Abstract

This thesis presents an implementation of a single-pass scan algorithm described by researchers Merrill & Garland, as an extension to the functional language Futhark. The work consists of a generalization of a prior implementation made by Persson & Nicolaisen, modified to work on regular segments.
In addition to generalization of the implementation, two major contributions in the form of *an analytical model to estimate optimal workload per thread based on type analysis* and *safe rewriting of index arithmetic to computationally cheaper calculations* are made.
The implementation and contributions are tested, and their respective benefits are documented using Futharks built-in benchmarking system.

UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's Thesis

Morten Clausen

Regular Segmented Single-pass Scan in Futhark

Supervisor: Cosmin Oancea & Troels Henriksen

Submission: 31st of May, 2021

Contents

1 Introduction & Motivation **3**
1.1 The Scan Operator . . . 3
1.2 Segmented Scan . . . 4
1.3 Contributions . . . 5

2 Background **6**

3 Preliminaries **7**
3.1 Nvidias Single-Pass Scan Algorithm . . . 7
3.2 Futhark . . . 9

4 Baseline algorithm **10**
4.1 Glossary . . . 10
4.2 Load & Map . . . 10
4.3 Transposition . . . 11
4.4 Thread-Level Scan . . . 11
4.5 Block-Level Scan . . . 12
4.6 Lookback Phase . . . 13
4.7 Result Distribution . . . 15
4.8 Initial Results . . . 16

5 Sequentiality Degree **19**
5.1 Analytical Model for Shared memory . . . 20
5.2 Analytical Model for Register memory . . . 21
5.3 Formula for Sequentiality Degree . . . 22
5.4 Benchmarks . . . 23

6 Optimizing Index Calculations **27**
6.1 Thread-level scan . . . 28
6.2 Block-Level Scan . . . 29
6.3 Lookback . . . 30
6.4 Result Distribution . . . 30
6.5 Benchmarks . . . 33

7 Implementation **35**

7.1 Determining Sequential Work 35

7.2 Load & Map and Transpose 36

7.3 Thread-Level Scan 36

7.4 Block-Level Scan 37

7.5 Lookback Phase 38

7.6 Result Distribution 41

8 Empirical Validation **42**

9 Conclusion & Future Work **44**

A Source Code **46**

B Benchmarked Programs **52**

B.1 LSSP 52

B.2 Radix-Sort 53

B.3 KD-Tree 54

1 Introduction & Motivation

1.1 The Scan Operator

A scan applies a binary associative operator $\oplus$ and a neutral element $\mathtt{ne}$ on a collection $[x_0, \ldots, x_n]$, resulting in a new sequence [2]:

$$\mathtt{scan} \oplus \mathtt{ne}\ [x_0 \ldots x_n] = [\mathtt{ne} \oplus x_0,\ \mathtt{ne} \oplus x_0 \oplus x_1, \ldots,\ \mathtt{ne} \oplus_{i=0}^{n} x_i]$$

Scan plays a fundamental role in parallel programming, as Blelloch has shown how to use it as a building block for parallel programs [1]. He provides examples of applications relying on scans to perform the majority of computations, such as radix-sort, quick-sort, the merging step of merge-sort as well as the construction of minimum spanning trees. With scan serving as a fundamental building block, it is desirable to have an efficient parallel implementation of the primitive.

One potential implementation is the *scan-then-propagate* strategy, which is illustrated in figure 1. The collection is divided into groups, and the scan is done in stages. During the first stage, each

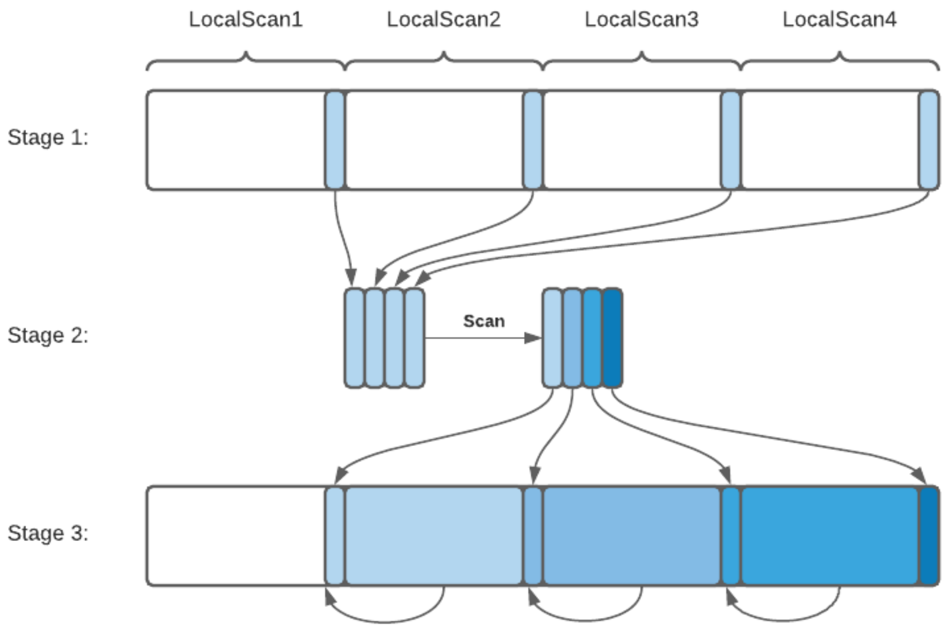


Figure 1: Overview of a two-pass parallel scan algorithm

group performs a local scan and writes this partial result to global memory. In the second stage, the last element from each group's local scan is collected, scanned and distributed back to their original position. The last element in each group now has the expected value. In the third and final stage, each group - except the first - reads the last value of the preceding block, and combines it with the partial values written in stage 1, which results in the final values for all elements.

This strategy, while clearly parallel, has the drawback of doing two passes over the scanned collection. Additionally, during stage 1 each element is read and a partial result is written, which is then read again in stage 3 before being combined and written back again. This results in ~$4n$ global memory movements (~$2n$ reads, ~$2n$ writes) during the scan. Comparing this to a purely sequential version which is able to perform the scan in a single pass, the optimal is ~$2n$ total memory movements. It is possible to achieve ~$3n$ memory movements by skipping the write-back of partial results in stage 1, but this makes fusing multiple operations more cumbersome, as the local scan would have to be performed again in stage 3. This essentially doubles the amount of operations each group performs, and is not an absolute improvement to the approach with ~$4n$ movements.

One strategy to achieve a single pass and ~$2n$ movements is the *chained-scan* approach where the input is again split into groups, and like in stage 1 of the *scan-then-propagate* strategy, each group performs a local scan. Instead of writing the partial result to global memory, each group waits on the previous group to calculate the final value of the last element. This value is then combined with the local scan, and the result is written to global memory. This results in the desired ~$2n$ global memory movements, but introduces a serial dependence between the groups. This extra latency can severely hinder throughput. To combat this, Merril & Garland has developed a generalized *chained-scan* with *decoupled-lookback*[6]. In this approach, instead of waiting, each group looks at the results of the predecessors until it either has enough partial results to calculate the final result, or finds a group that already has a final result available.

1.2 Segmented Scan

Sometimes it is beneficial to split the scanned collection into segments, such that the scan restarts at the beginning of each new segment. This plays an important role in flattening programs with nested parallelism, as each inner workload can be treated as a segment and turned into a flat representation with a segmented operation.

In the case of segmented scan, the scan operator also takes a collection of flags as argument, which specifies the beginning of segments.An example would be multiplication over a nested array `[[2,3],[4],[4,5]]`, which is converted to a flat array to avoid irregular parallelism, along with a flag array:

```
let xs = [2,3,4,4,5]
let flags = [true, false, true, true, false]
in sgm_scan (*) 1 xs flags -- [2,6,4,4,20]
```

The flags are constructed in such a way that `flag[i] == true` when `xs[i]` is the first element in a segment. A segmented scan can be expressed as a regular scan by modifying the operator:

$$\texttt{sgm\_scan} \oplus \texttt{ne xs flags} = \texttt{fst \$ unzip \$ scan} \oplus' (\texttt{ne}, \texttt{false})\ \texttt{zip}(\texttt{xs}, \texttt{flags})$$
$$\textit{where}$$
$$(v_1, f_1) \oplus' (v_2, f_2) = (\texttt{if}\ f_2\ \texttt{then}\ v_2\ \texttt{else}\ v_1 \oplus v_2, f_1 \vee f_2)$$

If the segments are all of equal length, the flags can however be omitted and instead inferred from the segment length. The segmented scan is then referred to as *regular*. Omitting the flags means saving memory operations, as otherwise each elements would require ~$2n$ reads along with ~n writes. Regular segmented scans naturally occur when the scan operator is combined with a map. For instance scanning each row of a matrix can be achieved with: `map (scan ⊕ ne) matrix`, which corresponds to a regular segmented scan.

1.3 Contributions

The goal of this thesis is to implement a chained-scan with decoupled lookback as described in section 1.1, in the compiler for the functional array language Futhark[1], such that it works on regular segmented scans mentioned in section 1.2.

An existing implementation done by Persson & Nicolaisen already exists, but is limited to non-segmented scans[8]. The Futhark compiler thus defaults to a two-pass approach when compiling segmented scans.

The contributions to the scan primitive in Futhark are therefore as follows:

1. Modify the existing single-pass scan implementation to also work on regular segments. The implementation is done in such a fashion where it does not require more memory movements than the non-segmented version.

2. Deriving a method for dynamically choosing the amount of sequential work each thread should perform. Having each thread perform some sequential work minimizes inter-thread communication, but the optimal amount depends on the type of operator provided for the scan.

3. Optimize index calculations in order to minimize the number of 64-bit modulo operations. Calculating relative indices within segments is most easily done by applying the modulo operator along with the size of segments, but this is a costly operation. Changing these calculations while preserving the original semantics thus results in better performance.

[1] `https://futhark-lang.org/`

2 Background

Since the invention of digital computers, computing power has been a valuable resource. Great efforts have been made to increase the amount of power per cost, as shown in an article by Moore [7], which later resulted in the now famous "Moore's law". This continuos doubling of transistor density lead to powerful single-processor architectures being the prevalent industry standard. However, as the data in figure 2 shows, we have reached a point where doubling the number of transistors no longer doubles performance[2].

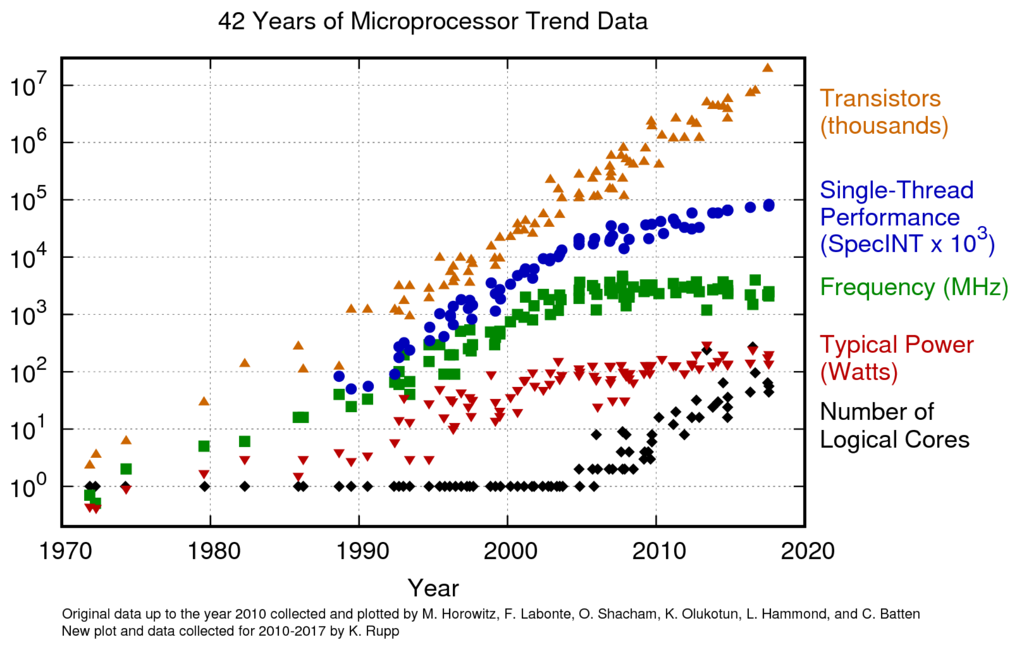


Figure 2: Data borrowed from K. Rupp

This trend have led to alternative architectures being more attractive, with multi-core processing or even massively parallel hardware becoming more common. To take full advantage of a parallel architecture, a different approach to programming is necessary, as reasoning about parallel and sequential code is inherently different. One model of approach as suggested by Blelloch, is to use scan operations as primitive building blocks[1]. This has led to a parallel programming paradigm where the programmer can abstract away from the low-level details of thread management, and instead think of parallel programs as combinations of the parallel primitives.

With this approach, the primitives becomes the performance bottleneck, and increasing performance is a question of implementing the primitive as efficiently as possible.

[2]https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

3 Preliminaries

3.1 Nvidias Single-Pass Scan Algorithm

Futhark is able to compile code using different backends. One of those is Nvidias parallel platform CUDA. CUDA uses a specific notion of work groups by grouping threads together in warps, which is further grouped together in blocks. This allows algorithms to be specified through what work should be done at the thread-level, and what communication should happen at each grouping layer. Nvidia researchers Merrill & Garland have developed such an algorithm for the scan operation, expressed in CUDA work groups.

The algorithm is based on a *chained-scan*, where each block is assigned a slice of the collection to be scanned. Each block then computes a local scan result of its given slice, and waits for its predecessor to communicate the value of the last element in its own scan result. This notion of predecessor is based on assigning a dynamic id to blocks as they start execution. Not all blocks are active at once, and cannot be expected to be scheduled in a predetermined order. In order to reduce latency when propagating results between blocks, it is beneficial to arrange for blocks close in time to also be close in space. As such, when a block spawns it is atomically assigned an id, which means it will work on the slice immediately following the slice of the block with $id - 1$, and the dynamic id is incremented.

When a block has received the value of the last element from the previous block, this value is combined with the local scan result, and the block can send the value of its last element to the next block. This solution can lead to high latency for the blocks assigned to the end of the collection, which is why Merrill & Garland propose a *decoupled lookback* procedure to resolve this issue:

1. Each block is assigned 3 fields:
    - Aggregate: The value of the last element in the slice after performing the local scan.
    - Prefix: The combined value of the current block's aggregate and the aggregates of the previous blocks.
    - Status: A flag having one of 3 values:
        - A: aggregate has been computed and is available
        - P: prefix has been computed and is available
        - X: No value is available, starting status of all blocks
2. Each block performs a local scan, and saves the value of the last element in the aggregate field, then sets its status to A. The first block ($id = 0$) copies the aggregate value to prefix and sets status to P, and jumps to step 5.
3. Each block performs a lookback to calculate an exclusive prefix. Starting at the immediately preceding block, the current block inspects the status and conditionally does the following:
    - X: Block until the status changes to not X.
    - A: The aggregate value of the inspected block is added to the exclusive prefix, and the lookback continues to the block preceding the inspected block.
    - P: The prefix of the inspected block is added to the exclusive prefix, and the lookback is terminated.

4. Each block combines the exclusive prefix with its aggregate and saves the result in the prefix field. The status is then set to P.

5. Each block combines the exclusive prefix with the elements obtained from the local scan, to finally create the total scan.

This decoupled lookback minimizes latency by having each block calculate the exclusive prefix on their own, removing the serial dependence between blocks.

3.2 Futhark

The actual implementation of an efficient segmented scan algorithm will happen through the expansion of the compiler for the functional language Futhark[5].

As part of the compiling, code is transformed in order to optimize the resulting program. One of these transformations is the act of fusing maps and scans together. The fusion happens when a map and a scan is applied to the same collection. A piece of code like:

```
let ys = map (+1) xs
in scan (*) 1 ys
```

can be turned into:

```
scanomap (*) (+1) 1 xs
```

The `scanomap` is semantically equivalent to:

$$\texttt{scanomap} \oplus \ \texttt{f ne xs} \equiv \texttt{scan} \oplus \texttt{ne} \ (\texttt{map f xs})$$

The benefit to combining the scan and the map into a single primitive is that it limits the amount of memory movements required, as the intermediate result is not manifested. This `scanomap` construct thus means that the implementation also has to handle potential mappings.

Note that every `scan` can be viewed as a `scanomap` where the mapping is the identity function, thus an implementation of `scanomap` is general enough to serve as an implementation of a `scan` with no mapping. The creation of the `scanomap` primitive is a case of horizontal fusion, where the output of one operation becomes the input of another. Futhark also does vertical fusion, meaning maps and scans on the same source collection are fused. This means if we have a `scanomap` $\oplus$ `f ne` and a `map g` on the same collection, the implementation must have a workflow corresponding to the illustration in figure 3.

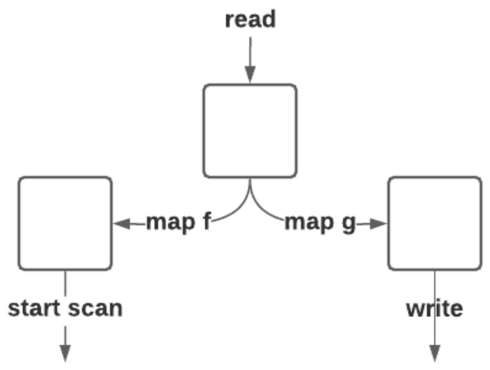


Figure 3: Handling of input during vertical and horizontal fusion of maps and scan

This again reduces the number of memory movements, as the input collection only needs to be read once before the map and the scanomap starts computation.

4 Baseline algorithm

4.1 Glossary

As mentioned in section 1.3, each thread will perform some sequential work on a number of elements, which we will refer to as m. Other relevant constants relating to the model of thread blocks include:

Constant	Semantic
`m`	The number of elements each thread will operate on at a given time.
`ne`	The neutral element used during the scan.
`dynamic_id`	The id assigned to blocks as they spawn, imposing a temporal ordering on blocks. The first block to spawn is assigned 0, and the id is incremented for each following block.
`block_size`	The number of threads within a single block
`thread_id`	The local id of a thread in regards to the block. A value between 0 and `block_size-1`.
`block_offset`	The global index at which a block starts, in regards to the entire input collection.
`sgm_size`	The size of the regular segments in the input collection. For multi-dimensional arrays, this corresponds to the size of the inner-most dimension.

Table 1: Glossary of constants values used in the single-pass algorithm

4.2 Load & Map

The fused construct illustrated in figure 3 has already been handled in the implementation of the non-segmented single-pass scan by Persson & Nicolaisen. Performing mappings can be done in exactly the same way whether the mapped collection is segmented or not. This part of the algorithm does thus not differ from the existing implementation, but is still included for completeness.

For simplicity we will use `float` as the data type in the pseudo-code, even though this in reality is dependant on the map function.

```
float[] local = new float[m];
int block_offset = dynamic_id * m * block_size;
for(int i = 0; i < m; i++){
  int phys_tid = block_offset + threadId + i * block_size;
  if(phys_tid < input.length)
    float elem = input[phys_tid];
    float map_out = g(elem);
    float scan_in = f(elem);
    g_dest[phys_tid] = map_out;
    local[i] = scan_in;
  else
    local[i] = ne;
}
```

The fused construct is implemented by having a single read per element on line 6, with line 7+9 handling the vertical fusion by applying `g` and writing the result back, while line 8+10 handles the horizontal fusion by applying `f` and keeping the result in local memory.

One thing to note about the calculation of `phys_tid` on line 4 is that the thread does not calculate m consecutive indices. Rather, each thread calculates and accesses m indices with a stride of `block_size`. This results in the entire block reading `block_size` consecutive elements from global memory each loop iteration, which ensures coalesced access and improves throughput[3].

4.3 Transposition

As a result of the memory coalescing shown in section 4.2, each thread now has m non-consecutive elements in local memory, which is not useful for computing the scan result. To group consecutive elements together in the same thread, shared memory is used as a staging buffer for the threads to transpose the elements. The transposition is carried out like so:

```
for(int i = 0; i < m; i++){
  int shared_idx = threadId + i * block_size;
  shr_mem[shared_idx] = local[i];
}
barrier();
for(int i = 0; i < m; i++){
  int shared_idx = threadId * m + i;
  local[i] = shr_mem[shared_idx];
}
```

The first loop places elements into shared memory with a stride of `block_size`, similar to how the load in section 4.2 was carried out. This then means the shared memory contains a slice of the transformed input with the ordering preserved, and thus the second loop can simple read from shared memory using a stride of 1. As a result, the local memory now contains a slice of m consecutive elements as needed.

4.4 Thread-Level Scan

With `local` containing consecutive elements, each thread can now perform a sequential scan on the m elements. Since this scan might cross between segments, each thread has to calculate the global indices of the elements they scan. If the global index of an element corresponds to the start of a new segment, it is not combined with the previous value, effectively restarting the scan.

```
int gIdx = block_offset + threadId * m;
for(int i = 1; i < m; i++){
  bool new_sgm = gIdx+i % sgm_size == 0;
  if(!new_sgm)
    local[i] = ScanOp(local[i-1], local[i]);
}
shr_mem[threadId] = local[m-1];
```

[3] https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

The last resulting element after the local scan is published in shared memory on line 7, in order to prepare for a block-level scan. Also note that the loop starts with `i=1`, as the first element in the local array has no predecessor to be combined with.

4.5 Block-Level Scan

At the block-level, the results each thread published in shared memory is scanned again. The goal is to obtain an accumulator value for each thread, which can be combined with the local results to obtain a partial scan of the entire block.

This scan in shared memory is done using the pre-existing `groupScan` construct, which needs a predicate, a scan operator and the array to be scanned. This predicate takes two indices and returns true whenever the two indices belong to different segments.

The `groupScan` can utilize any block-wide scan strategy to carry out the scan, but we will assume the used strategy is a scaled down version of the *scan-then-propagate* strategy mentioned in section 1.1. That is, the shared memory is divided into sub-blocks with length equal to the size of a warp. Each warp then performs a scan like stage 1 in figure 1. The first warp then collects the last element from the other warps, temporarily moving its own results to local memory. The first warp then scans the collected results and distributes them back (like stage 2), before restoring its own results from local memory. Each warp them combines the distributed result with its own scan result to obtain the final value (like stage 3). Since this scan is performed entirely in shared memory, the penalty of doing two passes is negligible.

The `groupScan` needs a way to determine if two elements belong to different segments. This is most easily done by converting the indices of the elements in shared memory back to their original global indices.

A formula for determining whether two global indices i_1, i_2 belong to different regular segments is $i_2 - i_1 > i_2$ `%` `sgm_size`, when $i_1 \leq i_2$. The elements being scanned are however results from slices of length m, so the indices have to be converted by multiplying with this length, and adding the `block_offset`.

```
 1  bool crossesSegment(int idx1, int idx2){
 2    int end   = idx2 * m + block_offset + m - 1;
 3    int start = idx1 * m + block_offset + m - 1;
 4    return end - start > end % sgm_size;
 5  }
 6  groupScan crossesSegment ScanOp shr_mem;
 7  float acc;
 8  if (threadId == 0)
 9    acc = shr_mem[grp_size-1];
10  else
11    acc = shr_mem[threadId-1];
```

After performing the block-level scan on lines 1-6, each thread reads the value at the position of its predecessor (line 11). The exception is the first thread which has no predecessor. This thread instead reads the last value of the entire block, as this is the `aggregate` value described in section 3.1.

At this point, each thread could combine the accumulator value from the `groupScan` with its own local values, but this action is delayed until after the lookback has been performed.

4.6 Lookback Phase

The lookback functions as described in section 3.1. Extra cases regarding segments can be added to optimize the lookback, in addition to the cases required for correctness. Originally, the first block would skip the lookback and immediately publish its result as a prefix, since it had no preceding blocks. This can be expanded to include any block which perfectly overlaps with the beginning of a new segment. That is, any block where the first element is also the first element of a segment can skip the lookback phase. This is handled on lines 2-7 in the code below. The variables `aggrs, prefixs` and `statuss` are global arrays containing the aggregates, prefixes and status flags of the blocks.

Furthermore, blocks that overlap segment boundaries still need the prefix from the preceding block, but the successor blocks do not. This means boundary-overlapping blocks can also immediately publish their result as a prefix before starting the lookback in order to speedup the lookback for other blocks. This is handled on lines 11-16 and 73-75 by calculating the distance to the next segment boundary (`sgm_size - sgm_idx`) and comparing it to the number of elements treated by the block.

During the lookback itself, only the first warp will be used to calculate the required prefix, as shown in line 9. `exchange` and `warpscan` are fixed arrays in shared memory which contains the aggregates and the flags of the blocks visited by the warp during each iteration of the loop on line 25. `shr_readOffset` is a singleton array also in shared memory, used to communicate the current lookback offset between the entire warp.

At each iteration of the loop, 32 blocks (size of a warp) are inspected for their aggregates and statuses. When looking at blocks outside the same segment as the block performing the lookback, the result of the block can be ignored and the flag can be treated as `P`, as it is not necessary to look past that block. Thus lines 31-38 ensures that block results are only included when part of the current segment.

The first thread then sequentially reduces the aggregates and flags from lowest dynamic id to highest (line 43-69). When the right-hand flag is `A` the aggregates are simply combined (line 58) and we increment how many aggregates we have used so far (line 59). The `used`-counter determines how far the lookback should be shifted for the next iteration of the loop. If the right-hand flag is not `A`, we keep the right-hand value and flag as the result, and reset the `used`-counter.

After the sequential reduction has been carried out, the thread checks is the resulting flag is `P` and stops the lookback loop if that is the case (line 64-65). Otherwise the lookback is shifted depending on the value of `used`, and the rest of the warp is informed of the new offset. (line 66-68). In any case, the first thread combines the calculated aggregate with its current prefix to obtain either a partial or total prefix, depending on the value of the final flag (line 69).

After the lookback is terminated, the first thread publishes the prefix of the block and sets the status to `P` (lines 72-75), unless the block crosses a segment boundary. In that case, this was already done on lines 11-16. The rest of the threads reads the prefix from the first thread (lines 79-80), and the final values can now be calculated.

```
1   float prefix = 0;
2   int sgm_idx = block_offset % sgm_size
3   bool block_new_sgm = sgm_idx == 0;
4   if(block_new_sgm && threadId == 0)
5     prefixs[dynamic_id] = acc;
6     statuss[dynamic_id] = P;
7     acc = 0;
8   barrier();
```

```
if(!block_new_sgm && threadId < warp_size)
  if(threadId == 0)
    if(sgm_size - sgm_idx >= block_size * m)
      aggrs[dynamic_id] = acc;
      statuss[dynamic_id] = A;
    else
      prefixs[dynamic_id] = acc;
      statuss[dynamic_id] = P;
    warpscan[0] = statuss[dynamic_id-1];
  warp_barrier();
  int status = warpscan[0];
  if(status == P)
    if(threadId == 0)
      prefix = prefixs[dynamic_id-1];
  else
    int readOffset = dynamic_id - warp_size;
    while(readOffset > -warp_size)
      float aggr = 0;
      int flag = X;
      int readI = readOffset + threadId;
      int prev_end = (readI + 1) * block_size * m - 1;
      bool same_sgm = block_offset - prev_end <= sgm_idx
      if(readI >= 0 && same_sgm)
        flag = statuss[readI];
        if(flag == P)
          aggr = prefixs[readI];
        else if(flag == A)
          aggr = aggrs[readI];
      else if(readI >= 0)
        flag = P;
      exchange[threadId] = aggr;
      warpscan[threadId] = flag;
      warp_barrier();
      flag = warpscan[warp_size-1];
      if(threadId == 0) # single-threaded reduce
        if(!(flag == P))
          float agg1 = 0;
          float agg2;
          int flag1 = X;
          int flag2;
          int used = 0;
          for(int i = 0; i < warp_size; i++)
            flag2 = warpscan[i];
            agg2 = exchange[i];
            if(!(flag2 == A))
              flag1 = flag2;
              agg1 = agg2;
              used = 0;
            else
              agg1 = ScanOp(agg1, agg2);
              used++;
          flag = flag1;
          aggr = agg1;
        else
          aggr = exchange[warp_size-1];
        if (flag == P)
          readOffset = -warp_size;
        else
          readOffset -= used;
        shr_readOffset[0] = readOffset;
        prefix = ScanOp(aggr, prefix)
      warp_barrier();
      readOffset = shr_readOffset[0];
  if (threadId == 0)
    if(sgm_size - sgm_idx >= block_size * m)
      prefixs[dynamic_id] = ScanOp(prefix, acc);
      statuss[dynamic_id] = P;
    exchange[0] = prefix;
    acc = 0;
  barrier();
  if (threadId != 0)
    prefix = exchange[0];
```

4.7 Result Distribution

Once the prefix has been calculated, it is time to combine it with the accumulated value from the group scan and the local value from the thread scan. During this final combination of values, there are two cases to consider:

1. The thread scan crossed between segments
2. The group scan crossed between segments

In case 1, it means that the accumulated value from the group scan should only be combined with the local values appearing before the segment crossing. This is handled in line 12-16 by calculating how many elements are left in the current segment, and then skipping the combining step if the loop exceeds that amount.

In case 2, the prefix value obtained from the lookback phase should only be combined with the accumulated value from the group scan, if the local slice of the thread overlaps with the segment the previous block ended in. This is calculated in lines 4-7 by finding the global index of the last element in the previous block, and the global index of the first element in the current thread.

If neither of the scans crossed between segments, the prefix value, accumulated value and local value are all combined.

Result distribution:

```
1  float x1; float y1;
2  float x2 = prefix; #from lookback phase
```

```
float y2 = acc; #from groupScan
int prev_block_end = block_offset-1;
int thread_start = block_offset + threadId * m;
bool prev_block_same_sgm =
  thread_start - prev_block_end <= thread_start % sgm_size;
if(prev_block_same_sgm)
  x1 = ScanOp(x2, y2);
else
  x1 = y2;
int prev_Idx = thread_start - 1;
int already_scanned = prev_Idx % sgm_size;
int remaining_sgm = sgm_size - already_scanned - 1;
for(int i = 0; i < m; i++){
  if(i < remaining_sgm)
    y1 = local[i];
    local[i] = ScanOp(x1, y1);
}
```

Before writing the final results back to global memory, the values are once again transposed in shared memory, to ensure the writing is also done in a coalesced fashion. This step is simply repeating what was done in section 4.3, but with lines 2 and 7 swapped.

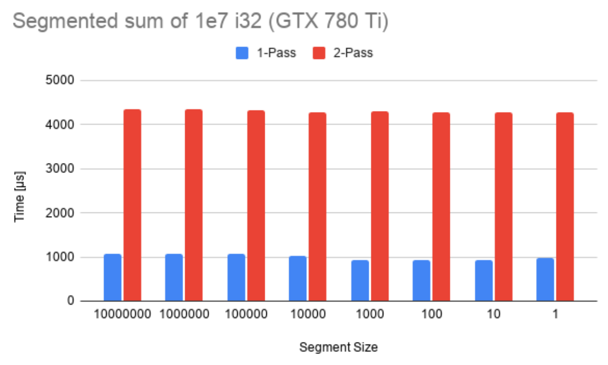
4.8 Initial Results

Futhark has a built-in benchmarking system, which in addition to timing the execution of compiled code also compares the results of parallel execution to the results obtained from sequential execution. Each benchmark thus also verifies correctness of the algorithm, by ensuring equivalence with the sequential counterpart. The comparison is an equivalence test and not an exact equality, as floating point arithmetic is non-associative. As a consequence, parallel algorithms working on floating points rarely obtain the exact values a sequential algorithm would. These results are instead verified to be within an acceptable margin of difference.

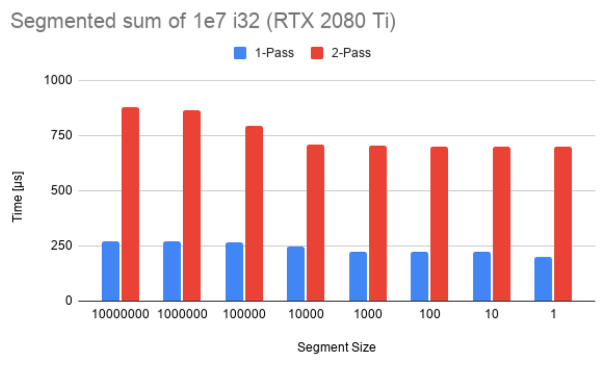
Testing out the modified algorithm on the pre-made micro benchmarks, which sums over segments of 32-bit integers, a clear speedup can be perceived. The times are averages of 100 runs, and with each thread scanning $m = 9$ elements. While the sizes of the segments vary, the number of segments are chosen such that the total number of elements are 10^7 in all cases.

Figure 4a shows that on a GTX 780 Titan, the single-pass algorithm provides a 4.0x-4.6x speedup on segmented sums compared to the two-pass version. On the faster RTX 2080 Titan (4b) the relative speedup is smaller but still quite significant, with a range of 2.9x-3.5x improvement. A thing to note is that the relative performance increases in the favor of the single-pass version as the size of the segments decreases. This is because smaller segments increases the likelihood of reaching the special cases during the lookback phase, where either the block aligns with a segment start or overlaps a segment boundary, which speeds up the lookback phase.

Figure 5 shows the same benchmarks, but expressed as operation speed rather than total running time. Since the benchmarks sum 10^7 integers of 4 bytes each, and each element is transferred twice (1 read, 1 write), the total amount of bytes moved is 80 MB. While the two-pass approach performs 2 reads and 2 writes per element and thus moves 160 MB of data, using 160 MB in the calculation of effective bandwidth makes comparing the results from the two approaches nonsensical.

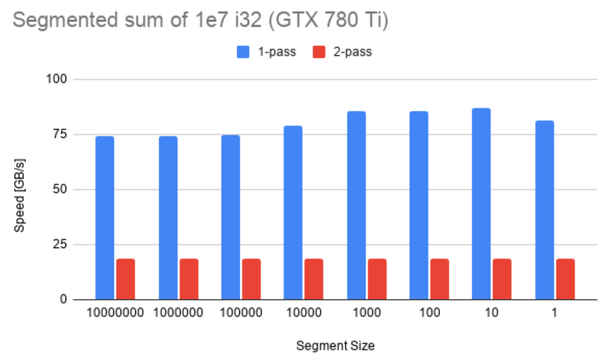


(a) Segmented sum on a GTX 780 Ti

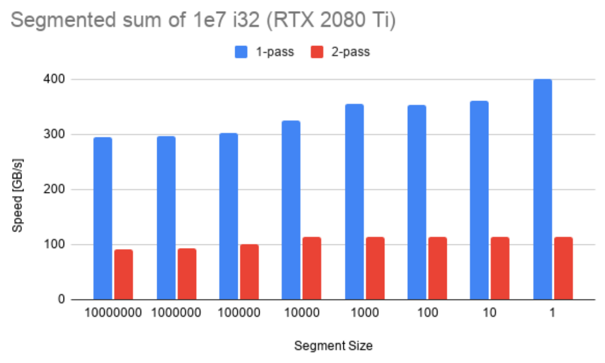


(b) Segmented sum on a RTX 2080 Ti

Figure 4: Running time of segmented sums



(a) Segmented sum on a GTX 780 Ti



(b) Segmented sum on a RTX 2080 Ti

Figure 5: Speed of segmented sums

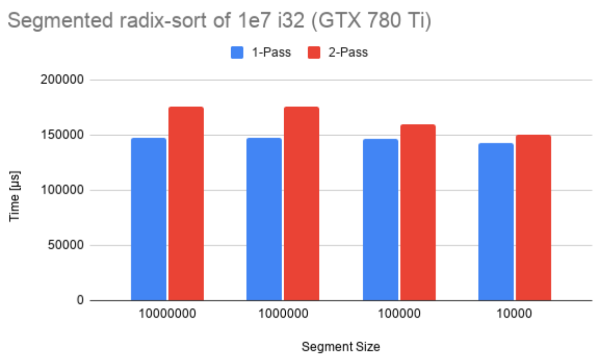
Rather than calculating the effective bandwidth of the actual implementation, we will calculate a bandwidth based on the specification of the optimal implementation, which as discussed in section 1.1 uses $2n$ memory movements.

The speed is thus found by dividing 80 MB with the running time of the relevant benchmark. The results are 74-86 GB/s on the GTX 780 Ti, and 295-400 GB/s on the RTX 2080 Ti. The respective memory bandwidth of these gpus are 336 GB/s and 616 GB/s [4,5]. Compared to the bandwidth, the implementation reaches 22%-25% of the theoretical limit on GTX 270 Ti, and 48%-65% on RTX 2080 Ti. The current implementation thus seems to benefit higher-end hardware more.

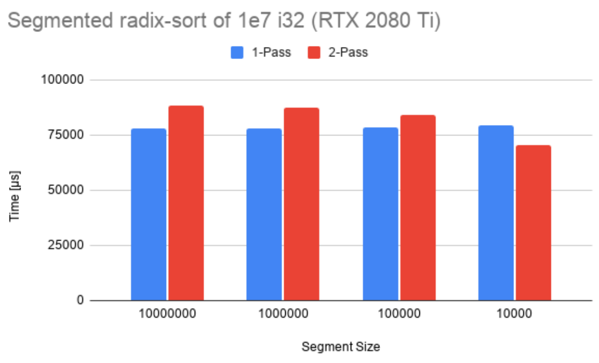
If we extend the benchmarks to not just segmented scans, but also segmented radix-sort which utilizes scan, we can see the difference in performance is not only due to hardware differences. Using the same parameters as before ($m = 9, runs = 100$), but only including segments of size $\geq 10^4$ so the sorting step is not too small, figure 6 shows that the speedup becomes only 1.2x for GTX 780 Ti and 1.1x for RTX 2080 Ti. In fact, the single-pass performs *worse* on segments of size $= 10^4$ on the 2080 Ti.

This discrepancy is explained in the following section, by taking a more detailed look at the scan operator used and the number of sequential elements per thread.

[4] `https://www.techpowerup.com/gpu-specs/geforce-gtx-780-ti.c2512`
[5] `https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-ti.c3305`



(a) Segmented radix-sort on a GTX 780 Ti



(b) Segmented radix-sort on a RTX 2080 Ti

Figure 6: Running times of radix-sort

5 Sequentiality Degree

As the values of each element in a scan depends on the values of the preceding elements, a parallel scan naturally requires a certain amount of communication between threads in order to propagate results. This inter-thread communication however adds latency to the execution, meaning that reducing the amount of inter-thread communication increases the overall speed of the implementation.

This fact is the reason behind letting each thread sequentially calculate a slice of elements, rather than assigning a single thread to each elements. The greater the value of m, the less inter-thread communication is needed. This of course carries the risk of starving the gpu of work, but as long as m is sufficiently small, this does not pose a real issue. More precisely, in a scan of n elements on a gpu with t concurrently active threads, as long as m is not greater than $\frac{n}{t}$, the amount of parallelism is unaffected.

On an ideal GPU, optimal performance would be achieved with $m = \left\lfloor \frac{n}{t} \right\rfloor$, but in reality hardware properties limit the range of m to somewhere between 1 and ~16. This limit is due to resources such as memory and registers being shared across threads. Threads are being managed by a streaming multiprocessor (SM), which has a set amount of registers and shared memory available. The more resources each thread requires, the fewer concurrent active threads the SM can spawn.

Assuming there is always enough work to not decrease the amount of parallelism, we are interested in choosing the greatest value of m that does not inhibit the number of concurrent active threads. This is done by inspecting the amount of resources available and comparing it to the amount required by the scan operator.

Concentrating on the results of the radix sort on the 2080 Ti with segment size $= 10^4$, if we repeat the benchmark with different values of m, the data paints a different picture than the results shown in section 4.8.

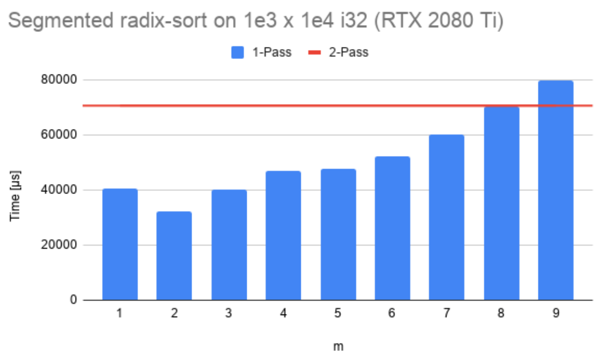


Figure 7: Benchmarking different values of m for radix-sorting

As depicted in figure 7, the performance goes from a 1.1x *slowdown* at $m = 9$ to a 2.2x speedup at $m = 2$. Inspecting the scan operator used in radix-sort, it can be seen to perform pairwise addition on a 4-tuple of 64-bit integers:

```
let pairwise op (a1,b1,c1,d1) (a2,b2,c2,d2) =
  (a1 `op` a2, b1 `op` b2, c1 `op` c2, d1 `op` d2)
[...]
let offsets = scan (pairwise (+)) (0,0,0,0) flags
```

Comparing this to a less resource-intensive operator such as addition on 32-bit floats, m can be set much higher. As figure 8 shows, increasing m increases performance until reaching the optimal value at $m = 15$. At $m = 16$ resource usage reaches a point where the number of active threads has to be reduced, worsening performance.

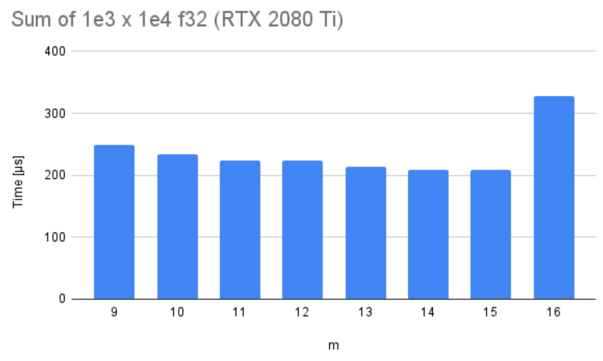


Figure 8: Benchmarking different values of m for floating-point addition

There are two main factors to consider in determining the size of m: shared memory and register memory. For the sake of example, we will use the specifications of GPUs with compute capability 7.5, as documented in [3].

5.1 Analytical Model for Shared memory

As specified in the technical documentation, blocks on a gpu with compute capability 7.5 can have a maximum of 64 KB of shared memory allocated. Dividing by the maximum number of possible threads in a block (1024), we get that each thread can have 64 bytes of shared memory available, corresponding to 16 32-bit words. This of course assumes that both of these values are set to the upper limit, which might not be the case in reality.

By inspecting the steps of the algorithm, we can identify where shared memory is used and possibly reused. During the reading step, as shown in section 4, all the elements are loaded from global to shared memory as part of a transposition, in order to ensure coalesced memory access. This step is also mirrored in the final write-back. Each thread naturally loads m elements, but the amount of memory used is not $m * t$, with t being the size of an element in bytes. Tuples are not treated as singular elements, rather each element within a tuple is processed separately. This means that the first element of all tuples is transposed, and the shared memory is reused when the second element is being transposed and so on. If we define $max(t)$ to be the size of the largest atomic type contained in t, the amount of shared memory each thread uses for the load/write steps becomes $m * max(t)$.

When doing the thread-level scan as presented in section 4.4, shared memory is only used in the end when each thread publishes the last element of the scanned slice. Since each thread only publishes a single element, this memory usage is independent of m. The intra-block scan from section 4.5 operates *in* shared memory, but does not use additional memory, because it reuses the shared memory allocated during the reading stage. This means the shared memory used per thread for these steps is the total size of t in bytes, which we will call $sum(t)$.

Doing the lookback presented in section 4.6, the result of previous blocks are loaded into shared memory, along with the flag that identifies whether the result is prefix, aggregate or unavailable. The amount of block results is equal to the size of a warp, which is 32 in this case. The flag value is contained in a single byte, which means the amount of shared memory used doing lookback is $(sum(t) + 1) * warp\_size$. This is not per thread, but for the entire block. If this amount of memory does not exceed the memory used in the previous step, the memory from before can simply be reused.

If we view the shared memory usage of the previous step for the entire block, it becomes $sum(t) * block\_size$. The size of the block has to be at least the size of a warp, so with a large enough $sum(t)$ and $block\_size > warp\_size$, we get $sum(t) * block\_size \geq (sum(t) + 1) * warp\_size$. The smallest possible size of $sum(t)$ is 1 byte, and by using that value we can determine the minimum block size needed to guarantee the lookback reuses memory from the scan step.

$$1 * block\_size \geq (1 + 1) * warp\_size$$
$$\Longrightarrow$$
$$block\_size \geq 2 * warp\_size$$

We will thus continue the calculations with the assumption that blocks contains at least 2 warps. We know the constraints on shared memory is 64 bytes per thread and have:

$$max(m * max(t), sum(t)) \leq 64$$

Since we want the maximal m and $sum(t)$ is independent of m, for cases where $sum(t) > 64$ we may as well choose an m such that $m * max(t) = sum(t)$. We can thus rewrite the constraint to:

$$m \leq \frac{max(64, sum(t))}{max(t)} \tag{1}$$

5.2 Analytical Model for Register memory

GPUs can also experience performance issues due to register pressure, which happens if execution requires more 32-bit registers than are available. Trying to determine the exact amount of registers a given piece of code will use is a difficult matter, so instead we will rely on some conservative assumptions. The most likely effect the value of m has on register usage, is the loops that run for m iterations, which might get vectorized and thus require registers for each value of the loop variables.

Since values represented with fewer than 32 bits still occupy a full 32-bit register, we need a slightly different notion of $sum(t)$. All types less than 32 bits in size need to be rounded up to one word. Thus we define:

$$sum\uparrow(t) = \frac{\Sigma_{t_{prim} \in t}\, max(size(t_{prim}), 4)}{4}$$

Where $size(t)$ returns the size of a type t in bytes.

The programming guide specifies a max of 64K 32-bit registers per block, which results in 64 registers per thread using the same assumption as in the previous section. We will also further assume that these 64 registers are general-purpose, and not further split into e.g. floating-point registers and address registers. The loop that relies on m and uses the most variables is the loop in the result distribution:

```
float[] local = float[m];
[...]
```

```
  float x1; float y1;
  float x2 = prefix; #from lookback phase
  float y2 = acc;
  [...]
  int last_tIdx = block_offset + threadId * m - 1;
  int already_scanned = last_tIdx % sgm_size;
  int remaining_sgm = sgm_size - already_scanned - 1;
  for(int i = 0; i < m; i++){
    if(i < remaining_sgm)
      y1 = local[i];
      local[i] = ScanOp(x1, y1);
  }
```

The loop variables are `i` (1 word), the address of `local[i]` (2 words), the scan-parameter `y1` ($sum \uparrow (t)$) and the result of the scan operation ($sum \uparrow (t)$). This means that in terms of m, $m * (2 * sum \uparrow (t) + 3)$ registers are needed. Depending on the scan operator, this number can vary quite a bit. If the operator is normal addition, the resulting code would most likely compile into instructions where one of the parameter registers are also used for the result, thus reducing the number of registers in use. Conversely, if the operator makes use of intermediary values, the number of registers would be greater.

The constants used in the loop of the result distribution is `remaining_sgm` (1 word) and `x1` ($sum \uparrow (t)$ words). `remaining_sgm` would generally need to be stored in 2 words depending on the segment size, but in the next section we will cover how to guarantee a value that fits within 1 word. We will thus for now assume 1 word is enough. This leaves us with $64 - 1 - sum \uparrow (t)$ registers for the loop variables:

$$64 - 1 - sum \uparrow (t) \geq m * (2 * sum \uparrow (t) + 3)$$
$$\Longrightarrow$$
$$m \leq \frac{63 - sum \uparrow (t)}{2 * sum \uparrow (t) + 3} \tag{2}$$

5.3 Formula for Sequentiality Degree

Combining the constraints from (1) and (2), we should choose the maximal integer m such that:

$$m \leq min\left(\frac{max(64, sum(t))}{max(t)}, \frac{63 - sum \uparrow (t)}{2 * sum \uparrow (t) + 3}\right) \tag{3}$$

This constraint is of course specific to gpus with compute capability 7.5, and should be more generic by relying on device properties. If we instead use $k_{mem} = \frac{max\_block\_mem}{max\_block\_size}$ and $k_{reg} = \frac{max\_block\_reg}{max\_block\_size}$, we can write:

$$m = max\left(min\left(\left\lfloor\frac{max(k_{mem}, sum(t))}{max(t)}\right\rfloor, \left\lfloor\frac{k_{reg} - 1 - sum \uparrow (t)}{2 * sum \uparrow (t) + 3}\right\rfloor\right), 1\right) \tag{4}$$

The inequality is changed to an equality, as we want the greatest value for m. Additionally, for very large type sizes, flooring the quotients might result in 0. As the algorithm cannot function with $m = 0$, m is bounded with a lower limit of 1.

The values of k_{reg} and k_{mem} should ideally be calculated dynamically by querying the device that executes the code, but as the size of the local arrays for each thread depends on the value of m, the result cannot be passed as a normal parameter. CUDA requires size-expressions for arrays to be constant, and while the run-time for the program corresponds to the compile-time of the kernel, the Futhark compiler code does not easily allow for the passing of values as compile-time constants.

Instead we will use static values for the hardware constants, and use $k_{reg} = 64, k_{mem} = 48$ on the RTX 2080 Ti and $k_{reg} = 64, k_{mem} = 36$ on GTX 780 Ti. The reason that k_{mem} is not 64 and 48 respectively, is that the devices have been configured to use 1/4 of the available memory as cache, meaning the max amount of shared memory is instead 3/4 of the specified maximum in the documentation.

Going back to the initial example of the radix-sort, even though the sorting is of 32-bit integers, the scan used is over 4-tuples of 64-bit integers, which explains why $m = 9$ was not a good fit. Using the values of the RTX 2080 Ti and memory sizes of a 4-tuple of int64, we can plug in the values to determine m:

$$m = max\left(min\left(\left\lfloor\frac{max(48,8)}{2}\right\rfloor, \left\lfloor\frac{64-1-8}{2*8+3}\right\rfloor\right), 1\right) = max(min(24,2),1) = 2$$

This gives us the $m = 2$ which the benchmarks in figure 7 showed to be the most efficient.

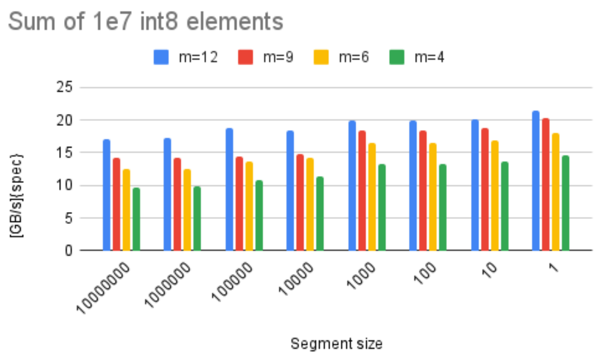
5.4 Benchmarks

To test how beneficial it is to have a semi-dynamically chosen m, we can test the performance of a scan on differently sized types. Figure 9 shows an additive scan on integers represented with 1, 2, 4 and 8 bytes on the GTX 780 Ti.

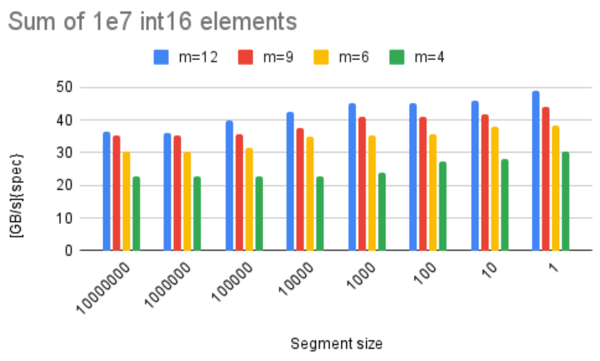
For the sake of brevity, the lower bound on m of 1 will be omitted in the following calculations, as the type sizes are small enough that the bound becomes irrelevant. With $k_{reg} = 64, k_{mem} = 36$, the values of m becomes:

$$\begin{aligned}
int8 : m &= min\left(\left\lfloor\frac{max(36,1)}{1}\right\rfloor, \left\lfloor\frac{64-1-1}{2*1+3}\right\rfloor\right) = min(36,12) = 12\\
int16 : m &= min\left(\left\lfloor\frac{max(36,2)}{2}\right\rfloor, \left\lfloor\frac{64-1-1}{2*1+3}\right\rfloor\right) = min(18,12) = 12\\
int32 : m &= min\left(\left\lfloor\frac{max(36,4)}{4}\right\rfloor, \left\lfloor\frac{64-1-1}{2*1+3}\right\rfloor\right) = min(9,12) = 9\\
int64 : m &= min\left(\left\lfloor\frac{max(36,8)}{8}\right\rfloor, \left\lfloor\frac{64-1-1}{2*2+3}\right\rfloor\right) = min(4,8) = 4
\end{aligned}$$

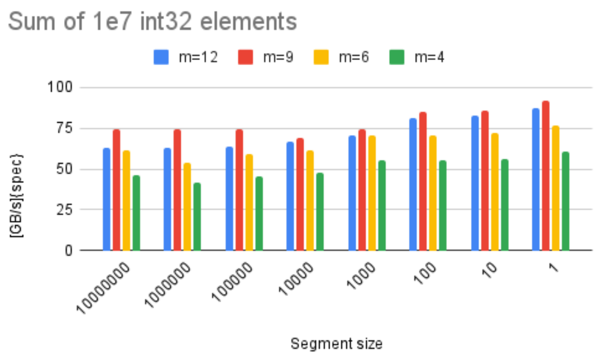
For each of the data types, m has also been fixed to various other values to see how the dynamically chosen one compares. In figures 9a-9c, the dynamic values performs best out of the four tested, while in 9d, $m = 6$ performs about 10% better than the dynamic $m = 4$.



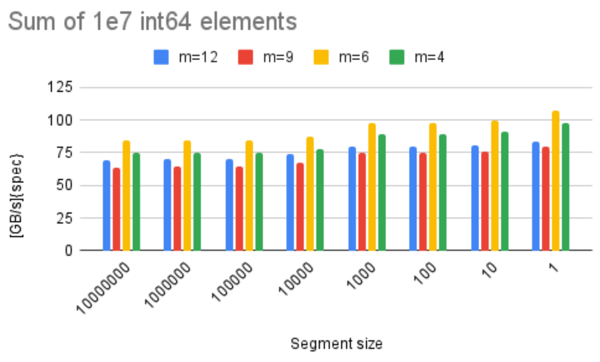
(a) For 1 byte elements, m is dynamically set to 12



(b) For 2 byte elements, m is dynamically set to 12



(c) For 4 byte elements, m is dynamically set to 9



(d) For 8 byte elements, m is dynamically set to 4

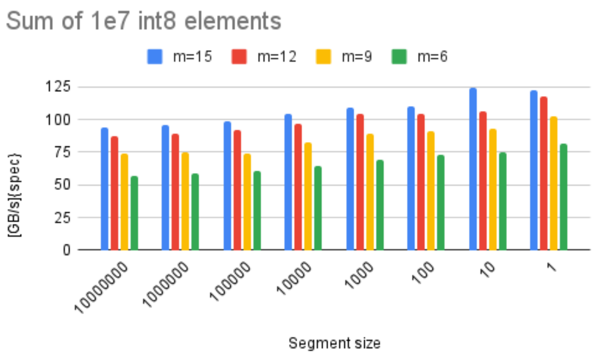
Figure 9: GTX 780 Ti executing with different values of m on different types

Figure 10 shows the same experiment repeated on the RTX 2080 Ti. Here the chosen values are:

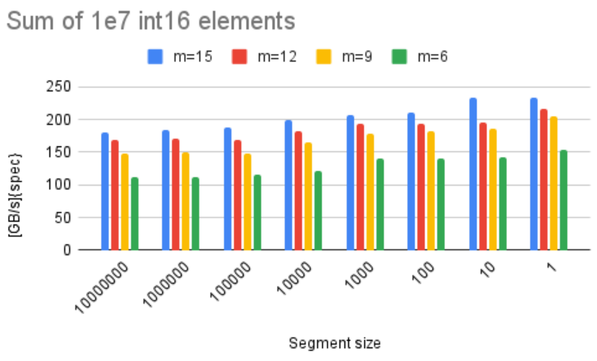
$$int8 : m = min\left(\left\lfloor\frac{max(48,1)}{1}\right\rfloor, \left\lfloor\frac{64-1-1}{2*1+3}\right\rfloor\right) = min(48,12) = 12$$
$$int16 : m = min\left(\left\lfloor\frac{max(48,2)}{2}\right\rfloor, \left\lfloor\frac{64-1-1}{2*1+3}\right\rfloor\right) = min(24,12) = 12$$
$$int32 : m = min\left(\left\lfloor\frac{max(48,4)}{4}\right\rfloor, \left\lfloor\frac{64-1-1}{2*1+3}\right\rfloor\right) = min(12,12) = 12$$
$$int64 : m = min\left(\left\lfloor\frac{max(48,8)}{8}\right\rfloor, \left\lfloor\frac{64-1-1}{2*2+3}\right\rfloor\right) = min(6,8) = 6$$

Looking at figures 10a-10c, the dynamic value is outperformed by $m = 15$. 15 seems to be the limit, as the performance worsens significantly at $m = 16$. A possibly explanation could be what was discussed in the section about register memory; Using operators that easily translate to machine instructions can allow for efficient reusing of registers. In fact, the two benchmarks in figure 9 where the register constraint was the limiting factor (int8 and int16) also perform better at $m = 15$, with a 3%-15% increase for int8 and 4%-16% for int16, depending on segment size.

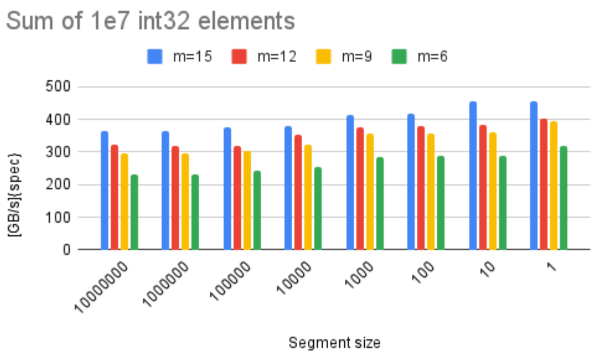
Figure 10d shows another important caveat to the method of choosing m. $m = 6$ performs best for segments where the size is ≤ 1000, but for sizes ≥ 10000 the best performing value is $m = 9$. For



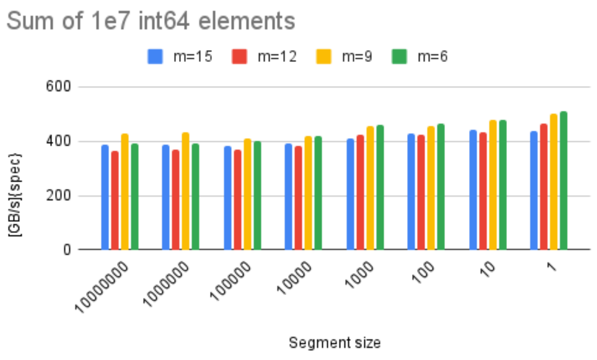
(a) For 1 byte elements, m is dynamically set to 12



(b) For 2 byte elements, m is dynamically set to 12



(c) For 4 byte elements, m is dynamically set to 12



(d) For 8 byte elements, m is dynamically set to 6

Figure 10: RTX 2080 Ti executing with different values of m on different types

segments of size = 1000 and below, the segment size is smaller than the number of elements each block handles. Thus each block never has to inspect more than one block during the lookback. At size = 10000 and above, the segment size is greater than the number of elements in each block, meaning more work has to be performed during lookback. During lookback, the more elements each block computes, the fewer blocks have to be inspected before hitting a segment boundary. This explains the change of optimal values for m between 1000 and 10000 segment size. What can be concluded from this is; The method for determining m only considers how to avoid exceeding register and shared memory. It does not take into account whether the benefit from increasing the elements per block and decreasing inter-thread communication outweighs the penalty from exceeding memory constraints.

This can further be seen in 10d due to the fact that $m = 12$ performs worse than $m = 9$ and $m = 6$ as a result of exceeding the memory constraints, but at segments size ≥ 10000, $m = 15$ performs better than $m = 12$ even though it exceeds the memory constraints even more. As mentioned, this is because increasing the elements per block lessens the workload during lookback, and the benefit starts to outweigh the drawback of exceeding memory.

The dynamic m does however seem to be a general improvement over the static value, as no one value for m would have performed great in a benchmark, without doing poorly in another.

6 Optimizing Index Calculations

Modulo calculations are generally very costly to perform, especially on 64-bit values. With the baseline algorithm described in section 4, there are currently used 64-bit modulo calculations on line 3 of the thread-level scan (sec 4.4), line 4 of the block-level scan (sec 4.5), line 2 of the lookback (sec 4.6) and line 7+13 of the results distribution (sec 4.7). To minimize the cost, we will look at a way to safely transform the calculations into 32-bit arithmetic operations.

The divisor in all the mentioned modulo calculations is `sgm_size`, and we cannot generally assume the size of segments to fit within 32 bits. We can however change the calculations on a case basis. The are two major cases to consider: when the size of segments is less than the number of elements treated by a block (1), and when the size is greater than or equal the number of elements treated by a block (2). We will refer to the number of elements treated by a block as the "group size".

The group size is the product of m and the number of threads in a group. Since we are now using equation 4 from section 5.3 to determine m, we can put an upper limit on this value by looking at hardware specifications. With the current model, the maximum possible shared memory per thread is 164 (Compute Capability 8.0), and the smallest data type passed to scan must take up 1 byte.

We therefore know that m will not be greater than $164/1$, and the number of threads in a block does not exceed 1024 (at least on current NVIDIA hardware). The maximum group size then becomes $164 \cdot 1024$, which easily fits within 32 bits. This does rely on the assumption that the value of $sum \uparrow (t)$ is not greater than the amount of shared memory per thread, but even if that is the case, as long as a scan is not performed on elements with a data size greater than 4MB each, the group size is still representable with 32 bits.

If we are in case (1) where the segment size is less than the group size, we know the segment size to be representable in 32 bits also.

If we are in case (2) where the segment size is larger than the group size, there is at most one segment crossing within each block. This case can further be divided into cases where the current block has no segment crossings (2a), and blocks that have exactly one segment crossing (2b). If we are in case (2b), we can calculate the relative index within the block where the segment crossing in located. This will be a value between 0 and group size-1, so is representable with 32 bits. In case (2a) this relative index can be set to any value equal to or greater than the group size, as there is no segment crossing within the current block.

On line 2 of the lookback (section 4.6), we calculated a 64-bit `sgm_idx` value, representing what element of the current segment the block starts at. The calculation of this value is left unchanged and will be the only 64-bit modulo operation we do not remove. The value could be computed by a single thread and propagated to the rest of the block, but in testing this approach it did not prove to be faster than having the threads individually compute the value.

Subtracting `sgm_idx` from `sgm_size` results in the distance to the first segment crossing in terms of number of elements, which gives us the relative index needed for case (2). Referring to this as a boundary index, the boundary index can be viewed as both the number of remaining elements in the current segment, as well as the index in the group where the first new segment starts. In case (2a) where there is no new segment within the current block, the boundary index is set to the group size. In case (2b) we know `sgm_size-sgm_idx` to be representable with 32 bits.

If we are in case (1), the size of segments fits within 32 bits, and we will store it in a 32-bit variable `sgm_sizeC`. When not int case (1) we will set this value to the group size.

The calculation of these values then becomes:

```
int64 sgm_idx      = block_offset % sgm_size;
int32 boundary_idx = (int32) min(m*block_size, sgm_size-sgm_idx);
int32 sgm_sizeC    = (int32) min(m*block_size, sgm_size);
```

With two variables taking one of two different values each, there are 4 combinations of assignments:

A $boundary\_idx = sgm\_size - sgm\_idx \wedge sgm\_sizeC = sgm\_size$:
With `sgm_sizeC` equal to `sgm_size`, the segment size must be less than the group size, which means we are in case (1).

B $boundary\_idx = m * block\_size \wedge sgm\_sizeC = m * block\_size$:
If `sgm_sizeC` is equal to the group size, we know its because `sgm_size` is not less than the group size, meaning we are in case (2).
Further, because the boundary index is also equal to the group size, the next segment crossing must not happen within the block and we are in case (2a).

C $boundary\_idx = sgm\_size - sgm\_idx \wedge sgm\_sizeC = m * block\_size$:
With similar reasoning as above, we are in case (2). Because `sgm_size-sgm_idx` is less than the group size, there is a segment crossing within the block and we are in case (2b).

D $boundary\_idx = m * block\_size \wedge sgm\_sizeC = sgm\_size$:
Looking at the definition of `boundary_idx` above, we must have $boundary\_idx \leq sgm\_sizeC$. With $boundary\_idx = m * block\_size$ this gives us $m * block\_size \leq sgm\_sizeC$.

`sgm_sizeC` is also upper bounded by the group size, so we have $sgm\_sizeC \leq m*block\_size$. Combining these two inequalities shows that we must have $sgm\_sizeC = m*block\_size$, which means this is just assignment B again, and we are in case (2a).

We have now associated the cases with corresponding variable assignments, and the correctness of the new calculations in the algorithm will be argued for on a case-by-case basis.

6.1 Thread-level scan

Only the calculation of new_sgm has to be changed at this step:

```
for(int i = 1; i < m; i++){
  bool new_sgm = (threadId*m+i-boundary_idx) % sgm_sizeC == 0;
  if(!new_sgm)
    local[i] = ScanOp(local[i-1], input[gIdx+i];
}
```

The key observation here is that the sub-expression $threadId * m + i$ will have a value between 0 and $block\_size * m - 1$.

- Case (1):
  With both $boundary\_idx$ and sgm_siceC being less than $block\_size * m$, new_sgm will be true when $\exists k.k \in \mathbb{N} \Rightarrow threadId * m + i = sgm\_sizeC * k + boundary\_idx$. More informally, new_sgm is true when the thread index reaches the boundary index, and for each full segment length after the boundary.

- Case (2a):
  Since we have $boundary\_idx = block\_size * m$, we get $threadId * m + i - boundary\_idx < 0$, and thus new_sgm will never be true as required.

- Case (2b):
  As $sgm\_sizeC = block\_size * m$ and $threadId * m + i - boundary\_idx < block\_size * m$, new_sgm will only be true exactly when $threadId * m + i - boundary\_idx = 0$, which happens at $threadId * m + i = boundary\_idx$.

6.2 Block-Level Scan

As before, the threads publish their last element before starting a intra-group scan. Since each thread only publishes a single element, the indices for this scan will have values between 0 and $block\_size - 1$. The predicate for determining if two indices belong to different segments is modified like so:

```
1 bool crossesSegment(idx1, idx2){
2   int end =   idx2*m+m-1;
3   int start = idx1*m+m-1;
4   return end - start > (end+sgm_sizeC-boundary_idx) % sgm_sizeC)
5 }
```

Similar to the thread-level scan, end and $start$ takes on values in the range $m-1$ and $block\_size*m-1$.

- Case (1):
  With $sgm\_sizeC = sgm\_size$ and $boundary\_idx = sgm\_size - sgm\_idx$, The calculation on line 4 becomes $end - start > end + sgm\_idx \mod sgm\_size$. Since $sgm\_idx = block\_offset \mod sgm\_size$, we get:

$$end + sgm\_idx \mod sgm\_size$$
$$=$$
$$end + (block\_offset \mod sgm\_size) \mod sgm\_size$$
$$=$$
$$end + block\_offset \mod sgm\_size$$

  We are then back to the original calculation used in section 4.5, meaning the original semantics are preserved.

- Case (2a):
  We have $sgm\_sizeC = block\_size * m = boundary\_idx$. The expression on line 4 thus corresponds to $end - start > end$ which will never be true.

- Case (2b):
  Note that since $sgm\_sizeC = block\_size * m$, $boundary\_idx \leq sgm\_sizeC$ and $end < block\_size * m$, we get that $end + sgm\_sizeC - boundary\_idx \mod sgm\_sizeC$ can only take one of two values, either $end + sgm\_sizeC - boundary\_idx$ or $end - boundary\_idx$.

With only 1 segment crossing, the predicate should only be true when $start < boundary\_idx \land boundary\_idx \leq end$. This corresponds to proving:

$$start < boundary\_idx \land boundary\_idx \leq end$$
$$\iff$$
$$end - start > end + sgm\_sizeC - boundary\_idx \mod sgm\_sizeC$$

To prove the ($\Rightarrow$) case, assume $start < boundary\_idx \land boundary\_idx \leq end$. Then we have $end + sgm\_sizeC - boundary\_idx \geq sgm\_sizeC$, which means $end + sgm\_sizeC - boundary\_idx \mod sgm\_sizeC = end - boundary\_idx$. Since $start < boundary\_idx$, we have $end - start > end - boundary\_idx$ and line 4 returns true.

For the ($\Leftarrow$) case, assume we have $end - start > end + sgm\_sizeC - boundary\_idx \mod sgm\_sizeC$. If $end + sgm\_sizeC - boundary\_idx \mod sgm\_sizeC = end - boundary\_idx$ we must have $boundary\_idx \leq end$. We also have $end - start > end - boundary\_idx$ which implies $start < boundary\_idx$ as required.

If we instead have $end + sgm\_sizeC - boundary\_idx \mod sgm\_sizeC = end + sgm\_sizeC - boundary\_idx$, we have $end - start > end + sgm\_sizeC - boundary\_idx$. However, since $boundary\_idx \leq sgm\_sizeC$ we get $sgm\_sizeC - boundary\_idx \geq 0$, and with $start > 0$ we arrive at a contradiction. This case can thus not happen, which concludes the proof.

6.3 Lookback

For the lookback phase, no changes are necessary. The calculation to determine if results from two different block belong to different segments relies on `sgm_idx` which we have kept, and no additional modulo operations are used during this step.

```
int readI = readOffset + threadId;
int prev_end = (readI + 1) * group_size * m - 1;
bool same_sgm = group_offset - prev_end <= sgm_idx
```

6.4 Result Distribution

For the final step, two calculations where performed to determine whether the result of the previous block should be applied to the current thread, and how many elements the thread could combine before crossing into a new segment.

```
bool block_new_sgm = sgm_idx == 0;
bool prev_block_same_sgm = !block_new_sgm && threadId*m < boundary_idx;
int remaining_sgm =
  sgm_sizeC - 1 -
  (threadId*m-1+sgm_sizeC-boundary_idx) % sgm_sizeC;
for(int i = 0; i < m; i++){
  if(i < remaining_sgm)
    local[i] = ScanOp(x1, local[i])
}
```

The first calculation is rather simple, as we just have to check that the current group does not perfectly overlap with a segment start, and that the first element of the thread comes before the boundary.

`remaining_sgm` does not always need to calculate the exact amount of elements remaining in the segment, only if the current thread actually crosses between segments. When the thread does not cross between segments, any value such that $remaining\_sgm \geq m$ will suffice.

All segment crossing will be located at indices $k * sgm\_size + boundary\_idx$ for some $k \in \mathbb{N}$. The starting index of threads can similarly be expressed as an integer quotient of sgm_sizeC with a remainder:

$$\exists q, r \in \mathbb{N}. threadId * m = q * sgm\_sizeC + r \wedge r < sgm\_sizeC$$

Threads that overlap segments thus do so at $k = q + 1$ when $r > boundary\_idx$, and at $k = q$ when $r \leq boundary\_idx$.

We will start by proving case (1). When $r > boundary\_idx$, threads need to calculate

$$\begin{gathered} remaining\_sgm \\ = \\ k * sgm\_size + boundary\_idx - threadId * m \\ = \\ (q+1) * sgm\_size + boundary\_idx - q * sgm\_sizeC + r \\ = \\ sgm\_size + boundary\_idx - r \end{gathered}$$

Rewriting the calculation on line 5 gives us:

$$\begin{gathered} threadId * m - 1 + sgm\_sizeC - boundary\_idx \mod sgm\_sizeC \\ = \\ (q+1) * sgm\_sizeC + r - 1 - boundary\_idx \mod sgm\_sizeC \\ = \\ r - 1 - boundary\_idx \end{gathered}$$

$remaining\_sgm$ then becomes:

$$\begin{gathered} sgm\_sizeC - 1 - (r - 1 - boundary\_idx) \\ = \\ sgm\_sizeC - r + boundary\_idx \end{gathered}$$

Since we are in the case (1) we have $sgm\_sizeC = sgm\_size$ and we get what we needed.

When $r \leq boundary\_idx$, threads need to calculate:

$$\begin{gathered} remaining\_sgm \\ = \\ k * sgm\_size + boundary\_idx - threadId * m \\ = \\ q * sgm\_size + boundary\_idx - q * sgm\_sizeC + r \\ = \\ boundary\_idx - r \end{gathered}$$

The difference of r and $boundary\_idx$ is never more than sgm_sizeC, as we have $0 \leq r \leq boundary\_idx \leq sgm\_sizeC$. Rewriting of line 5 then becomes:

$$\begin{gathered} threadId * m - 1 + sgm\_sizeC - boundary\_idx \mod sgm\_sizeC \\ = \\ (q+1) * sgm\_sizeC + r - 1 - boundary\_idx \mod sgm\_sizeC \\ = \\ sgm\_sizeC + r - 1 - boundary\_idx \end{gathered}$$

$remaining\_sgm$ then becomes:

$$\begin{gathered} sgm\_sizeC - 1 - (sgm\_sizeC + r - 1 - boundary\_idx) \\ = \\ boundary\_idx - r \end{gathered}$$

This means that all threads calculate the exact value for $remaining\_sgm$, and not just those that overlap a segment boundary. We are therefore guaranteed to have $remaining\_sgm \geq m$ when threads do not cross. This concludes case (1)

In case (2a) we are guaranteed that no threads overlap a segment crossing. We have $sgm\_sizeC = boundary\_idx = block\_size * m$, so $remaining\_sgm = sgm\_sizeC - threadId * m = block\_size * m - threadId * m$. Since $threadId < block\_size$, we get $remaining\_sgm \geq m$ as wanted.

In case (2b) we know there is exactly one segment crossing at $boundary\_idx$. Thus only the thread that fulfills $threadId * m \leq boundary\_idx < (threadId + 1) * m$ needs to calculate the precise value for $remaining\_sgm$.
With $threadId * m \leq boundary\_idx$, we get $threadId * m - 1 - boundary\_idx < 0$ and thus line 5 is:

$$\begin{gathered} threadId * m - 1 + sgm\_sizeC - boundary\_idx \mod sgm\_sizeC \\ = \\ threadId * m - 1 + sgm\_sizeC - boundary\_idx \end{gathered}$$

Therefore $remaining\_sgm = boundary\_idx - threadId * m$, which is the same as the distance from the start of the thread to the boundary, equivalent to the number of remaining elements.

The other threads in the block must just needs the inequality of $remaining\_sgm \geq m$ satisfied.
If we have $theadId * m \leq boundary\_idx$, but do not have $boundary\_idx < (threadId + 1) * m$, we still get $remaining\_sgm = boundary\_idx - threadId * m$. Since $boundary\_idx \geq (threadId + 1) * m$ we get $remaining\_sgm \geq (threadId + 1) * m - threadId * m = m$ as required.

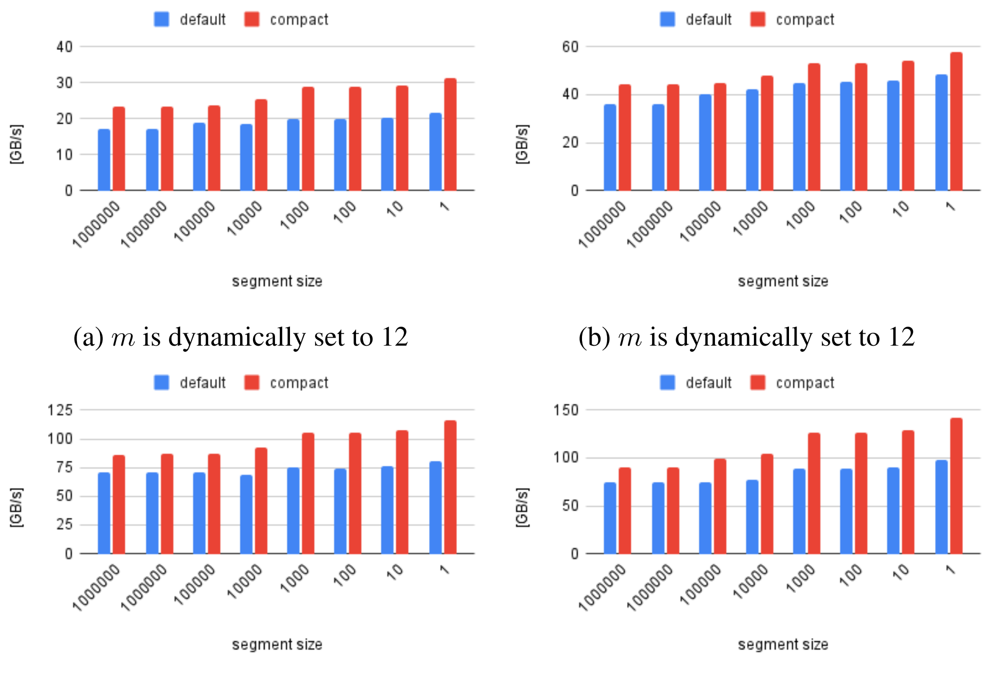
If $threadId * m > boundary\_idx$, line 5 gives us:

$$\begin{gathered} threadId * m - 1 + sgm\_sizeC - boundary\_idx \mod sgm\_sizeC \\ = \\ threadId * m - 1 - boundary\_idx \\ \leq \\ (block\_size - 1) * m - 1 \end{gathered}$$

This means $remaining\_sgm \geq block\_size * m - 1 - ((block\_size - 1) * m - 1) = m$.

6.5 Benchmarks

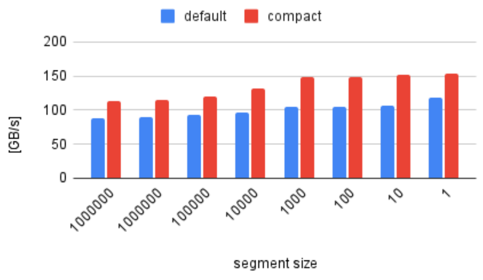
Repeating the benchmarks from section 5.4 with the new index calculations, a speedup of between 20%-40% is obtained on both gpus. Figure 11 shows the results of the GTX 780 Ti and figure 12 shows the results of the RTX 2080 Ti. Similar to the results in section 4.8, the performance increase gets larger as the segment size becomes smaller. As the lookback is skipped more often for small segments, the index calculations takes up a relatively larger part of the computation, which explains why scans on smaller segments benefit more from the optimization.



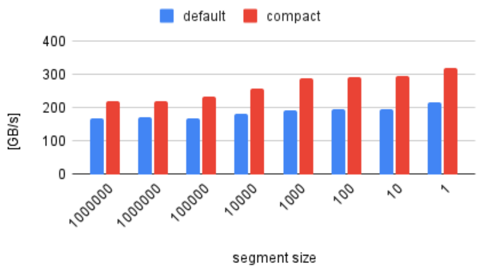
(a) m is dynamically set to 12 (b) m is dynamically set to 12

(c) m is dynamically set to 9 (d) m is dynamically set to 4

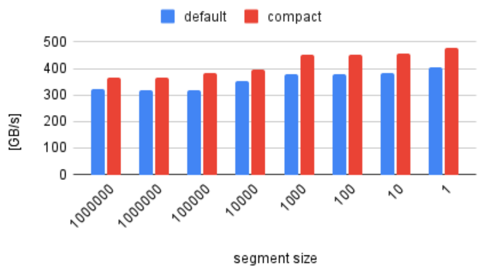
Figure 11: Comparison of new vs. old index calculations on the GTX 780 Ti



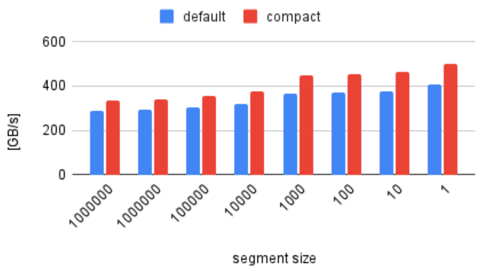
(a) m is dynamically set to 12



(b) m is dynamically set to 12



(c) m is dynamically set to 12



(d) m is dynamically set to 6

Figure 12: Comparison of new vs. old index calculations on the RTX 2080 Ti

7 Implementation

The Futhark compiler is written in Haskell, and implementation of the single-pass segmented scan is done by creating a new Haskell module for the algorithm.

The module is located at src/Futhark/CodeGen/ImpGen/Kernels/SegScan/SegSinglePass.hs.

Adding the single-pass algorithm as possible compilation output is achieved by adding a guard clause when the chosen backend is CUDA. The clause is on lines 6-11, and imposes some restrictions on the scan operator. The provided operator cannot be vectorized (line 10), and the data type it operates on cannot be arrays (line 11). If this does not hold, the compilation will default to the two-pass approach.

Additionally, the general compilation of scans in Futhark allows for providing several scan operators, but we have only covered how to handle one operator in section 4. Therefore on line 7, multiple possible scan operators are combined into a single do-all operator, so the case is still covered by the presented single-pass algorithm.

This does however come with a drawback, as the resulting operator also has the combined resource cost of all the individual operators. As discussed in section 5, the amount of sequential work each thread should perform is constrained by the type size of the operator. With individual operators it could have been possible to reuse shared memory and registers by applying the operators one at a time. This would result in the constraining factor is the operator with the greatest type size, rather than the combined type size of all the operators.

```haskell
 1 compileSegScan pat lvl space scans kbody = sWhen (0 .<. n) $ do
 2   target <- hostTarget <$> askEnv
 3   case target of
 4     CUDA
 5       ...
 6       | all ok scans ->
 7         SegSinglePass.compileSegScan pat lvl space (combineScans scans) kbody
 8       where
 9         ok op =
10           segBinOpShape op == mempty
11             && all primType (lambdaReturnType (segBinOpLambda op))
12     _ -> TwoPass.compileSegScan pat lvl space scans kbody
13   where
14     n = product $ map toInt64Exp $ segSpaceDims space
```

7.1 Determining Sequential Work

In section 5.3 we arrived at equation 4 for determining the value of m. Implementing this is then simply a question of calculating the required values $sum(t)$, $sum\uparrow(t)$ and $max(t)$.

Line 1 converts the return type of the scan operator to a flat list of primitive types. This list serves as the basis for calculating the necessary values. `primByteSize` returns the size of a primitive type in bytes, so calculating $sum(t)$ is easily done with a fold over the type list, as shown on line 2. Similarly on line 5, $max(t)$ is calculated by choosing the maximum value after converting the types to their byte sizes. Line 3 and 4 calculates $sum\uparrow(t)$ by using a modified `primByteSize` with a lower bound of 4 bytes.

On lines 10 and 11 we have the static hardware constants for the GTX 780 Ti. Lines 12 and 13 then use these constants to calculate the two constraints we arrived at in sections 5.1 and 5.2. Note that the flooring is done implicitly by utilizing integer division. Finally line 14 implements the equation from section 5.3.

```
1    tys = map (\(Prim pt) -> pt) $ lambdaReturnType $ segBinOpLambda scanOp
2    sumT = foldl (\bytes typ -> bytes + primByteSize typ) 0 tys
3    primByteSize' = max 4 . primByteSize
4    sumT' = foldl (\bytes typ -> bytes + primByteSize' typ) 0 tys `div` 4
5    maxT = maximum (map primByteSize tys)
6    -- RTX 2080 Ti constants (CC 7.5)
7    -- k_reg = 64
8    -- k_mem = 48
9    -- GTX 780 Ti constants (CC 3.5)
10   k_reg = 64
11   k_mem = 36
12   mem_constraint = max k_mem sumT `div` maxT
13   reg_constraint = (k_reg-1-sumT') `div` (2*sumT'+3)
14   m = fromIntegral $ max 1 $ min mem_constraint reg_constraint
```

7.2 Load & Map and Transpose

The steps of loading, mapping and transposing the input before starting the scan is unaffected by the generalization of segmenting the scan. This part of the implementation is therefore exactly the same as the implementation by Persson & Nicolaisen, and not described in greater detail for the sake of brevity.

7.3 Thread-Level Scan

As described in section 4.4, the thread-level scan computes the global index of the second element it treats (lines 1-3), as the first element has no local predecessor the be combined with. The scan is then carried out in a for-loop (line 4), with a check to avoid combining elements from different segments. Lines 8-10 compile the check as described in section 6.1. Lines 12-22 apply the scan operator on two neighboring elements if the check passes. Lines 24-26 then write the last element to shared memory for preparation of the block-level scan.

```
1    globalIdx <-
2      dPrimVE "gidx" $
3        (kernelLocalThreadId constants * m) + 1
4    sFor "i" (m -1) $ \i -> do
5      let xs = map paramName $ xParams scanOp
6          ys = map paramName $ yParams scanOp
7
8      isNewSgm <-
9        dPrimVE "new_sgm" $
10         (globalIdx + sExt32 i - boundary) `mod` segsize_compact .==. 0
11
12     sUnless isNewSgm $ do
```

```
    forM_ (zip privateArrays $ zip3 xs ys tys) $ \(src, (x, y, ty)) -> do
      dPrim_ x ty
      dPrim_ y ty
      copyDWIMFix x [] (Var src) [i]
      copyDWIMFix y [] (Var src) [i + 1]

    compileStms mempty (bodyStms $ lambdaBody $ segBinOpLambda scanOp) $
      forM_ (zip privateArrays $ bodyResult $
        lambdaBody $ segBinOpLambda scanOp) $ \(dest, res) ->
          copyDWIMFix dest [i + 1] res []

forM_ (zip prefixArrays privateArrays) $ \(dest, src) ->
  copyDWIMFix dest [sExt64 $ kernelLocalThreadId constants]
              (Var src) [m - 1]
sOp localBarrier
```

7.4 Block-Level Scan

As described in section 6.2, the predicate for determining if the indices of two elements belong to different segments is implemented in lines 1-5. The predicate is then passes to the groupScan construct on lines 10-16, which performs a block-level scan in shared memory, in the manner described in section 4.5.

The first thread then reads the last element from the groupScan result (lines 18-20), with the other threads reading the value at position threadId-1 (lines 21-23). All threads store the read value as a local accumulator value.

```
1   let crossesSegment =
2     Just $ \from to ->
3     let from' = (from + 1) * m - 1
4         to'   = (to   + 1) * m - 1
5     in (to' - from') .>. (to'+segsize_compact-boundary) `mod` segsize_compact
6
7   scanOp' <- renameLambda $ segBinOpLambda scanOp
8   accs <- mapM (dPrim "acc") tys
9
10  groupScan
11    crossesSegment
12    (tvExp numThreads)
13    (kernelGroupSize constants)
14    scanOp'
15    prefixArrays
16  sOp localBarrier
17
18  let firstThread acc prefixes =
19        copyDWIMFix (tvVar acc) []
20                    (Var prefixes) [sExt64 (kernelGroupSize constants) - 1]
21      notFirstThread acc prefixes =
22        copyDWIMFix (tvVar acc) []
23                    (Var prefixes) [sExt64 (kernelLocalThreadId constants) - 1]
```

```
sIf (kernelLocalThreadId constants .==. 0)
  (zipWithM_ firstThread accs prefixArrays)
  (zipWithM_ notFirstThread accs prefixArrays)
sOp localBarrier
```

7.5 Lookback Phase

The compiler implementation responsible for the lookback phase is 170+ lines of code and rather cumbersome. We choose to instead focus on the part of the implementation which has changed to handle the inclusion of segments. The entire source code listing can be found in appendix A.

Corresponding to the start of the code in section 4.6, lines 1-3 initializes the prefixes to their corresponding neutral element. Line 4 then calculates whether the block perfectly starts a new segment. If it does, line 6-14 makes the first thread in the block publish the block aggregate as a prefix (line 8-9), and set the status of the block to `P` (line 12). Afterwards the local accumulator of the first thread which stored the block aggregate is reset to the neutral element (line 13-14).

```
1   prefixes <-
2     forM (zip scanOpNe tys) $ \(ne, ty) ->
3       dPrimV "prefix" $ TPrimExp $ toExp' ty ne
4   blockNewSgm <- dPrimVE "block_new_sgm" $ sgmIdx .==. 0
5
6   sWhen (blockNewSgm .&&. kernelLocalThreadId constants .==. 0) $ do
7     everythingVolatile $
8       forM_ (zip incprefixArrays accs) $ \(incprefixArray, acc) ->
9         copyDWIMFix incprefixArray [tvExp dynamicId] (tvSize acc) []
10    sOp globalFence
11    everythingVolatile $
12      copyDWIMFix statusFlags [tvExp dynamicId] (intConst Int8 statusP) []
13    forM_ (zip scanOpNe accs) $ \(ne, acc) ->
14      copyDWIMFix (tvVar acc) [] ne []
```

The next step altered with segment handling is the early publishing of the block prefix if the first and last element within the block are in different segments. This is the first part of the actual lookback, which only starts if the block did not perfectly start a new segment, and only includes the first warp of threads (line 1).

The first thread checks if the index of the next segment crossing is equal to the number of elements treated by the block (lines 2-3). As discussed in section 6, this only happens when there are no segment crossings within the block. If there are no crossing, the thread proceeds as normal to publish the block aggregate and set the status to `A` (lines 4-10). Otherwise, the aggregate can be published as a prefix, as other blocks do not need to perform a lookback past the segment crossing. Lines 13-19 thus makes the first thread write the aggregate as a prefix and set the block status to `P`.

```
1   sWhen (bNot blockNewSgm .&&. kernelLocalThreadId constants .<. warpSize) $ do
2     sWhen (kernelLocalThreadId constants .==. 0) $ do
3       sIf (boundary .==. sExt32 (unCount group_size * m))
4         ( do
5           everythingVolatile $
6             forM_ (zip aggregateArrays accs) $ \(aggregateArray, acc) ->
```

```
            copyDWIMFix aggregateArray [tvExp dynamicId] (tvSize acc) []
        sOp globalFence
        everythingVolatile $
          copyDWIMFix statusFlags [tvExp dynamicId] (intConst Int8 statusA) []
      )
      (
        do
        everythingVolatile $
          forM_ (zip incprefixArrays accs) $ \(incprefixArray, acc) ->
            copyDWIMFix incprefixArray [tvExp dynamicId] (tvSize acc) []
        sOp globalFence
        everythingVolatile $
          copyDWIMFix statusFlags [tvExp dynamicId] (intConst Int8 statusP) []
      )
    copyDWIMFix warpscan [0] (Var statusFlags) [tvExp dynamicId - 1]
  sOp localFence
```

The last part of the lookback that is modified is when the first warp reads the values and statuses of the blocks. Lines 1-3 starts by setting the read offset to 32 blocks before the current block, and line 4 defines the reading should stop when the offset is 32 blocks before the very first block.

Lines 5-8 defines the predicate that determines whether an inspected block belongs to the same segment as the block performing the lookback. It takes a read offset, which is also the dynamic id of a block, and calculates the global index of the last element in that block (line 6-7). It then finds the distance from that element to the first element in the current block, and checks whether that interval overlaps a segment boundary on line 8.

The predicate is used on line 16, where the reading continues as normal on line 17-30 by reading either the aggregate or prefix depending on the status of the block, if the last element in the block belongs to the same segment as the start of the current block. If not, lines 31-32 makes the thread overwrite the flag to `P` without reading a value, which means the aggregate is still the neutral element, as declared on line 11-12.

```
1  readOffset <-
2    dPrimV "readOffset" $
3      sExt32 $ tvExp dynamicId - sExt64 (kernelWaveSize constants)
4  let loopStop = warpSize * (-1)
5      sameSegment readIdx =
6        let startIdx = sExt64 (tvExp readIdx + 1) *
7                       kernelGroupSize constants  * m - 1
8        in tvExp blockOff - startIdx .<=. sgmIdx
9  sWhile (tvExp readOffset .>. loopStop) $ do
10   readI <- dPrimV "read_i" $ tvExp readOffset + kernelLocalThreadId constants
11   aggrs <- forM (zip scanOpNe tys) $ \(ne, ty) ->
12     dPrimV "aggr" $ TPrimExp $ toExp' ty ne
13   flag <- dPrimV "flag" statusX
14   used <- dPrimV "used" (0 :: Imp.TExp Int8)
15   everythingVolatile $
16     sIf (tvExp readI .>=. 0 .&&. sameSegment readI)
17       (do
18       copyDWIMFix (tvVar flag) []
19                   (Var statusFlags) [sExt64 $ tvExp readI]
20       sIf
21         (tvExp flag .==. statusP)
22         ( forM_ (zip incprefixArrays aggrs) $ \(incprefix, aggr) ->
23             copyDWIMFix (tvVar aggr) []
24                         (Var incprefix) [sExt64 $ tvExp readI]
25         )
26         ( sWhen (tvExp flag .==. statusA) $ do
27             forM_ (zip aggrs aggregateArrays) $ \(aggr, aggregate) ->
28               copyDWIMFix (tvVar aggr) []
29                           (Var aggregate) [sExt64 $ tvExp readI]
30         ))
31       (sWhen (tvExp readI .>=. 0) $
32         copyDWIMFix (tvVar flag) [] (intConst Int8 statusP) [])
```

After the lookback, the first thread combines the block aggregate with the prefix obtained from the lookback, if it did not already publish a result before the lookback started. Like prior to the lookback, line 4 checks that there are no segment crossings within the block. It then passes the prefix from the lookback and the block aggregate -which is stored in `acc`- to the scan operator (line 6-8). The combined result is then published as a prefix (line 10-12) and the status is set to `P` line (15-16). Lines 18-19 stores the prefix in shared memory in order to propagate the value to the other threads in the block, and line 20-21 resets `acc` to the neutral element for the first thread.

```
1  sWhen (kernelLocalThreadId constants .==. 0) $ do
2    let xs = map paramName $ take (length tys) $ lambdaParams scanOp
3        ys = map paramName $ drop (length tys) $ lambdaParams scanOp
4    sWhen (boundary .==. sExt32 (unCount group_size * m))
5      ( do
6        forM_ (zip xs prefixes) $ \(x, prefix) -> dPrimV_ x $ tvExp prefix
7        forM_ (zip ys accs) $ \(y, acc) -> dPrimV_ y $ tvExp acc
8        compileStms mempty (bodyStms $ lambdaBody scanOp) $
9          everythingVolatile $
10           forM_ (zip incprefixArrays $ bodyResult $ lambdaBody scanOp) $
11             \(incprefixArray, res) ->
12               copyDWIMFix incprefixArray [tvExp dynamicId] res []
13       sOp globalFence
14       everythingVolatile $
15         copyDWIMFix statusFlags [tvExp dynamicId]
16                     (intConst Int8 statusP) []
17     )
18   forM_ (zip exchanges prefixes) $ \(exchange, prefix) ->
19     copyDWIMFix exchange [0] (tvSize prefix) []
20   forM_ (zip3 accs tys scanOpNe) $ \(acc, ty, ne) ->
21     tvVar acc <~~ toExp' ty ne
```

7.6 Result Distribution

The result distribution is the last part of the compiler that is changed. Lines 1 and 2 gets the parameters from the scan operation. `scanOp` and `scanOp'` is the same operation, only with renamed parameters to avoid naming conflicts, as we are writing to two sets of parameters at the same time.

Lines 4-9 writes the accumulator value and the prefix from the block-level scan and the lookback phase to the parameters of the scan operator. Then lines 11-14 combines the prefix and accumulator if there is no segment boundary between the start of the current thread and the end of the previous block. If there is, the distribution continues with just the accumulator value, as shown on line 17.

Lines 19-22 then calculates the remaining segment or `stopping_point` as described in section 6, with lines 24-31 generation the for loop which calculates the final values of the scan. Line 25 ensures that the accumulator is only combined if the elements is before the stopping point.

```
 1  let (xs, ys) = splitAt (length tys) $ map paramName $ lambdaParams scanOp
 2      (xs', ys') = splitAt (length tys) $ map paramName $ lambdaParams scanOp'
 3
 4      forM_ (zip4 (zip prefixes accs) (zip xs xs') (zip ys ys') tys) $
 5        \((prefix, acc), (x, x'), (y, y'), ty) -> do
 6          dPrim_ x ty
 7          dPrim_ y ty
 8          dPrimV_ x' $ tvExp prefix
 9          dPrimV_ y' $ tvExp acc
10
11      sIf (kernelLocalThreadId constants * m .<. boundary .&&. bNot blockNewSgm)
12        (compileStms mempty (bodyStms $ lambdaBody scanOp') $
13          forM_ (zip3 xs tys $ bodyResult $ lambdaBody scanOp') $
14            \(x, ty, res) -> x <~~ toExp' ty res)
15        (forM_ (zip xs accs) $
16          \(x, acc) ->
17            do copyDWIMFix x [] (Var $ tvVar acc) [])
18
19      stop <-
20        dPrimVE "stopping_point" $
21          segsize_compact - (kernelLocalThreadId constants * m - 1 +
22          segsize_compact - boundary) `rem` segsize_compact
23
24      sFor "i" m $ \i -> do
25        sWhen (sExt32 i .<. stop - 1) $ do
26          forM_ (zip privateArrays ys) $ \(src, y) ->
27              copyDWIMFix y [] (Var src) [i]
28          compileStms mempty (bodyStms $ lambdaBody scanOp) $
29            forM_ (zip privateArrays $ bodyResult $ lambdaBody scanOp) $
30              \(dest, res) ->
31                copyDWIMFix dest [i] res []
```

8 Empirical Validation

As mention in section 1.2, a regular segmented scan naturally occurs when mapping a scan over regular sized inputs. In order to benchmark the final implementation on different applications, we can thus take some programs that rely on the scan operator and apply them on a list of equally sized input problems.

The programs used for benchmarking and validation can be found in appendix B, but we give here a high-level overview on how the programs benefit from the scan operator:

- Longest Satisfying Streak Problem (LSSP):
  LSSP is a problem about finding the longest interval within a given array, such that the elements in the interval satisfies a provided predicate, e.g. being sorted or only being positive numbers.

  Although not immediately apparent how to solve this with a scan, the elements can be transformed to include extra information such as the longest streak that must start with the first element, longest streak that must end with the last element, longest streak encountered so far and so on. This transforms the problem into that of a ”near homomorphism”, a concept introduced by Gorlatch[4]. The scan operator can then use this extra information to compute the desired result.

- Radix-Sort:
  As mentioned in the introduction, radix-sorting can be implemented by using a scan to calculate the indices of elements during partitions. Radix-sorting performs a split of elements based on whether a specific bit is set or not, and scanning over those bits with the addition-operator results in the indices the element should be moved to during the split.

- KD-Tree:
  A KD-tree is a data-structure used for partitioning k-dimensional points and storing them for faster searching. The construction of the tree is done by sorting the points on a dimension, choosing the median, and splitting the points into two equally sized parts based on their relation to the median. This process is repeated on the two new parts until some lower threshold is hit.

  The sorting can be done using a radix-sort as just discussed, and since the construction creates equally sized partitions (with padding for odd length inputs), this already utilizes an underlying regular segmented scan. The construction of a KD-tree does thus not need to be mapped onto multiple inputs before we can bench the effects of the segmented scan implementation.

	GTX 780 Ti			RTX 2080 Ti		
	Time [µs]		Speedup	Time [µs]		Speedup
Application	2-Pass	1-Pass		2-Pass	1-Pass	
LSSP	7 444	5 343	x1.39	1 839	768	x2.39
Radix-Sort	151 045	121 267	x1.25	70 808	29 177	x2.43
KD-Tree	606 422	469 235	x1.29	268 952	140 461	x1.91

Table 2: Time measurement and speedups of different applications

The running times and improvements of the 3 applications can be seen in table 2. Each benchmark has been run 100 times, and the reported values are the averages of these runs.

The LSSP program is instanced to calculate the longest sorted streak from 10^5 32-bit integers with 100 problem inputs. The radix-sort sorts 10^4 32-bit integers and is given 1000 problem inputs. The KD-tree construction is provided one large problem input of around $2 \cdot 10^6$ 5-dimensional points, with 32-bit floats as coordinates.

On the GTX 780 Ti, the speedup is a modest 25%-39%. While still a significant speedup, the results are outshined by the 91%-143% speedup gained on the RTX 2080 Ti. This difference in performance increase is explained by the hardware specifications mentioned in section 5. The GTX 780 Ti has 3/4 the amount of shared memory available that the RTX 2080 Ti has, and as a result the RTX 2080 Ti is able to reduce inter-thread communication to a larger degree without suffering a performance penalty. This means that scans carried out on gpus with more resources get a higher performance boosts.

9 Conclusion & Future Work

In section 4 we have shown how to generalize Merrill & Garland's single-pass scan algorithm to also account for regular segments. This generalization makes use of short-circuiting calculations that span across multiple segments, further reducing latency when blocks propagate partial results to successors.

We additionally constructed an analytical model in section 5, to statically determine the amount of sequential work each thread should perform, with the assumption that device threads would not be starved of work. The model benefits the compiler by making it produce kernels with reduced inter-thread communication, without straining the resources of the device. The model conservatively chooses an amount of sequential work that is estimated to not require more memory resources than available, even though we have proved there are cases when it can be beneficial to set the amount of sequential work even higher.

We have shown how to safely change the segment calculations introduced in section 4 into 32-bit representable arithmetic. This change is proven in section 6 to preserve the original semantics, while increasing the performance of the generated code.

Finally, we have presented the extension to the Futhark compiler in section 7, and validated the quality of the compilation output by benchmarking scan-reliant Futhark programs in section 8.

There are still limitations to the implementation, which opens the possibility of further improvements:

- The model for determining sequential work uses hardware specifications in its calculations. In the implementation these values are hardcoded, meaning application of the model does not give accurate results for all gpus. Changing the implementation to query the device for these constants would allow for better general use.
- The calculation of the prefix during the lookback phase is done by a single thread. The calculation is done in such a way that it is guaranteed to fit within a warp, so applying a warp-level parallel calculation instead could lead to increased performance.
- The implementation only supports scanning with a single operator. The compiler can fuse multiple scans together, resulting in multiple operators which has to be combined into one for the current approach to function. Generalizing the implementation to accept multiple operators and reusing resources between application of each operator would limit the total resource usage. This would also mean updating the model for determining sequential work, as lower resource usage allows each thread to perform more sequential work without exceeding resource limits.

References

[1] G. E. Blelloch. “Scans as primitive parallel operations”. In: *IEEE Transactions on Computers* 38.11 (1989), pp. 1526–1538. DOI: 10.1109/12.42122.

[2] G. E. Blelloch, S. Chatterjee, and M. Zagha. “Scan primitives for vector computers”. In: *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. 1990, pp. 666–675.

[3] *CUDA C++ Programming Guide*. NVIDIA Corporation. 2788 San Tomas Expressway, Santa Clara, CA 95051, 2021.

[4] S. Gorlatch. “Systematic efficient parallelization of scan and other list homomorphisms”. In: *Euro-Par’96 Parallel Processing*. Ed. by Luc Bougé et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 401–408. ISBN: 978-3-540-70636-6.

[5] T. Henriksen. “Design and Implementation of the Futhark Programming Language”. PhD thesis. Universitetsparken 5, 2100 København: University of Copenhagen, Nov. 2017.

[6] D. Merrill and M. Garland. *Single-pass Parallel Prefix Scan with Decoupled Look-back*. Tech. rep. NVIDIA Corporation, 2016.

[7] G. E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (1965).

[8] M. Persson and A. Nicolaisen. “Implementing Single-Pass Scan in the Futhark Compiler”. MSc Project, Department of Computer Science, University of Copenhagen. 2020.

A Source Code

```haskell
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE TypeFamilies #-}

-- | Code generation for segmented and non-segmented scans.  Uses a
-- fast single-pass algorithm, but which only works on NVIDIA GPUs and
-- with some constraints on the operator.  We use this when we can.
module Futhark.CodeGen.ImpGen.Kernels.SegScan.SegSinglePass (compileSegScan) where

import Control.Monad.Except
import Data.List (zip4)
import Data.Maybe
import qualified Futhark.CodeGen.ImpCode.Kernels as Imp
import Futhark.CodeGen.ImpGen
import Futhark.CodeGen.ImpGen.Kernels.Base
import Futhark.IR.KernelsMem
import qualified Futhark.IR.Mem.IxFun as IxFun
import Futhark.Transform.Rename
import Futhark.Util (takeLast)
import Futhark.Util.IntegralExp (IntegralExp (mod, rem), divUp, quot)
import Prelude hiding (quot, mod, rem)

xParams, yParams :: SegBinOp KernelsMem -> [LParam KernelsMem]
xParams scan =
  take (length (segBinOpNeutral scan)) (lambdaParams (segBinOpLambda scan))
yParams scan =
  drop (length (segBinOpNeutral scan)) (lambdaParams (segBinOpLambda scan))

alignTo :: IntegralExp a => a -> a -> a
alignTo x a = (x `divUp` a) * a

createLocalArrays ::
  Count GroupSize SubExp ->
  SubExp ->
  [PrimType] ->
  InKernelGen (VName, [VName], [VName], VName, VName, [VName])
createLocalArrays (Count groupSize) m types = do
  let groupSizeE = toInt64Exp groupSize
      workSize = toInt64Exp m * groupSizeE
      prefixArraysSize =
        foldl (\acc tySize -> alignTo acc tySize + tySize * groupSizeE) 0 $
          map primByteSize types
      maxTransposedArraySize =
        foldl1 sMax64 $ map (\ty -> workSize * primByteSize ty) types

      warpSize :: Num a => a
      warpSize = 32
      maxWarpExchangeSize =
        foldl (\acc tySize -> alignTo acc tySize + tySize * fromInteger warpSize) 0 $
          map primByteSize types
      maxLookbackSize = maxWarpExchangeSize + warpSize
      size = Imp.bytes $ maxLookbackSize `sMax64` prefixArraysSize `sMax64` maxTransposedArraySize

      varTE :: TV Int64 -> TPrimExp Int64 VName
      varTE = le64 . tvVar

  byteOffsets <-
    mapM (fmap varTE . dPrimV "byte_offsets") $
      scanl (\off tySize -> alignTo off tySize + toInt64Exp groupSize * tySize) 0 $
        map primByteSize types

  warpByteOffsets <-
    mapM (fmap varTE . dPrimV "warp_byte_offset") $
      scanl (\off tySize -> alignTo off tySize + warpSize * tySize) warpSize $
        map primByteSize types

  sComment "Allocate_reused_shared_memeory" $ return ()

  localMem <- sAlloc "local_mem" size (Space "local")
  transposeArrayLength <- dPrimV "trans_arr_len" workSize

  sharedId <- sArrayInMem "shared_id" int32 (Shape [constant (1 :: Int32)]) localMem
  sharedReadOffset <- sArrayInMem "shared_read_offset" int32 (Shape [constant (1 :: Int32)]) localMem

  transposedArrays <-
    forM types $ \ty ->
      sArrayInMem
        "local_transpose_arr"
        ty
        (Shape [tvSize transposeArrayLength])
        localMem

  prefixArrays <-
    forM (zip byteOffsets types) $ \(off, ty) -> do
      let off' = off `quot` primByteSize ty
      sArray
        "local_prefix_arr"
        ty
        (Shape [groupSize])
        $ ArrayIn localMem $ IxFun.iotaOffset off' [pe64 groupSize]

  warpscan <- sArrayInMem "warpscan" int8 (Shape [constant (warpSize :: Int64)]) localMem
  warpExchanges <-
    forM (zip warpByteOffsets types) $ \(off, ty) -> do
```

```
      let off' = off `quot` primByteSize ty
      sArray
        "warp_exchange"
        ty
        (Shape [constant (warpSize :: Int64)])
        $ ArrayIn localMem $ IxFun.iotaOffset off' [warpSize]

  return (sharedId, transposedArrays, prefixArrays, sharedReadOffset, warpscan, warpExchanges)
-- | Compile 'SegScan' instance to host-level code with calls to a
-- single-pass kernel.
compileSegScan ::
  Pattern KernelsMem ->
  SegLevel ->
  SegSpace ->
  SegBinOp KernelsMem ->
  KernelBody KernelsMem ->
  CallKernelGen ()
compileSegScan pat lvl space scanOp kbody = do
  let Pattern _ all_pes = pat
      group_size = toInt64Exp <$> segGroupSize lvl
      n = product $ map toInt64Exp $ segSpaceDims space
      sumT :: Integer
      maxT :: Integer
      sumT = foldl (\bytes typ -> bytes + primByteSize typ) 0 tys
      primByteSize' = max 4 . primByteSize
      sumT' = foldl (\bytes typ -> bytes + primByteSize' typ) 0 tys `div` 4
      maxT = maximum (map primByteSize tys)
      -- TODO: Make these constants dynamic by querying device
      -- RTX 2080 Ti constants (CC 7.5)
      -- k_reg = 64
      -- k_mem = 48 --12*4
      -- GTX 780 Ti constants (CC 3.5)
      k_reg = 64
      k_mem = 36 --9*4
      mem_constraint = max k_mem sumT `div` maxT
      --reg_constraint = (k_reg `div` sumT) - 6
      reg_constraint = (k_reg-1-sumT') `div` (2*sumT'+3)
      m :: Num a => a
      m = fromIntegral $ max 1 $ min mem_constraint reg_constraint
      num_groups = Count (n `divUp` (unCount group_size * m))
      num_threads = unCount num_groups * unCount group_size
      (gtids, dims) = unzip $ unSegSpace space
      dims' = map toInt64Exp dims
      segment_size = last dims'
      scanOpNe = segBinOpNeutral scanOp
      tys = map (\(Prim pt) -> pt) $ lambdaReturnType $ segBinOpLambda scanOp
      statusX, statusA, statusP :: Num a => a
      statusX = 0
      statusA = 1
      statusP = 2
      makeStatusUsed flag used = tvExp flag .|. (tvExp used .<<. 2)
      unmakeStatusUsed :: TV Int8 -> TV Int8 -> TV Int8 -> InKernelGen ()
      unmakeStatusUsed flagUsed flag used = do
        used <-- tvExp flagUsed .>>. 2
        flag <-- tvExp flagUsed .&. 3

  -- Allocate the shared memory for output component
  numGroups <- dPrimV "numGroups" $ unCount num_groups
  numThreads <- dPrimV "numThreads" num_threads

  globalId <- sStaticArray "id_counter" (Space "device") int32 $ Imp.ArrayZeros 1
  statusFlags <- sAllocArray "status_flags" int8 (Shape [tvSize numGroups]) (Space "device")
  (aggregateArrays, incprefixArrays) <-
    fmap unzip $
      forM tys $ \ty ->
        (,) <$> sAllocArray "aggregates" ty (Shape [tvSize numGroups]) (Space "device")
          <*> sAllocArray "incprefixes" ty (Shape [tvSize numGroups]) (Space "device")

  sReplicate statusFlags $ intConst Int8 statusX

  sKernelThread "segscan" num_groups group_size (segFlat space) $ do
    constants <- kernelConstants <$> askEnv

    (sharedId, transposedArrays, prefixArrays, sharedReadOffset, warpscan, exchanges) <-
      createLocalArrays (segGroupSize lvl) (intConst Int64 m) tys

    dynamicId <- dPrim "dynamic_id" int32 :: ImpM lore r op (TV Int64)
    sWhen (kernelLocalThreadId constants .==. 0) $ do
      (globalIdMem, _, globalIdOff) <- fullyIndexArray globalId [0]
      sOp $
        Imp.Atomic DefaultSpace $
          Imp.AtomicAdd
            Int32
            (tvVar dynamicId)
            globalIdMem
            (Count $ unCount globalIdOff)
            (untyped (1 :: Imp.TExp Int32))
      copyDWIMFix sharedId [0] (tvSize dynamicId) []

    let localBarrier = Imp.Barrier Imp.FenceLocal
        localFence = Imp.MemFence Imp.FenceLocal
        globalFence = Imp.MemFence Imp.FenceGlobal

    sOp localBarrier
    copyDWIMFix (tvVar dynamicId) [] (Var sharedId) [0]
    sOp localBarrier

```

```haskell
blockOff <-
  dPrimV "blockOff" $
    sExt64 (tvExp dynamicId) * m * kernelGroupSize constants
sgmIdx <- dPrimVE "sgm_idx" $ tvExp blockOff `mod` segment_size
boundary <-
  dPrimVE "boundary" $
    sExt32 $ sMin64 (m * unCount group_size) (segment_size - sgmIdx)
segsize_compact <-
  dPrimVE "segsize_compact" $
    sExt32 $ sMin64 (m * unCount group_size) segment_size
privateArrays <-
  forM tys $ \ty ->
    sAllocArray
      "private"
      ty
      (Shape [intConst Int64 m])
      (ScalarSpace [intConst Int64 m] ty)
sComment "Load␣and␣map" $
  sFor "i" m $ \i -> do
    -- The map's input index
    phys_tid <- dPrimVE "phys_tid" $
      tvExp blockOff + sExt64 (kernelLocalThreadId constants)
        + i * kernelGroupSize constants
    zipWithM_ dPrimV_ gtids $ unflattenIndex dims' phys_tid
    -- Perform the map
    let in_bounds =
          compileStms mempty (kernelBodyStms kbody) $ do
            let (all_scan_res, map_res) = splitAt (segBinOpResults [scanOp]) $ kernelBodyResult kbody

            -- Write map results to their global memory destinations
            forM_ (zip (takeLast (length map_res) all_pes) map_res) $ \(dest, src) ->
              copyDWIMFix (patElemName dest) (map Imp.vi64 gtids) (kernelResultSubExp src) []

            -- Write to-scan results to private memory.
            forM_ (zip privateArrays $ map kernelResultSubExp all_scan_res) $ \(dest, src) ->
              copyDWIMFix dest [i] src []

        out_of_bounds =
          forM_ (zip privateArrays scanOpNe) $ \(dest, ne) ->
            copyDWIMFix dest [i] ne []

    sIf (phys_tid .<. n) in_bounds out_of_bounds

sComment "Transpose␣scan␣inputs" $ do
  forM_ (zip transposedArrays privateArrays) $ \(trans, priv) -> do
    sOp localBarrier
    sFor "i" m $ \i -> do
      sharedIdx <-
        dPrimVE "sharedIdx" $
          sExt64 (kernelLocalThreadId constants)
            + i * kernelGroupSize constants
      copyDWIMFix trans [sharedIdx] (Var priv) [i]
    sOp localBarrier
    sFor "i" m $ \i -> do
      sharedIdx <- dPrimV "sharedIdx" $ kernelLocalThreadId constants * m + i
      copyDWIMFix priv [sExt64 i] (Var trans) [sExt64 $ tvExp sharedIdx]
  sOp localBarrier

sComment "Per␣thread␣scan" $ do
  -- We don't need to touch the first element, so only m-1
  -- iterations here.
  globalIdx <-
    dPrimVE "gidx" $
      (kernelLocalThreadId constants * m) + 1
  sFor "i" (m -1) $ \i -> do
    let xs = map paramName $ xParams scanOp
        ys = map paramName $ yParams scanOp
    -- determine if start of segment
    isNewSgm <-
      dPrimVE "new_sgm" $ (globalIdx + sExt32 i - boundary) `mod` segsize_compact .==. 0
    -- skip scan of first element in segment
    sUnless isNewSgm $ do
      forM_ (zip privateArrays $ zip3 xs ys tys) $ \(src, (x, y, ty)) -> do
        dPrim_ x ty
        dPrim_ y ty
        copyDWIMFix x [] (Var src) [i]
        copyDWIMFix y [] (Var src) [i + 1]

      compileStms mempty (bodyStms $ lambdaBody $ segBinOpLambda scanOp) $
        forM_ (zip privateArrays $ bodyResult $ lambdaBody $ segBinOpLambda scanOp) $ \(dest, res) ->
          copyDWIMFix dest [i + 1] res []

sComment "Publish␣results␣in␣shared␣memory" $ do
  forM_ (zip prefixArrays privateArrays) $ \(dest, src) ->
    copyDWIMFix dest [sExt64 $ kernelLocalThreadId constants] (Var src) [m - 1]
  sOp localBarrier

let crossesSegment =
      Just $ \from to ->
      let from' = (from + 1) * m - 1
          to'   = (to   + 1) * m - 1
      in (to' - from') .>. (to'+segsize_compact-boundary) `mod` segsize_compact

scanOp' <- renameLambda $ segBinOpLambda scanOp

accs <- mapM (dPrim "acc") tys
sComment "Scan␣results␣(with␣warp␣scan)" $ do
```

```haskell
      groupScan
        crossesSegment
        (tvExp numThreads)
        (kernelGroupSize constants)
        scanOp'
        prefixArrays

      sOp localBarrier
      let firstThread acc prefixes =
            copyDWIMFix (tvVar acc) [] (Var prefixes) [sExt64 (kernelGroupSize constants) - 1]
          notFirstThread acc prefixes =
            copyDWIMFix (tvVar acc) [] (Var prefixes) [sExt64 (kernelLocalThreadId constants) - 1]
      sIf
        (kernelLocalThreadId constants .==. 0)
        (zipWithM_ firstThread accs prefixArrays)
        (zipWithM_ notFirstThread accs prefixArrays)

      sOp localBarrier
    prefixes <-
      forM (zip scanOpNe tys) $ \(ne, ty) ->
        dPrimV "prefix" $ TPrimExp $ toExp' ty ne
    blockNewSgm <- dPrimVE "block_new_sgm" $ sgmIdx .==. 0
    sComment "Perform_lookback" $ do
      sWhen (blockNewSgm .&&. kernelLocalThreadId constants .==. 0) $ do
        everythingVolatile $
          forM_ (zip incprefixArrays accs) $ \(incprefixArray, acc) ->
            copyDWIMFix incprefixArray [tvExp dynamicId] (tvSize acc) []
        sOp globalFence
        everythingVolatile $
          copyDWIMFix statusFlags [tvExp dynamicId] (intConst Int8 statusP) []
        forM_ (zip scanOpNe accs) $ \(ne, acc) ->
          copyDWIMFix (tvVar acc) [] ne []
      -- end sWhen

      let warpSize = kernelWaveSize constants
      sWhen (bNot blockNewSgm .&&. kernelLocalThreadId constants .<. warpSize) $ do
        sWhen (kernelLocalThreadId constants .==. 0) $ do
          sIf (boundary .==. sExt32 (unCount group_size * m))
            ( do
              everythingVolatile $
                forM_ (zip aggregateArrays accs) $ \(aggregateArray, acc) ->
                  copyDWIMFix aggregateArray [tvExp dynamicId] (tvSize acc) []
              sOp globalFence
              everythingVolatile $
                copyDWIMFix statusFlags [tvExp dynamicId] (intConst Int8 statusA) []
            )
            (
              do
              everythingVolatile $
                forM_ (zip incprefixArrays accs) $ \(incprefixArray, acc) ->
                  copyDWIMFix incprefixArray [tvExp dynamicId] (tvSize acc) []
              sOp globalFence
              everythingVolatile $
                copyDWIMFix statusFlags [tvExp dynamicId] (intConst Int8 statusP) []
            )
          copyDWIMFix warpscan [0] (Var statusFlags) [tvExp dynamicId - 1]
        -- sWhen
        sOp localFence

        status <- dPrim "status" int8 :: InKernelGen (TV Int8)
        copyDWIMFix (tvVar status) [] (Var warpscan) [0]

        sIf
          (tvExp status .==. statusP)
          ( sWhen (kernelLocalThreadId constants .==. 0) $
              everythingVolatile $
                forM_ (zip prefixes incprefixArrays) $ \(prefix, incprefixArray) ->
                  copyDWIMFix (tvVar prefix) [] (Var incprefixArray) [tvExp dynamicId - 1]
          )
          ( do
              readOffset <-
                dPrimV "readOffset" $
                  sExt32 $ tvExp dynamicId - sExt64 (kernelWaveSize constants)
              let loopStop = warpSize * (-1)
                  sameSegment readIdx =
                    let startIdx = sExt64 (tvExp readIdx + 1) * kernelGroupSize constants * m - 1
                    in tvExp blockOff - startIdx .<=. sgmIdx
              sWhile (tvExp readOffset .>. loopStop) $ do
                readI <- dPrimV "read_i" $ tvExp readOffset + kernelLocalThreadId constants
                aggrs <- forM (zip scanOpNe tys) $ \(ne, ty) ->
                  dPrimV "aggr" $ TPrimExp $ toExp' ty ne
                flag <- dPrimV "flag" statusX
                used <- dPrimV "used" (0 :: Imp.TExp Int8)
                everythingVolatile $
                  sIf (tvExp readI .>=. 0 .&&. sameSegment readI)
                    (do
                    copyDWIMFix (tvVar flag) [] (Var statusFlags) [sExt64 $ tvExp readI]
                    sIf
                      (tvExp flag .==. statusP)
                      ( forM_ (zip incprefixArrays aggrs) $ \(incprefix, aggr) ->
                          copyDWIMFix (tvVar aggr) [] (Var incprefix) [sExt64 $ tvExp readI]
                      )
                      ( sWhen (tvExp flag .==. statusA) $ do
                          forM_ (zip aggrs aggregateArrays) $ \(aggr, aggregate) ->
                            copyDWIMFix (tvVar aggr) [] (Var aggregate) [sExt64 $ tvExp readI]
                          used <-- (1 :: Imp.TExp Int8)
                      ))
                    (sWhen (tvExp readI .>=. 0) $
                      copyDWIMFix (tvVar flag) [] (intConst Int8 statusP) [])
                -- end sIf
                -- end sWhen
                forM_ (zip exchanges aggrs) $ \(exchange, aggr) ->
                  copyDWIMFix exchange [sExt64 $ kernelLocalThreadId constants] (tvSize aggr) []
                tmp <- dPrimV "tmp" $ makeStatusUsed flag used
                copyDWIMFix warpscan [sExt64 $ kernelLocalThreadId constants] (tvSize tmp) []
                sOp localFence

                (warpscanMem, warpscanSpace, warpscanOff) <-
                  fullyIndexArray warpscan [sExt64 warpSize - 1]
                flag <-- TPrimExp (Imp.index warpscanMem warpscanOff int8 warpscanSpace Imp.Volatile)
                sWhen (kernelLocalThreadId constants .==. 0) $ do
                  -- TODO: This is a single-threaded reduce
                  sIf
                    (bNot $ tvExp flag .==. statusP)
                    ( do
                        scanOp'' <- renameLambda scanOp'
                        let (agg1s, agg2s) = splitAt (length tys) $ map paramName $ lambdaParams scanOp''

                        forM_ (zip3 agg1s scanOpNe tys) $ \(agg1, ne, ty) ->
                          dPrimV_ agg1 $ TPrimExp $ toExp' ty ne
                        zipWithM_ dPrim_ agg2s tys

                        flag1 <- dPrimV "flag1" statusX
                        flag2 <- dPrim "flag2" int8
                        used1 <- dPrimV "used1" (0 :: Imp.TExp Int8)
                        used2 <- dPrim "used2" int8
                        sFor "i" warpSize $ \i -> do
                          copyDWIMFix (tvVar flag2) [] (Var warpscan) [sExt64 i]
                          unmakeStatusUsed flag2 flag2 used2
                          forM_ (zip agg2s exchanges) $ \(agg2, exchange) ->
                            copyDWIMFix agg2 [] (Var exchange) [sExt64 i]
                          sIf
                            (bNot $ tvExp flag2 .==. statusA)
                            ( do
                                flag1 <-- tvExp flag2
                                used1 <-- tvExp used2
                                forM_ (zip3 agg1s tys agg2s) $ \(agg1, ty, agg2) ->
                                  agg1 <~~ toExp' ty (Var agg2)
                            )
                            ( do
                                used1 <-- tvExp used1 + tvExp used2
                                compileStms mempty (bodyStms $ lambdaBody scanOp'') $
                                  forM_ (zip3 agg1s tys $ bodyResult $ lambdaBody scanOp'') $
                                    \(agg1, ty, res) -> agg1 <~~ toExp' ty res
                            )
                        flag <-- tvExp flag1
                        used <-- tvExp used1
                        forM_ (zip3 aggrs tys agg1s) $ \(aggr, ty, agg1) ->
                          tvVar aggr <~~ toExp' ty (Var agg1)
                    )
                    -- else
                    ( forM_ (zip aggrs exchanges) $ \(aggr, exchange) ->
                        copyDWIMFix (tvVar aggr) [] (Var exchange) [sExt64 warpSize - 1]
                    )
                  -- end sIf
                  sIf
                    (tvExp flag .==. statusP)
                    (readOffset <-- loopStop)
                    (readOffset <-- tvExp readOffset - zExt32 (tvExp used))
                  copyDWIMFix sharedReadOffset [0] (tvSize readOffset) []
                  scanOp''' <- renameLambda scanOp'
                  let (xs, ys) = splitAt (length tys) $ map paramName $ lambdaParams scanOp'''
                  forM_ (zip xs aggrs) $ \(x, aggr) -> dPrimV_ x (tvExp aggr)
                  forM_ (zip ys prefixes) $ \(y, prefix) -> dPrimV_ y (tvExp prefix)
                  compileStms mempty (bodyStms $ lambdaBody scanOp''') $
                    forM_ (zip3 prefixes tys $ bodyResult $ lambdaBody scanOp''') $
                      \(prefix, ty, res) -> prefix <-- TPrimExp (toExp' ty res)
                -- end sWhen
                sOp localFence
                copyDWIMFix (tvVar readOffset) [] (Var sharedReadOffset) [0]
          )
        -- end sWhile
        -- end sIf
        sWhen (kernelLocalThreadId constants .==. 0) $ do
          scanOp'''' <- renameLambda scanOp'
          let xs = map paramName $ take (length tys) $ lambdaParams scanOp''''
              ys = map paramName $ drop (length tys) $ lambdaParams scanOp''''
          sWhen
            (boundary .==. sExt32 (unCount group_size * m))
            ( do
              forM_ (zip xs prefixes) $ \(x, prefix) -> dPrimV_ x $ tvExp prefix
              forM_ (zip ys accs) $ \(y, acc) -> dPrimV_ y $ tvExp acc
              compileStms mempty (bodyStms $ lambdaBody scanOp'''') $
                everythingVolatile $
                  forM_ (zip incprefixArrays $ bodyResult $ lambdaBody scanOp'''') $
                    \(incprefixArray, res) -> copyDWIMFix incprefixArray [tvExp dynamicId] res []
              sOp globalFence
              everythingVolatile $ copyDWIMFix statusFlags [tvExp dynamicId] (intConst Int8 statusP) []
            )
          forM_ (zip exchanges prefixes) $ \(exchange, prefix) ->
            copyDWIMFix exchange [0] (tvSize prefix) []
          forM_ (zip3 accs tys scanOpNe) $ \(acc, ty, ne) ->
            tvVar acc <~~ toExp' ty ne
      -- end sWhen
```

```haskell
      -- end sWhen

      sWhen (bNot $ tvExp dynamicId .==. 0) $ do
        sOp localBarrier
        forM_ (zip exchanges prefixes) $ \(exchange, prefix) ->
          copyDWIMFix (tvVar prefix) [] (Var exchange) [0]
        sOp localBarrier
    -- end sWhen
    -- end sComment

    scanOp''''' <- renameLambda scanOp'
    scanOp'''''' <- renameLambda scanOp'

    sComment "Distribute␣results" $ do
      let (xs, ys) = splitAt (length tys) $ map paramName $ lambdaParams scanOp'''''
          (xs', ys') = splitAt (length tys) $ map paramName $ lambdaParams scanOp''''''

      forM_ (zip4 (zip prefixes accs) (zip xs xs') (zip ys ys') tys) $
        \((prefix, acc), (x, x'), (y, y'), ty) -> do
          dPrim_ x ty
          dPrim_ y ty
          dPrimV_ x' $ tvExp prefix
          dPrimV_ y' $ tvExp acc

      sIf (kernelLocalThreadId constants * m .<. boundary .&&. bNot blockNewSgm)
        (compileStms mempty (bodyStms $ lambdaBody scanOp'''''') $
          forM_ (zip3 xs tys $ bodyResult $ lambdaBody scanOp'''''') $
            \(x, ty, res) -> x <~~ toExp' ty res)
        (forM_ (zip xs accs) $
          \(x, acc) ->
            do copyDWIMFix x [] (Var $ tvVar acc) [])
      -- calculate where previous thread stopped, to determine number of
      -- elements left before new segment.
      stop <-
        dPrimVE "stopping_point" $
          segsize_compact - (kernelLocalThreadId constants * m - 1 + segsize_compact - boundary) `rem` segsize_compact
      sFor "i" m $ \i -> do
        sWhen (sExt32 i .<. stop - 1) $ do
          forM_ (zip privateArrays ys) $ \(src, y) ->
            -- only include prefix for the first segment part per thread
              copyDWIMFix y [] (Var src) [i]
          compileStms mempty (bodyStms $ lambdaBody scanOp''''') $
            forM_ (zip privateArrays $ bodyResult $ lambdaBody scanOp''''') $
              \(dest, res) ->
                copyDWIMFix dest [i] res []

    sComment "Transpose␣scan␣output" $ do
      forM_ (zip transposedArrays privateArrays) $ \(trans, priv) -> do
        sOp localBarrier
        sFor "i" m $ \i -> do
          sharedIdx <-
            dPrimV "sharedIdx" $
              sExt64 (kernelLocalThreadId constants * m) + i
          copyDWIMFix trans [tvExp sharedIdx] (Var priv) [i]
        sOp localBarrier
        sFor "i" m $ \i -> do
          sharedIdx <-
            dPrimV "sharedIdx" $
              kernelLocalThreadId constants
                + sExt32 (kernelGroupSize constants * i)
          copyDWIMFix priv [i] (Var trans) [sExt64 $ tvExp sharedIdx]
      sOp localBarrier

    sComment "Write␣block␣scan␣results␣to␣global␣memory" $
        sFor "i" m $ \i -> do
          flat_idx <-
            dPrimVE "flat_idx" $
              tvExp blockOff + kernelGroupSize constants * i
                + sExt64 (kernelLocalThreadId constants)
          zipWithM_ dPrimV_ gtids $ unflattenIndex dims' flat_idx
          sWhen (flat_idx .<. n) $ do
            forM_ (zip (map patElemName all_pes) privateArrays) $ \(dest, src) ->
              copyDWIMFix dest (map Imp.vi64 gtids) (Var src) [i]

    sComment "If␣this␣is␣the␣last␣block,␣reset␣the␣dynamicId" $
      sWhen (tvExp dynamicId .==. unCount num_groups - 1) $
        copyDWIMFix globalId [0] (constant (0 :: Int32)) []
```

B Benchmarked Programs

B.1 LSSP

```
type int = i32
let max (x:int, y:int) = i32.max x y
let scanOp (pred2 : int -> int -> bool)
          (x: (int,int,int,int,int,int))
          (y: (int,int,int,int,int,int))
        : (int,int,int,int,int,int) =
  let (lssx, lisx, lcsx, tlx, firstx, lastx) = x
  let (lssy, lisy, lcsy, tly, firsty, lasty) = y

  let connect= pred2 lastx firsty || tlx == 0 || tly == 0
  let newlss = if connect then max (max (lcsx + lisy, lssx), lssy)
                          else max (lssx, lssy)
  let newlis = if lisx == tlx && connect then tlx + lisy else lisx
  let newlcs = if lcsy == tly && connect then tly + lcsx else lcsy
  let newtl  = tlx + tly
  let first  = if tlx == 0 then firsty else firstx
  let last   = if tly == 0 then lastx else lasty in
  (newlss, newlis, newlcs, newtl, first, last)

let mapOp (pred1 : int -> bool) (x: int) : (int,int,int,int,int,int) =
  let xmatch = if pred1 x then 1 else 0 in
  (xmatch, xmatch, xmatch, 1, x, x)

let lssp (pred1 : int -> bool)
          (pred2 : int -> int -> bool)
          (xs    : []int ) : int =
  let (x,_,_,_,_,_) =
        last <|
        scan (scanOp pred2) (0,0,0,0,0,0) <|
        map (mapOp pred1) xs
  in  x

-- ==
-- entry: lssp_bench_sorted
-- compiled random input { [1][10000000]i32 } auto output
-- compiled random input { [10][1000000]i32 } auto output
-- compiled random input { [100][100000]i32 } auto output
-- compiled random input { [1000][10000]i32 } auto output
entry lssp_bench_sorted [m] [n] (xs : [m][n]i32) =
  let pred1 _ = true
  let pred2 x y = x <= y
  in map (lssp pred1 pred2) xs
```

B.2 Radix-Sort

Source: https://github.com/diku-dk/futhark-benchmarks/blob/master/misc/knn-by-kdtree/lib/github.com/diku-dk/sorts/radix_sort.fut

```
local let radix_sort_step [n] 't (xs: [n]t) (get_bit: i32 -> t -> i32)
                                  (digit_n: i32): [n]t =
  let num x = get_bit (digit_n+1) x * 2 + get_bit digit_n x
  let pairwise op (a1,b1,c1,d1) (a2,b2,c2,d2) =
    (a1 `op` a2, b1 `op` b2, c1 `op` c2, d1 `op` d2)
  let bins = xs |> map num
  let flags = bins |> map (\x -> if x == 0 then (1,0,0,0)
                                 else if x == 1 then (0,1,0,0)
                                 else if x == 2 then (0,0,1,0)
                                 else (0,0,0,1))
  let offsets = scan (pairwise (+)) (0,0,0,0) flags
  let (na,nb,nc,_nd) = last offsets
  let f bin (a,b,c,d) = match bin
                        case 0 -> a-1
                        case 1 -> na+b-1
                        case 2 -> na+nb+c-1
                        case _ -> na+nb+nc+d-1
  let is = map2 f bins offsets
  in scatter (copy xs) is xs

let radix_sort [n] 't (num_bits: i32) (get_bit: i32 -> t -> i32)
                      (xs: [n]t): [n]t =
  let iters = if n == 0 then 0 else (num_bits+2-1)/2
  in loop xs for i < iters do radix_sort_step xs get_bit (i*2)

let radix_sort_int [n] 't (num_bits: i32) (get_bit: i32 -> t -> i32)
                          (xs: [n]t): [n]t =
  let get_bit' i x =
    -- Flip the most significant bit.
    let b = get_bit i x
    in if i == num_bits-1 then b ^ 1 else b
  in radix_sort num_bits get_bit' xs

-- ==
-- random input { [1000][10000]i32 } auto output
entry main [n] [m] (data : [n][m]i32) =
  map (radix_sort_int i32.num_bits i32.get_bit) data
```

B.3 KD-Tree

Source: https://github.com/diku-dk/futhark-benchmarks/blob/master/misc/knn-by-kdtree/buildKDtree.fut

```
-- ==
-- entry: main
--
-- compiled input @ valid-data/kdtree-ppl-32-m-2097152.in
-- output @ valid-data/kdtree-ppl-32-m-2097152.out

import "lib/github.com/diku-dk/sorts/radix_sort"
import "util"

let iota32 n = (0..1..<i32.i64 n) :> [n]i32

local let closestLog2 (p: i32) : i32 =
    if p<=1 then 0
    else let (_,res) = loop (q,r) = (p,0)
                         while q > 1 do
                              (q >> 1, r+1)
           let err_down = p - (1 << res)
           let err_upwd = (1 << (res+1)) - p
           in  if err_down <= err_upwd
                 then res else res+1

let computeTreeShape (m: i32) (defppl: i32) : (i32, i32, i32, i32) =
    let def_num_leaves = (m + defppl - 1) / defppl
    let hp1 = closestLog2 def_num_leaves in
    if hp1 <= 0 then (-1, 0, m, m)
    else let h = hp1 - 1
           let num_leaves = 1 << (h+1)
           let ppl  = (m + num_leaves - 1) / num_leaves
           in  (h, num_leaves-1, ppl, num_leaves*ppl)

local let updateBounds [n] [d2] (level: i32) (median_dims: [n]i32)
                                (median_vals: [n]f32)
                                (node_ind: i32)
                                (lubs_cur: *[d2]f32) : *[d2]f32=
    let d = d2 / 2
    let ancestor = 0
    let (_, res) =
      loop (ancestor,lubs_cur) for i < level do
        let k = level - i - 1
        let ancestor_child = compute_Kth_ancestor k node_ind
        let anc_dim = median_dims[ancestor]
        let lub_ind = if  (ancestor_child & 1) == 0
                      then anc_dim
                      else i32.i64 d+anc_dim
        let anc_med = median_vals[ancestor]
        let lubs_cur[lub_ind] = if !(f32.isinf anc_med)
```

```
                                then anc_med
                                else lubs_cur[lub_ind]
        in  (ancestor_child, lubs_cur)
    in  res

local let findClosestMed [n] (cur_dim: i32) (median_dims: [n]i32)
                             (node_ind: i32) : i32 =
    let cur_node = node_ind
    let res_ind  = -1i32
    let (_, res) =
        loop (cur_node, res_ind)
          while (cur_node != 0) && (res_ind == (-1i32)) do
            let parent = getParent cur_node
            let res_ind = if median_dims[parent] == cur_dim then parent else -1
            in  (parent, res_ind)
    in  res
let mkKDtree [m] [d] (height: i32) (q: i64) (m' : i64)
                      (input: [m][d]f32) :
            (*[m'][d]f32, *[m']i32, *[q]i32, *[q]f32, *[q]i32) =

          let inputT = transpose input
          let lbs = map (reduce_comm f32.min f32.highest) inputT |> opaque
          let ubs = map (reduce_comm f32.max f32.lowest ) inputT |> opaque
          let lubs = lbs ++ ubs

          let num_pads = m' - m
          let input' = input ++ (replicate num_pads (replicate d f32.inf))
                       :> [m'][d]f32
          let indir  = iota32 m'

          let median_vals = replicate q 0.0f32
          let median_dims = replicate q (-1i32)
          let clanc_eqdim = replicate q (-1i32)
          let ( indir' : *[m']i32
              , median_dims': *[q]i32
              , median_vals': *[q]f32
              , clanc_eqdim': *[q]i32
              ) =
            loop ( indir  : *[m']i32
                , median_dims: *[q]i32
                , median_vals: *[q]f32
                , clanc_eqdim: *[q]i32 )
              for lev < (height+1) do
                let nodes_this_lvl = 1 << i64.i32 lev
                let pts_per_node_at_lev = m' / nodes_this_lvl
                let indir2d = unflatten nodes_this_lvl pts_per_node_at_lev indir

                let (med_dims, anc_same_med) =
                    map (\(i: i32) ->
                            let node_ind = i + i32.i64 nodes_this_lvl - 1
```

```
                  let diffs =
                    map (\i ->
                      f32.abs(lubs_cur[i+i32.i64 d] - lubs_cur[i]))
                    (iota32 d)
                  let (cur_dim, _) =
                    reduce_comm (\ (i1,v1) (i2,v2) ->
                                    if v1 >= v2 then (i1, v1)
                                                else (i2, v2) )
                                (-1, f32.lowest)
                                <| zip (iota32 d) diffs
                  let prev_anc = findClosestMed cur_dim median_dims
                                 node_ind
                  in  (cur_dim, prev_anc)
              ) (iota32 nodes_this_lvl)
          |> unzip

      let chosen_columns = map2 (\indir_chunk dim ->
                                  map (\ind -> input'[ind, dim]
                                      ) indir_chunk
                                ) indir2d med_dims

      let (sorted_dim_2d, sort_inds_2d) =
          map2 zip chosen_columns
            (replicate nodes_this_lvl (iota32 pts_per_node_at_lev))
          |> map (radix_sort_float_by_key (\(l,_) -> l)
             f32.num_bits f32.get_bit)
          |> map unzip |> unzip

      let med_vals = map  (\sorted_dim ->
                              let mi = pts_per_node_at_lev/2
                              in (sorted_dim[mi] + sorted_dim[mi-1])/2
                          ) sorted_dim_2d

      let indir2d' = map2(\ indir_chunk sort_inds ->
                              map (\ind -> indir_chunk[ind]) sort_inds
                        ) indir2d sort_inds_2d

      let this_lev_inds = map (+ (nodes_this_lvl-1))
                              (iota nodes_this_lvl)
      let median_dims' = scatter median_dims this_lev_inds med_dims
      let median_vals' = scatter median_vals this_lev_inds med_vals
      let clanc_eqdim' = scatter clanc_eqdim this_lev_inds anc_same_med
      let indir'' = flatten indir2d' :> *[m']i32

      in  (indir'', median_dims', median_vals', clanc_eqdim')

let input'' = map (\ ind -> map (\k -> input'[ind, k]) (iota32 d) )
              indir' :> *[m'][d]f32
in  (input'', indir', median_dims', median_vals', clanc_eqdim')
```

```

let main0 (m: i32) (defppl: i32) =
    computeTreeShape m defppl

let main [m] [d] (defppl: i32) (input: [m][d]f32) =
    let (height, num_inner_nodes, _, m') = computeTreeShape (i32.i64 m) defppl
    let (leafs, indir, median_dims, median_vals, clanc_eqdim) =
        mkKDtree height (i64.i32 num_inner_nodes) (i64.i32 m') input
    let r = i64.i32 (m' / 64)
    in  (leafs[:r], indir[:r], median_dims, median_vals, clanc_eqdim)
```