# A Single Pass Scan Using look-back

dwp992 & lsh789

November 2023

## 1 Introduction

Throughout this course we've covered many topics in parallel programming, among these one of the primary building blocks of parallel programming: The array scan. The scanning algorithms we've seen up until now have either been fully sequential or have in some way been multi-pass, i.e., more than 2 reads/writes are made to global memory per element. In this project we seek to implement a parallel single pass scanning algorithm, utilizing shared memory and registers for the vast amount of our computations, and only performing computations in global memory when doing inter-block computations. We experiment with different methods, such as having this inter-block communication being performed using an auxiliary block and using global aggregate and prefix arrays with blocks computing prefixes either sequential or in parallel. Furthermore, using these methods along with a list of baselines we experiment with different block sizes, register sizes per thread, and array size.

## 2 Background

This project is based on the paper "Single-pass Parallel Prefix Scan with Decoupled Look-back" [1]. In this paper the authors go through explaining their implementation of a single pass scan, meaning we can scan with only 2 global IO accesses per element. (Not counting a few extra needed per block). The important part of the algorithm they introduce, is the decoupled look-back scan, that will scan over what previous blocks have gotten. The paper also proposes a number of optimisations, some of then we implemented and some of them will be discussed later on.

### 2.1 Decoupled look-back

Normally when we would do a scan we would need 3 global IO accesses per element, since we would first need a read, when calculating the reduction of each block and scanning over the result of these, this result will then be used in a second pass through were we then scan each block while adding the prefix we found from the reduction/block-level-scan this gives us another global read. And finally we need to write the result. What we can see from this normal implementation is that we 2 of the 3 global accesses are reading the same memory, the only problem is that we are reading it with bad temporal locality, so we can not keep the data. This is what we try to fix by doing a single-pass scan, since we only pass through the data once, we only do one global read. And then we calculate the reduction and result right away.

One of the issues with this way of implementing it. Is that we now have a dependency between the blocks, so in order for block 2 to calculate the result of it's scan, it needs the prefix value from block 1. In the paper they then introduce the look-back algorithm that lets the blocks update a global array with their prefix and aggregate values that future blocks then can use. This is done by the aggregate value being just a reduction of all the values in the block, and the prefix can be found by adding this aggregate to the previous blocks prefix. Specifically we have 3 global arrays, each with the length that is the number of blocks. The first array, a flag array, where the flag can be X, if the neither the aggregate nor the prefix value has been calculated, A, if the aggregate value has been calculated but not the prefix value, and P if the prefix and aggregate value has been calculated. The other 2 arrays are then the aggregate and prefix arrays, that contain the prefix and aggregate values. But for this we also need to ensure that the blocks are being calculated in certain order, so we do not just have all calculating threads waiting for block 1, that is in queue to being spawned, the paper suggest a dynamic scheduling of the blocks, which we utilised in our implementation.

## 3 Our implementation

This section describes the look-back algorithm we've implemented, the kernel that executes the

scan, and the associated functions we've defined. Listing 3 describes the look-back kernel and listing 1 contains the code for our look-back algorithm. All our kernel related code can be found in the `spsKernels.cu.h` file, and our tests which verify the algorithms can be found in the `testSPS.cu` file. The algorithms we've implemented verify against a sequential CPU scan, and using the accompanying code while in the `/src` directory one can execute the code that verifies the algorithms by calling `make`.

## 3.1 The look-back Scan Kernel

Listing 3 takes pointers to the input and output arrays, along with a size. We also pass as arguments 4 pointers to different global memory locations. The first is a dynamic ID that is shared across blocks. The second, third, and fourth are pointers to the flag array, aggregate array, and inclusive prefix array described in section 2. The arrays are implemented using the `volatile` keyword to ensure that memory is visible for all threads as soon as it's written, and the dynamic ids are handled using atomic additions. Specifically, the `atomicAdd` function is used to add to the global variable and return the value before adding. Line 9-10 initializes the shared memory that we'll need. We create a shared array of size $Q * B$ and a shared buffer of size $B$. The shared array accompanying each block is reserved for part of the array the block scans. The shared buffer stores scans performed by each thread on `Q` elements. The kernel code contains 6 different device functions:

1. `copyGlb2Shr` - Copies memory from global to shared memory in a coalesced fashion. Equivalent to the function used in assignment 2.
2. `threadScan` - Each thread scans $Q$ elements from the shared memory and writes the resulting value to the shared buffer at the index equal to the thread id.
3. `blockScan` - Performs an inclusive scan equivalent to the optimized warp scan from assignment 2.
4. `threadAdd` - Adds the correct inclusive prefix to all values in the shared memory. After this function call, all values in shared memory have values representing an inclusive scan for the block only.
5. `look-backScan` - The look-back scan function looks through the flag, aggregate and prefix arrays to compute the appropriate exclusive prefix to add to the respective blocks shared memory. This function is described in detail in subsection 3.2.
6. `threadAddVal` - Each thread adds the appropriate exclusive prefix to the $Q$ elements it's responsible for.
7. `copyShr2Glb` - Copies the shared memory to a location in the global array in a coalesced fashion. This is functionally equivalent to the one we used in assignment 2.

Originally the design choice to use a separate shared memory buffer was made as it made the algorithm easier to construct in pseudo code. Furthermore, this choice makes the whole algorithm more easy to understand and explain. A potential drawback could be inefficiency in using $B * 4$ extra bytes of shared memory per block, since if $B = 1024$, this shared buffer would use $4*1024/49152 = 0.083 \approx 8\%$ of the shared memory per block.

## 3.2 The look-back Algorithm

In accordance with section 2 we have implemented the look-back algorithm using 3 different arrays in global memory: A flag array, an aggregate array, and an inclusive prefix array. this section covers the look-back version where only 1 thread computes the exclusive prefix and updates its own aggregate value and inclusive prefix values. The implementation of this can be seen in listing 1. The lines 11 to 34 are computed only for the final thread in the block.

Line 11-16 covers the case in which the dynamic id is 0 - We set both the aggregate value and the prefix value equal to the final value in the shared memory for the block. It then memory fences and then sets its flag to be P.

Line 18-34 covers the cases where the dynamic ids are greater than 0. First the aggregate value is set to the final value in the shared memory for the block. After a memory fence we set the flag value to A. We then define a `grab_id` - an index of where we want values to add to our current aggregate value to produce an inclusive prefix. In line 25-29, while the flag at `grab_id` is not P and the `grab_id` is greater than zero, the aggregate value at `grab_id` is added to an accumulating variable and the id is decremented. Finally, once a P flag is reached, the prefix value at the dynamic id is set to the current accumulation plus

the prefix value at the `grab_id`. After a fence, set the dynamic id flag to P.

Finally after a inter-block thread synchronization, in line 37, computed by all threads in the block, a prefix variable is created - this is the exclusive prefix value returned. It is computed by the prefix value minus the aggregate value accompanying the dynamic id.

### 3.3 Parallelization of the look-back algorithm

One of the optimisations that were suggested in the paper was a parallelized look-back, we decided to implement this in our implementation, the implementation can be seen in Listing 2. The code works much like the normal look-back, but instead of going through each element in the arrays(being the flag/aggregate/prefix arrays), one at a time, we look at a number of these in parallel instead.

The first decision was how many elements we should look at in the arrays at a time, we decided to look at 32 elements in each iteration of the look-back, 32 specifically here being the warpsize on the hardware we used, since we are doing a reduction on the values each thread read, so therefore we thought that it would be faster to ensure that this reduction is done in a single warp. For faster communication between the threads.[1]

In Listing 2 we can see that in line 17-31 we first handle the first block, and set the values, much like the sequential look-back.

In line 34-103 we have a do-while loop, which will do the parallel look-back, the loop is to ensure that we keep looking back until we reach an element that has a P flag, so that we have a prefix value to start and not just aggregate values.

In the loop in lines 39-60 we start by copying, the elements in the flag/aggregate/prefix arrays into shared memory, which is done by the threads in the first warp. We ensure that when we copy we have a useful result, meaning each thread will wait until it reads either a P or A in the flag array, before copying, in order to handle that there might be some elements with the X flag set, due to its block not haven calculated the aggregate value yet. Note that we ensure not to read at negative indices, and we also ensure that the element we read with the lowest index is written to index 0 in shared memory.

In lines 62-67 we set the rest of the values to 0. This is done so it will not change the result. The reason for this being that the values will then be a right identity/neutral element for the operator we are reducing with. We only need it to be a right identity, since we only add these values at the end of the array.

We then have a syncthreads in line 69 to ensure that all of this data has been written before we do the reduction.

In line 70-93 We have a loop where the first warp does a reduction over all of the elements we have just copied to shared memory.[2] The operator we use is defined in lines 79-86, and basically says if the right hand side is a prefix value, then we just keep that flag and value from the right hand side, otherwise we sum up the values and keep the flag that was on the left hand side. One thing to note about this reduce is that we decided to define it such that the result is stored in the left hand sides place, and therefore the result of the whole reduction will always be stored at index 0.

After the reduction, in lines 95-97 we update the shared prefix variable with the result. Before we start the loop again if we did not find a prefix value.

After the loop in lines 108-112 we update the prefix array with the result we found, and in line 115 each thread returns the prefVal, which is the value of the reductions we did, which should be added to each element in the block to complete the scan, as discussed in subsection 3.1 about the kernel.

### 3.4 SPS using an auxiliary block

We also implemented the method mentioned in the old slides written by professor Henriksen, wherein the construction of the inclusive prefix array is done by one auxiliary block that is given the dynamic id of -1. As values in the aggregate array become available, this auxiliary block sequentially computes the prefix array. Blocks that have computed an aggregate value wait for the prefix value to be computed, and once it has the block adds the exclusive prefix to its section of the scanned array.

[1] With our implementation there is also an argument for not wanting this number being to big, due to us always reducing through all the elements we read, since the first prefix value hopefully will be close, and we then might do a lot of unnecessary calculations.

[2] This reduction was inspired by the warpscan and reduction we used in assignment 2.

# 4 Throughput for different array sizes

## 4.1 Experiment

Our first experiment focused on exploring how the throughput changed as we increased the array size. We measured throughput in terms of GB/s read/written to global memory on average over 100 runs for both GPU and CPU. For randomly initialized arrays of sizes between $2^{10}$ and $2^{30}$ we measure the performance of our implementations and some baseline computations. Our baseline computations are the `cudaMemcpy`, a naive memcpy kernel, a sequential scan running on CPU, and the GPU inclusive scan algorithm from assignment 2. We measure the performance of our sequential look-back method, our modified parallel look-back method, and our implementation of single pass scan using an auxiliary block.

For these experiments we heuristically found the combinations $Q = 30, B = 256$, $Q = 10, B = 256$, and $Q = 10, B = 1024$ to achieve great results for all methods when compared to other parameters. Our intuition for why these parameters work will is that they utilize have greater of the shared memory available to each block, leading to an overall faster algorithm, as we can make many more computations in shared memory.

We expect to see the `cudaMemcpy` and naive memcpy kernel to achieve the highest throughput, as it is the de facto high ceiling of how well our algorithm can perform. We expect the parallel look-back scan to outperform the sequential look-back scan, but that both of them would outperform the rest of the methods. We do expect the auxiliary method to fare worse than the look-back method but better than the GPU inclusive scan.

## 4.2 Results

The results of this experiment can be seen in figures 1, 2, and 3. From all figures we can clearly see that `cudaMemcpy` and our naive memcpy implementation achieved the highest throughput in the experiments, with one instance of `cudaMemcpy` for $N = 21$ in figure 2 having a throughput of almost 1500 GB/s, nearing the theoretical limit of 1555 GB/s on the used hardware.

We see that in figures 3 and 1 the parallel look-back outperforms all other scan methods tested, while in 2 the baseline GPU scan outperforms both the parallel and sequential look-back scan. Our hypothesis for this is that as we lower batch size, the chunk size of the will increase accordingly for the baseline GPU scan, while for our look-back scan implementation suffers from smaller computations in shared memory, and more computations in global memory.

We also generally see that the auxiliary block version has performance similar to a sequential CPU scan. This was surprising to us as we assumed the mentioning of it in the supplementary slides indicated the solution. This very low throughput might be due to our implementation, but so far we haven't found the culprit.

Finally, from this data we can also measure the performance difference between our implementations and our baseline naive memcpy, which can be found in table 1. We see first of all that the auxiliary block method clearly underperforms. At $b = 256, Q = 30$ we do however achieve 71% of the throughput of memcpy for our sequential look-back, and 84% of the throughput for our parallel look-back.

| | B=256 Q=30 | B=256 Q=10 | B=1024 Q=10 |
|---|---|---|---|
| naiveMemcpy | 1.00x | 1.00x | 1.00x |
| Seqlook-back | 0.71x | 0.34x | 0.61x |
| Parlook-back | 0.84x | 0.49x | 0.62x |
| AuxBlock | 0.03x | 0.01x | 0.01x |

Table 1: Throughput performance vs naiveMemcpy

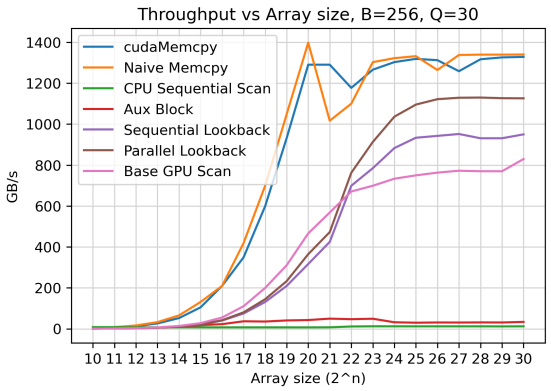



Figure 1: throughput measurements for $B = 256, Q = 30$.

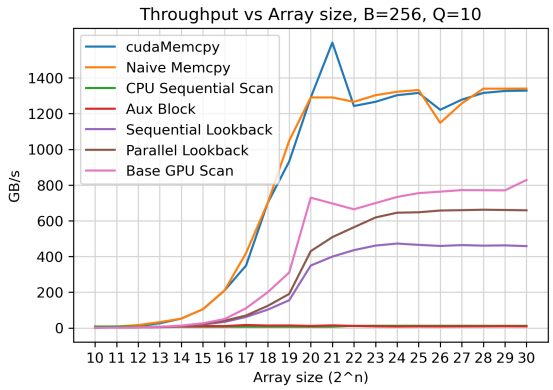



Figure 2: throughput measurements for $B = 256, Q = 10$.

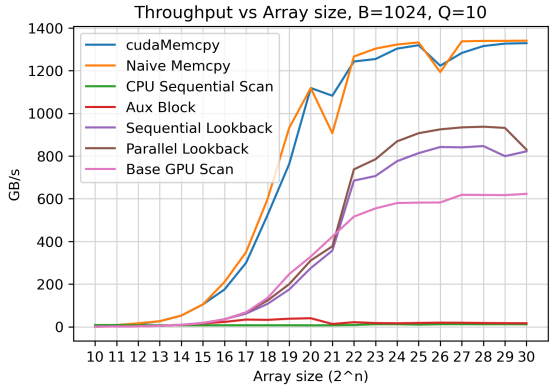



Figure 3: throughput measurements for $B = 1024, Q = 10$.

# 5 Changes in throughput as we vary B, Q, and N

## 5.1 Experiments

Our next experiment was to decide on lists of parameter values for each of $B$, $Q$, and $N$:

- $B$: 2, 4, 7, 8, 10, 13, 16, 20, 24, 30, 32, 40.
- $Q$: 32, 64, 128, 256, 512, 1024.
- $N$: $2^i \; \forall \; i \in [10, 30]$.

For each pairing of these parameters we computed the throughput of what we saw to be the best performing scan method we implemented - the parallel look-back scan. After this we found the highest throughput measured, which was 1133.64 GB/s with the parameters $B = 256, N = 2^{30}, Q = 30$. Using these values we created heatmaps of the three different slices created by fixing either of the parameters and varying the two others. These figures are found in figure 4.

## 5.2 Results

The heatmaps seen in figure 4 show that as we change each of parameters the change in throughput varies. In figure 4a we see that as we increase $N$ but leave $Q$ fixed, apart form using a block size of 32 or 64 there isn't a large difference between the block sizes used. Furthermore, for lower $N$s there is little to no difference between throughput for different batch sizes.

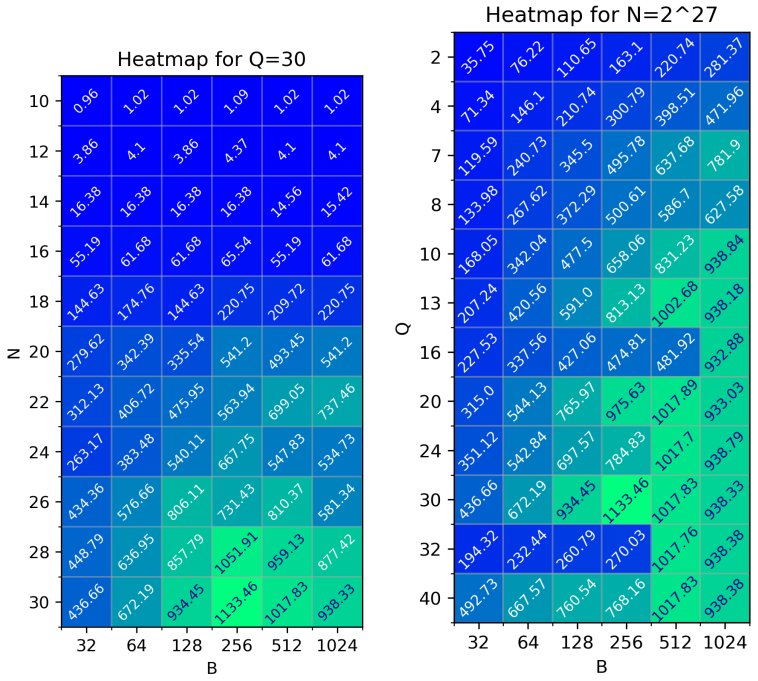
We can see a similar pattern in figure 4c: For lower $N$s there is a neglegible difference in throughput regardless of $Q$. As we increase $N$, the value of $Q$ becomes more important. After an array size of $2^{20}$ we start seeing throughput increases with Q and reaches the max measured throughput at $Q = 30$. Most significantly, for some values ($Q = 16, 32$) we see a collapse in performance. The reason for this, comes from the interesting way that the GPU handles shared memory. We see this low performance because we are experiencing shared memory bank conflicts caused by the fact that each thread accesses shared memory through different memory banks, and the ids for these banks are computed by dividing the memory address by 32. Thus, by setting $Q = 32$, all 32 reads/writes to shared memory performed by a thread is done sequentially as they all end up using the same memory bank.

In figure 4b we see that varying $B$ changes the throughput measured more drastically than changing $Q$. While the range in $B$-values is shorter, even when making larger steps along the $Q$-values we don't see as drastic a change. Specifically, For any $Q \geq 10$ if we change the $Q$ the throughput stays relatively fixed, while a change in block size can alter the throughput noticeably. This is likely due to the fact that by increasing the block size we are simply able to revert many more computations away from global memory and into shared memory, and due to the fact that scanning inside a block is quite efficient with our implementation.

# 6 Conclusion

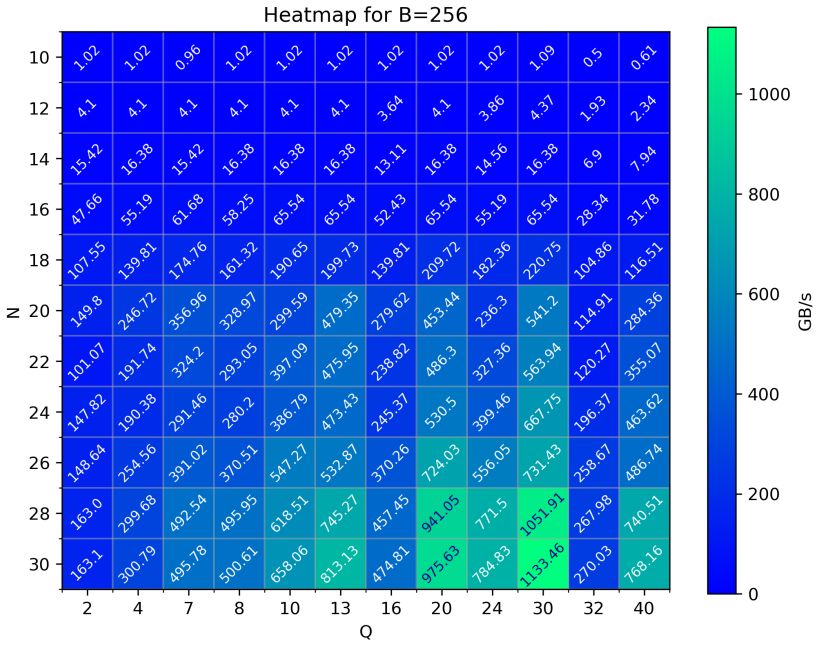
This report covers our venture into implementing an efficient single pass scan. Utilizing shared memory and registers to handle intra-

block scans, along with different methods of handling inter-block computations, we've implemented different single pass scan methods which we've experimented with. We've experimented with different block sizes, number of registers per thread, and the array size. We find that our implementations verify when compared to a basic sequential scan. We find that our implementation of parallel look-back scan achieves up to 84% of the throughput of a naive memcpy implementation with a sequential implementation not far behind, and that our auxiliary block implementation achieves terrible results. The highest throughput we achieve is 1133.64 GB/s with a block size of 256 and a per thread register count of 30. We also find that Varying block size can be essential to the throughput of the method.




(a) Heatmap of different block sizes versus array length for $Q = 30$.

(b) Heatmap of different block sizes vs Q for array length $= 2^{30}$




(c) Heatmap of different Q's vs array length for block size $= 256$

Figure 4: Heatmaps made by fixing one of $Q, B, N$ and varying the others.

## References

[1] Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled look-backs. 2016.

# 7 Appendix

# A Code for the sequential look-back scan function

Listing 1: look-back kernel

```
template<typename T>
__device__ inline T look-backScan(volatile T* agg_mem,
                                  volatile T* pref_mem,
                                  volatile uint32_t* flag_mem,
                                  T* shr_mem, uint32_t dyn_idx,
                                  uint32_t tid) {
    #define X 0
    #define A 1
    #define P 2
  // Handle look-back differently depending on dynamic id.
    if (tid == B - 1 && dyn_idx == 0) {
    T agg_val = shr_mem[(Q - 1) * B + tid];
    agg_mem[dyn_idx] = agg_val;
        pref_mem[dyn_idx] = agg_val;
        __threadfence();
        flag_mem[dyn_idx] = P;

    } else if (tid == B - 1 && dyn_idx > 0) {
    T agg_val = shr_mem[(Q - 1) * B + tid];
        agg_mem[dyn_idx] = agg_val;
        __threadfence();
        flag_mem[dyn_idx] = A;
        uint32_t grab_id = dyn_idx - 1;
        // isnt there a bug here when we encounter an X?
        while (flag_mem[grab_id] != P) {
            if (flag_mem[grab_id] == A && grab_id > 0) {
                agg_val = agg_mem[grab_id] + agg_val;
                grab_id--;
            }
        }
    pref_mem[dyn_idx] = agg_val + pref_mem[grab_id];
        __threadfence();
        flag_mem[dyn_idx] = P;
    }

  __syncthreads();
  T prefix = pref_mem[dyn_idx] - agg_mem[dyn_idx];

    return prefix;
}
```

## B Code for the parallel look-back function

Listing 2: parallel look-back function

```
template<typename T>
__device__ inline T look-backScanWarp(volatile T* agg_mem,
                     volatile T* pref_mem,
                     volatile uint32_t* flag_mem,
                     T* shr_mem, uint32_t dyn_idx,
                     uint32_t tid) {
    // first we define some usefull notions.
    uint32_t lane = tid & (WARP - 1);
    uint32_t look_idx = dyn_idx;
    int k = lgWARP;
    T agg_val = shr_mem[Q * B -1]; // The aggregate value for this block.
    // Some shared memory usefull for doing the reduce of the
    // flag/aggregate/prefix arrays.
    volatile __shared__ uint32_t shrFlag[WARP];
    volatile __shared__ int32_t shrVal[WARP];
    volatile __shared__ int32_t prefVal; // set to the result of the
       reduce
  // Handle look-back differently depending on dynamic id.
    // If block 0 just set the prefix value to be the aggregated result.
    if (tid == B - 1 && dyn_idx == 0) {
    pref_mem[dyn_idx] = agg_val;
        __threadfence();
        flag_mem[dyn_idx] = P;
    // If not block 0 we update the aggregate array.
    } else if (tid == B - 1 && dyn_idx > 0) {
        agg_mem[dyn_idx] = agg_val;
        prefVal = 0;
        __threadfence();
        flag_mem[dyn_idx] = A;
    }
    // Block 0 can return 0 already
    if (dyn_idx == 0) return 0;
    // Otherwise we need to loop, where we do the reduction over the
    // arrays. and calculate the prefix value. Saved in PrefVal
    do {
        // only the threads in the warp should be used.
        if (tid == lane){
            // Select the n lanes such that id we are in block n+1 then
               we
            // at max need n lanes to calculate the prefix. in the
               Reduction.
            if (((int32_t)look_idx-(int32_t)lane) > 0) {
                // First we copy the flag and values from global to the
                   shared
                // memory we allocated earlier.
                // For this we use grab_id to know from where in global
                   memory we read
                // and put_id for where in shared memory we write it.
                // we do some calculations since lane 0 reads the element
```

```
45                  // just before this block, and so on, but we therefore
                        want the last lane
46                  // that reads a value to put it's value into index 0 in
                        shared memory.
47                  int32_t grab_id = (look_idx-1) - lane;
48                  int32_t put_id = min((look_idx-1),WARP-1)-lane;
49                  while (flag_mem[grab_id] == X) {/*wait until we do not
                        read an X*/ }
50                  // Copy the value over.
51                  if (flag_mem[grab_id] == A){
52            shrVal[put_id] = agg_mem[grab_id];
53            __threadfence();
54                        shrFlag[put_id] = A;
55                  } else if (flag_mem[grab_id] == P){
56            shrVal[put_id] = pref_mem[grab_id];
57            __threadfence();
58                        shrFlag[put_id] = P;
59                  }
60              }
61              // If we were not a lane that should copy, just write 0, ie.
                    Neutral element.
62              else{
63          shrVal[lane]  = 0;
64          __threadfence();
65                  shrFlag[lane] = X;
66              }
67          }
68          // After the values has been copied to shared memory sync up the
                threads.
69          __syncthreads();
70          #pragma unroll
71          // We then do a loop to do the reduce, see reduce in PBBKernel or
                warpScan above.
72          for (int d = 0; d < k; d++) {
73              if (tid == lane && dyn_idx > 0 && ((int32_t)look_idx-(int32_t
                    )lane) > 0) {
74                  int h = 1 << d;
75                  if (lane % (h<<1) ==0) {
76                      // The operator we use in the reduction, that just
                            takes the second
77                      // value if the P-flag is set, otherwise it sums it
                            up, note that the result is
78                      // kept in the first Values place in the reduction,
                            in contrast to warpScan.
79                      if (shrFlag[lane+h]==P) {
80              shrVal[lane] = shrVal[lane+h];
81              __threadfence();
82                          shrFlag[lane] = P;
83                      }
84                      else { // flag2 = A and flag1 = A or P
85                          shrVal[lane] += shrVal[lane+h];
86                      }
```

```
87                  }
88
89              }
90              // synchronise the threads in each iteration, since the
                    reduction of 2 elements
91              // is used by another lane in the next iteration for further
                    reduction.
92              __syncthreads();
93          }
94          // We add the reduced value to prefVal
95          if (tid == 0){
96              prefVal += shrVal[0];
97          }
98          // and continue the loop, by reducing the look_idx
99          look_idx -= WARP;
100         // and continuing untill we have gotten a P flag.
101         // note that if we ever encounter a P flag in the flag array
102         // then the reduced value will have the P flag set as well.
103     } while (shrFlag[0] != P);
104     // Lastly we update the prefix array with the prefix value.
105
106   __syncthreads();
107
108     if (tid == 0){
109         pref_mem[dyn_idx] = agg_val+ prefVal;
110         __threadfence();
111         flag_mem[dyn_idx] = P;
112     }
113     // And return prefVal which is what we need to add to all elements
114     // in this block.
115     return prefVal;
116 }
```

## C Code for the look-back scan kernel

Listing 3: look-back kernel

```
template <typename T>
__global__ void SinglePassScanKernel2(T* d_in, T* d_out,
                                      const size_t N, int32_t* IDAddr,
                                      volatile uint32_t* flagArr,
                                      volatile T* aggrArr,
                                      volatile T* prefixArr,
                                      bool par_redux) {
    // Allocate shared memory
    __shared__ T blockShrMem[Q * B];
    volatile __shared__ T blockShrBuf[B];

    // Step 1 get ids and initialize global arrays
    uint32_t tid = threadIdx.x;
    int32_t dynID = getDynID(IDAddr, tid);
    uint32_t globaloffset = dynID * B * Q;

    // Step 2 copy the memory the block will scan into shared memory.
    copyGlb2Shr<T>(globaloffset, N, T(), d_in, blockShrMem, tid);

    // Step 3 Do the scan on the block
    // First scan each thread
    threadScan<T>(blockShrMem, blockShrBuf, tid);

    // Do the scan on the block level
    blockScan<T>(blockShrBuf, tid);

    // Save the result in shrmem.
    threadAdd<T>(blockShrMem, blockShrBuf, tid);

    // Step 4 use look-back scan to find the inclusive prefix value
  T prefix = T();
  if (!par_redux) {
    prefix = look-backScan<T>(aggrArr, prefixArr, flagArr, blockShrMem,
        dynID, tid);
  } else {
    prefix = look-backScanWarp<T>(aggrArr, prefixArr, flagArr,
        blockShrMem, dynID, tid);
  }

    // Step 5 Sum the prefix into the scan
    threadAddVal<T>(blockShrMem, prefix, tid, dynID);

    // Step 6 Copy the result into global memory
    copyShr2Glb<T>(globaloffset, N, d_out, blockShrMem, tid);
}
```

# D Code for the auxiliary block scan kernel

Listing 4: Auxiliary scan kernel

```
1  template<typename T>
2  __global__ void SinglePassScanAuxKernel(T* d_in, T* d_out,
3                                          const size_t N, int32_t* IDAddr,
4                                          volatile uint32_t* flagArr,
5                        volatile T* aggrArr,
6                                          volatile T* prefixArr) {
7      // Step 1 get a dynamic id
8      uint32_t tid = threadIdx.x;
9    uint32_t num_blocks = gridDim.x;
10     int32_t dynID = getDynID(IDAddr, tid);
11
12   __syncthreads();
13     // If the first dynamic id, of -1 then we are the prefix block
         instead.
14     // an optimisation might be to let id 0 do it, but it still
         calculates the
15     // first block.
16     if (dynID < 0 && tid == 0) {
17         T prefix = T();
18         for (uint32_t counter = 0; counter < num_blocks - 1; counter++) {
               // 1 block is aux block
19       while (flagArr[counter] == X) {}
20             // Flag should be A
21             T tmp = aggrArr[counter];
22             prefix = prefix + tmp;
23             aggrArr[counter] = prefix;
24             __threadfence();
25             flagArr[counter] = P;
26         }
27     } else if (dynID >= 0) {
28
29         // Step 1.5 calculate some id's and stuff we will use
30         uint32_t globaloffset = dynID * B * Q;
31
32         // Step 2 copy the memory the block will scan into shared memory.
33     __shared__ T blockShrMem[Q * B];
34         volatile __shared__ T blockShrBuf[B];
35         copyGlb2Shr<T>(globaloffset, N, T(), d_in, blockShrMem, tid);
36
37         // Step 3 Do the scan on the block
38         // First scan each thread
39         threadScan<T>(blockShrMem, blockShrBuf, tid);
40
41         // Do the scan on the block level
42         blockScan<T>(blockShrBuf, tid);
43
44         // Save the result in shrmem.
45         threadAdd<T>(blockShrMem, blockShrBuf, tid);
46
```

```
47          // Step 4 Update aggregate array
48          if (tid == B - 1 && dynID < num_blocks - 1) {
49              T res = blockShrMem[(Q - 1) * B + tid];
50              aggrArr[dynID] = res;
51              __threadfence();
52              flagArr[dynID] = A;
53          }
54
55          // Let block 0 calculate the prefix, we wait for it.
56          while (flagArr[dynID] != P) {}
57
58      T prefix = T();
59      if (dynID > 0) {
60            prefix = aggrArr[dynID - 1];
61      }
62      __threadfence();
63
64          // Step 7 Sum the prefix into the scan
65          threadAddVal<T>(blockShrMem, prefix, tid, dynID);
66
67          // Step 8 Copy the result into global memory
68          copyShr2Glb<T>(globaloffset, N, d_out, blockShrMem, tid);
69      }
70      // Step 9 Die!
71  }
```