) to the leaves (just like a mirror-image of up-sweep phase). It is depicted in the picture below:

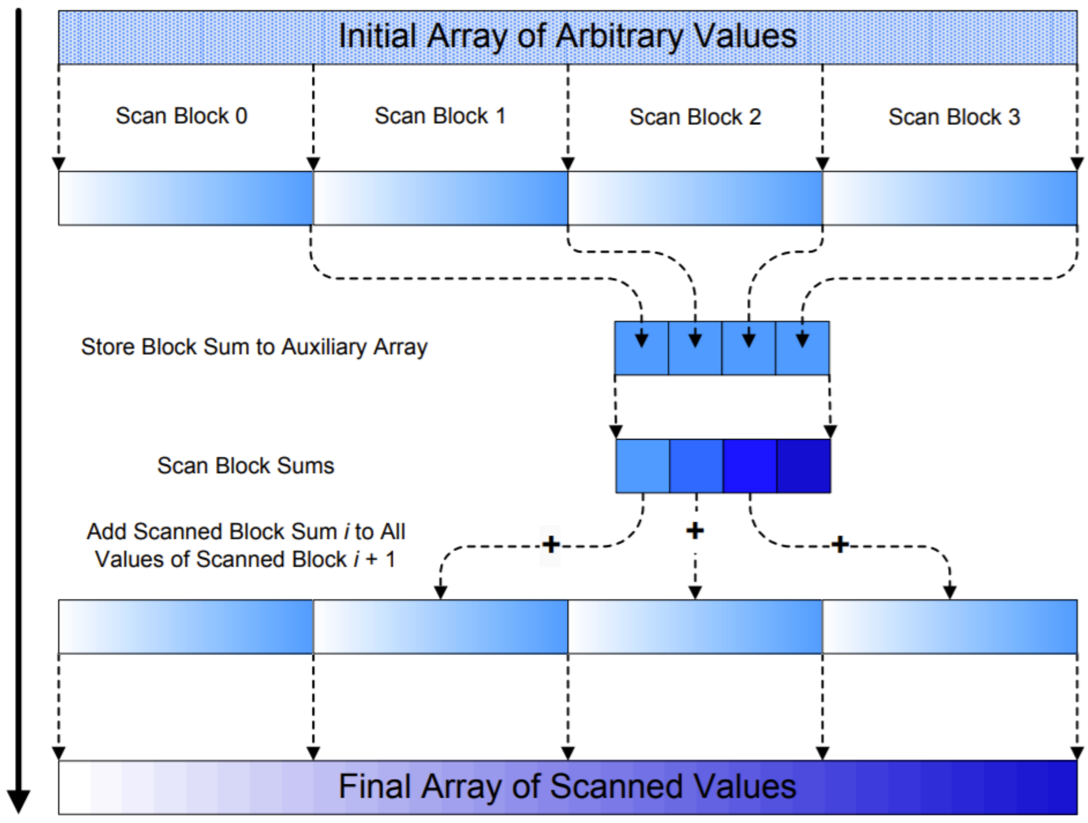


x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_7)$
Zero
x_0 | $\Sigma(x_0..x_1)$ | x_2 | $\Sigma(x_0..x_3)$ | x_4 | $\Sigma(x_4..x_5)$ | x_6 | 0
x_0 | $\Sigma(x_0..x_1)$ | x_2 | 0 | x_4 | $\Sigma(x_4..x_5)$ | x_6 | $\Sigma(x_0..x_3)$
x_0 | 0 | x_2 | $\Sigma(x_0..x_1)$ | x_4 | $\Sigma(x_0..x_3)$ | x_6 | $\Sigma(x_0..x_5)$
0 | x_0 | $\Sigma(x_0..x_1)$ | $\Sigma(x_0..x_2)$ | $\Sigma(x_0..x_3)$ | $\Sigma(x_0..x_4)$ | $\Sigma(x_0..x_5)$ | $\Sigma(x_0..x_6)$

The above depictions are shown for a particular block of the GPU with a certain number of threads (precisely, no_of_threads = array_size / 2; since at the bottom most level, all threads deal with 2 elements in the array).

This algorithm has been extended to work on multiple GPU blocks so as to support larger array input sizes. This requires additional computation of block sums.

Each block computes its scan on its own and also notes down its entire sum (block sum). Now using this array of blocksums we can compute the scan of the entire array by first computing the scan of the block sums array, and then adding blockSumScan[j] to each element in the jth block. This idea is depicted in the picture below:


Initial Array of Arbitrary Values
Scan Block 0
Scan Block 1
Scan Block 2
Scan Block 3
Store Block Sum to Auxiliary Array
Scan Block Sums
Add Scanned Block Sum *i* to All Values of Scanned Block *i* + 1
+
+
+
Final Array of Scanned Values

A separate Code file was created for the serial algorithm to measure the time taken.

Runnning the Code:

The input array size can be controlled by editing the 'ARR_SIZE' macro defined at the top (line 17).

The operation to be performed (addition / minimum / maximum) can be controlled by editing the 'OPERATOR' macro defined at the top (line 16)

To show verbose output, the code can be compiled with an optional command line argument 'v' or 'V'. This will print the either entire or first 100 elements from input and output arrays.

To compile, type : nvcc scan_CUDA.c

To run, type : ./a.out

For verbose output , type : ./a.out v   OR   ./a.out V

(b) Serial Fraction:

On testing with the code file containing the serial scan algorithm, the serial fraction was found as follows for varying inputs:

Input arr size (N)	**Total Exec. Time (ms)**	**Serial Portion Exec. Time (ms)**	**Serial Fraction (f)**
4096	0.079000	0.066000	0.835443
32768	0.601000	0.601000	0.825291
131072	2.509000	2.052000	0.817856
1048576	20.16200	16.48500	0.817627

On *average*, the ***serial fraction f*** comes out to be around **0.824054.**

(c) Performance Parameters:

By Amdahl's law, speedup is calculated as : S(p) = 1 / (f + ((1-f) / p))

⇨ S(p) = 1 / (0.824 + (0.176 / p))

Using the above formula we get the speedups mentioned in the last column of the table below:

Amdahl's law seems to be ***underestimating*** the speedups obtained while using **GPUs**.

This can be explained due to the fact that:

- Amdahl's law does *not* take into account the size of *input data*.
- It is well known that GPUs are built for handling large amounts of data parallelly and perform extremely well compared to CPUs as the size of input data increases. This is shown by the increasing observed speedup in the above table as size of input increases.
- Hence, purely based on the number of processing elements and serial fraction Amdahl's law underestimates speedups obtained from GPUs.

Log2(N)	**Input arr size (N)**	**Serial Exec. Time (ms)**	**Parallel Exec. Time (ms)**	**Speedup**	**Block Count**	**No. of concurrent processing elements (Streaming MPs) (p)**	**Efficiency**	**Speedup by Amdahl's law**
10	1024	0.024	0.09824	0.244299674	1	1	0.244299674	1
12	4096	0.079	0.111328	0.709614832	2	2	0.354807416	1.0965
15	32768	0.609	0.212992	2.85926232	16	16	0.178703895	1.1976
17	131072	2.566	0.451584	5.682220805	64	16	0.3551388	1.1976
20	1048576	20.376	2.518376	8.09092844	512	16	0.505683027	1.1976
22	4194304	81.597	9.40976	8.671528286	2048	16	0.541970518	1.1976
25	33554432	656.255	74.220285	8.841989761	16384	16	0.55262436	1.1976
27	134217728	2613.813	284.837074	9.176519627	65535	16	0.573532477	1.1976

(d) Iso-efficiency function

Rows 2 and 4 (corresponding to N = 4096 and N = 131072 respectively) can help us figure out the iso-efficiency function since the efficiency remains constant in both rows (= 0.355).

When *p* increases from 2 to 16 (increases by a factor of 8), then N increases from 4096 to 131072 (by a factor of 32).

This means that N increases as a function of : $p^{\frac{\log_2 32}{\log_2 8}} = p^{\frac{5}{3}}$

Thus, we can state the asymptotic iso-efficiency function for our parallel system as :

$$\boldsymbol{\theta}(\boldsymbol{p}^{\frac{5}{3}})$$

(e) Estimated no. of processors to solve the problem cost-optimally

For cost-optimality, we must have:

$$pT_P = \theta(W)$$

$$\Rightarrow W + T_0(W,p) = \theta(W)$$

$$\Rightarrow T_0(W,p) = O(W)$$

$$\Rightarrow W = \Omega\big(T_0(W,p)\big)$$

Since we have our iso-efficiency function : $f(p) = \theta\left(p^{\frac{5}{3}}\right)$, then the following equation must hold for cost-optimality :

$$W = \Omega\big(f(p)\big)$$

Since we know that $W = \theta(n)$ (i.e. the best-known serial algorithm for computing scan involves operations of linear order in input; to be precise It involves exactly $n - 1$ addition operations), we have:

$$n - 1 = \Omega\left(p^{\frac{5}{3}}\right)$$

$$\Rightarrow n - 1 \geq p^{\frac{5}{3}}$$

$$\Rightarrow \frac{3}{5}\log_2(n-1) \geq \log_2 p$$

$$\Rightarrow p \leq (n-1)^{\frac{3}{5}}$$

This gives us an upper bound / maximum number of processing elements *p* in order to a achieve cost-optimal, parallel scan algorithm for a given *n*-sized input array.