Code Generation for Stencils in Futhark

Maya Saietz

December 2020

Abstract

In this thesis, I investigate two different optimizations – tiling and partitioning – for stencil computations on parallel architectures through prototyping in CUDA and C, and implement code generation for stencils in the Futhark compiler's multicore backend. The prototyping reveals that the two optimizations in combination yield significant performance increase for multithreaded C programs, compared to an unoptimized version. When benchmarking the code generated by the Futhark compiler implementation, the fully optimized version again far outperformed the non-optimized version, but did not outperform implementations of the same stencils in terms of non-stencil-specific Futhark constructs.

Contents

1 Introduction

A *stencil* computation is a computation that updates elements of an array based on nearby elements. A simple example is a function that blurs an image by replacing each pixel's color with the average of the nearby (original) pixels' colors.

Stencil computations are used in a wide variety of fields, including medical imaging, physics simulations, and machine learning. They are parallel, in the sense that the elements of the array can be updated in any order, but creating optimal code for a stencil computation on parallel architectures is non-trivial and requires extensive knowledge of the hardware to best take advantage of data locality.

Futhark[3] is a statically typed data-parallel functional programming language, targeting GPUs and multicore arcitechtures. The purpose of Futhark is to allow programmers to write high-level code that compiles to efficient low-level code. The Futhark compiler does a variety of optimizations before generating the intermediate-level imperative code which is eventually compiled to C. The Futhark compiler does not currently implement any stencil-specific optimizations, which means that stencil computations implemented in Futhark are unlikely to perform optimally.

Several high-level domain-specific languages have been created[9, 2, 7, 1, 11, 4] to simplify the process of writing code – including stencil computations – that can be compiled to highly performant low-level code for various architectures. Many of these use different kinds of tiling[7, 11, 1, 4], vectorization[11, 4], and some use other loop transformations such as skewing[8, 4].

1.1 Contributions

In this thesis I investigate the performance increase of loop tiling and partitioning on simple stencil computations through prototyping in C and CUDA, with the goal of deciding how to best implement code generation for a stencil construct in the Futhark compiler. For tiling, I also investigate how tile shape impacts performance.

I then implement three different versions of code generations for stencils into the Futhark compiler, based on these prototypes, and measure the performance of the generated code.

1.2 Overview

In section 2, I briefly go over some notation and terminology that will be used throughout this report. In section 3, I give a summary of some common stencil optimizations, and in section 4 I explain the loop transformations I use to implement tiling and partitioning.

In section 5, I investigate the performance impact of tiling and partitioning on GPUs and multicore CPUs through prototyping in CUDA and C. I also take a look at how tile sizes impact performance.

In section 6 I present three different implementations of code generation for stencil computations in the Futhark compiler, all for the multicore CPU backend. In section 7, I measure the performance of these three implementations.

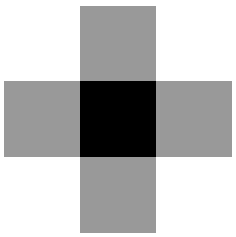


Figure 1: A 2D stencil shape. The black point is (0, 0), and the gray points are the stencil shape points `[(-1, 0), (0, -1), (1, 0), (0, 1)]`.

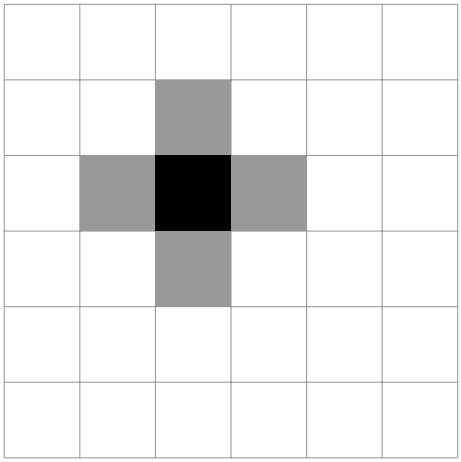


Figure 2: A 2D stencil neighborhood – here, the stencil shape has been applied to a specific point on a grid. Values at the gray points will be read, and the calculated value will be written to the location corresponding to the black point, but in the output array.

2 Notation and terminology

In this text, I'll refer to the set of *relative* indices used to compute an updated value as the *stencil shape*. An example of a 2-dimensional stencil shape could be the list `[(-1, 0), (0, -1), (1, 0), (0, 1)]`. This shape is shown in figure 1.

Given some specific index `(x, y)` (in the case of a 2-dimensional stencil), we can use the stencil shape to get the *neighborhood indices* of this point. For the example stencil shape above, the neighborhood indices would be the list computed from `[(x-1, y), (x, y-1), (x+1, y), (x, y+1)]`, as shown in figure 2. Given the neighborhood indices, we compute the actual *neighborhood* by extracting the corresponding values from the input array.

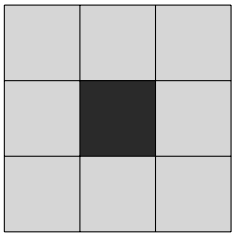
The updated value is computed by a pure function, the *stencil function*, which takes the neighborhood as input.

3 Related work

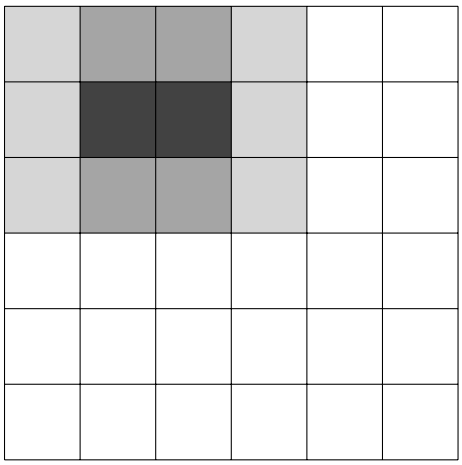
Stencil computations have been widely studied, and a lot of optimization strategies have been invented. The most common class of optimizations is those exploiting locality. Figure 3b shows how, for a 3×3 square stencil shape (see figure 3a), the neighborhoods of two adjacent points share six points.

On GPUs, we want to minimize the number of times we read the same point from global memory, so there we can imagine storing the values from this overlap area in shared memory so two threads – each one updating one of the two adjacent points – can both access these values quickly. We'll want to store a whole tile of input data, all the data needed for a thread block, in shared memory. There will still be some overlap at the boundaries of the tiles, where the values will be read by two or more thread blocks, but in total, there will be far fewer global memory reads. Strategies that try to tile the input data to avoid unnecessary reads from global memory have been extensively explored, as has the idea of *time tiling*, where we calculate multiple iterations of a stencil in one go, at the cost of increasing the overlap area between tiles[1, 2, 4].

On a CPU, we also use tiling[11, 1, 8], for similar reasons – here, the goal is to take advantage of values automatically being cached. Looking again at figure 3b, calculating the first of these two values will result in the overlap area being cached, and so calculating the second value will be faster. In fact, more values than just the highlighted overlap area will be cached – how many depend on the cache line size in question. However, if the array is large, and we simply continue to update values horizontally,



(a) A 3×3 neighborhood shape.



(b) When updating two adjacent points (black), there is some overlap in the areas we read (black and dark gray area).

Figure 3: Overlap between neighborhoods of two adjacent points, for a 3×3 stencil.

we will eventually eject from the cache those values that were cached at the beginning, even though most of them will be used again when the next row is updated. Updating a smaller rectangular area – a tile – before moving on to the next area, can lead to better use of the cache, and so to better performance.

Somewhat analogously to time tiling on GPUs, there is also the idea of time skewing (a combination of loop skewing and blocking)[8] stencil loops on CPUs to allow a value calculated in iteration t to be cached and reused in iteration $t + 1$. Time-skewed loops can then be parallelized with different strategies such as pipeline and wavefront parallelization[1, 8, 10]. With pipeline parallelization, explicit synchronization is used to ensure that all required loop tiles have been evaluated. With wavefront parallelization, tiles are grouped such that tiles in "group 0" depend on nothing, and tiles in higher-numbered groups only depend on values calculated by tiles in lower-numbered groups. Then, each group can be scheduled collectively. Pipeline parallelization guarantees some data reuse across time iterations, while wavefront parallelization does not, but this comes at the cost of having to spawn threads explicitly, which means each tile is bound to a specific thread.

Loop tiling will be explored thoroughly in this text. Time tiling and time skewing is only applicable to iterative stencil computation, and in this thesis, I focus on optimizing a single sweep of a stencil computation, so no work is done involving these optimizations. The main optimization is tiling, and the parallelization strategy is straight-forward.

It's worth noting that if we execute an untiled stencil loop on a multicore architecture, simply making each iteration into a task and executing them in parallel, we are likely to get more cache misses than we would if we executed it sequentially. This makes tiling even more important in threaded programs than it is in sequential programs, where some cached data will still be reused.

3.1 Handling boundary regions

When updating values near the edges of the array, we're faced with a problem: Sometimes, some of the neighborhood indices will be out of bounds. In some stencils, this is handled by substituting zeroes for the non-existent neighborhood values, and in others, it's handled by duplicating the nearest existing value. Yet other stencils "wrap around", using values from the other end of the array.

Lift[9] is a high-level, data-parallel language, meant to be used as an intermediate language for compilers. When implementing stencils in Lift[2], Gorlatch and Duback chose the approach of padding

```
for (int i = 0; i < N; i++) {
  // body
}
```

(a) A for-loop before strip-mining

```
for (int ii = 0; ii < N; ii += T) {
  for (int i = ii; i < min(ii+T, N); i++) {
    // body
  }
}
```

(b) Applying strip-mining with stride `T`

Figure 4: Strip-mining a loop

the input array, adding a *pad* primitive to the language. The user passes a function to *pad*, which is used to determine what values to pad the array with. This strategy is very flexible, but it isn't clear what sort of code would be generated to do this padding of the array.

Meanwhile, Holewinski, Pouchet and Sadayappan[4] propose a nameless DSL involving *subgrids* to which entirely different functions can be applied – thus, one could make the boundary areas into subgrids that use a slightly different stencil function than the central area of the input array, but the strategy is more flexible than that. The grid can be partitioned in whatever ways the user desires, and completely different stencil functions can be applied to different subgrids.

A more straightforward approach is to write code that detects whether the index needed is out of bounds. This will typically involve an if-statement, which can be quite the performance hit, so we'd like to avoid doing this check for the points that are safely in the center area of the array. That is, we'd like to calculate the smallest bounding box of the stencil shape, and for any point where this bounding box doesn't include any out-of-bound points, we'll want to proceed without checking for out-of-bound reads. This is similar to the subgrid approach[4], but without exposing the mechanism to the user, and is the approach is used by the REPA library for Haskell[6].

4 Code transformations

As mentioned in previous sections, the main optimization that I want to apply to my code is tiling. Tiling is a combination of two well-known transformations, loop *strip-mining* (also sometimes called *sectioning*), and loop *interchange*. Additionally, I want to process part of the array – the boundary areas – separately. For this, we use a technique usually associated with pipelining or unrolling, where we extract some iterations of the loop into a prologue or epilogue. I'll briefly go over these three transformations here, looking at some pseudocode examples, and then in later subsections we'll see how they're applied to stencil code.

4.1 Loop strip-mining

Strip-mining a loop refers to taking a loop and turning it into two (nested) loops, as seen in figure 4. The inner loop keeps its stride of 1, but outer loop uses a larger stride, so that each iteration of the outer loop contains several iterations of the inner loop. The loop body remains unchanged.

This transformation is always safe to apply, as it doesn't actually change the order of execution of any statements within the loop body, either within an iteration or across iterations.[5]

4.2 Loop interchange

Loop interchange is a transformation in which we take a perfect loop nest, and we move one of the outer loops one step inwards (see figure 5). This transformation is not always safe to apply, since it changes the order of iterations. We can use dependency analysis to figure out whether it's safe to interchange two specific loops: If we construct a direction matrix of the loop nest, and if exchanging

column i and j of the matrix does *not* result in a $>$ direction as the left-most non-$=$ direction of any row, then it is safe to exchange the loops at depth i and j[5].

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
    // body
  }
}
```

(a) A loop nest before loop interchange

```
for (int j = 0; j < M; j++) {
  for (int i = 0; i < N; i++) {
    // body
  }
}
```

(b) We interchange the two loops, leaving the body unchanged.

Figure 5: Interchanging two loops

For stencil computations this is very straightforward – there are no dependencies between loop iterations at all, so the direction matrix will contain only $=$ directions, and so we can safely interchange our loops.

4.3 Extracting prologues and epilogues

Figure 6 shows how we can separate out the first and last iterations of a loop into a prologue and epilogue. This transformation does not change the order of iterations of a loop, and so is always safe to perform.

```
for (int i = 0; i < N; i++) {
  // body
}
```

(a) Original loop

```
for (int i = 0; i < M; i++) {
  // body
}

for (int i = M; i < N-K; i++) {
  // body
}

for(int i = N-K; i < N; i++) {
  // body
}
```

(b) Prologue and epilogue has been extracted

Figure 6: Extracting a prologue and epilogue from a loop

4.4 Tiling: Combining loop interchange and strip-mining

If we have a perfect loop nest with no dependencies between iterations, then we can first apply strip-mining to all loops in the loop nest (doubling the number of loops), and then apply loop interchange repeatedly to move each of the loops with stride 1 inwards. The result is a loop nest where there is a set of outer loops that iterate over tiles, and a set of inner loops that iterate over elements of each tile. Figure 7 shows an example of this.

5 Prototyping

Before doing anything to the Futhark compiler, I spent some time writing prototypes for the two backends I was interested in. The goal was to test how much speedup I could get by applying various

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
    // body
  }
}
```

(a) Loop nest before tiling

```
for (int ii = 0; ii < N; ii += T) {
  for (int jj = 0; jj < M; jj += S) {
    for (int i = ii;
             i < min(ii+T, N); i++) {
      for (int j = jj;
               j < min(jj+S, M); j++) {
        // body
      }
    }
  }
}
```

(b) Loop nest after tiling – each loop has been strip-mined, and then the `i`-loop has been exchanged inwards once.

Figure 7: Tiling a 2-level loop nest with tile size $S \times T$.

optimizations, compared to a naive implementation for the respective architectures.

In both cases, I used a simple smoothing stencil to benchmark – sum the values in the region surrounding each point, and divide by the size of the region. An example of a 3×3 stencil of this shape can be seen in figure 3a. We say that the stencil shown in figure 3a has a radius of 1, while a 5×5 stencil of the same shape would have a radius of 2.

Boundaries are handled by repeating the nearest existing value. For example, if we need to read the value at location (-1, 0), we'll instead use the value at location (0, 0).

5.1 GPU prototyping in CUDA

The optimization I wanted to test out for GPUs was to load part of the input array from global memory into shared memory, and then have the block work on that, so that most values would only need to be read from global memory once.

Appendix A shows a straight-forward, naive implementation of a stencil computation in CUDA – all reads and writes are performed directly on global memory. A tiled version can be seen in appendix B – here I've modified the kernel so that each thread only reads from global memory once, storing the read value in shared memory, after which all reads are from shared memory.

The results from the GPU prototyping were disappointing – the only way to get speedup was by making the stencil shape very large. For smaller stencils, such as the 3×3 one used in the two given examples, the tiled version either performed the same as the naive version, or slightly worse.

5.2 Multicore prototyping in C with OpenMP

Parallelizing a stencil computation for a multicore architecture isn't particularly difficult – within each stencil convolution, there are no data dependencies, so all we have to do is share out the loop iterations among the available cores.

However, on large arrays, there is some performance to be gained by optimizing for the various available caches. This is technically not unique to multicore architectures – the same benefits theoretically apply to single-threaded applications – but with a single thread, there is already decent cache use by default, since we're updating points in order. The moment we start scheduling iterations in an arbitrary order, we lose this benefit, and tiling is how we get it back and even improve it.

The idea is, essentially, that we tile the loops to take advantage of the locality inherent to stencil computations. Take for example a 2-dimensional stencil. When we compute the first point, a rectangular area of data will be read. A block of some size (often 64 bytes) per row will be cached in the

L1 cache. As we continue computing points in a horizontal direction, the elements we read from our array will already be cached – until we reach the end of the first set of blocks. At that point, we read a whole fresh set of blocks into our cache. As we keep going, we'll eventually fill up the cache, and the first set of blocks will be ejected, even though most of those values are going to be read again once we compute the second row.

If we tile the loop such that it computes rectangular sections of points, then we can reuse the already cached blocks more times. This reduces the number of times that we need to read from the high-latency main memory, at the cost of having deeper loop nests.

Most modern CPUs come with three levels of cache, and there can be benefits to doing up to three levels of tiling, optimizing for each of those caches. In this report, I only work with a single level of tiling. My aim is the L1 cache, which is smallest and fastest, but as the following sections will show, I get good results with tile sizes that are too large to fit in the L1 cache, indicating that I'm probably getting some speedup out of the L2 and L3 cache as well. This matches the observations of Rahman et al.[8], who found in their experiments that tuning for the L3 cache yielded more consistent impact on performance than tuning for the L2 and L1 caches.

See appendix C for a link to the full prototype implementation.

5.2.1 Naive implementation

The naive implementation of the 2-dimensional smoothing stencil for multicore, seen in figure 8, simply uses two nested loops, which are automatically parallelized using OpenMP. Note that `RADX` and `RADY` are constants – if those two values are not known at compile time, then none of the performance increases described later in this section apply.

```
1  const int l = (2 * RADX + 1) * (2 * RADY + 1);
2
3  #pragma omp parallel for collapse(2)
4  for(int i = 0; i < H; i++) {
5      for(int j = 0; j < W; j++) {
6          float acc = 0;
7          for(int i1 = 0; i1 < (2 * RADY + 1); i1++) {
8              for(int j1 = 0; j1 < (2 * RADX + 1); j1++) {
9                  acc += xs_in[MIN(MAX(i + i1 - RADY, 0), H-1) * W +
10                             MIN(MAX(j + j1 - RADX, 0), W-1)];
11             }
12         }
13         xs_out[i * W + j] = acc / l;
14     }
15 }
```

Figure 8: Untiled 2D stencil computation in C

Here, `W` and `H` are the width and height of the input array, and the smoothing stencil is of size `(2 * RADX + 1) * (2 * RADY + 1)`.

5.2.2 Partitioning

In figure 8, there are calls to the macros `MIN` and `MAX`, each of which expand to a conditional operator (`?:`), in the innermost loop. As discussed in section 3.1, this is not necessary for *most* iterations, only for the ones where the neighborhood extends beyond the boundaries of the input array. Branching is generally bad for performance, so we would like to only do so when necessary.

One solution – the one I will explore here – is to partition the input array into four boundary regions (in the case of a 2D stencil) and one inner region, and then do separate loops for each region.

Figure 9 shows what this looks like for the untiled version of the code, albeit with three of the four boundary-region loops omitted, as they are nearly identical to the first one and take up a lot of space. Note how the last loop nest, which calculates updates for the inner region, no longer needs to use `MIN` and `MAX` to clamp the array indices. The exact same loops can be used for the tiled version of the code.

```
const int l = (2 * RADX + 1) * (2 * RADY + 1);

// Top edge
#pragma omp parallel for collapse(2)
for(int i = 0; i < RADY; i++) {
    for(int j = 0; j < W; j++) {
        float acc = 0;
        for(int i1 = 0; i1 < (2 * RADY + 1); i1++) {
            for(int j1 = 0; j1 < (2 * RADX + 1); j1++) {
                acc += xs_in[MIN(MAX(i + i1 - RADY, 0), H-1) * W
                             + MIN(MAX(j + j1 - RADX, 0), W-1)];
            }
        }
        xs_out[i * W + j] = acc / l;
    }
}

// ... similar loops for bottom, right, and left edge ...

#pragma omp parallel for collapse(2)
for(int i = RADY; i < H-RADY; i++) {
    for(int j = RADX; j < W-RADX; j++) {
        float acc = 0;
        for(int i1 = 0; i1 < (2 * RADY + 1); i1++) {
            for(int j1 = 0; j1 < (2 * RADX + 1); j1++) {
                acc += xs_in[(i + i1 - RADY) * W + (j + j1 - RADX)];
            }
        }
        xs_out[i * W + j] = acc / l;
    }
}
```

Figure 9: Untiled, partitioned 2D stencil computation in C. Three of the four boundary-area loops have been cut for brevity.

5.2.3 Tiled implementation

To create a tiled version with tile size `TW * TH`, we give the two outer loops a stride of `TH` and `TW` respectively, and then add inner loops for computing the points within a tile. This corresponds to stripmining both of the outer loops, and then interchanging the outermost of the stride-1 loops inwards. We do this to the last loop from figure 9, and get the result seen in figure 10. The same transformation is applied to the un-partitioned code from figure 8 to get a version that is tiled but not partitioned.

5.2.4 Performance measurements

All measurements are made on a Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz. This CPU has 8 L1 data caches of 32KB each, each of which is shared between two cores, as well as larger L2 and L3

```
const int l = (2 * RADX + 1) * (2 * RADY + 1);

// ... boundary loops ...

int imax, jmax;

#pragma omp parallel for collapse(2)
for(int i = RADY; i < H-RADY; i += TH) {
    for(int j = RADX; j < W-RADX; j += TW) {
        imax = MIN(H-RADY, i+TH);
        jmax = MIN(W-RADX, j+TW);
        for(int ii = i; ii < imax; ii++) {
            for(int jj = j; jj < jmax; jj++) {
                float acc = 0;
                for(int i1 = 0; i1 < (2 * RADY + 1); i1++) {
                    for(int j1 = 0; j1 < (2 * RADX + 1); j1++) {
                        acc += xs_in[(ii + i1 - RADY) * W + (jj + j1 - RADX)];
                    }
                }
                xs_out[ii * W + jj] = acc / l;
            }
        }
    }
}
```

Figure 10: Tiled 2D stencil computation in C

caches. The cache line size is 64 bytes for all three cache levels.

For each measurement, I've run the code ten times. The input for the 2D stencils is a 10000×10000 array of randomly generated 32-bit floating point numbers.

I've compared four different versions of the code – with and without tiling, and with and without partitioning. For the tiled versions, I've compared a variety of tile sizes.

Figure 11 shows the performance of a 3×3 stencil with no parallelization. Here we see that tiling can actually make performance worse, but that partitioning makes it drastically better. This tells us that when it comes to Futhark's sequential backend, we should avoid implementing tiling, but we should definitely implement partitioning.

Figure 12 shows the performance of the same code as figure 11, but now using OpenMP to parallelize the execution. Here we see that tiling alone gives us some performance increase, but not very much, and that partitioning alone also gives us some performance increase, but not very much. Put together, however, these two optimizations yield a much larger performance increase.

When we increase the size of the stencil to, for example, 9×9, we get less benefit from tiling. Figure 13 shows this – here, tiling without partitioning does almost nothing for us (with some tile sizes it actually decreases performance), but partitioning *and* tiling is still better than only partitioning. This result – that tiling is less beneficial for larger stencil shapes – was surprising to me. I would have expected that a stencil that reads more points would benefit more from execution strategies that increase cache hits.

Upon investigating, using the `perf` tool, I found out that for the 9×9 stencil, the tiled version (both with and without partitioning) had fewer L1 cache misses than the untiled version, but *more* total cache misses. Figure 14 shows output from the `perf` tool without tiling, and figure 15 shows what happens when we tile with the tile shape that performed best in the earlier benchmark (figure 13). Partitioning was used in both figure 14 and 15, but the result without partitioning is similar.

Finally, we also compare accesses per second across different stencil shapes, choosing a good but

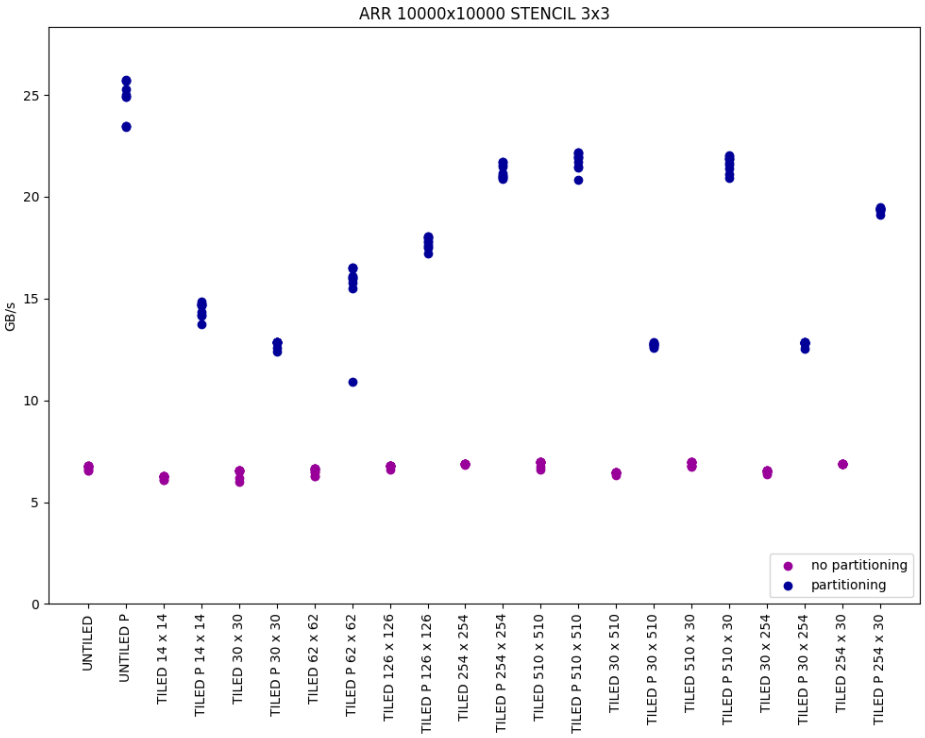


Figure 11: Performance of untiled and tiled 3x3 stencils without parallelization, with and without partitioning.

not fine-tuned tile shape for each (for the 2D stencils, I've gone with shapes where a tile of 512×30 points will be read, and for the 3D stencil, I ran benchmarks within a smaller range and picked the best). The result can be seen in figure 16, and shows, oddly enough, that the 3×3 stencil is something of an outlier.

5.3 Choosing the tile size

Tiling the loops and parallelizing the execution of the tiles yields speedup, but some tile shapes and sizes are better than others. How can we choose a tile size that gives us as much speedup as possible? The search space is quite enormous – testing every possible tile size, even for just a single stencil shape, would take far too much time, so we need some way to narrow it down.

The search for an optimal tile size was further hampered by only having access to one specific CPU for benchmarking – I cannot say for certain that any strategy that yields good results on the Xeon will also work well on some different CPU.

The approach used by Patus[1] is to do automatic tuning within a pre-determined range for each dimension, but we're still left with the question of how to choose that range. In this section, I explore in more detail how different tile shapes affect the performance of a 3×3 stencil, and whether there are any useful patterns to be found.

The ideas I had going into this were that it would probably be better for performance if the values read by a tile all fit in the cache, and that the X-dimension of the tile should probably align with the cache line size. I've done benchmarks that test both of these assumptions, as well as some more general searches for areas worth exploring.

Figure 17a shows what happens when the tiles are very small – we try tiles with each dimension ranging from size 1 to 14. Figure 17b shows a similar measurement, but with larger tile sizes.

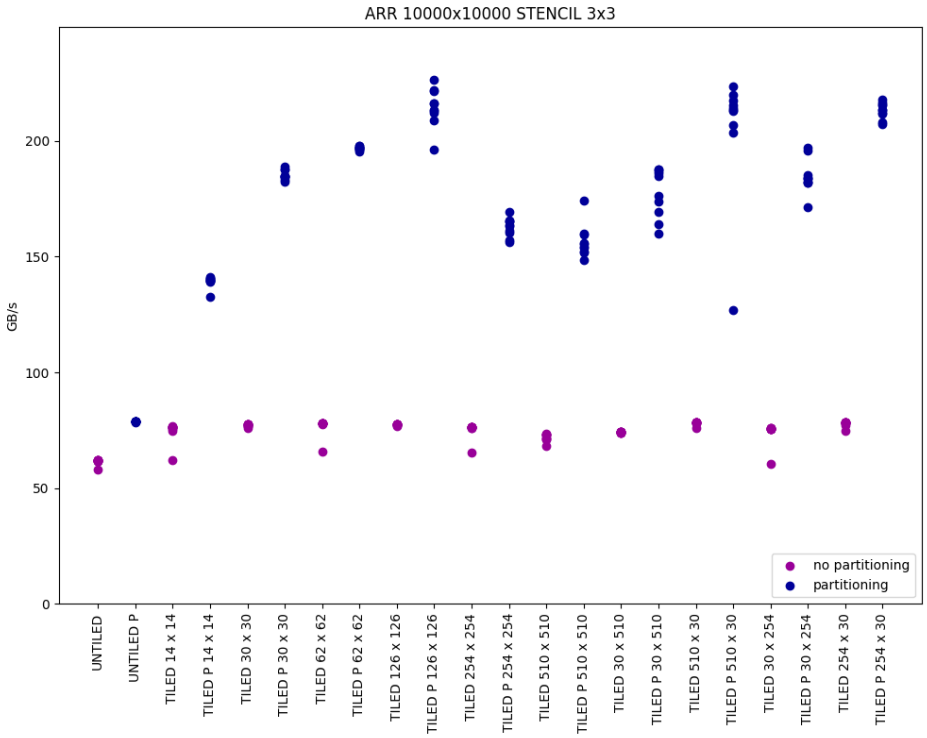


Figure 12: Performance of untiled and tiled 3x3 stencils, with and without partitioning, using OpenMP to parallelize.

Intuitively, very small tiles – in the context of a multithreaded program – are bad because they cannot take advantage of caching – one tile will be processed, resulting in some values being cached, but we don't know when the neighboring tiles will be processed, so we don't know if those values will still be in the cache by then. Interestingly, though, figure 17a shows that small tiles that are wider than they are tall actually lead to fairly good performance. The best results from figure 17a have an average of just over 200GB/s worth of memory accesses, which is not much worse than the best results I've gotten with larger tile sizes.

At larger tile sizes, we have figure 17b, which shows a performance jump when the tile width goes from 20 to 30. With a tile width of 140 and a tile height of 100, we get better average performance than we did in any of the runs from figure 17a.

Our benchmark uses floats, each of which takes up 4 bytes. With a cache size of 32K, we can fit, for example, 64×128 or 128×64 floats in the cache. Figure 19a shows what happens when we try tiles with dimensions close to 128×64. Specifically, I have tested tile shapes that are a bit larger along each dimension, and surprisingly, it doesn't seem that going above this size hurts performance. The best-performing tile size from this set was 124×69, which – because of the stencil shape – reads a tile of size 126×71, or 35784 bytes.

In figure 17b, the most promising tile size was 140×100. In figure 19b, we zoom in on this area and try similar tile sizes. This gives us the best result so far, with an average of 221.48 GB/s processed, at tile size 148×99.

In figure 12, we saw benchmarks for a few tile sizes – mostly squares, with dimensions ranging from 14 to 510. These numbers are chosen such that the dimensions of the data that gets *read* by the tile are powers of two. For example, to update a 14×14 square, we need to read values from a 16×16 square. Why powers of two? Mostly because I had to chose a starting point, in this huge search space, and space in the context of computing tends to be measured in powers of two. Setting that aside for

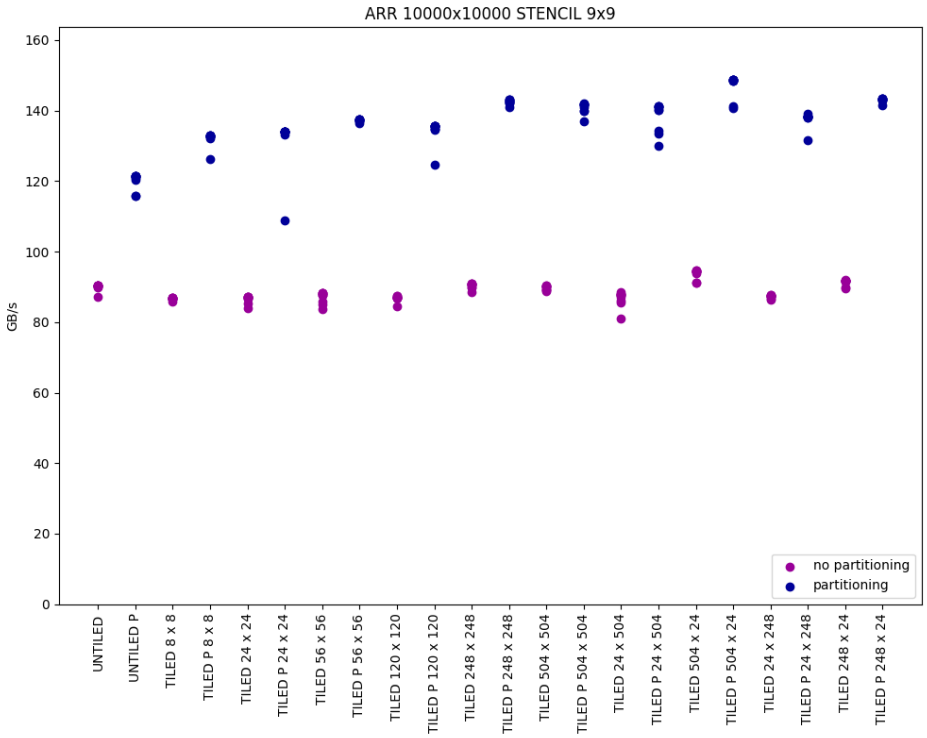


Figure 13: Performance of untiled and tiled 9×9 stencils, with and without partitioning, using OpenMP to parallelize.

```
Performance counter stats for './main_2d_untiled_par':

     18,505.55 msec task-clock:u              #   11.564 CPUs utilized
54,391,139,141      cycles:u                  #    2.939 GHz
37,980,591,487      instructions:u            #    0.70  insn per cycle
     6,579,997      cache-references:u        #    0.356 M/sec
     1,276,608      cache-misses:u            #   19.401 % of all cache refs
   131,791,847      L1-dcache-load-misses:u

   1.600227040 seconds time elapsed

  18.227138000 seconds user
   0.281986000 seconds sys
```

Figure 14: Output from `perf` when running a 9×9 stencil with partitioning but no tiling.

now, these benchmarks showed good results for larger tile sizes, especially 126×126 and 510×30. In figure 18 I've tested tile sizes close to these two sizes. Especially figure 18b shows good results, with all benchmarks processing at least 200GB/s on average over 10 runs. The best result is 220.37 GB/s at a tile size of 501×22, which is on par with the best result from figure 19b.

What can we say about patterns from all of this? There are definitely areas of the search space that are better than others, but when we zoom in on a small area, there isn't a particularly clear pattern. We can look at which of these tile sizes line up neatly with the cache line size along the X dimension,

```
Performance counter stats for './main_2d_tiled_par 504 24':

        15,334.86 msec task-clock:u              #   10.237 CPUs utilized
   44,762,963,760      cycles:u                  #    2.919 GHz
   31,749,751,819      instructions:u            #    0.71  insn per cycle
        5,612,797      cache-references:u        #    0.366 M/sec
        1,574,614      cache-misses:u            #   28.054 % of all cache refs
      129,268,532      L1-dcache-load-misses:u

      1.497939954 seconds time elapsed

     14.960280000 seconds user
      0.378097000 seconds sys
```

Figure 15: Output from `perf` when running a 9×9 stencil with partitioning and tiling, tile size 504×24.

and see that this particular property doesn't seem to have a measurable impact. We can look at tile sizes that exceed the size of the L1 cache compared to those that don't, and again this does not seem to be a deciding factor in how well the benchmark performs.

Doing a search with a larger stride, and then zooming in on a smaller area based on that, seems to be a decent strategy for finding a good tile size. This suggests that autotuning could be used to determine tile sizes.

6 Implementation in the Futhark compiler

In this project, I've implemented code generation for stencil computations for the Futhark compiler's multicore backend, which is implemented in Haskell. I decided against working on the GPU backend, as my prototyping in CUDA did not yield particularly interesting results. I've implemented three different versions of stencil code generation for the multicore backend: An unoptimized version, a tiled version, and a version that is both tiled and partitioned. Tile sizes are currently hardcoded, as the multicore backend does not have a concept of tunable size parameters, but this is something that would be fairly simple to add at a later date.

Links to the full implementations can be found in appendix C.

6.1 Source language

In the source language, a stencil computation is represented by a call to one of three functions: `stencil_1d`, `stencil_2d`, or `stencil_3d`. Figure 20 shows an example of a call to `stencil_2d`. Each of the stencil-functions has four parameters: The neighborhood shape, given as a list of tuples, the stencil function, an invariant array, and the input array. In figure 20, the neighborhood shape is `[(-1, -1), (1, 1)]`, the function `f` ignores the invariant element and adds the two elements of its neighborhood together, the invariant array just contains empty tuples, and finally the input array `xs` is provided by the caller.

I'll use the stencil computation from figure 20 as a running example in the rest of this section.

6.2 Internal representation

In order to do stencil-specific optimizations, the abstract syntax tree used by the compiler needs to have a node type for stencil computations. In fact, since the compiler uses different representations at different phases of compilation, two different constructs were required. The actual data types were

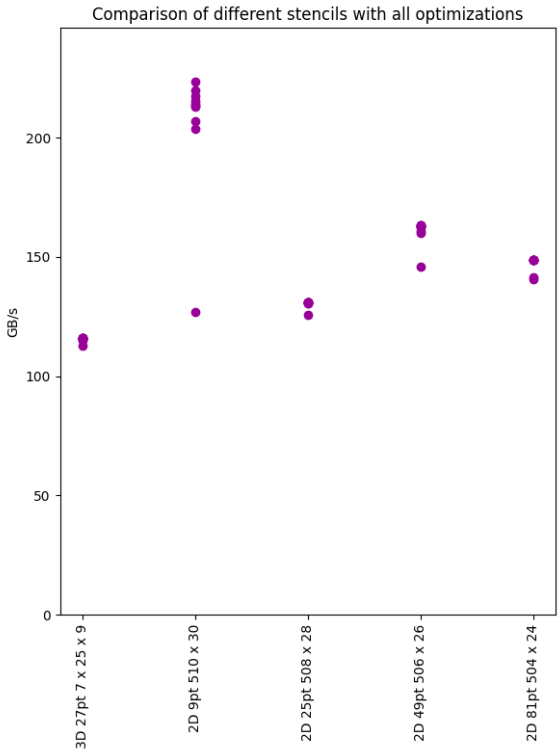


Figure 16: Comparison of several different stencils, including a 3D 27-point stencil.

designed and implemented by my supervisors rather than myself, but I'll give an overview of how they work here for context.

During compilation, an *internalize* pass is run (See `src/Futhark/Internalize.hs`), which turns a call to one of the three stencil functions into a `SOAC`, using the `Stencil` constructor seen in figure 21.

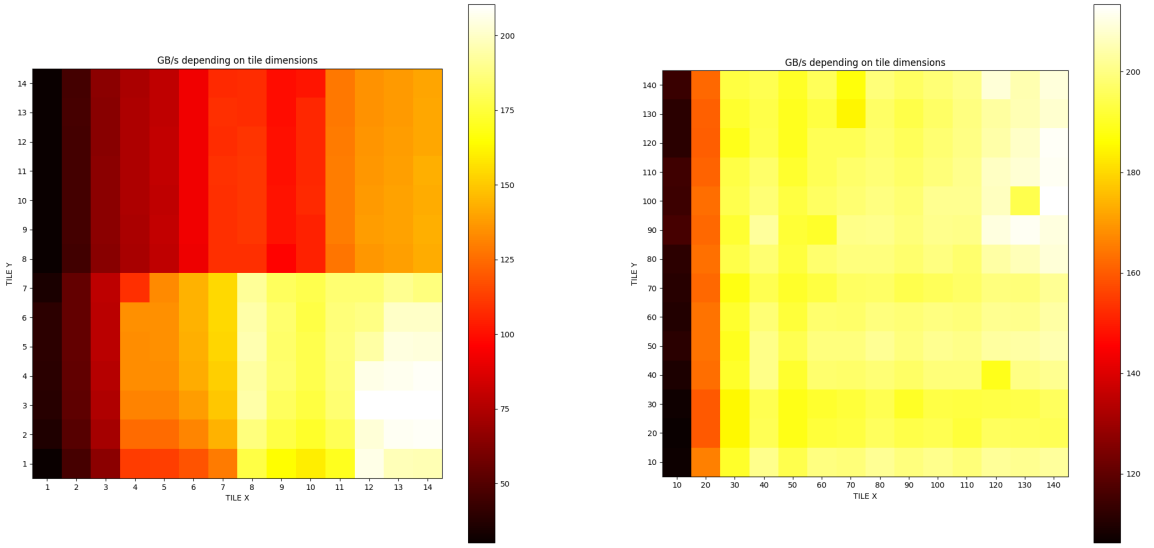
Here, `StencilIndexes` is a type containing the relative indices of the neighborhood, either as static values or as dynamic expressions. We distinguish between these two cases because it is hard to do any optimizations when we do not know the bounding box of the neighborhood at compile time. My focus has been exclusively on the cases where we know the exact shape of the stencil at compile time.

Figure 22 shows what happens to our example stencil computation from figure 20 when it gets internalized. The only surprising thing here is that the neighborhood indices are dynamic – the internalizer always does that, and a later *simplify*-pass takes care of turning them into static indices if possible (as it is in our example). Aside from that, all expressions have been assigned names, and those names (`"is"`, `"cs"`, etc.) are used rather than the actual expressions.

Later on, the compiler runs a pass that turns the SOAC representation into something more suitable for the specific backend – in the case of the multicore backend, this pass is called ExtractMulticore, and the code can be found in `src/Futhark/Pass/ExtractMulticore.hs`. This pass turns a `SOAC`-value built with the `Stencil`-constructor from figure 21 into a `StencilOp`, as seen in figure 23. Currently, this transformation only works for stencils with static indices.

This object is generally combined with some other data (an iteration space, some type information), and is what we use for the actual code generation. Together, the `StencilOp` and associated data form a `SegStencil`, which is one of several types of `SegOp` (others include `SegMap`, `SegScan`, and more).

The `stencilIndexes` are arranged such that each inner list represents one dimension. Transpose the list of lists, and we get the actual points. The `stencilArrays` are the input arrays (usually a single array, but arrays of tuples are represented as multiple arrays). Each `VName` is the name of an input array. The `stencilOp` is a lambda function which takes a parameter for each point in the neighborhood.



(a) Tile dimensions ranging from 1 through 14.

(b) Tile dimensions ranging from 10 through 140, with an increment of 10.

Figure 17: Memory accesses (GB/s) depending on tile dimensions. We measure average memory accesses per second of a 9-point 2D stencil over ten runs, varying the tile size along each dimension.

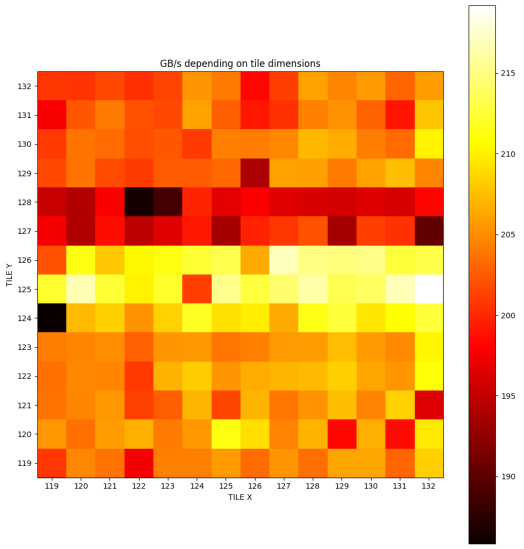
Figure 24 shows the example stencil from figure 20 in this representation. Note that the constant array has completely disappeared – the ExtractMulticore pass generates a separate set of statements for loading values from the constant array.

6.3 Code generation without tiling

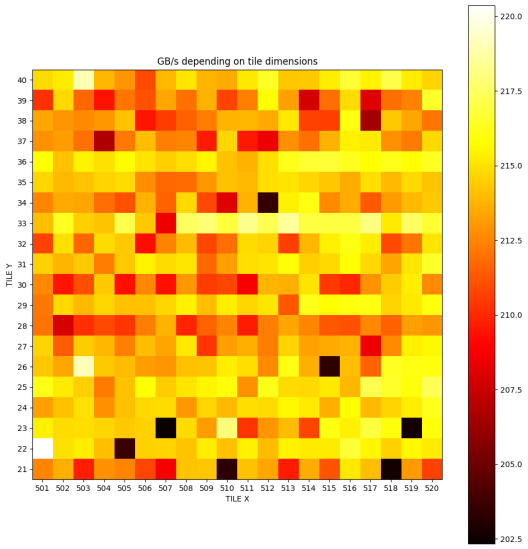
In this section I will show some excerpts from the code generation part of the compiler. Along with each excerpt, I will show an example of the kind of C-code that might be generated, using the running example from figure 20. The code generation presented in this section does no tiling or partitioning, and is fairly straightforward, but it introduces some concepts that help illustrate some of the challenges I faced when I did implement those optimizations.

The compiler code for untiled stencil loops is fairly straightforward. The input array is represented as a flat array regardless of the dimensionality of the stencil, and we have the actual dimensions separately. The actual parallelization is handled by Futhark's existing scheduler, so our only concern is generating the code for the loop body. We do this with a Haskell function called `compileSegStencil`, the head of which is shown in figure 25. It has four pieces of input – a pattern telling us where the result from the lambda function will be stored, a `SegSpace` which is essentially the dimensions we're iterating over, a `StencilOp` as seen in figure 23, and finally a kernel body, which is the code that takes care of reading elements from the constant array. The output is a value of type `MulticoreGen Imp.Code` – `MulticoreGen` is a monad for code generation, and the Futhark compiler already contains a lot of combinators and helper functions for working in this monad. `Imp.Code` is the type of ASTs for the low-level imperative intermediate language that this part of the compiler generates.

Two of the basic combinators for the `MulticoreGen`-monad are `collect` and `emit`. `emit` takes a little piece of AST and 'emits' it, that is, makes it part of the generated code. `collect`, meanwhile, takes a `MulticoreGen ()` value and gathers all the `emit`ted code, returning a `MulticoreGen Imp.Code` value. So when, in figure 25, we start the body of `compileSegStencil` with a call to `collect`, this means that inside the do-block, any call to `emit` will result in the given code being part of the return value. Many of the combinators used will also emit code.

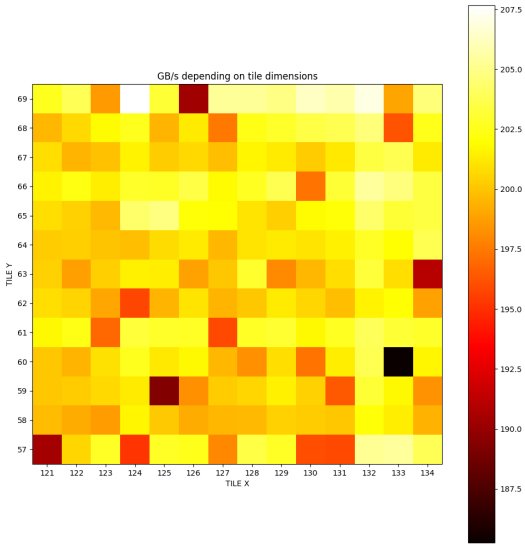


(a) Tile dimensions ranging from 119 through 132.

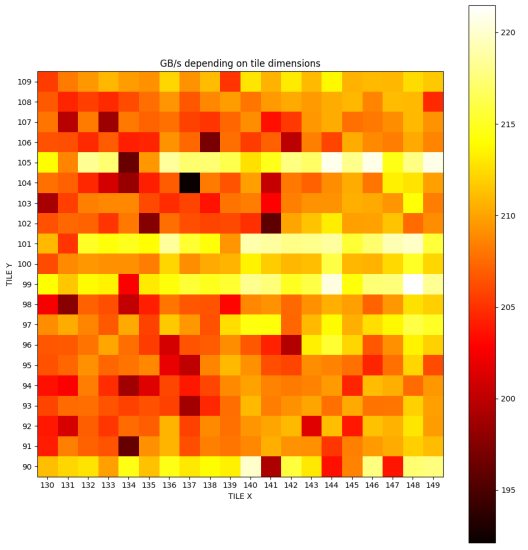


(b) Tile dimensions ranging from 501 through 520 along the X dimension, and from 21 through 40 along the Y dimension.

Figure 18: Memory accesses (GB/s) depending on tile dimensions. We measure average number of memory accesses per second of a 9-point 2D stencil over ten runs, varying the tile size along each dimension.



(a) Tile dimensions ranging from 121 through 134 along the X dimension, and from 57 through 69 along the Y dimension.



(b) Tile dimensions ranging from 130 through 149 along the X dimension, and from 90 through 109 along the Y dimension.

Figure 19: Memory accesses (GB/s) depending on tile dimensions. We measure average number of memory accesses per second of a 9-point 2D stencil over ten runs, varying the tile size along each dimension.

```
entry main [n][m] (arr : [n][m]i32)  =
  let f _ nbh = nbh[0] + nbh[1]
  in  stencil_2d [(-1, -1), (1, 1)] f (map (map (const ())) xs) xs
```

Figure 20: Example of a 2D stencil computation in the Futhark source language

```
| Stencil
  -- Dimensions of input arrays in each dimension; length of list
  -- is dimensionality of stencil ('r').
  [SubExp]
  -- Size of stencil neighbourhood ('p').
  SubExp
  -- Neighbourhood indexes.  Conceptually a 'p'-size array of
  -- 'r'-tuples, but we encode it as 'r' arrs each of size [p]i64.
  StencilIndexes
  -- ...invariant... -> [p]a -> ... -> [p]a -> b.
  (Lambda lore)
  [([Int], VName)] -- List of (invariant,arr)
  [VName] -- [...ds...]a, input to stencil.
```

Figure 21: General stencil construct, part of the `SOAC` datatype.

```
Stencil
  [Var (VName (Name "n") 4027), Var (VName (Name "m") 4028)]
  (Constant (IntValue (Int64Value 2)))
  (StencilDynamic [VName (Name "is") 4143, VName (Name "is") 4144])
  (Lambda ...)
  [([],VName (Name "cs") 1064)]
  [VName (Name "as") 1065]
```

Figure 22: The example stencil from figure 20 after internalization. Representation of the actual stencil function elided for brevity.

```
data StencilOp lore = StencilOp
  { -- | Encodes @[r][p]i64@ array.
    stencilIndexes :: [[Integer]],
    stencilArrays :: [VName],
    stencilOp :: Lambda lore
  }
  deriving (Eq, Ord, Show)
```

Figure 23: Stencil construct for code generation

```
StencilOp {
    stencilIndexes = [[-1,1],[-1,1]],
    stencilArrays = [VName (Name "arr") 4390],
    stencilOp = Lambda ...
}
```

Figure 24: The example stencil from figure 20 after the ExtractMulticore pass. Representation of the actual stencil function elided for brevity.

We declare a variable to be used for the flat loop index, and then calculate the unflattened indices based on this and the array dimensions. We do the unflattening using existing functions `dPrimV_`, which declares a variable and assigns it a value, and `unflattenIndex`, which does what the name implies. Figure 26 shows the declarations of the various index variables, both in the compiler code and the generated code. Note that, in figure 26b on line 5-10, two variables are declared – these represent the indices of the point we're going to update, and there are two of them because the stencil used in this example is 2-dimensional.

All code from figures 27b, 28b, and 29b is inside the loop started in figure 26b.

A stencil can, in addition to the input array, also use one or more constant (or invariant) arrays – these are arrays that remain the same from one iteration to the next. The kernel body, `kbody`, contains the code for loading these values from the constant arrays, and `kbres` is the list of variable names assigned to those values after the kernel body. Figure 27a shows how we compile the kernel body, declare variables for the parameters to the lambda that will be used for constant values, and set those variables to the correct values using the brilliantly named `copyDWIMFix` function ("DWIM" is short for "Do What I Mean"). Figure 27b shows the very boring output when we run this on our example stencil – it doesn't use the constant array for anything, so a default value of `1` (or `true`) gets used (this is how Futhark's empty tuples are represented in C).

Similarly, we need to declare variables for the actual neighborhood and read the corresponding values from the input array. The code that does this is shown in figure 28. Here, `stencilIdxs` is a function that generates the neighborhood indices based on the current index and the neighborhood shape. The output shown in figure 28b includes a somewhat complex index expression (and there's a second one which I've left out as it's not much different) – this is what `stencilIdxs` generates.

Finally, having created variables for all the lambda parameters and assigned the correct values to all of these parameters, all that's left to do is compile the lambda body itself, and write its result to the correct position of the output array. This is done by the code shown in figure 29. Here, `space` determines where in the output array the result is written – this becomes important once we implement tiling.

6.4 Code generation with tiling

Adding tiling to the code generation involved some fairly major changes. The iteration space of the loop is now different – instead of having an iteration per array element, we now have an iteration per tile. This means that the unflattening operation seen in figure 26 no longer gives us the indices for the array element we're updating, but for the tile we're updating.

Instead of generating code handling the update of a single array element, we now need to generate code that uses a (nested) sequential loop to update an entire tile. The various indices (of the element to be updated as well as the neighborhood of that element) new need to be calculated based on the tile index *and* the index of the inner sequential loop.

Generating the inner sequential loop nest is done by an auxiliary function named `innerLoop`. It takes as input a list of tuples – one tuple for each dimension, with each tuple containing the tile size, tile index, and array size along that dimension. It builds up, in an accumulator parameter, a list of the inner loop index – at the end, it will have one loop index per dimension. The part of the function that generates the loop nest can be seen in figure 30. It does some calculation to find out how many iterations should be done along this dimension (if we are nearing the edge of the array, we should stop before falling over), and then sets up a for-loop, with a recursive call determining the body of that for-loop.

The recursive case of `innerLoop` generates the loops themselves, and the base case – when the list of tuples is empty – generates the body of the innermost loop. It is inside this body that we need to set up the parameters for the lambda function, and compile the body of the lambda function itself.

Figure 31 shows how the body of the inner loop is generated. The `stencilIdxs` function has been expanded to take a few more parameters, but otherwise does about what it did in the untiled version of the code. On line 7-8 we read values from the input array. After this, we need to compile the

```
compileSegStencil ::
  Pattern MCMem ->
  SegSpace ->
  StencilOp MCMem ->
  KernelBody MCMem ->
  MulticoreGen Imp.Code
compileSegStencil pat space sten kbody = collect $ do
  ...
```

Figure 25: Head of the function that generates code for the stencil loop body.

```
let (is, ns) = unzip
          $ unSegSpace space
    ns' = map toInt64Exp ns
iter <- dPrim "iter" $ IntType Int64
body <- collect $ do
  zipWithM_ dPrimV_ is $
    unflattenIndex ns' $ tvExp iter
  ...
```

(a) Compiler code

```
int64_t iter_4410 = start;

for (; iter_4410 < end;
       iter_4410++) {
  int64_t gtid_4399 =
    squot64(iter_4410, m_4389);
  int64_t gtid_4400 =
    iter_4410 - squot64(iter_4410,
                        m_4389)
                * m_4389;
  ...
}
```

(b) Generated code

Figure 26: Code generation for the first part of the loop body: getting the flat and unflattened indices.

```
compileStms mempty
    (kernelBodyStms kbody) $ do
  dLParams const_params
  forM_ (zip const_params kbres) $
    \(param, r) ->
      copyDWIMFix (paramName param)
                  [] r []
```

(a) Compiler code

```
bool x_4393 = 1;
```

(b) Generated code

Figure 27: Code generation for fetching constant values and initializing constant parameters

```
let idxs = stencilIdxs (tvExp iter)
                       (zip sten_idxs ns')
let idxs_with_vnames =
  concatMap
    (\vn -> map (vn,)
                (transpose idxs))
    arrs

dLParams sten_params
forM_ (zip sten_params
           idxs_with_vnames) $
  \(param, (arr, idxs')) ->
    copyDWIMFix (paramName param)
                [] (Var arr)
                idxs'
```

(a) Compiler code

```
int32_t x_elem_4401;
int32_t x_elem_4402;

x_elem_4401 =
  ((int32_t *) arr_mem_4404.mem)[
    smin64(n_4388 -
      (int64_t) 1,
      smax64((int64_t) 0,
        sdiv_safe64(iter_4410,
          m_4389) +
        (int64_t) -1)) *
    m_4389 +
    smin64(m_4389 -
      (int64_t) 1,
      smax64((int64_t) 0,
        sub64(iter_4410,
          mul64(sdiv_safe64(
              iter_4410,
              m_4389),
            m_4389)) +
    (int64_t) -1))];
x_elem_4402 = //similar expression
```

(b) Generated code

Figure 28: Code generation for declaring non-constant lambda parameters and assigning them the correct values from the input array.

lambda body, but here we run into a challenge: Normally, the location to which the result of a `SegOp` should be written is determined by a `SegSpace` given as input to the function that compiles the `SegOp`. However, this is a tiled loop, and the `SegSpace` we are given as input does not describe locations in the array – it describes tiles. So we need to create a new `SegSpace`, containing variable names that have been assigned the actual indices we want to write to. This is what the code on lines 12-16 in figure 31 do.

Lines 18-20 look a lot like the code from figure 29a, but use the newly constructed `spaceOut` instead of `space`. The result is that the lambda body is compiled and its result stored at the location described by `spaceOut`.

The compiler code in figure 31 doesn't actually generate that much output – it's essentially the same as what's shown in figure 28 and 29, but with slightly more complex index calculations, since we now have an additional set of indices from the sequential loop nest. For an example of such a calculation, see figure 32.

6.5 Partitioning

So far, we've only been worrying about what happens inside the body of our parallelized loop. When we want to add partitioning to the Futhark compiler, we also have to look at the loop itself, because we need to break it up into multiple loops. For an n-dimensional stencil, we'll end up with a total of $2n + 1$ loops – one for the central part of the array, and for each dimension, one for the low indices and one for the high indices.

The actual loop is generated by a function called `compileMCOp`, which can be found in `src/Futhark/CodeGen/ImpGen/Multicore.hs`. This function generates loops not just for stencils, but for other SOACs as well. It's where the iteration space for the loop is determined and the scheduling information is set up, and it then calls `compileSegOp`, which is just a wrapper that pattern matches on

```
1 compileStms mempty
2     (bodyStms $
3       lambdaBody sten_lam) $
4   zipWithM_
5       (compileThreadResult space)
6       (patternElements pat) $
7     map (Returns ResultMaySimplify) $
8       bodyResult $
9       lambdaBody sten_lam
```

(a) Compiler code

```
1 int32_t defunc_1_f_res_4397;
2 defunc_1_f_res_4397 =
3   add32(x_elem_4401, x_elem_4402);
4 ((int32_t *) mem_4407.mem)[
5   gtid_4399 * m_4389 +
6   gtid_4400] =
7     defunc_1_f_res_4397;
```

(b) Generated code

Figure 29: Code generation for the lambda body and writing the result to the output array.

the given `SegOp` and calls the relevant `compileSeg*`-function, such as `compileSegStencil`, to generate the loop body. The assumption made in this function is that one `SegOp` will correspond to one iteration space. There can be more than one parallelized loop, but of there is, then they loop over the same space.

This is of course not the case for our tiled and partitioned stencil computations – we have a number of loops that iterate over varying numbers of elements of the input array, and then we have one loop that instead iterates over tiles. So we need to add a case to `compileMCOp` which handles stencils separately from the other types of `SegOp`.

Figure 33 shows the type of `compileMCOp` as well as the pattern matching for the case where a stencil is getting compiled. The interesting part here `sten`, which is the actual `StencilOp`.

Before generating the loops, we need to figure out the iteration spaces and offsets. Figure 34 shows how the index space would be partitioned for a 2D stencil such as the example from figure 20, on a small grid. Now, parallel loops in Futhark get as input an iteration number, which is a flat index – if a loop is over a 5×5 index space, the loop indices range from 0 through 24. Knowing the index space, we can easily unflatten the index, as we've seen in previous parts of this section. When we're iterating over the middle area of an array, however, we also need to know the offsets for the loop – essentially, we need to know the unflattened index that iteration 0 should access. Calculating these offsets for the main loop is fairly trivial, but the boundary area loops get a little tricky.

Figure 35 shows how we calculate the offsets. First we find the bounding box of the stencil shape – this is just the smallest and largest relative index along each dimension. Those along with the shape of the input array let us easily find iteration space of the inner array, `nsInner`. The variable `lBounds` is used in several calculations, but it is also used as the offsets for the inner loop.

The next thing we do is calculate the iteration space and offsets for the loops that work on the low indices of each dimension – that is, the topmost and leftmost area of figure 34. Note that the leftmost blue area has offsets `(0, 0)`, while the topmost green area has offsets `(1, 0)` – the offset in the first dimension has been replaced by the corresponding value from `lBounds`. If we expand this to more than two dimensions, what we'll see is that each dimension replaces one more zero with the corresponding value from `lBounds`. For example, for a 3D stencil with shape `[(-1, -2, -1), (1, 1, 1)]`, we'd get offset `(0, 0, 0)`, `(1, 0, 0)`, and `(1, 2, 0)`. Intuitively, if we think of a 3-dimensional box, each loop shaves off a side, making the box a little smaller along one dimension each time. Line 8-9 of figure 35 shows how this is calculated in Haskell by using `tails` and `inits` from the standard library – this is a pattern we will see a couple of times.

The actual iteration spaces for the low-index loops is calculated on line 10-11 with a similar technique, but here the lists we combine are different. Looking again at figure 34, we see that the blue and green areas have sizes that, along each dimension, are always equal to values pulled from either `ns` (the array dimensions), `nsInner`, or `lBounds`. Imagine a 3D array with dimensions $X \times Y \times Z$, and let's say that `bounds` is $[(-x_l, x_u), (-y_l, y_u), (-z_l, z_u)]$. The dimensions of the first loop will be $x_l \times Y \times Z$, the second loop will have dimensions $(X - x_l - x_u) \times y_l \times Z$, and the final loop will have dimensions

```
innerLoop ((d, i, a):ds) ts = do
  iters <- newVName "seq_iters"
  let prev_iters = i * d
      rem_iters = a - prev_iters
  dPrimV_ iters $ sMin64 rem_iters d
  sFor "t" (toInt64Exp $ Var iters) $ \t ->
        innerLoop ds (t:ts)
```

(a) Compiler code

```
seq_iters_4411 = smin64(n_4388 - gtid_4399 * (int64_t) 256,
                        (int64_t) 256);
for (int64_t t_4412 = 0; t_4412 < seq_iters_4411; t_4412++) {
    int64_t seq_iters_4413 = smin64(m_4389 - gtid_4400 *
                                    (int64_t) 256, (int64_t) 256);

    for (int64_t t_4414 = 0; t_4414 < seq_iters_4413; t_4414++) {
        // ... body ...
    }
}
```

(b) Generated loops

Figure 30: Part 1 of the auxiliary function for generating a sequential loop nest over a tile.

$(X - x_l - x_u) \times (X - y_l - y_u) \times z_l$. That is, for the nth loop, the first n dimensions will be pulled from `nsInner`, then there will be a number pulled from `lBounds`, and the remaining dimensions will be taken from `ns`.

Lines 12-17 calculate the offsets and iteration space for the opposite boundaries, the "high-index" ones, in a very similar manner.

Once we've calculated all of these lists of indices and offsets, we can take the generalized `SegOp` scheduling code – that is, the original version of `compileMCOp` – and create an adapted version of it that schedules all of our many loops. A part of that code is shown in figure 36, and the rest of the function follows a very similar pattern. Note line 15-16 where we make a call to `compileStencilBoundaryLoop`, which is a function very similar to the old `compileSegStencil` from section 6.3. For the inner loop, we do the exact same thing, except instead of calling `compileStencilLoopBoundary`, we call `compileSegStencilInner`, which is essentially the code from section 6.4, but with an added parameter for offsets.

7 Benchmarks

I've compared all three versions of my implementation. Additionally, each benchmark has also been implemented with the use of Futhark's `tabulate_2d` or `tabulate_3d` primitives. The results can be seen in table 1.

The benchmarks used were implemented by my colleagues, Christian Charlie Virt and Jonathan Wraa-Hansen, who are doing work on stencil code generation for Futhark's GPU backend. The benchmarks can be found at `https://github.com/Quartzinin/futhark-stencil-benchmarks`, though it should be noted that I've increased the sizes of the input arrays to 10000×10000 for 2D arrays, and $512 \times 512 \times 512$ for 3D arrays.

All benchmarks have been run by using Futhark's `futhark-bench`-tool, which runs the code ten times and reports the average runtime in microseconds. For the tiled stencil implementations, I've used a tile size of 148×99 for the 2D benchmarks, and $64 \times 64 \times 16$ for the 3D benchmarks. This is a fairly arbitrary and not very well-tuned choice, but in the current implementation, there's not way to

```
innerLoop [] ts = do
  let ts' = reverse ts
      idxs = map (uncurry5 stencilIdxs) $
        zip5 is sten_idxs dimsI64 ts' tileDimsI64
      idxs_with_vnames = concatMap (\vn -> map (vn,) (transpose idxs)) arrs

  forM_ (zip sten_params idxs_with_vnames) $ \(param, (arr, idxs')) ->
    copyDWIMFix (paramName param) [] (Var arr) idxs'

  flatName <- newVName "idx_out_flat"

  out_ns <- replicateM (length is) (newVName "idx_out")
  let is' = zipWith4 (\i t n d -> inBounds' (toInt64Exp (Var i) * d + t) n)
              is ts' dimsI64 tileDimsI64
      spaceOut = SegSpace flatName (zip out_ns (map Var out_ns))
  zipWithM_ dPrimV_ out_ns is'

  compileStms mempty (bodyStms $ lambdaBody sten_lam) $
    zipWithM_ (compileThreadResult spaceOut) (patternElements pat) $
      map (Returns ResultMaySimplify) $ bodyResult $ lambdaBody sten_lam
```

Figure 31: Part 2 of the auxiliary function for generating a sequential loop nest over a tile.

```
x_elem_4401 = ((int32_t *) arr_mem_4404.mem)[smin64(n_4388 -
                                                    (int64_t) 1,
                                                    smax64((int64_t) 0,
                                                           gtid_4399 *
                                                           (int64_t) 256 +
                                                           t_4412 +
                                                           (int64_t) -1)) *
                                             m_4389 +
                                             smin64(m_4389 -
                                                    (int64_t) 1,
                                                    smax64((int64_t) 0,
                                                           gtid_4400 *
                                                           (int64_t) 256 +
                                                           t_4414 +
                                                           (int64_t) -1))];
```

Figure 32: Example of code generated by figure 31 for reading an element from the input array

```
compileMCOp ::
  Pattern MCMem ->
  MCOp MCMem () ->
  ImpM MCMem HostEnv Imp.Multicore ()
compileMCOp _ (OtherOp ()) = pure ()
compileMCOp pat (ParOp _ op@(SegStencil _ _ sten _ kbody)) = do
  ...
```

Figure 33: Type signature and parameters of `compileMCOp`

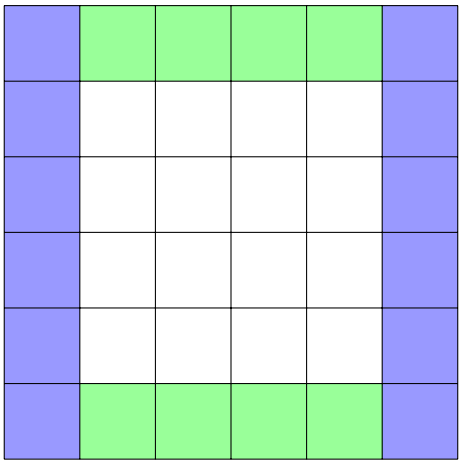


Figure 34: Boundary areas for a small grid for a stencil shape that extends one point in each direction, such as `[(-1, -1), (1, 1)]`. The white area in the middle is processed by the main loop, and each of the colored boundary areas are processed by a separate loop.

Program \Implementation	Tabulate	Stencil untiled	Stencil tiled	Stencil tiled + partitioned
Jacobi 2D 5pt	300 534 μs	1 215 607 μs	376 386 μs	300 742 μs
Jacobi 2D 9pt	372 119 μs	1 545 113 μs	462 686 μs	319 560 μs
Jacobi 3D 7pt	522 489 μs	14 055 745 μs	630 668 μs	574 936 μs
Jacobi 3D 13pt	644 428 μs	24 823 523 μs	808 403 μs	758 718 μs

Table 1: Programs and their average runtime over 10 runs using different implementations.

adjust tile sizes without re-compiling the compiler itself.

The first thing to note is that the untiled stencil implementation performs terribly compared to the tabulate-based one. This isn't surprising – Futhark's tabulate primitive is fairly well-optimized (when internalized, it becomes a `SegMap`, which is perhaps the most widely used parallel construct in the Futhark language), whereas the untiled stencil implementation performs no optimizations at all.

The tiled implementation is a lot faster than the untiled – it's still worse than tabulate, but it's actually in the same ballpark. When we add partitioning, we get a bit more speedup, and in the case of the 2D jacobi-stencil with 9 points, we actually outperform tabulate. For the other benchmarks, it is possible that we could equal or outperform the tabulate-based implementation by choosing better tile sizes.

Table 2 shows the speedup we get from the two optimizations. Something that stood out to me is that, in the prototypes, we didn't see huge speedups from tiling alone, but in these benchmarks we do. There's a chance that this indicates some flaw in the untiled implementation.

```
space = getSpace op
ns = map (toInt64Exp . snd) $ unSegSpace space
idxs = stencilIndexes sten
bounds = map (\xs -> (fromInteger (minimum xs),
                      fromInteger (maximum xs))) idxs
nsInner = zipWith (\(mi, ma) n -> n + mi - ma) bounds ns
lBounds = map (negate . fst) bounds
offsetsLower = init $ zipWith (++) (inits lBounds)
                                   (tails (replicate numDims 0))
nsLower = zipWith (++) (map init $ tail $ inits nsInner)
                       (zipWith (:) lBounds $ tail $ tails ns)
offsetsUpper = zipWith (++)
  (inits lBounds)
  (zipWith (:) (zipWith (+) nsInner lBounds)
               (tails (replicate (numDims-1) 0)))
nsUpper = zipWith (++) (map init $ tail $ inits nsInner)
                       (zipWith (:) (map snd bounds) $ tail $ tails ns)
```

Figure 35: Calculating the offsets and iteration spaces for the various loops.

Speedup		
Program	Tiled	Tiled + partitioned
Jacobi 2D 5pt	69.04%	75.26%
Jacobi 2D 9pt	70.05%	79.32%
Jacobi 3D 7pt	95.51%	95.91%
Jacobi 3D 13pt	96.74%	96.94%

Table 2: Speedup percentage for different programs when going from the untiled implementation to the tiled versions.

8 Conclusion

In this thesis, I've measured performance of several different prototype implementations of simple stencil computations, using various loop transformations to improve performance. I've also explored the effect of tile size on performance. Based on these prototypes, I've implemented code generation for stencil computations in the Futhark compiler, in three different versions. I've explained the details of the implementation and shown examples of the generated C-code, and I've measured the performance of each implementation using several different Futhark stencil programs.

In the prototype implementations, the speedup from tiling alone was unimpressive, but in the Futhark implementation tiling is where nearly all the speedup comes from. The untiled implementation in Futhark is also incredibly slow when compared to a stencil implemented with pre-existing Futhark primitives. This could indicate that there is some flaw in my untiled implementation that negatively impacts performance.

Programs that use my implementation are typically outperformed by programs that use Futhark's pre-existing `tabulate_2d` and `tabulate_3d` primitives, but not by much. With further work, the code generation for the stencil-specific construct can likely be improved to outperform `tabulate_2d` and `tabulate_3d`. There are several pieces of low-hanging fruit: Multi-level tiling to better use the L2 and L3 caches is straight-forward to implement. Time skewing for iterative stencils would be worth exploring, and with it various scheduling models.

```
nsubtasks <- dPrim "num_tasks" $ IntType Int32
retvals <- getReturnParams pat op
s <- segOpString op

-- For each dimension, we need 2 loops (one at each boundary of the array)
forM_ (zip4 nsLower offsetsLower nsUpper offsetsUpper) $
  \(loopStart, offsetsStart, loopEnd, offsetsEnd) -> do
    -- Loop for the "lower" boundary
    loop_lower_ns <- replicateM numDims (newVName "idx_lower")
    zipWithM_ dPrimV_ loop_lower_ns loopStart

    let loopStartSpace = SegSpace (segFlat space)
                $ zip (map fst $ unSegSpace space)
                $ map Var loop_lower_ns
    codeStart <- compileStencilBoundaryLoop
                     pat loopStartSpace offsetsStart ns sten kbody
    let iterationsStart = product loopStart
        schedulingInfoStart = Imp.SchedulerInfo (tvVar nsubtasks)
                                                (untyped iterationsStart)
        nonFreeStart = segFlat loopStartSpace :
                       tvVar nsubtasks :
                       map Imp.paramName retvals
    freeParamsStart <- freeParams codeStart nonFreeStart
    let taskStart = Imp.ParallelTask codeStart (segFlat loopStartSpace)
    emit $ Imp.Op
         $ Imp.Segop s freeParamsStart taskStart Nothing retvals
         $ schedulingInfoStart (decideScheduling' op codeStart)

    -- (similar loop for "upper" boundary is elided for brevity)
```

Figure 36: Generating boundary area loops

References

[1] M. Christen, O. Schenk, and H. Burkhart. "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures". In: *2011 IEEE International Parallel Distributed Processing Symposium*. 2011, pp. 676–687. DOI: 10.1109/IPDPS.2011.70.

[2] Bastian Hagedorn et al. "High Performance Stencil Code Generation with Lift". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 100–112. ISBN: 9781450356176. DOI: 10.1145/3168824. URL: https://doi.org/10.1145/3168824.

[3] Troels Henriksen et al. "Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. URL: http://doi.acm.org/10.1145/3062341.3062354.

[4] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. "High-Performance Code Generation for Stencil Computations on GPU Architectures". In: *Proceedings of the 26th ACM International Conference on Supercomputing*. ICS '12. San Servolo Island, Venice, Italy: Association for Computing Machinery, 2012, pp. 311–320. ISBN: 9781450313162. DOI: 10.1145/2304576.2304619. URL: https://doi.org/10.1145/2304576.2304619.

[5] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. ISBN: 1558602860.

[6] Ben Lippmeier and Gabriele Keller. "Efficient Parallel Stencil Convolution in Haskell". In: *Proceedings of the 4th ACM Symposium on Haskell*. Haskell '11. Tokyo, Japan: Association for Computing Machinery, 2011, pp. 59–70. ISBN: 9781450308601. DOI: 10.1145/2034675.2034684. URL: https://doi.org/10.1145/2034675.2034684.

[7] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[8] Faizur Rahman, Qing Yi, and Apan Qasem. "Understanding stencil code performance on multicore architectures". In: May 2011, p. 30. DOI: 10.1145/2016604.2016641.

[9] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. "Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation". In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO '17. Austin, USA: IEEE Press, 2017, pp. 74–85. ISBN: 9781509049318.

[10] Gerhard Wellein et al. "Efficient Temporal Blocking for Stencil Computations by Multicore-Aware Wavefront Parallelization". In: *2009 33rd Annual IEEE International Computer Software and Applications Conference*. Vol. 1. 2009, pp. 579–586. DOI: 10.1109/COMPSAC.2009.82.

[11] C. Yount et al. "YASK—Yet Another Stencil Kernel: A Framework for HPC Stencil Code-Generation and Tuning". In: *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. 2016, pp. 30–39. DOI: 10.1109/WOLFHPC.2016.08.

Appendices

A CUDA prototype, unoptimized

Naive implementation of a 9-point 2D stencil kernel in CUDA. Each thread reads nine values directly from global memory.

```
__global__
void smooth_2d_naive_kern(
    const uint64_t W,      // array size
    const uint64_t H,      // array size
    float* d_out,         // device output
    float* d_in           // device input
) {
    const uint64_t i = (blockDim.y * blockIdx.y) + threadIdx.y;
    const uint64_t j = (blockDim.x * blockIdx.x) + threadIdx.x;

    if(i < H && j < W) {
        d_out[i*W+j] = (
                d_in[max((int)i-1, (int)0)*W+max((int)j-1, (int)0)] +
                d_in[max((int)i-1, (int)0)*W+j] +
                d_in[max((int)i-1, (int)0)*W+min((int)j+1, (int)W-1)] +

                d_in[i*W+max((int)j-1, (int)0)] +
                d_in[i*W+j] +
                d_in[i*W+min((int)j+1, (int)W-1)] +

                d_in[min((int)i+1, (int)H-1)*W+max((int)j-1, (int)0)] +
                d_in[min((int)i+1, (int)H-1)*W+j] +
                d_in[min((int)i+1, (int)H-1)*W+min((int)j+1, (int)W-1)]
            ) / 9;
    }
}
```

B CUDA prototype, tiled

Tiled implementation of a 9-point 2D stencil kernel in CUDA. Each thread first reads a value from global memory and saves it in shared memory. The threads are syncronized, and each thread then calculates its output value, performing nine reads from shared memory.

```
__global__
void smooth_2d_tiled_kern(
    const uint64_t W,     // array size
    const uint64_t H,     // array size
    float* d_out,        // device output
    float* d_in          // device input
) {
    extern __shared__ float subgrid[];

    // Position for which this thread computes an output value
    const uint64_t i = ((blockDim.y-2) * blockIdx.y) + threadIdx.y;
    const uint64_t j = ((blockDim.x-2) * blockIdx.x) + threadIdx.x;

    // Position from which this thread fetches an input value
    const uint64_t fetch_i = min(max((int)i - 1, 0), (int)H-1);
    const uint64_t fetch_j = min(max((int)j - 1, 0), (int)W-1);

    subgrid[threadIdx.x*blockDim.y+threadIdx.y] = d_in[fetch_i * W + fetch_j];
    __syncthreads();

    if(threadIdx.x < blockDim.x-2 &&
       threadIdx.y < blockDim.y-2 &&
       i < H && j < W) {

        d_out[i*W+j] = (
                subgrid[threadIdx.x * blockDim.y + threadIdx.y] +
                subgrid[threadIdx.x * blockDim.y + threadIdx.y+1] +
                subgrid[threadIdx.x * blockDim.y + threadIdx.y+2] +

                subgrid[(threadIdx.x+1) * blockDim.y + threadIdx.y] +
                subgrid[(threadIdx.x+1) * blockDim.y + threadIdx.y+1] +
                subgrid[(threadIdx.x+1) * blockDim.y + threadIdx.y+2] +

                subgrid[(threadIdx.x+2) * blockDim.y + threadIdx.y] +
                subgrid[(threadIdx.x+2) * blockDim.y + threadIdx.y+1] +
                subgrid[(threadIdx.x+2) * blockDim.y + threadIdx.y+2]
            ) / 9;
    }
}
```

C Where to find the code

The three different Futhark compiler implementations can be found at the following links:

- `https://github.com/diku-dk/futhark/tree/stencils-mc`
  No optimizations.
- `https://github.com/diku-dk/futhark/tree/stencils-mc-tiled`
  Tiling, but no partitioning.
- `https://github.com/diku-dk/futhark/tree/stencils-mc-partitioned`
  Tiling and partitioning.

The multicore prototype implementation can be found at `https://github.com/Tayacan/stencil-prototypes-mc`.