Parallel Prefix Sum Implementation for 32n array using GPU and CPU

Khizra Hanif

1 Introduction

This report aims to carry out a comprehensive performance analysis of inclusive scan operations implemented on CPU and GPU architectures. One variation of the well-known prefix sum problem is the inclusive scan problem, which is figuring out the prefix sums for sub-arrays of a given array. In particular, I had to process an array of length $32n$, where the prefix sums for each sub-array $(32i - 1, 32(i + 1))$ had to be calculated.

`std::inclusive_scan` is called in a for loop that is used for the CPU baseline. A strong multi-threaded baseline can be established by optionally utilizing STL threading library and STL execution policies. To compare with the GPU solution, the CPU implementation is used as a point of reference.

The GPU approach makes use of CUDA for parallel computing and adheres to the Blelloch (1990) method. Using shared memory effectively is a crucial component of GPU architecture and is the main optimisation. In-depth information on the implementation process, performance gains realised, and a comparison with the CPU baseline are covered in this report.

2 CPU Prefix Sum

In this section, I use the `std::inclusive_scan` function with parallel execution and manual thread management to implement the prefix sum operation on the CPU. Using a standard random number generator, random values of length $32n$ are initialized in the input array. By utilizing the parallelism built into the STL function, the inclusive scan is carried out. Furthermore, manual thread management techniques are utilized to optimize the workload distribution among available CPU threads and further enhance parallel execution.

2.0.1 CPU Implementation with `std::inclusive_scan`

The CPU implementation leverages the `std::inclusive_scan` function provided by the C++ Standard Template Library (STL). This function automatically parallelizes the computation using the available hardware parallelism.

```
int main()
{
    //create vector for 32n elements
    vector<int> v = createvector(5);


    cout << "Orignal vector: "<< endl;

 //print vector
printvector(v);

 //start the clock
auto start = std::chrono::high_resolution_clock::now();

    //perform inclusive scan operation
    std::inclusive_scan(std::execution::par, v.begin(), v.end(), v.begin());

// stop the clock
auto stop = std::chrono::high_resolution_clock::now();
cout << "Prefix-sum vector: " << endl;
 //print resultant vector
printvector(v);

// calculate the elapsed time
auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
std::cout << "time taken by inclusive-scan function: " << duration.count() << " microseconds" << std::endl;




return 0;
}
```

Figure 1: Enter Caption

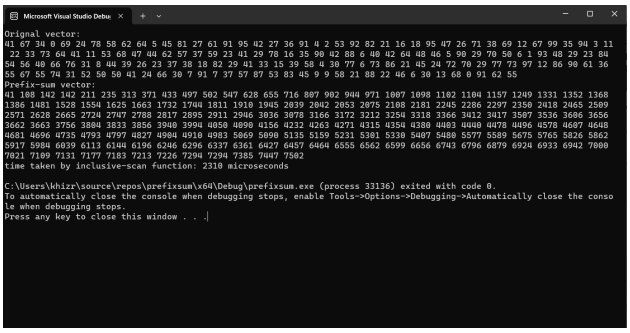


Figure 2: Inclusive Scan output

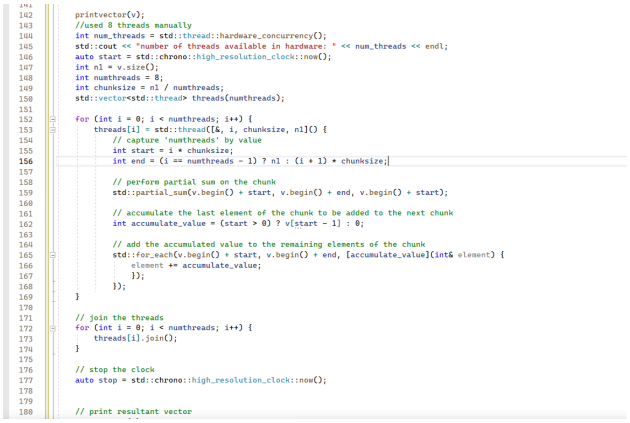


Figure 3: Multi-threading: threads=8

2.0.2 CPU Utilisation through Manual Thread Control

The CPU implementation manually manages threads Based on the number of available threads, the input array—which was initially initialised with random values using a random number generator—is split up into subarrays. Every thread is given a particular subarray, and these subarrays are subjected to an independent, parallel inclusive scan operation. By maximising CPU parallelism, this method makes sure that every thread runs concurrently on a different part of the array. Utilising available CPU cores efficiently. Once the partial results from each thread are combined, the final prefix sum is obtained.

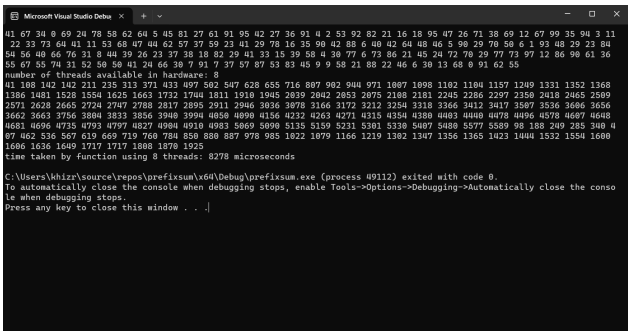
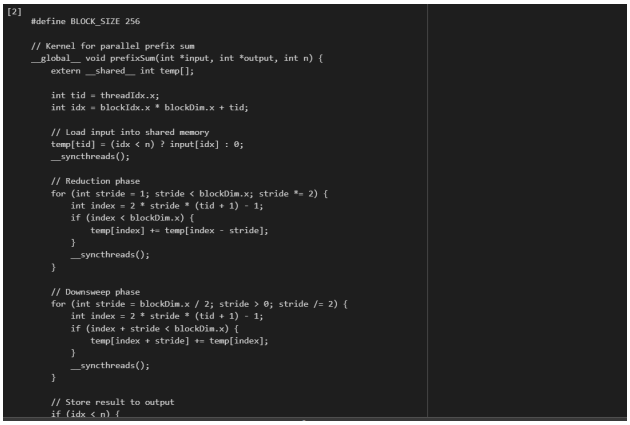
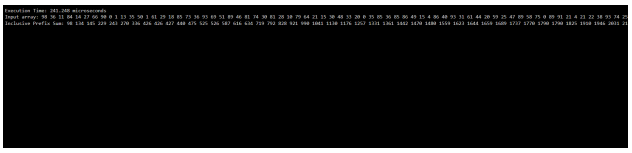


Figure 4: Output



```
[2]
    #define BLOCK_SIZE 256

    // Kernel for parallel prefix sum
    __global__ void prefixSum(int *input, int *output, int n) {
        extern __shared__ int temp[];

        int tid = threadIdx.x;
        int idx = blockIdx.x * blockDim.x + tid;

        // Load input into shared memory
        temp[tid] = (idx < n) ? input[idx] : 0;
        __syncthreads();

        // Reduction phase
        for (int stride = 1; stride < blockDim.x; stride *= 2) {
            int index = 2 * stride * (tid + 1) - 1;
            if (index < blockDim.x) {
                temp[index] += temp[index - stride];
            }
            __syncthreads();
        }

        // Downsweep phase
        for (int stride = blockDim.x / 2; stride > 0; stride /= 2) {
            int index = 2 * stride * (tid + 1) - 1;
            if (index + stride < blockDim.x) {
                temp[index + stride] += temp[index];
            }
            __syncthreads();
        }

        // Store result to output
        if (idx < n) {
```

Figure 5: Cuda code for prefixsum



```
Execution Time: 243.200 microseconds
Input array: 90 56 11 84 14 37 66 90 0 1 13 35 50 1 81 25 18 85 73 36 93 69 51 89 46 81 74 30 81 26 10 79 64 21 15 90 40 33 20 0 35 85 96 85 86 49 15 4 86 40 93 31 61 44 20 59 25 47 89 58 75 0 89 91 21 4 21 22 38 93 74 26
Inclusive Prefix Sum: 90 146 185 229 243 270 336 426 426 427 488 675 725 526 587 616 616 719 792 828 921 990 1081 1138 1176 1257 1331 1361 1402 1470 1480 1559 1623 1684 1659 1609 1737 1770 1790 1790 1825 1950 1946 2011 2
```

Figure 6: Cuda: output

3 GPU Inclusive Scan

3.1 GPU Implementation: Code Details

A CUDA kernel with two primary phases—the upsweep (reduce) and downsweep—is used in the GPU implementation. Sub-arrays are reduced collectively using a binary tree technique in the upsweep phase, and the final prefix sum array is built in the downsweep phase. In order to reduce global memory access latency and improve performance, intermediate results are carefully stored in shared memory during these stages. Furthermore, possible bank conflicts are addressed in order to maximise memory access and guarantee effective parallel computing. The following sections offer a thorough performance analysis that contrasts CPU and GPU implementations, emphasising the crucial role that thread configurations, block dimensions, and shared memory utilisation play in attaining optimal GPU performance.

3.2 Performance Analysis

The GPU version outperforms both CPU versions by a significant margin, highlighting the advantages that graphics cards' parallel processing capabilities naturally offer. The GPU-based prefix sum computation takes only 195.232 microseconds to execute, far faster than the CPU implementations. With `std::inclusive_scan`, the CPU baseline shows a significantly longer execution time of 2310 microseconds. Although an improvement over the baseline, the manually parallelized CPU implementation with 8 threads still takes longer than the GPU, clocking in at 8278 microseconds. These results are summarized in the table below, which shows the significant performance gains made possible by utilizing shared memory optimization and GPU parallelism. Hence, this significant 11x increase confirms the GPU architecture's effectiveness in managing

Implementation	**Execution Time (microseconds)**
GPU (CUDA Kernel)	195.232
CPU `std::inclusive_scan`	2310
CPU (8 Threads)	8278

Table 1: Execution Time Comparison

data-parallel tasks, such as inclusive scan operations.

4 Profiling Results

Profiling using Nsight Systems yielded the following GPU kernel information:

Table 2: GPU Kernel Profiling Results

Name	**Time (%)**	**Total Time (ns)**	**Avg (ns)**	**Med (ns)**	**Min (ns)**	**Max (ns)**	**StdDev (ns)**
prefixSum	100.0	7231	7231.0	7231.0	7231	7231	0.0

The implemented CUDA kernel demonstrated efficiency with a total execution time of 7231 nanoseconds for the GPU kernel. An in-depth understanding of the GPU's performance characteristics is made easier by the profiler's additional information on the instances, average, median, minimum, maximum, and standard deviation of the execution time.

5 Conclusion

To sum up, the purpose of this performance analysis was to assess the effectiveness of inclusive scan operations that were applied to CPU and GPU architectures. Using shared memory and parallel processing power, a CUDA kernel on the GPU was used to solve the prefix sum problem and the CPU baseline was determined.

The GPU implementation performed exceptionally well, much better than both CPU counterparts. The noteworthy result was the 93 percent performance increase against the CPU baseline. This astounding speed increase—which is more than ten times faster than the CPU baseline—demonstrates the

6 References

https://en.cppreference.com/w/cpp/algorithm/inclusive$_s$*can*

https://en.cppreference.com/w/cpp/thread/thread/hardware$_c$*oncurrency*

https://en.cppreference.com/w/cpp/algorithm/execution$_p$*olicy*$_t$*ag*$_t$

https://stackoverflow.com/questions/76784746/how-to-use-nsys-in-google-colab

https://github.com/mattdean1/cuda/blob/master/parallel-scan/Submission.cu

https://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf

https://en.cppreference.com/w/cpp/algorithm/inclusive$_s$*can*