UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

PMPH final project

Mikkel Willén (bmq419), Matilde Broløs (jtw868)

Single Pass Scan

November 7, 2022

Contents

1 Project description

The goal of this project is to implement a robust single-pass-scan in CUDA. Scans are very important, but slow implementations of scans causes bottlenecks. The common implementation makes $O(4n)$ memory operations, whereas the single-pass-scan should be able to do the work with just $O(2n)$ memory operations. For comparison, the reduce-then-scan approach utilizes $O(3n)$ memory operations.
For this project we try to implement the single-pass-scan algorithm in CUDA. We want to start with an implementation more similar to the chained scan, and then alter this to be the single-pass-scan, thus comparing the runtimes.
In section 2 we describe the single-pass-scan algorithm as presented in the paper [1]. In section 3 we present our implementation of the single-pass-scan, and the considerations that we had along the way. In section 4 we present the performance of our implementation(s), and lastly we conclude in section 5.
Enjoy!

2 The single-pass-scan algorithm

For this project we attempt to implement the single-pass parallel prefix scan with decoupled look-back, as presented in [1]. The algorithm builds upon the chained-scan, which is also a single-pass scan.

2.1 Chained-scan

In the chained scan each thread is assigned a partition of M elements to work on, and the high-level description of the work of eac block can be explained in the following three steps:

1. Each thread in the block reduce its M elements, to find the partition aggregate, and from that the block aggregate is computed. If the block is the first block, this aggregate is published as the inclusive prefx of the block.

2. The block wait for the previous block to have computed its inclusive prefix. When it recieves the value, it computes the block inclusive prefix and publishes it.

3. Each thread now uses the block exclusive prefix to compute the scan of its M elements.

The problem with this approach is latency of signalling between thread blocks. **???**
This approach requires a block to wait patiently while the previous blocks compute their aggregates, and this part of the scan is thus sequential in that only one block at a time can compute theirinclusive prefix. We would like to be able to parallelize the work further, and therefore we look at the *single-pass parallel scan with decoupled look-back*.

2.2 Single-pass parallel scan with decoupled look-back

This algorithm ensures that a partition is not dependent on its immediate predecessor for the prefix result, and thus reduce the propagation latencies. It works firstly by first of all not only publishing the block inclusive prefix when that is ready, but taking

advantage of publication of the block aggregate to speed up the look-back. This means that a block can check if the preceeding blocks have computed either the prefix or the aggregate, and if it finds a prefix it can stop searching, but if it finds an aggregate it can add that to its own exclusive prefix and keep on searching.
Secondly, it implements a parallel look-back where multiple threads can work together to find the block prefix, and thus decrease the runtime.
The algorithm follows six steps:

1. Initializing the partition descriptors. Each partition has a $flag$ which can be either A if the aggregate has been computed, a P if the prefix has been computed, or an X if nothing has been computed yet. Furhtermore is has a value A and P which holds the aggregate and prefix (respectively) as soon as it has been computed. Both A and P is initialized to the neutral element, and the flag is initialized to X.
2. In this step all processors syncronize, to ensure that all descriptors have been set appropriately.
3. Now each processor computes the partition aggregate, publishes this, and sets its flag to A. If the partition is the very first partition, the prefix is also set, and the flag is set accordingly.
4. Now the partition exclusive prefix is found by using the decoupled look-back. This is done by initialising a $block\_exclusive\_prefix$, and then fetching the flag of the predecessor and acting accordingly:
   - If the flag is X, wait for the flag to change.
   - If the flag is A, add the aggregate to the block_exclusive_prefix and loop again, looking at the next predecessor.
   - If the flag is P then add the prefix to the block_exclusive_prefix and stop looping.
5. Now the inclusive prefix for each partition is computed by adding the exclusive prefix to the aggregate, and setting the descriptor flag to P.
6. Lastly, each partition computes the scan of its own elements again, but this time adding the exclusive prefix, and updating the values in global memory.

Here we see that all steps can be computed in parallel, except for step 4, where a partition must wait for its predecessors to compute either the aggregate or inclusive prefix. It is also important that we synchronize in step 2, so that all descriptors will be instantiated when the following steps are executed.

This algorithm should ensure some interesting properties, which makes it faster than other scans. Firstly, the waiting time for each processor will be decreased, assuming that the system is relatively fair. That means that each processor will have computed their aggregates in about the same amount of time, because of fair scheduling. This also means that the aggregates are highly likely to be available to the processors when they are ready for computing their exclusive prefix. Secondly, the look-back is bound by a constant, as there are a finite amount of predecessors that a processor will have to look at, and that the partitions are of constant size. Thirdly, we have accelerated signal propagation. In the chained version, the publication of an inclusive prefix will stop the immediate decendent from waiting, however in this version it is possible that it releases all decendents from waiting.

3 Implementation

For the implementation of the single-pass scan algorithm, our approach was to first implement a chained-scan, and then extend that version into the single-pass scan, as we wanted to compare the results of the two approaches.
Our implementation is split into five different files. We have `sPSKernels.cu.h`, `rTSKernels.cu.h` and `helperKernels.cu.h` which contain the kernels and kernels helper functions that we use in our project.

Kernels and kernel functionality. `sPSKernels.cu.h` contain the single-pass scan implementation, with the various strategies and optimisations that we have attempted.
`rTSKernels.cu.h` contain the implementation of reduce-then-scan, from the second assignment of the course, which we use to compare our results with.
`helperKernels.cu.h` contain some helper functionality that we use in both the single-pass scan and the reduce-then-scan kernel.

Host skeleton. The `hostSkel.cu.h` file contains the skeleton of our implementation. This is also based on the handed out code from the second assignment.
It should be noted that we have different versions of the implementation, and therefore we take as input an integer parameter, ranging from 0 to 3, indicating which version we want to run.

Tests. The `testSPS.cu` file contains the tests of our implementation. We test both the validity and performance of the implementation.
First we compute a sequential scan directly on the CPU, to have a reference solution to compare with our result, in order to ensure validity.
Then we compute the scan using a reduce-then-scan approach, to get something to compare performance with. It is also checked that this vrsion validates, and the runtime in microseconds and the GB/sec is reported.
Lastly we run our implementation, and checks validity by comparing the result with the reference solution, and reports the performance in the same way as for reduce-then-scan.

Makefile. We have provided a Makefile, with which you can compile the program, and run the tests different versions of the impelmentation. To test on a specific version simply run

```
$ make v<version>
```

where $version$ is the number of version you want to run. Eg. for running the tests if version two, run

```
$ make v2
```

in the terminal.

3.1 Skeleton

The way we build up our implementation is by following the algorithm as directly as possible. This means that we follow the six steps of the algorithm, but allow for different versions of step 4. This means that we must first initialise the partition descriptors

of all partitions. We use blocks as our partitions, and let each thread in a block scan over `CHUNK` elements. Now to implement the partition descriptors we use three arrays, one for storing the flags of all blocks, one for storing the aggregate values, and one for storing the prefix values. The flags are either 0, 1 or 2, representing X, A and P respectively. These arrays are allocated in global memory, as all blocks need to be able to access them, and to get the updated values from other blocks. To ensure that the arrays are initialized before step 3, i.e. to ensure the synchronization of step 2, we allocate the two arrays before starting the kernel. This is done in the host skeleton file. Here we also initialise the flag array to be filled with 0s.
Now, after initialisation of these arrays, we start the kernel, giving the arrays as input. To make the order in which blocks compute their results follow the order in which blocks get shcedule ressources for computing, we want to use a dynamically allocated block id, instead of the physical block id. This will ensure that preceeding blocks, i.e. blocks with lower block ids, have been started earlier. The way that we dynamically allocate the block id is by instantiating a `int* block_num` to zero in the host skeleton file, and then passing the pointer to the kernel. Now a block computes its block id by calling an atomic add on the `block_num`, ensuring that the next block will get the next block id. We furthermore ensure that only one thread of the block computes the block id, and that the block id is shared block-wide, as seen in the listing below.

```
volatile __shared__ int block_id;
if (threadIdx.x == 0) {
    block_id = atomicAdd(block_num, 1);
} __syncthreads();
```

Now we are finally ready to move on to step 3: computing and recording the block-wide aggregate. This is done in parallel, by having eavh thread compute the scan of its own chunk elements. Each thread has a local array of chunk elements, which is used for storing the results of the scan over the chunk. This array is ensured to be in local memory, as `CHUNK` is a constant.
Now the thread copies chunk elements from global to shared memory in a coalesced fashion. It can then compute the scan of the chunk elements by a simple sequential loop, and store the values in the chunk array, as seen below.

```
typename OP::RedElTp acc = OP::identity();
#pragma unroll
for (uint32_t i = 0; i < CHUNK; i++) {
    typename OP::RedElTp elm = shmem[shmem_offset + i];
    acc = OP::apply(acc, elm);
    chunk[i] = acc;
}
__syncthreads();
```

Next we want to compute the aggregate of the block. This is done by letting each thread publish their chunk aggregate to shared memory, and then scanning over the array, to get the chunk inclusive prefix (the sum of aggregates up til and including the current chunk). Each thread then again publishes the chunk inclusive prefix to shared memory, so that each thread can fetch their chunk exclusive prefix. Finally each thread initialize their local block exclusive prefix to zero, and we are ready for publishing the block aggregate and finding the block exclusive prefix. The implementation is shown below.

```
// 3.1 last elem of each thread is placed in shmem_buf.
shmem[threadIdx.x] = acc;
__syncthreads();

// 3.2 block-wise scan over the shmem_buff
```

```
acc = scanIncBlock<OP>(shmem, threadIdx.x);
__syncthreads();

shmem[threadIdx.x] = acc;
__syncthreads();

// chunk_exl_prefix is set to be the previous chunks inclusive prefix
typename OP::RedElTp chunk_exl_prefix =
    (threadIdx.x == 0) ? OP::remVolatile(shmem[blockDim.x - 1]) : OP::
    remVolatile(shmem[threadIdx.x - 1]);
__syncthreads();

block_exl_prefix = OP::identity();
```

The next part of the implementation is determined by the version, and both publishes the aggregate values, computes the block exclusive prefix, and publishes the block inclusive prefix. We will present the approach for each version in the next subsections.

After steps 4 and 5 have been computed, we are ready for adding the block exclusive prefix and chunk exclusive prefix to the scan results of each chunk elements, an copying the result to global memory. Again this is done with a simple sequential loop, followed by a coalesced copy from shared to global memory, as seen below.

```
// 3.3 update the per thread-scan with the corresponding prefix
#pragma unroll
for (uint32_t i = 0; i < chunk; i++) {
    shmem[shmem_offset + i] = op::apply(chunk_exl_prefix, chunk[i]);
} __syncthreads();

// 6. write back from shared to global memory in coalesced fashion.
copyfromshr2glbmem<typename op::redeltp, chunk>
    (block_offs, n, d_out, shmem);
```

And now the scan is complete. We will now look at the different versions for completing step 4 and 5.

3.2 Version 0: Chained-scan

The first version is the simple chained-scan version, which does not use aggregate values, and instead just waits for the previous block to have computed its inclusive prefix.
The implementation simply ensures that the first block publishes its aggregate as the prefix, updates the flag to P, and then lets one thread of each block wait for the previous block to compute its prefix. The inclusive is then computed and published, and the exclusive prefix is shared with the rest of the block by placing the value in shared memory.
The code can be seen in the listing below:

```
if (version == 0) { // chained version
    if (threadIdx.x == 0) {
        if (block_id == 0) {
            d_ps[block_id] = chunk_exl_prefix;
            __threadfence();
            d_fs[block_id] = 2;
            chunk_exl_prefix = OP::identity();
            sh_vs[0] = OP::identity();
        } else {
```

```
            uint32_t prev = block_id - 1;
            while(d_fs[prev] != 2) {
                // wait
            }
            block_exl_prefix = OP::apply(block_exl_prefix, d_ps[prev]);
        }
        sh_vs[0] = block_exl_prefix;
    } __syncthreads();
}
```

Finally thread zero publishes the inclusive prefix to global memory. The implementation of this is the same for both version 0, 1 and 2.

3.3 Version 1: Decoupled look-back

This approach is very similar to that of version 0, however now we start to utilise the aggregates for computing the prefixes faster with less blocking.
Version 1 and 2 uses the same implementation for publishing the aggregate values to global memory, and updating the flags. It simply ensures that the first block publishes its aggregate as the prefix, setting the flag to P, and all other blocks publishes its aggregates, setting the flag to A.
Now version 1 implements the decoupled look-back by a simple change to the while-loop of the chained version. The alteration is shown below:

```
while(d_fs[prev] != 2) {
    if (d_fs[prev] == 1) {
        block_exl_prefix = OP::apply(block_exl_prefix, d_as[prev]);
        prev -= 1;
    }
}
```

We see that the difference is that the thread waits for either an aggregate or a prefix, updates the block exclusive prefix accordingly, and either loops again looking at the next preceeding block, or stops looping if it has found a prefix.
Now we have the basic version of a single-pass scan with decoupled look-back. In the next subsection we will take a look at how we attempted to optimise this, resulting in versions 2 and 3.

3.4 Version 2 + 3: Optimisations

The two optimisations that we attempted are bound together in that the optimisation of version 2 is actually a sub part of the optimisation of version 3.
We attempt to make the look-back in step 4 parallel, thus utilizing multiple threads in the block. This is done by the following optimisations:

Version 2 Using a WARP of threads to copy the results of the preceeding WARP blocks into shared memory, and then having one thread to loop through the result, computing the exclusive prefix. If none of the preceeding WARP blocks have completed their prefix, the window is slides down another WARP blocks, continuing until a prefix has been found.

Version 3 Using a WARP of threads to both copy the results into shared memory, and to compute the exclusive prefix by a warp scan approach.

3.4.1 Version 2 - WARP Parallel copying, decoupled look-back.

First we do the simple version, in which only the copy is parallel, but the computation is still handled by just one thread.
To do this we let only the first WARP threads enter a loop, where they continue copying WARP elements from the flag and aggregate/prefix array to shared memory, until the exclusive prefix has been computed. This is done by defining where in shared memory we put flags and values, which are put in `sh_fs` and `sh_vs` respectively.
For each block, thread zero will look through the shared memory and compute the result, breaking out of the loop if a prefix is found, and updating the first element in shared memory with the result so that all threads can fetch it.
The code is as follows:

```
if (version == 2) { // WARP thread copy - single thread lookback:
    int32_t window_offset = block_id - WARP;
    int32_t loop_stop = - WARP;
    if (block_id != 0 && threadIdx.x < WARP) {
        while(window_offset > loop_stop) {
            int lookup_block_id = window_offset + threadIdx.x;
            if (lookup_block_id >= 0) {
                while (d_fs[lookup_block_id] == 0) {}  // wait for some
     result
                uint32_t flag = d_fs[lookup_block_id];
                sh_fs[threadIdx.x] = flag;
                if (flag == 1) {
                    sh_vs[threadIdx.x] = d_as[lookup_block_id];
                } else if (flag == 2) {
                    sh_vs[threadIdx.x] = d_ps[lookup_block_id];
                }
            }

            if (threadIdx.x == 0) {
                // do lookback
                int i = 0;
                while (i < WARP) {
                    int index = (WARP - 1) - i;
                    uint32_t flag = sh_fs[index];
                    if (flag == 1) {
                        block_exl_prefix = OP::apply(block_exl_prefix,
                                                     sh_vs[index]);
                        i++;
                    } else if (flag == 2) {
                        block_exl_prefix = OP::apply(block_exl_prefix,
                                                     sh_vs[index]);
                        window_offset = loop_stop;
                        i = WARP;
                    }
                }
            }
            window_offset -= WARP;
        }

        if (threadIdx.x == 0) {
            sh_vs[0] = block_exl_prefix;
        }
    }
    __syncthreads();
}
```

We do not need to worry about syncronization of threads, as the first WARP threads execute in lock step.

3.4.2 Version 3 - WARP parallel copying and parallel decoupled look-back.

Now we attempt to also make the computation of the exclusive prefix in parallel, exploiting the first WARP of threads. This version has publication of aggregates and prefixes built-in, thus not sharing implementation with the other versions.

For this we still copy from memory as in version 2, but now we use the approach of a warp scan to compute the result (using slides `Lab2-RedScan.pdf`). The code is also inspired by the handed out code for assignment 2, which defines a warp inclusive scan, and the implementation in opencl from [2]. The only difference is that we do not want to reduce unconditionally, as we want to check the flag before applying the operator on two elements. This is done by, when looking at two threads with ids tid and $tid - h$, checking the flag according to tid, and if the flag is already set to P we do nothing. If not, we apply the operator to the values of the two threads, using the result as the new value for thread with tid, and updating the flag to be that of $tid - h$. This makes sense as, if thread $tid - h$ has already computed its prefix, the flag will be propagated. The implementation of the parallel warp scan is seen below.

```
// WARP-Level inclusive scan
if (sh_fs[WARP - 1] != 2) {
    const uint32_t lane = threadIdx.x & (WARP - 1);
    #pragma unroll
    for(int d = 0; d < lgWARP; d++) {
        uint32_t h = 1 << d;
        if (lane >= h) {
            uint32_t flag1 = sh_fs[threadIdx.x];
            uint32_t flag2 = sh_fs[threadIdx.x - h];
            typename OP::RedElTp value1 = sh_vs[threadIdx.x];
            typename OP::RedElTp value2 = sh_vs[threadIdx.x - h];
            if (flag1 == 1) {
                sh_fs[threadIdx.x] = flag2;
                sh_vs[threadIdx.x] = OP::apply(value1, value2);
            }
        }
    }
}
```

When determining whether to loop on, each thread must check the flag of the $WARP-1$ element in shared memory, as this indicates whether we need to loop on. Also, the block exclusive prefix is updated with the value of $WARP - 1$, so that the result is gradually accumulated.

4 Performance

Now we will present the validity and performance of our solution, reporting the runtime of the different versions, and discussing the robustness of the implementations. We run all our tests on the GPU4.

4.1 Expectations

Before we look at the actual numbers, we want to discuss our expectations for the different versions.
We expect the chained-scan to perform significantly worse than the single-pass scan decoupled look-back versions. We would assume that version 2 would run faster than

version 1, as we parallelize parts of the implementation, and expect version 3 to run much faster than the other versions, as it parallelizes the entire look-back and computation part of the algorithm.

4.2 Validity

We determine whether our solutions are viable by comparing the result to a sequential scan performed on the CPU. In Table 1 we report the validity of the different versions.

array length \ version	version 0	version 1	version 2	version 3
5003565	*valid*	*valid*	*valid*	*valid*
50003565	*valid*	*valid*	*valid*	*valid*
500003565	*valid*	*valid*	*valid*	*invalid*
1000003565	*valid*	*valid*	*valid*	*invalid*

Table 1: Validity of the different versions, at a certain array length, on GPU4

From the table we see that the different version are quite robust, validating on different sizes of arrays, with the exception of version 3. It seems we get a race condition somewhere, that we have not been able to locate, and the possibility of the race condition occuring increases with the number of elements in the array. It also sometimes happens for smaller array sizes, but most of the time it succeeds for these smaller arrays.

4.3 Performance

Now lets look into the runtime of the different versions. We compare to a reduce-then-scan approach, to see how the solution performs in comparison.

array length \ version	reduce-then-scan	version 0	version 1	version 2	version 3
5003565	137	3047	383	644	380
50003565	1230	20037	1597	15321	1005
500003565	11996	208068	12109	1518315	–
1000003565	23924	418624	23360	5410449	–

Table 2: Runtime of the different versions measured in microsecs, given a certain array length, on GPU4

From Table 2 we see that version 0 runs quite slowly for all array sizes, as one would expect. Version 1 runs quite fast, showing the improvement in runtime from chained-scan to single-pass scan with decoupled look-back. In contrast to what we expected, version 2 runs rathler slowly, but still not as slowly as the chained-scan version. Lastly version 3 runs very fast, but sadly does not validate on larger array sizes.

We also want to take a look at the throughput, to compare to the device throughput. In Table 3 the throughput, measured in GB/s, is presented. The device throughput of GPU4 is 616 GB/s.

array length \ version	reduce-then-scan	version 0	version 1	version 2	version 3
5003565	438.27	26.53	142.62	92.23	156.77
50003565	438.27	19.71	142.62	93.23	597.06
500003565	500.17	28.84	495.5	3.95	–
1000003565	500.84	28.67	513.7	2.22	–

Table 3: Runtime of the different versions measured in GB/s, given a certain array length

From Table 3 we see that for sufficiently large array sizes, version 1 gets throughput values very similar to reduce-then-scan. The reduce-then-scan approach is relatively stable, whereas our implementations are more unstable. It furthermore shows that version 3 almost reaches device throughput when it validates, but at aforementioned it does not always validate, and we have never seen it validate of larger array sizes. Table 4 shows the effeciency in percentage of device throughput.

array length \ version	reduce-then-scan	version 0	version 1	version 2	version 3
5003565	71%	4.3%	23.15%	15%	25.4%
50003565	71%	3.2%	23.15%	15%	96.9%
500003565	81.2%	4.7%	80.4%	0.64%	–
1000003565	81.3%	4.65%	83.4%	0.36%	–

Table 4: Percentage of device throughput, for the different versions

This table makes it clear that version 3 is extremely efficient when working, but it is also very unstable. Version 1 has a good percentwise throughput, especially when the array is larger, whereas version 2 gets slower when the array size increase.

An interesting, and somewhat depressing, detail is that running the implementation on GPU3 or A100 does not work. Somehow our solution is not robust, and therefore we only test on GPU4.

5 Conclusion

From the results of examining validity, performance, and robustness of our different implementation versions, we see that our implementation of the unoptimized single-pass scan with decoupled look-back is quite efficient and robust, as it validates of different array sizes, and computes the result fairly quickly.
Against our expectations, version 3 does not perform very well, which we assume is due to a bug in our code. We expected it to be more robust than it was.
All in all we did not get our implementation to be as robust as we wanted, as we could not run on other GPUs, and could not make the optimizations work as intended.
For future work it would be good to ensure that the implementation runs on more GPUs, and to figure out why version 3 sometimes fails on larger array sizes. It would also be nice to try to implement more of the optimizations from the article.
However, the unoptimized version was more effective than we had expected, and we are quite content with the performance of that version, even though we would have liked the others to perform better.

References

[1] Duane Merrill and Michael Garland. “Single-pass Parallel Prefix Scan with Decoupled Lookback”. In: 2016.

[2] *OpenCL repository*. `https://github.com/coancea/OpenCL-Repo/blob/master/Examples/Scan/SinglePassScanKer.cl`. Accessed: 2022-11-07.