FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Optimisation and GPU code generation of Stencils for Futhark

Master thesis

Christian Charlie Virt
skm861@alumni.ku.dk

Jonathan Wraa-Hansen
qck925@alumni.ku.dk

Supervisors
Troels Henriksen and Cosmin Eugen Oancea

May 31, 2021
Department of Computer Science,
University of Copenhagen

Abstract

Stencils are a common problem in the area of scientific computing. Exploitation of parallel computing is a central part when optimising for faster execution times of stencils running on large amounts of data. For this reason stencils are well suited to be run in a GPGPU setting. However, programming stencils to run on massively-parallel hardware is a time-consuming and error-prone exercise. For this reason it is useful to be able to express these stencils in a more abstract form, in a high-level programming language. Then a compiler will translate the stencil into more efficient and parallel computations in a GPGPU setting. Futhark is a high-level programming language, with the purpose of producing efficient multi-threaded CPU, CUDA and OpenCL programs. However, it has no native support for stencils. This thesis concerns the implementation of code generation for a stencil construct for the Futhark OpenCL and CUDA back-ends of the compiler. We investigate many designs for running stencils in a GPGPU setting, and analyse these different designs. We then choose the most efficient and robust prototype, to guide our implementation of code generation of the stencil construct in the Futhark compiler. The implemented stencil construct provides significant speedups compared to what could already be done with a nested map implementation in Futhark. For some hardware and stencils we achieve up to three times speedup.

Contents

1 Introduction

Performing dense matrix computations is a common problem in scientific computing. These problems include solving partial differential equations, performing image processing, and geometric modeling. The class of algorithms used to perform these computations are commonly refereed to as stencils [9]. An example of a stencil could be computing 2D Jacobi iterations

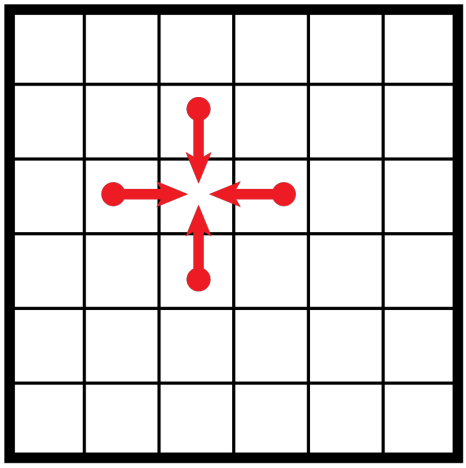


Figure 1: A von Neumann neighbourhood in 2D with 5 points. Image from https://upload.wikimedia.org/wikipedia/commons/e/ec/2D_von_Neumann_Stencil.svg

The 2D Jacobi iterations uses a von Neumann stencil neighbourhood consisting of 5 points. The points in the stencil consists of the center point at index (i, j) and surrounding points $(i-1, j), (i+1, j), (i, j-1), (i, j+1)$. For each (i, j) point in a 2D matrix input, the mean of the 5 points in the neighbourhood are computed and stored in an output array. With 2D Jacobi iterations we repeat this computation n times using the output array as the new input array[1]. It is essential to consider that a point from the input matrix is used several times when computing a stencil, therefore tiling approaches which increase the reuse of points is a primary factor for the optimisation of stencils. Performance optimisation of stencils is a widely researched topic since problems involving stencils often are computationally intensive. Among the research of optimising stencils, the exploitation of parallel computing is a central aspect. Some do experimental evaluation and design based on CPU multicore hardware [7] [6]. However, the focus in this thesis will be on exploring methods to optimise stencils, in order to compute stencils efficiently

[1]Example from https://en.wikipedia.org/wiki/Iterative_Stencil_Loops

on GPUs. Furthermore, we will extend the Futhark compiler to support optimised stencil constructs. Futhark is a programming language which is meant to produce efficient multi threaded CPU, CUDA and OpenCL programs. This thesis extends the Futhark compiler's code generation of sequential C, CUDA, and OpenCL. The stencil constructs should be compiled to optimised efficient parallel code for CUDA and OpenCL, as long as the points of the stencil are known at compile-time. We will not extend or change the Futhark programming language itself. The stencil constructs will be represented as built-in functions for the Futhark language.

1.1 Contributions

The following contains the primary contributions made by this thesis.

- Prototyping of different designs for running stand-alone stencils. This includes both designing, implementing, evaluating, and documenting several different designs for 1D, 2D, and 3D stencils.
- Implementation for handling of a stencil construct in several modules of the Futhark compiler. The representation of this stencil construct was provided to us among other things. The implemented parts include how the stencil-construct is evaluated in the interpreter (for 2D and 3D), type-assignment for stencils during internalisation (for 2D and 3D), first-order-transformation (which is the conversions of the stencil construct into a sequential representation for use when generating C code), and finally generation of OpenCL/Cuda code in the GPU pipeline (excluding the handling of the invariant array).
- Description and documentation of the implemented parts in the Futhark compiler, along with how this is connected to the prototyping.
- Evaluation of our implementations in the Futhark compiler, which includes both an evaluation of the correctness of the implementations, but also on the topic of practical viability in terms of run-time performance vs a reference implementation.

1.2 Limitations

- The fusion of stencils is not a topic which is covered or used in this thesis. This means that neither fusion of separate stencils nor fusion on iterative stencils will be investigated or implemented.

2 Background

This section explains what a stencil is, in more details. This section also provides important notation with regards to how we express stencil neighbourhoods in e.g. section 9. Furthermore we also provide a more detailed background of Futhark.

2.1 Stencils

In this paper, a stencil with d indices in the stencil neighbourhood is called a *d-point stencil*. To further elaborate on how stencils can be represented we will delve deeper into the Jacobi2D example. The Jacobi2D algorithm is common for solving Laplace's differential equation for square domains [1]. For this example, we compute the Jacobi2D stencil on a `N`×`M` input matrix, and the computation is repeated a number of times (`NumIterations`) to increase the accuracy of the solution.

```
1  computeJacobi2D(input[N,M], NumIterations)
2    output[N,M]
3    for k = 0 to NumIterations
4      for i = 0 to N
5        for j = 0 to M
6          output[i,j] = 0.2*( input[i-1,j] + input[i,j-1]
7                            + input[i  ,j] + input[i,j+1]
8                            + input[i+1,j])
9      for i = 0 to N
10       for j = 0 to M
11         input[i,j] = output[i,j]
12   return output
```

There is no dependency between the `input` and the `output` when computing the stencil for each point (i, j) in the `input` on lines 4-8. However, when repeating the computation a dependency arises between the `output` matrix and the updated `input` matrix. Therefore the outermost loop over k cannot be parallel. Typically stencils have a relatively high degree of potential parallelism due to the lack of dependencies when computing the stencil function. Here the stencil function can be seen on line 13. Therefore the degree of parallelism is typically in the same order of magnitude as the input size.

Stencil notation
In this paper, we have used the following notation to concisely describe the shape of 1D stencils.

[-2,...,2]

To indicate that the 1D stencil contains the indices [-2,1,0,1,2] for the stencil neighbourhood.

We used a more involved but hopefully elegant notation to concisely describe the indices for 2D and 3D stencil neighbourhoods.
Consider two sets of indices Y and X. For a 2D matrix the set Y corresponds to the range of stencil indices with respect to the row index i. The set X corresponds to the range of stencil points with respect to the column index j.

$$Y \times X = \{(i, j) | i \in Y, j \in X\}$$

$$\{-1, 0, 1\} \times \{0, 1\} = \{(-1, 0), (-1, 1), (0, 0), (0, 1), (1, 0), (1, 1)\}$$

This is also extended to 3D matrices with three sets of indices instead of two.

$$Z \times Y \times X = \{(z, i, j) | z \in Z, i \in Y, j \in X\}$$

$$\{-1, 1\} \times \{0, 1\} \times \{0\} = \{(-1, 0, 0), (-1, 1, 0), (1, 0, 0), (1, 1, 0)\}$$

Stencil mathematical example Let $N \in \mathbb{Z}_+^D$ denote the size of a D-dimensional input A

$$N = \begin{pmatrix} N_1 \\ N_2 \\ \vdots \\ N_D \end{pmatrix}$$

such that N_d is the size of the d'th dimension of A. Let $P \in \mathbb{Z}_+$ denote the number of points in the stencil input indices I.

For example, given a two-dimensional (D=2) input A, then $A_{n,m}$ denotes element (n, m) in A for $0 \leq n < N_1$ and $0 \leq m < N_2$, alternatively one could say that A_k^{flat} is the k'th entry in a flattened A (denoted A^{flat}) with $k = n \cdot N_2 + m$ for $0 \leq k < N_1 \cdot N_2$.

B denotes the result of applying the stencil on the entire input A, where the size of B is also N. Let X_k denote some stencil neighbourhood for the $k'th$ entry in A^{flat}, then x_i is the i'th value in the neighbourhood and the size of the neighbourhood is P. Let $g(X_k)$ denote the value of applying some function g to a stencil neighbourhood X_k. Then b_k is the k'th entry in B^{flat} where $b_k = g(X_k)$. Let $maxIdx(I, d)$ denote the maximum index in I for the d'th dimension and $minIdx(I, d)$ denote the minimum index in I for the d'th dimension. Then $W(I, d) = maxIdx(I, d) - minIdx(I, d) + 1$ denotes the width of the the stencil for the d'th dimension.

Here A can be represented as a two-dimensional 2×4 matrix and I is a three-point stencil, which can be represented by a two-dimensional 2×3 matrix.

$$A = \begin{bmatrix} 5 & 2 & 6 & 4 \\ 10 & 4 & 5 & 1 \end{bmatrix}$$

$$I = \begin{bmatrix} -1 & 0 & 2 \\ -2 & 0 & 1 \end{bmatrix}$$

$$g(X_k) = X_{k,0} + X_{k,1} + X_{k,2}$$

Then we have the following values (assuming that the out-of-bounds values are padded with the values of the edges).

$$N = \begin{pmatrix} N_1 \\ N_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$$

$$P = 3$$

$$minIdx(I, 1) = -1$$

$$maxIdx(I, 1) = 2$$

$$W(I, 1) = maxIdx(I, 1) - minIdx(I, 1) + 1 = 4$$

$$minIdx(I, 2) = -2$$

$$maxIdx(I, 2) = 1$$

$$W(I, 2) = maxIdx(I, 2) - minIdx(I, 2) + 1 = 4$$

$$\begin{aligned} b_1 = g(X_1) &= X_{1,0} + X_{1,1} + X_{1,2} \\ &= A_{-1,-2} + A_{0,0} + A_{2,1} \\ &= 5 + 5 + 4 \\ &= 14 \end{aligned}$$

$$B = \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \\ b_5 & b_6 & b_7 & b_8 \end{bmatrix} = \begin{bmatrix} 14 & 12 & 12 & 7 \\ 19 & 14 & 11 & 4 \end{bmatrix}$$

2.2 Futhark

Futhark is a programming language which is meant to handle relatively smaller parts of larger programs that are computationally intensive. Futhark is not meant to be used as a general-purpose language, instead it provides support such that Futhark compiled code can be integrated with other languages such as Python and C. The Futhark language is a purely functional array language [4, p. 12][2]. An array language allows the application of operations on entire arrays. Many of these operations consists of Second-Order Array Combinators (SOACs). The SOACs consists of array operations (e.g. `scan`, `reduce`, and `map`) which are similar to the higher order functions found in other functional programming languages. These operations are naturally suited for parallelisation. The SOACs differ from their higher order function counterparts in terms of passing functions as parameters. The SOACs only allow syntactic anonymous functions as parameters. The main design principle of the Futhark language is to be expressive such that complex programs can be written conveniently. However, there are some restrictions since too much expressiveness can limit the opportunity for the compiler to perform optimisations. A limitation is that Futhark does not support irregular arrays. However, it does support nested parallelism of regular arrays. Therefore we have not considered stencils applied to irregular arrays in this thesis.

2.2.1 Compiler design

The Futhark compiler is written in Haskell. The compiler initially transforms a Futhark source program into an internal intermediate representation, also referred to as the *core intermediate representation*. The compiler consists of three pipelines:

- Sequential C pipeline.
- GPU pipeline (OpenCL or CUDA).
- Multi-Core pipeline.

Each pipeline moves the program through a series of passes, where each pass conceptually takes a program as input and outputs a program. The number of passes and the types of passes differs between the pipelines. However, several of the initial passes are the same for every pipeline. Some passes takes

[2]See also Futhark website: https://futhark-lang.org/index.html

a program in some intermediate representation and rewrites the program into the same intermediate representation. Other passes produce an entirely new intermediate representation. When a pass simply rewrites a program, it could be a *simplifying* pass, which applies optimisations such as copy propagation and dead-code elimination. A *transforming* pass might take some intermediate representation and then produce a first-order intermediate representation. Such a transforming pass is used in the sequential C pipeline [4, p. 70-71]. A first-order intermediate representation is a program where all functions are in first-order. Functions in first-order are treated as values that can be assigned to variables, which then can be passed as parameters to functions, or returned from other functions.

2.2.2 Overview of compiler extensions for stencil constructs

Since all pipelines use the same core intermediate representation, we must extend the Futhark compiler such that it transforms the Futhark stencil constructs into the core intermediate representation. We will elaborate on how we extended the Futhark compiler to produce the core intermediate representation for Futhark stencil constructs in section 3.2. However, much of this implementation was provided by our supervisor Troels Henriksen. Therefore our own contribution to this part is relatively small compared to the rest of our implementations.

In terms of the sequential C pipeline we only need to extend the first-order transformation pass, in order to enable sequential C code generation of the stencil construct. Our first-order transformation pass implementation is presented in section 3.4. In terms of the GPU pipeline we need to extend the pass that produces a imperative intermediate representation, which is the final pass before OpenCL and CUDA code generation. The implementation of stencils for the GPU pipeline is presented at section 7. In terms of the Multi-Core pipeline, we do not have any extensions since multi-threaded CPU code generation is not in the scope of this thesis. Finally, Futhark also has an interpreter which we need to extend for the stencil construct. The extension of the interpreter is presented in section 3.3.

3 Extending the Futhark compiler with the stencil construct

This section concerns the representation of the stencil construct which was added to the compiler, as well as the addition to the different modules of the compiler that we made, excluding the GPU code generation which is in section 7.

3.1 Stencil representation in the Futhark prelude

We extend Futhark with three new functions for handling stencils. We refer to these functions as SOACs. The type of the three SOACs are denoted by

```
stencil_1d:
[d]i64 -> (c -> [d]a -> b) -> [n]c -> [n]a -> [n]b

stencil_2d:
[d](i64,i64) -> (c -> [d]a -> b) -> [n][m]c -> [n][m]a -> [n][m]b

stencil_3d:
[d](i64,i64,i64) -> (c -> [d]a -> b) -> [n][m][z]c -> [n][m][z]a -> [n][m][z]b
```

and will handle one-dimensional, two-dimensional, and three-dimensional stencils, respectively. The type annotation has similarities to the type annotation used in Futhark. For the `stencil_2d` function (an similarly for the other dimensions), the type expression can be described the following way, ordered left to right.

1. The first argument `[d](i64,i64)` represents an array of length `d` where each element is a tuple with 64-bit integers.
2. The second argument is a function that takes two arguments, firstly a value of type `c` and secondly an array with length `d` with elements of type `a`, then it returns type `b`.
3. The third argument is a $\texttt{n} \times \texttt{m}$ array with elements of type `c`. This array will be referred to as the `auxiliary array` or the `invariant array`. This primary purpose of this array is to allow access to other arrays inside the stencil function, as the alternative is to collect their neighbours as well.
4. The fourth argument is a $\texttt{n} \times \texttt{m}$ array with elements of type `a`. This is the array on which we collect the neighbours of.
5. The final part of the type expression is the return type. The `stencil_2d` function returns a $\texttt{n} \times \texttt{m}$ array with elements of type `b`.

Below (figure 2) we have an example of using the `stencil_2d` function in Futhark.

```
let hotspot_2d_iteration [M][N]
    (step:f32, cap:f32, rx:f32, ry:f32, rz:f32, amb_temp:f32)
    (temp: [M][N]f32) (power: [M][N]f32)
    : [M][N]f32 =
   let ixs = [(-1,0),(0,-1),(0,0),(0,1),(1,0)] in
   let f (C_pow:f32) (v:[5]f32) : f32 =
     let delta : f32 =
           ((step / cap) *
            (C_pow
             + (((v[3] - v[2]) + (v[1] - v[2])) / rx)
             + (((v[4] - v[2]) + (v[0] - v[2])) / ry)
             + (amb_temp - v[2]))
           ) / rz
     in v[2] + delta
   in stencil_2d ixs f power temp
```

Figure 2: The core part of the 2D stencil program Hotspot-2D, shown in futhark

The example (figure 2) is the primary part of the program Hotspot-2D which appear in places such as the Rodinia benchmark suite. Some equivalent imperative pseudo-code that does the same can be seen below (figure 3):

```
void hotspot_2d_iteration(
  long M, long N,
  float step, float cap, float rx, float ry, float rz,
  float amb_temp, float **temp, float **power, float **output){
  for(long i=0; i<M; i++){
    for(long j=0; j<N; j++){
      float C_pow = power[i][j];
      float v0 = temp[max(0,i-1)][j];
      float v1 = temp[i][max(0,j-1)];
      float v2 = temp[i][j];
      float v3 = temp[i][min(N-1,j+1)];
      float v4 = temp[min(M-1,i+1)][j];
      float delta = ((step / cap) *
              (C_pow
                + (((v3 - v2) + (v1 - v2)) / rx)
                + (((v4 - v2) + (v0 - v2)) / ry)
                + (amb_temp - v2))
              ) / rz;
      float res = v2 + delta;
      output[i][j] = res;
    }
  }
}
```

Figure 3: The core part of the 2D stencil program Hotspot-2D, shown as imperative pseudo-code

It should however be noted that can create a fast implementation of the Hotspot-2D program in Futhark without using the stencil-construct. It does however require some different writing. An example of this stencil program, that does the same, but without the stencil-constuct can be seen in figure 4.

```
let hotspot_2d_iteration_maps [M][N]
    (step:f32, cap:f32, rx:f32, ry:f32, rz:f32, amb_temp:f32)
    (temp: [M][N]f32) (power: [M][N]f32)
    : [M][N]f32 =
  let Mm1 = M-1
  let Nm1 = N-1
  in tabulate_2d M N (\i j ->
       let C_pow : f32 = #[unsafe] power[i,j]
       let v0 : f32 = #[unsafe] temp[i64.max 0 (i-1), j]
       let v1 : f32 = #[unsafe] temp[i, i64.max 0 (j-1)]
       let v2 : f32 = #[unsafe] temp[i, j]
       let v3 : f32 = #[unsafe] temp[i, i64.min Nm1 (j+1)]
       let v4 : f32 = #[unsafe] temp[i64.min Mm1 (i+1), j]
       let delta : f32 =
             ((step / cap) *
              (C_pow
               + (((v3 - v2) + (v1 - v2)) / rx)
               + (((v4 - v2) + (v0 - v2)) / ry)
               + (amb_temp - v2))
             ) / rz
       in v2 + delta
  )
```

Figure 4: The core part of the 2D stencil program Hotspot-2D, shown in futhark without using the stencil construct, and implemented to be reasonably fast. The `#[unsafe]` tag disables bounds-checking on indexing.

The Futhark example without the stencil-construct is however not as easy to read as with the stencil-construct, as many things have been made very explicit, and had they not been there it would either be not fast or run incorrectly.

Another thing that should have been made clear by the programs above, is that we chose to handle elements on the edge of the array in such a way that if they would access an index out-of-bounds then they are directed to the nearest in-bounds index.

3.1.1 Internal Futhark compiler representation of stencils

There are two internal constructs used for stencil indices in Futhark.

- StencilDynamic
- StencilStatic

If the stencil indices are static and known at compile-time, then the `StencilStatic` construct is used, we refer to this as a *static stencil*. If the stencil indices are known at run-time and not compile-time, then the `StencilDynamic` construct is used, we refer to this as a *dynamic stencil*. For this thesis, the aim is to optimise the static stencils. Consider a stencil function `f`:

```
let f _ (xs : [3]f32) = (xs[0] + xs[1] + xs[2])/3
```

When optimizing the static stencils, then the internal Futhark compiler representation of the stencil function `f` is different from the representation in the Futhark code. Rather than having an array `xs` as parameter, the function `f` will have each element of `xs` as a separate parameter. In the body of `f` the new parameters will replace the occurrences of `xs[0]`, `xs[1]`, ..., `xs[d]`, with its corresponding parameter name. Below we have an example of the internal Futhark compiler representation used for GPU compilation, based on the example shown above.

```
1 \{x_4463 : bool, x_elem_4479 : f32, x_elem_4480 : f32,
    x_elem_4481 : f32}
2     : {f32} ->
3       let {x_4467 : f32} = fadd32(x_elem_4479, x_elem_4480)
4       let {x_4469 : f32} = fadd32(x_4467, x_elem_4481)
5       let {defunc_1_f_res_4470 : f32} = fdiv32(x_4469, 3.0f32)
6       in {defunc_1_f_res_4470}
```

At line 1 we see the parameters of the stencil function `f`. The first parameter `x_4463` of type `bool` corresponds to the first argument of the function `f`, namely the empty tuple `()`. The following three parameters `x_elem_4479`, `x_elem_4480`, and `x_elem_4481` corresponds to the elements `xs[0]`, `xs[1]`, and `xs[2]`, respectively.

3.1.2 Limitations of the internal Futhark compiler representation

Due to this internal representation, we must bind all elements of `xs` to variables before computing the function `f`. Elements of `xs` are loaded into registers when bound to a variable. The GPU has a limited amount of registers per thread and group, therefore one might hit the maximum number of registers when executing a stencil. However, if the stencil function body is simple enough (e.g. the mean computed by `f`), then the CUDA or OpenCL compiler can interleave the load to register operations and arithmetic operations. If the load to register operations and arithmetic operations are interleaved, then the minimum required registers per thread will be reduced. It must however be stressed that this specific optimisation is entirely reliant on the compiler and one should not expect it to happen for any compiler (or even different version of the same compiler). The function above (3-point mean in 1D) is one such an example where the loads can be interleaved Upon inspecting the PTX-assembly code of this function is does look like the CUDA compiler (that it

was compiled with) can do this. Below is a snippet of the inlined lambda function (and the final write), in which we can see that the compiler happened to be able to do this. In the case that PTX-assembly is unfamiliar to the reader, then registers (such as %f10) are symbolic and will later be mapped to actual registers.

```
ld.shared.f32   %f10, [%rd2];
ld.shared.f32   %f11, [%rd2+4];
add.f32         %f12, %f10, %f11;
ld.shared.f32   %f13, [%rd2+8];
add.f32         %f14, %f12, %f13;
div.rn.f32      %f15, %f14, 0f40400000;
st.global.f32   [%rd11], %f15;
```

If the operations cannot be interleaved and/or the required registers is larger than the available registers per group or thread, then the elements of `xs` will spill to globally memory. An example of where the loads are not fully interleaved the following lambda function (using the same neighbours as before):

```
let f _ v = ((v[0]+v[1]+v[2]) / 3f32)-(v[0]*v[1]+v[2])
```

and the PTX-assembly of the inlined lambda function is:

```
ld.shared.f32   %f10, [%rd2];
ld.shared.f32   %f11, [%rd2+4];
add.f32         %f12, %f10, %f11;
ld.shared.f32   %f13, [%rd2+8];
add.f32         %f14, %f12, %f13;
div.rn.f32      %f15, %f14, 0f40400000;
fma.rn.f32      %f16, %f10, %f11, %f13;
sub.f32         %f17, %f15, %f16;
st.global.f32   [%rd11], %f17;
```

Where one can see that the loads are not fully interleaved between operations (as one cannot be done in this example, without reloading values) meaning that it needs to map these symbolic registers to different actual registers.

Spilling elements to global memory will significantly increase the running time compared to the case where no spilling occurs.

Currently there is no method to check the required number of registers per thread in the Futhark compiler. Therefore we have no mechanism to choose a different compilation strategy for the case where we encounter a program that will spill elements to global memory.

3.2 Overview of the extensions to Futhark compiler modules

In order to implement stencils for the Futhark compiler we have the following goals

1. Extend various modules in Futhark in order to enable type checking and internalisation[3] of the `stencil_2d` and `stencil_3d` SOACs[4].

2. Extend the Futhark interpreter.

3. Extend Futhark code generation for sequential C, OpenCL and CUDA back-ends.

We extended the following Futhark modules:

- `Language.Futhark.Prop` to provide type information of the stencil SOACs.
- `Futhark.Internalise` to transform the stencil SOACs into an internal Futhark compiler intermediate representation.
- `Language.Futhark.Interpreter` to implement stencils for the Futhark interpreter.
- `Futhark.Transform.FirstOrderTransform` to implement a transformation of the stencil from a Futhark compiler intermediate representation to a first order representation which is used to generate sequential C code.
- `Futhark.CodeGen.ImpGen.Kernels` to add a case for GPU compilation of static stencils.

We extended the `Language.Futhark.Prop` module with type information of the `stencil_2d` and `stencil_3d` functions, while the `stencil_1d` type information was provided by our supervisor Troels Henriksen. The type information consists of the type variables including array size variables, the function parameter types and the function return type. This type information can then be used by the Futhark type checking module. We extended the `Futhark.Internalise` module with transformations from Futhark abstract syntax source code of `stencil_2d` and `stencil_3d` to the core intermediate representation. An implementation of the transformation of `stencil_1d` was provided by our supervisor Troels Henriksen. We will only show the `stencil_2d` implementation since the transformation of the other stencil constructs are very similar

```
1 handleSOACs [TupLit [is, lam, inv, arr] _] "stencil_2d" = Just $ \desc -> do
2     is' <- internaliseExpToVars "stencil_is" is
3     inv' <- internaliseExpToVars "stencil_arr" inv
4     arr' <- internaliseExpToVars "stencil_arr" arr
5     lam' <- internaliseStencilLambda internaliseLambda is' lam inv' arr'
6     let arr_type = mapM lookupType arr'
```

[3]By "internalisation" we mean the transformation of the stencil SOACs into the internal Futhark compiler representation of stencils.

[4]A full implementation of parsing, interpreting, and type-checking of the `stencil_1d` SOAC was provided by our supervisor Troels Henriksen

```
7       w1 <- arraysSize 0 <$> arr_type
8       w2 <- arraysSize 1 <$> arr_type
9       p <- arraysSize 0 <$> mapM lookupType is'
10      letTupExp' desc $
11          I.Op $
12          I.Stencil [w1, w2] p (StencilDynamic is') lam' (map ([],) inv') arr'
```

The stencil construct initially consist of expressions which are transformed using helper functions `internaliseExpToVars`, `internaliseStencil` and `internaliseLambda`. Since we did not implement the helper functions, we will omit their details. On line 2 we bind `is'` to a list of variable names `VName`. In this case, each `VName` is a variable bound to the stencil indices expression. There is a `VName` for every dimension in the stencil. In this case there will be two `VName` in the list. The `inv'` and `arr'` bindings on line 3-4 correspond to the invariant array and input array, respectively. In this case the input or invariant array might contain tuples. If any array contain tuples in the source language, then the array are transformed into separate arrays in the core intermediate representation. Therefore we would get a `VName` for every entry in the tuple. The `lam'` binding on line 5 represents the stencil function, which is an anonymous function `Lambda` in the core intermediate representation. The bindings `w1` and `w2` on lines 7-8 correspond to the number of rows and number of columns in the input array, respectively. Then `p` on line 9 correspond the number of points in the stencil. Finally on lines 10-12 we bind the core intermediate representation `Stencil` to the name `desc`. Initially, all stencils are transformed to `StencilDynamic`. However, during the passes of a pipeline it is determined whether the stencil indices are known at compile-time or run-time. If the stencil indices are determined to be known at compile-time then the `StencilDynamic` construct will be replaced by a `StencilStatic` construct where the stencil indices are represented by the Haskell `Integer` type rather than `VName`.

3.3 The interpreter

The `stencil_1d` implementation in the Futhark interpreter was provided by our supervisor Troels Henriksen. The implementations of the stencil SOACs in the interpreter focus on correctness and simplicity above all else. The performance, or rather lack thereof, is therefore of little concern as long as the interpreter evaluates small examples correctly within a reasonable time-frame. We implemented `stencil_2d`, `stencil_3d` for the interpreter, however, Both implementations share a lot of implementation details with the initially provided `stencil_1d`. As an example, we will describe the `stencil_2d` implementation for the interpreter. The interpreter works directly on the abstract syntax of the Futhark source code.

3.3.1 Stencil 2D interpreter implementation

The interpreter consists of a interpreter context `Ctx`, containing the prelude functions such as `stencil_1d`, `stencil_2d`, and `stencil_3d`. As an example, we extend the `Ctx` with a `stencil_2d` function definition with the code seen in Figure 5. For a d-point stencil with a $n \times m$ input `as`, we compute a three dimensional array `hoods` of size $n \times m \times d$ where each innermost subarray of size d contains the indices for computing the stencil function for particular point of the output.

```
1  def "stencil_2d" = Just $
2        TermPoly Nothing $ \t -> return $
3          ValueFun $ \v ->
4            case (fromTuple v, unfoldFunType t) of
5              (Just [is, f, cs, as], ([_], ret_t))
6                | Just row_shape <- typeRowShape ret_t,
7                  ValueArray (ShapeDim n (ShapeDim m elm_shape)) as_arr <- as,
8                  Just ixss <-
9                    let from2 [k, l] = Just (k, l)
10                       from2 _ = Nothing
11                   in mapM
12                        (from2 <=< fromTuple)
13                        $ snd $ fromArray is -> do
14                 -- We Hardcode the boundary condition to repeat edge element.
15                 let bound i = max 0 . min i . (\x -> x - 1)
16                 let getElem (i, j) =
17                       case as_arr ! i of
18                         ValueArray _ row -> row ! j
19                         _ -> error "Bad input"
20                     hood i j =
21                       toArray' elm_shape $
22                         map
23                           ( getElem
24                               . ( \(k, l) ->
25                                     ( fromIntegral $ bound (i + asInt64 k) n,
26                                       fromIntegral $ bound (j + asInt64 l) m
27                                     )
28                                 )
29                           )
30                           ixss
31                     hoods = map (\i -> map (hood i) [0 .. m - 1]) [0 .. n -1]
32                 toArray (ShapeDim n row_shape)
33                   <$> forM
34                     (zip (snd $ fromArray cs) hoods)
35                     ( \(cvs, rows) ->
36                         toArray (ShapeDim m elm_shape) <$>
37                           zipWithM (apply2 noLoc mempty f)
38                             (snd $ fromArray cvs) rows
39                     )
40               | otherwise ->
41                 error $ "Bad return type: " ++ pretty ret_t
42             _ ->
43               error $
44                 "Invalid arguments to stencil_2d intrinsic:\n"
45                   ++ unlines [pretty t, pretty v]
```

Figure 5: Futhark interpreter implementation of `stencil_2d`.

The case-expression on line 4 to line 45 has two potential cases. The case on

line 5 matches a function type with four arguments, the second match on line 42 is a wildcard that will match any function that does not contain four arguments and leads to an error. The input array `as_arr` and the outer dimension `n` and inner dimension `m` are defined in line 7. The stencil indices are defined as a list of tuples `ixss` in line 8. We use the `bound` function defined at line 15 to bound the stencil indices. Let `i` be the index into the first dimension and let `j` be the index into the second dimension. Then `getElem` defined at line 16 is used to get element $\texttt{as\_arr}_{\texttt{i,j}}$ from the input array. The `hood` function defined at line 20 is used to compute the `hoods` array mentioned previously. At lines 32 to 39 we apply the stencil function and write the result to the output array for all elements. As mentioned previously, the implementation is not meant to be optimal in terms of runtime or memory usage. The memory usage is larger than optimal since we compute all the stencil indices for all elements and store the entire result in `hoods`. One could optimise this approach in terms of memory usage by eliminating the `hoods` array. Then one could iteratively for each output element compute a single "hood" to be applied in the stencil function.

3.4 First-order-transformation of stencils

The purpose of this pass in the compiler is to transform non-trivial parallel constructs onto the First-order representation, which is a sequential representation. The First-order representation is similar to standard Futhark code, except it has only sequential constructs.

To better illustrate what the First-order-transformer will create, the pseudocode is presented the generated representation. The only difference between the stencils with Dynamic indices vs the ones with Static indices are that the static ones are unrolled in the inner-most loop. Note that an implicit assumption is that the abstract types are primitives or a tuple of primitives or tuples (and recursively so on). The assumption is based on the fact that stencils not living up to the assumption are converted into map-nests prior to reaching this point, and are therefore not handled by the stencil generator.

```
1  -- pseudocode of the generated representation in 2D.
2  stencilSequential(
3      y_ixs:[D]int64,
4      x_ixs:[D]int64,
5      lambda:(c->[D]a->b),
6      invariants:[M][N]c,
7      variants[M][N]a)
8      : [M][N]b =
9  let result = newEmptyArray(b, shapeOf(variants));
10 let max_index_y = M-1
11 let res = loop (yx_mat = result) for gidy < M do {
12     let max_index_x = N-1
13     let res_xs = loop (x_mat = yx_mat) for gidx < N do {
14         let invar = invariants[gidy, gidx];
15         let temporary_array = newEmptyArray(a, shapeOf(ixs))
16         let vars = loop (tmp = temporary_array) for k < D do {
17             let relative_y = y_ixs[k]
18             let relative_x = x_ixs[k]
19             let bounded_y = max(0,min(M-1,(gidy+relative_y)))
20             let bounded_x = max(0,min(N-1,(gidx+relative_x)))
```

```
21              in tmp with [k] = variants[bounded_y, bounded_x]
22          in x_math with [gidy, gidx] = lambda(invar, vars)
23      in yx_mat with [gidy, :] = res_xs
24  in res
```

Listing 1: Pseudocode of result of the First order transformer

In the following part the indivial parts of the transformation are shown. The following program used as a example:

```
entry main [M][N] (as : [M][N]f32): [M][N]f32 =
    let cs = (map (map (const 2f32)) as)
    let inds = [(-1i64,-1i64),(0i64,0i64),(1i64,1i64)]
    let f c (xs : [3]f32) = (c + xs[0] + xs[1] + xs[2]) / 3
    in stencil_2d inds f cs as
```

Note that some of the variable names appear differently in these example compared to what the compiler produces, as the compiler adds a number at the end of each variable (presumably to make sure they are always unique). The numbers have been removed. In cases where a distinction is needed, a `_x` or `_y` is added to make the variables distinct, or in the cases of the outer loops it is changed to `gid_x` and `gid_y`.

```
1  map_arrs <- resultArray []
       returns_tp
2  outer_loops <- loopNest
       innerBody map_arrs []
       inputShape [] []
3  letBind pat outer_loops
```

Listing 2: First order transformer

```
1  let {result : [M][N]f32} = scratch(f32, M, N)
2  let {max_index_y : i64} = sub_nw64(M, 1i64)
3  let {defunc_3_stencil_2d_res : [M][N]f32} =
4  loop {stencil_out_y : *[M][N]f32} = {result}
5  for gid_y:i64 < M do {
6    let {max_index_x : i64} = sub_nw64(N, 1i64)
7    let {res : [M][N]f32} =
8      loop {stencil_out_x : *[M][N]f32} = {stencil_out_y}
9      for gid_x:i64 < N do {
10         ...
11         ...
12         ...
13   }
14   in {res}
15 }
16 in {defunc_3_stencil_2d_res}
```

Listing 3: Result of first order transformation

Figure 6: The loops and the output of the stencil

Comments for (figure 6):

Line 1 left: Corresponds to line 1 right. This creates the empty array used as output.

Line 2 left: Corresponds to line (2-15) right minus 3. This create the loop nest iterating over input/output. The actual code creating the loop nest will not be shown as it is too needlessly long.

Line 3 left: Corresponds to line (3,16). This binds the result of the stencil to the declared output variable.

```
let boundIx li max_index
    rel_off = do
      relix <- letSubExp "
    index" $ BasicOp $ BinOp
     (Add Int64
    OverflowUndef) rel_off
    li
      lower_bounded_index <-
    letSubExp "
    lower_bounded_index" $
    BasicOp $ BinOp (SMax
    Int64) relix (intConst
    Int64 0)
      letSubExp "
    upper_bounded_index" $
    BasicOp $ BinOp (SMin
    Int64)
    lower_bounded_index
    max_index
```

Listing 4: First order transformer

```
let {index : i64} = add_nw64(-1i64, gid_y)
let {lower_bounded_index : i64} = smax64(index, 0i64)
let {upper_bounded_index : i64} = smin64(
    lower_bounded_index, max_index)
```

Listing 5: Result of first order transformation

Figure 7: The bounding of the stencil indices

Comments for (figure 7): Line 2-4 corresponds to the entire right. The lines take care of the edge-extention scheme the we use, so before any indexing we direct it towards to closest in-bounds index by using `max(0,min(M,index))`. The (-1i64) is simply one of the indices in the example.

```
let load_elements ixs =
      forM (zip variants
    variants_tp) $ \(vname,
    tp) ->
        letSubExp "
    input_element" $ BasicOp
     $ Index vname $
    fullSlice tp $ map
    DimFix ixs
```

Listing 6: First order transformer

```
let input_element = as[upper_bounded_index_y,
    upper_bounded_index_x]
```

Listing 7: Result of first order transformation

Figure 8: The loader of the element(s) from the variant array(s)

Comments for (figure 8): The loader needs to read from each element of the tuple (of possibly one 1 element), recall that arrays of tuples in the source language are handled as tuples of arrays in the compiler's intermediate language.

```
1 let innerBody out_pars list_of_index
      list_of_max_ix = do
2       forM_ (zip invariantParams invariants
      ) $ \(p, (_, arr)) -> do
3         arr_t <- lookupType arr
4         letBindNames [p] $ BasicOp $ Index
      arr $ fullSlice arr_t $ map DimFix
      list_of_index
5       temp_array <- temporaryArray temp_tp
6       ...
```

Listing 8: First order transformer

```
1 let {x : f32} = = defunc_3_map_res_4567[
      gid_y, gid_x]
2 let {temporaryArray : [3i64]f32} =
      scratch(f32, 3i64)
```

Listing 9: Result of first order transformation

Figure 9: The loading of invariant elements and declaration of the temporary neighbourhood array

Comments for (figure 9): Line (1-5) left into line 1 right, for each array in the tuple of arrays, we load an element. Line 5 left into line 2 right, this declares the temporary array which is eventually passed to the lambda function.

```
1  StencilStatic is -> do
2    let ixs_len = toInteger $ length $ head
       is
3    let ix_unroll idx temp_arr_idx write_arr
       = do
4         ixs <- forM (zip3 idx list_of_index
      list_of_max_ix) $
5           \(rel, gix, imax) ->
6             boundIx gix imax . intConst
       Int64 $ rel
7         input_elements <- load_elements ixs
8         letwith write_arr (intConst Int64
       temp_arr_idx) input_elements
9    let funls = zipWith ix_unroll (transpose
       is) [0 .. ixs_len -1]
10   temp <- foldM (flip ($)) temp_array funls
11   pure $ map (BasicOp . SubExp . Var) temp
12 mapM_ (\(vp, vl) -> letBindNames [vp] vl) $
      zip variantParams exps
```

Listing 10: First order transformer

```
1 ... -- previous indices and loads have
      been unrolled, and are not shown.
2 ... -- bound current index
3 let {input_element : f32} = as[
      upper_bounded_index_y,
      upper_bounded_index_x]
4 let {lw_dest : [3i64]f32} = lw_dest with
      [2i64] = input_element
5 let {xs : [3i64]f32} = lw_dest
```

Listing 11: Result of first order transformation

Figure 10: The load of the final elements into the temporary array.

Comments for (figure 10): Note from line 1 left, that what we are concerned with here is the case where the indices are statically know.

Line 2 left: Look up the number of points of the stencil.

Line 3-8 left: Corresponds to line 3-4 right. This function that unrolls a single iteration (given an array and indices).

Line 9 left: This supplies the indices to the unroll function.

Line 10-11 left: This passes the temporary array through the unrolled indices, and loads the elements.

Line 10-11 left: Corresponds to line 5 right. This binds the temporary array to the lambda function input-name.

```
1 mapM_ addStm $ bodyStms $
      lambdaBody lam
2 let lambda_res = bodyResult $
       lambdaBody lam
3 out_var <-
4   letwithNDim
5     (map paramName out_pars)
6     list_of_index
7     lambda_res
8 pure $ map Var out_var
```

Listing 12: First order transformer

```
1 ... -- invariant value was loaded into variable 'x'
2 let {y : f32} = xs[0i64]
3 let {x : f32} = fadd32(x, y)
4 let {y : f32} = xs[1i64]
5 let {x : f32} = fadd32(x_4572)
6 let {y : f32} = xs[2i64]
7 let {x : f32} = fadd32(x, y)
8 let {defunc_1_f_res : f32} = fdiv32(x, 3.0f32)
9 let {lw_dest : [M][N]f32} = stencil_out_x with [gid_y,
      gid_x] = defunc_1_f_res
10 in {lw_dest}
```

Listing 13: Result of first order transformation

Figure 11: The insert the lambda function and a write the output.

Comments for (figure 11):

Line 1 left: Corresponds to line 2-8 right. This is the inlined lambda function.

Line 2-8 left: Corresponds to 9-10 right. This updates the current index in the output with the output of the lambda function, and finished that iteration with this value.

4 Stencil problem

This section describes the stencil problem as sequential code and how to transform it into efficient parallel code.

4.1 Sequential stencil problem

Here we have the sequential code, that will compute the stencil according to the stencil representation in the previous section. Note that all values for a point. We chose to handle out of bounds accesses, by padding the input with edge values. Padding the input with edge values corresponds to bounding the indices read from the input with a function `bound`.

```
1 bound(idx,maxIdx) = min(max(0,idx),maxIdx)
2 divUp(x,y) = (x + (y-1))/y
```

For the function `bound`, `idx` is an index, and `maxIdx` is the highest possible index. The function `min(a,b)` returns the smallest value between `a` and `b`, and `max(a,b)` returns the largest value. The `computeStencil` function for two-dimensional stencils takes a three arguments. First argument is a two-dimensional `N` $\times$ `M` `input` array with elements of type `T`. The second argument is an array `inds` of `D` indices, where each index has two values for the first and second dimension, respectively. The third argument is a function `f` which computes the stencil for a given point with `D` values of type `T`.

```
1 computeStencil(T input[N,M], int64 inds[D,2], (T[D] -> T) f)
2   T output[N,M]
```

```
  T stencilValues[D]
  for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
      for (k = 0; k < D; k++)
        int64 x = bound(i+inds[k,0],N-1)
        int64 y = bound(j+inds[k,1],M-1)
        stencilValues[k] = input[x,y]
      out[i,j] = f(stencilValues)
```

The `computeStencil` function first declares an `output` array and a `stencilValues` array. The output array will have the same shape as the input array. The length of the `stencilValues` array depends on the number of indices `D` in the stencil. We load the values for a particular `i,j` into `stencilValues` before computing the stencil function `f`. As mentioned earlier, we must load the values before computing `f`, due to the design decision of the stencil representation in Futhark.

4.2 Transformations into a parallel version

This section concerns the above mentioned sequential version into a version that can meaningfully be run on in parallel.

Firstly some assumptions need to be stated

1. The lambda-function being run is pure. This means that it only depends on its input, and the the calls to it can freely be moved around as long as the input is the same.

2. The input(s) array (read-only) and output array (write-only) are distinct arrays, hence there are no dependencies between reads from the input and writes to the output.

These 2 assumptions mean that doing dependency vector analysis on the input and output is redundant, as there are no dependencies between them. The lambda-function is pure, and from this we have that the are no dependencies implicit to it.

Privatising variables Firstly we notice that the loops in the loop nest are not immediately parallel because of the `stencilValues` array. However this can be made local to the second loop in the nest. There are then no write-after-read dependencies between the iterations of the loop nest, and all 3 loops are therefore parallel.

```
computeStencil(T input[N,M], int64 inds[D,2], (T[D] -> T) f)
  T output[N,M];
  for (i = 0; i < N; i++) // parallel
    for (j = 0; j < M; j++) // parallel
      T stencilValues[D]
```

```
      for (k = 0; k < D; k++) // parallel
        int64 x = bound(i+inds[k,0],N-1);
        int64 y = bound(j+inds[k,1],M-1);
        stencilValues[k] = input[x,y];
      out[i,j] = f(stencilValues);
```

Stripmining Stripmining refers to the following transformation, where the size of a strip is any constant B, but usually a power of 2 and small (less than 1024):

```
for (i = 0; i < N; i++)
// into
for (ii = 0; ii < (divUp(N,B)); ii+=B)
  for(i=ii; i < min((ii + B),N); i++)
```

The loops need not be parallel for this to be valid.

In our example we will stripmined both of the outer loops with a stripsizes of 32 and 32 for each of the outer axis respectively, to simulate a 32x32 group. This is valid when the loop iterator is constant with regards to the loop body, which it is when we have a parallel loop.

```
computeStencil(T input[N,M], int64 inds[D,2], (T[D] -> T) f)
  T output[N,M];
  for (ii = 0; ii < (divUp(N,32)); ii+=32) // parallel
    for(i=ii; i < min((ii + 32),N); i++) // parallel
      for (jj = 0; jj < (divUp(M,32)); jj+=32) // parallel
        for(j=jj; j < min((jj + 32),M); j++) // parallel
          T stencilValues[D]
          for (k = 0; k < D; k++) // parallel
            int64 x = bound(i+inds[k,0],N-1);
            int64 y = bound(j+inds[k,1],M-1);
            stencilValues[k] = input[x,y];
          out[i,j] = f(stencilValues);
```

Loop interchange Loop interchange refers to swapping 2 loops:

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    // body
// into
for (j = 0; j < M; j++)
  for (i = 0; i < N; i++)
    // body
```

To determine if the transformation on these loops (which need not be parallel) is valid, one would usually use a theorem concerning dependency vector analysis. However a corollary of this theorem states that parallel loops can always be moved inwards in the loop nest, however we may need to array-expand some variables to avoid write-after-read dependencies.

In this example the loops are parallel and only the innermost loop has any dependencies on the variable (`stencilValues`), but will no be part of the transformation. The second and third loop will be interchanged.

```
computeStencil(T input[N,M], int64 inds[D,2], (T[D] -> T) f)
  T output[N,M];
  for (ii = 0; ii < (divUp(N,32)); ii+=32) // parallel
    for (jj = 0; jj < (divUp(M,32)); jj+=32) // parallel
      for(i=ii; i < min((ii + 32),N); i++) // parallel
        for(j=jj; j < min((jj + 32),M); j++) // parallel
          T stencilValues[D]
          for (k = 0; k < D; k++) // parallel
            int64 x = bound(i+inds[k,0],N-1);
            int64 y = bound(j+inds[k,1],M-1);
            stencilValues[k] = input[x,y];
          out[i,j] = f(stencilValues);
```

Renaming and moving conditions around These things are not rules, as they are valid when the variables are constant with regards to the function body. The modification are that we make `for(ii=i;...)` and `for(jj=j;...)` into `for(ii=0; ...) gidy = i*32 + ii` and `for(jj=0; ...) gidx = j*32 + jj`. These new variables `gidx` and `gidy` need not be constant to the loop body. These variables are move inwards in the loop nest. Additionally the handling for making sure we don't write out of bound is moved into the start of the loop body.

```
computeStencil(T input[N,M], int64 inds[D,2], (T[D] -> T) f)
  T output[N,M];
  for (by = 0; by < (divUp(N,32)); by+=32) // parallel
    for (bx = 0; bx < (divUp(M,32)); bx+=32) // parallel
      for(ty = 0; ty < 32; ty++) // parallel
        for(tx = 0; tx < 32; tx++) // parallel
          gidy = by*32 + ty;
          gidx = bx*32 + tx;
          if(gidy < N && gidx < M)
            T stencilValues[D]
            for (k = 0; k < D; k++) // parallel
              int64 x = bound(gidx+inds[k,0],N-1);
              int64 y = bound(gidy+inds[k,1],M-1);
              stencilValues[k] = input[x,y];
            out[gidy,gidx] = f(stencilValues);
```

This corresponds to how a naive solution to the stencil problem could be run in parallel if done in group of size 32x32. The variables `by` and `bx` would then be the group ids, the group size is then 32 by 32, and the thread ids are the variables `ty` and `tx`.

Spacial Locality / Coalescing When creating programs that need to process very large volumes of data, it is important to consider how to best access

the main memory. When doing this on a CPU or GPU there are 2 general rules of thumb, which are *Spatial Locality*, which refers to accessing adjacent or nearby addresses in memory, and *Temporal locality*, which refers to accessing addresses that have previously been accessed not long ago. The reason for these rules of thumb is based on how caches usually work, where they load chunks from main memory (meaning adjacent elements are already loaded), and if a address is re-accessed within a short time-span, then there is a good chance that this chunk has not been evicted from the cache.

For GPUs there is an restriction on the rule regarding *Spatial Locality*. This variant of *Spatial Locality* is called *Coalescing*. The restriction is that within a thread-group, memory accesses should be ordered such that threads with adjacent IDs should access adjacent memory addresses. Even if multiple reads are needed, this rule need still be followed. An example of this, where multiple accesses are done (here 4 per thread), is the following.

```
// assume a thread-group size 'B'
// where threadID is in the range 0 to (but not including) B.
for(int i=0; i<4; i++){
  long idx = i*B + threadID;
  outputArray[idx] = 2 + inputArray[idx];
}
```

The Coalescing rule is due to the memory abstraction model of GPUs, where memory loads are handled as transactions of chunks of data from/to memory, made by small groups of threads. As an example, on NVidia GPUs that have compute-capability version 3.0 (this is old, but is still usable for the example), these transaction are handle per Warp basis (32 adjacent threads), meaning that a Warp of threads will ideally (if addresses matches the transaction-size alignment and the size of an element is 4 bytes or less) only spawn 1 memory transaction for the 32 access made by a Warp of threads ([5]). If the addresses are unaligned, spread out, or the size of each element is larger then 4 bytes, then several transaction will be made to fetch the desired data, which puts strain on the memory system.

Temporal locality / Shared memory The problem domain of this paper is stencils, in which each element of the input array are read multiple times. For this reason it is possible to optimize for temporal locality, fx. by storing a subset of the input array in a small intermediate array, and then directing future reads to that small array. This makes sense on GPUs as they have (what in CUDA terminology is called) Shared memory (in OpenCL terminology this is called Local memory). The amount of shared memory available per thread-group is very limited (fx. a upper limit of 48KiB per thread-group although using less is usually preferable), but has much faster access time than main

[5] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-3-0

memory (although not as fast as registers). This memory has the same lifetime as a thread-group. This memory is shared by the threads of the thread-group, which means that if multiple threads need to access a single element from the input array, then it need only be loaded once (per thread-group) from main memory into the shared memory, and then the rest of the threads of the thread-group can read it from there (after synchronizing the threads such that no race condition occur).

Using this knowledge one can create a variant of the stencil which uses shared memory to create a 2d array of constant size for each group and reads from this. If we for example have a 5 point 2d-stencil containing the relative points $[(-1,0),(0,-1),(0,0),(0,1),(1,0)]$ for the (y,x) axis, then a stencil evaluation using shared memory could look like this:

```
1  // assume there is a compile time constant array: int64 inds[D
       ,2]
2  // containing the (y,x) relative indices of the stencil.
3  computeStencil(T input[N,M], (T[D] -> T) f)
4    T output[N,M];
5    for (by = 0; by < (divUp(N,32)); by+=32) // parallel
6      for (bx = 0; bx < (divUp(M,32)); bx+=32) // parallel
7        T shared_array[34,34];
8        for(ty = 0; ty < 32; ty++) // parallel
9          for(tx = 0; tx < 32; tx++) // parallel
10           gidy = by*32 + ty;
11           gidx = bx*32 + tx;
12           for(int cy=0; cy < 34; cy+=32)
13             for(int cx=0; cx < 34; cx+=32)
14               int64 y = bound(gidy-1+cy,N-1);
15               int64 x = bound(gidx-1+cx,M-1);
16               shared_array[cy+ty,cx+tx] = input[y,x];
17       // synchronize the threads of the thread-group
18       for(ty = 0; ty < 32; ty++) // parallel
19         for(tx = 0; tx < 32; tx++) // parallel
20           gidy = by*32 + ty;
21           gidx = bx*32 + tx;
22           if(gidy < N && gidx < M)
23             T stencilValues[D]
24             // unroll the loop
25             for (k = 0; k < D; k++)
26               stencilValues[k] = shared_array[ty+inds[k,0], tx+
       inds[k,1];
27             out[gidy,gidx] = f(stencilValues);
```

This example does 34x34=1156 read from global memory to satisfy 32x32 writes, as opposed to the version not using shared memory which does 5x32x32=5120 read from global memory to satisfy the same amount of writes. This highlights the primary advantage of the shared memory version. However the shared-memory version does have more overhead compared to the naive version, as well as noting the 5x32x32=5120 reads are directed to shared memory, and

while this memory is faster than global memory is still not free of cost.

This variant shown above is similar to the prototype we made called "2d-bigTile-inlinedIndexes-squareLoader-multiDim", where the name is supposed to indicate that it is 2D, has (what will end up as) inlined stencil indices, has multi-dimensional iteration space for the grid of group, loads in elements into shared memory in chunks of the same size and shape as the thread-group, and the shared-memory array has a base size of thread-group-size (here the base is 32x32) and is extended depending on the width of the stencil (here 3x3 which extends the shared memory array by 1 in each direction, meaning an increase of 2 and 2 on each axis).

The stencil evaluation strategy above does however have other problems, such as an unbalanced amount of work for the threads, but that will be dealt with in depth in the prototype section of this paper.

Multiple writes per thread One may have notice that if each thread in the thread-group had to process multiple element, each one in adjacent chunks of the grid, then we can load all the elements of these adjacent chunks into the shared memory array to begin with and afterwards do the function evaluation and writing to output. An advantage of this is that it will decrease the total number of elements that need to be loaded, due to elements being on the border of chunks in the grid sometimes only needing to be loaded once. There are however a number of drawback/limitations of this approach, first and foremost that the amount of shared memory available per group is rather small (both the actual hard-limit and the practical limit for if we want most of the threads to actually run).

Consider the example were we have 32x32 thread-group and we amplify the amount of work along each axis by 2, then each chunk in the grid is responsible for a 64x64 area. If we use a stencil which requires 9 points from the square formed by the -2 to 2 range along each axis, and use the same loading style, then we load 68x68=4624 elements from the input to write 64x64 elements to the output. If used the regular 32x32 work area then we would require 4 chunks of 34x34, that is 4x34x34=5184 loads from main memory. This reduces the number of reads required to satisfy a area of 64x64 down to (4624/5184=89%) and reduces the number of chunks by potentially a factor of 4 (we still have to consider chunks on the edges of the 2d input array, as they get scaled up to 64x64 even if this does not make sense). This does however decrease the degree of parallelism by a factor of 4, which can be a problem on tiny input as then we may have idle threads in that case, which with the standard shared-memory array version could have been running as it has a higher degree of parallelism.

Another reason for doing multiple writes per thread is that if the chunks processed are adjacent (especially if it is along the x-axis) then there is a good change that some of the elements already exist in the cache, meaning fast

loading times. This is however not easy to work with as the relevant chunks may already have been evicted from the cache depending on what the other groups on are doing, and there is no easy way to find this out as eviction-strategy is hardware dependent, as well as the size of the cache.

Percentage of working threads / Occupancy Given the previous 2 paragraph one may think that increasing the number of writes per thread is always advantageous. We do however need to consider that we are in fact running this on hardware with a cap on the number of threads available, shared memory available and number of registers available.

- If one increase the work area for each thread-group to a high amount, and the input array is small, then perhaps not all threads can be running as we have effectively sequentialized some work by increasing the work area per thread-group. In such cases it would be advantageous to decrease the degree of sequentialization (multi-write) to increase the parallelism.
- If the requested amount of registers and shared memory is too high for the hardware to run all possible threads of a time, then it will run the program with a decreased amount of thread-group compared to what the potential upper limit is (or in rare cases fail to run anything). If this happens then the performance will generally be worse than a slightly less sequentialized version (under the assumption that increased sequentialization increase register and shared memory usage).

To learn more about this on the hardware that we are running on, and how this has influenced our designs, look to section (5.5) and section (9.1.2)

5 Prototyping

This section presents the prototype designs that were made and the configurations of these. The code for these prototypes are in a github repository of ours [6].

Assumptions:

1. Throughout this prototyping we assumed the neighbourhood indices were know at compile time, and we choose some simple patterns of indices highlighting some useful information about how the algorithms scaled on different input.

2. The initial designs had a restriction that for each dimension d of the stencil indices I, the property $minIdx(I, d) \leq 0 \leq maxIdx(I, d)$ must hold. This assumption is not present in the later designs.

The first of the assumption regarding compile time know constants for the stencil indices was chosen as many of our designs rely on the information the is obtained from that, and also for choosing which design to run.

The second assumption was initially made for simplicity rather than any design constraints. The assumption is sane as all stencils we used from the Rodinia benchmark suite, had stencils which had this property. The amount of optimization made available from making this assumption is however non-existent, so there was no reason to introduce it for any other reason than simplicity of design.

Design choices imposed by the design of the stencil function by the Futhark compiler Initially the prototyping did not take into concern what would be possible of practical to do inside the Futhark compiler, and some design were initially made (and later dropped although for other reasons) which could not be implemented in the compiler given certain restrictions (initially that "Constant" memory was not available).

However a more noticeable design restriction, when running with stencils having many points, is that the lambda function takes an array of values for each thread, meaning that an intermediate array need to be made in each thread to store the values. This restriction does make the stencil construct more expressive as we don't need to provide a function that folds over the input, but it does make it necessary to store a small array in each thread, which can take many registers.

[6] https://github.com/Quartzinin/Stencil-prototyping

General prototype benchmark configuration These are the configurations for all prototyping benchmarks, unless the individual benchmark explicitly states it to be different.

All of the prototypes mentioned below were made in Cuda as it is a GPGPU language/runtime. One would expect that implementations in OpenCL would have the same degree of scaling and comparable performance as they work on the same abstraction of hardware, and that we are limiting ourselves to functionality that is available in both.

The benchmarks are made on dense stencils, where we take the mean of these points. Some benchmarks are made on a uniformly weighted Jacobi-function, but those that do this explicitly state it to be so.

The default datatype of the input are 32-bit floating point numbers.

The prototype benchmarks are not iterative stencils, as opposed to most of the benchmarks shown in section 9.2, which are iterative stencils, and are used to benchmark the GPU-code generation of stencils in the Futhark compiler.

Dimension	input size
1D	[row length]=$[2^{24}+2]$
2D	[column length, row length]=$[2^{12}+4,\ 2^{12}+2]$
3D	[height length, column length,row length]=$[2^{8}+8,\ 2^{8}+4,\ 2^{8}+2]$

Table 1: Prototype benchmarks sizes

Prototyping Goals The prototyping goal for the paper concerns finding and benchmarking different approaches for efficiently running stencils in a GPGPU setting.

5.1 The reference/base method

The reference method is what one would get if one naively makes some nested maps over the index range of the array input. These indices are then used in combination with the stencil indices to extract the desired neighbours from the input array in a array of values for each thread, and then lambda function is applied, after which a write is performed. This method is later referred to as *Global read* with *Inlined indices* and *SingleDim grid* and *Lengths spans*, as it naively reads from global memory using the inlined indices. The latter parts of the name refers to how multidimensional grids are layed out, and how the flat index is unflattened. This odd detail is due to how multi-dimensional input is handled in the Futhark compiler.

Overall structure of versions using shared memory While we made multiple designs with tiles (big-tile, small-tile, multi-write big-tile, sliding tile), they all had some parts in common.

Consider the case of 2D stencils. These approaches divides the input array into 2D chunks. The read-tile is the chunk of values read from the input array. The values are cooperatively read by threads of the thread-group assigned to the chunk. The write-tile is the chunk of values of the output array that will be written to by some group. As the write-tile are writes to the output array, it is implied that they are the result of the lambda function being run for that index, and that the requirements of the lambda function has been supplied.

The read-tile must contain all the requirements for evaluating the stencil function for the write-tile indices. Ideally the read-tile is no larger than the minimum requirements for the write-tile as that would mean unused reads from global memory.

All of the tiled version have in common that given a write-tile, the read-tile is then the smallest rectangle (in 2D) that contain all of the dependencies for the write-tile. This approach is good if the stencils are dense or small, as the read-tile is an easy pattern, and the number of unused reads is small. In the case of wide stencils with few points in higher dimensions, then this approach is poor as there are many unused reads.

Reuse factor The primary motivation for making tiled version is that one can reuse loads from global memory within a thread-group by storing the in shared memory. This factor is however not the only important part of the algorithm designs, and should be weighted in comparisons with the other factors for performance.

The reuse factor is a fraction of how many reads from global memory the naive *Global read* method does per 1 read of the tiled versions. This obviously does not directly take the number of memory transactions into account, as the reads are unaligned (and therefore a clean closed form expression is not easy to derive), and the *Global read* is mostly reliant on the caches (which is not easily mapped into a formula).

Assume we have a P-point stencil. Then the mean use/reuse per thread-group of an element loaded from main memory is:

$$\text{mean reuse per thread-group} = \frac{P \cdot |\text{write-tile-flat}|}{|\text{read-tile-flat}|}$$

This does obviously not take out of bound elements into account, but the actual reuse would approach this as the input size increases.

We have the following variables, using vector notation to describe multiple dimensions: Given a 3D stencil with The read-tile and write-tile per thread-group for the different designs have the following sizes:

Variables	Type	Description
D	$\mathbb{N}$	Dimensionality of the input
P	$\mathbb{N}$	Number of points of the P-point stencil
I	$\mathbb{Z}^{D \times P}$	The points of the stencil e.g. $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \end{bmatrix}$ represents the points $[(-1,-2),(0,0),(1,2)]$
$I_{i,j}$	$\mathbb{Z}^P$	The j'th element of the i'th row of I
G	$\mathbb{N}^D$	Shape of the multi-dimensional group $(G_1, \ldots, G_D)$ e.g. with $D = 3$ outermost dimensions comes first as $[2, 4, 32]$
W	$\mathbb{N}^D$	The number of writes on each dimension $(W_1, \ldots, W_D)$. e.g with $D = 3$ we could have $[2, 2, 1]$
G_{flat}	$\mathbb{N}$	The group size denoted by $\prod_{d=1}^{D} G_d$
axisMins	$\mathbb{Z}^D$	The minimum value in each row $[\min_n(I_{1,n}), \ldots, \min_n(I_{D,n})]$.
axisMaxs	$\mathbb{Z}^D$	The maximum value in each row $(\max_n(I_{1,n}), \ldots, \max_n(I_{D,n}))$.
haloWidths_i	$\mathbb{Z}$	axisMaxs_i - axisMins_i
haloWidths	$\mathbb{Z}^D$	[haloWidths_1, …, haloWidths_D]
write tile	$\mathbb{N}^D$	The tile shape of the output array that we write to, for some group
read tile	$\mathbb{N}^D$	The tile shape of the input array that we read from, for some group
windowIters	$\mathbb{N}$	Number of iterations for the sliding window

Table 2: Variables used in the formulas. We use matrix and vector notation for these variables

5.2 Basic designs

5.2.1 Basic versions that were discarded:

All early version had in common, that they would load the indices into (Cuda-)constant memory and the use them from there. This differs from the later version were they are inlined (at compile time) in the code, and that the loops over these indices are unrolled.

Temporary array version: This version was created as a reference point of the most naive implementation of a stencil. It writes all possible views of the input array based on the index array to a temporary global array (of length $N^{flat} \cdot P$), where each thread then reads its own view. This is wasteful with regards to memory as no intermediate array is technically needed, and indexing into the temporary array will be strided. In conclusion it is naive and very slow.

Design name	Tile	Shape :: $\mathbb{N}^D$ (unless otherwise specified)
Global-Read-1D	read	$\mathrm{P} \cdot G_{flat}$
	write	G_{flat}
Small-Tile	read	G
	write	$[G_1 - \mathrm{haloWidths}_1, \ldots, G_D - \mathrm{haloWidths}_D]$
Big-Tile	read	$[G_1 + \mathrm{haloWidths}_1, \ldots, G_D + \mathrm{haloWidths}_D]$
	write	G
Multi-Write Big-Tile	read	$[G_1 \cdot W_1 - \mathrm{haloWidths}_1, \ldots, G_D \cdot W_D - \mathrm{haloWidths}_D]$
	write	$[G_1 \cdot W_1, \ldots, G_D \cdot W_D]$
Sliding-Small-Tile-2D	read	$[G_1 \cdot \mathrm{windowIters} + haloWidths_1,\ G_2]$
	write	$[G_1 \cdot \mathrm{windowIters}, G_2 - \mathrm{haloWidths}_2]$

Table 3: The size of the per-group read-tiles and write-tiles for the different designs. Global-read is annotated in 1D but has the same flat size in all dimensions. Sliding-Small-Tile-2D only exists in 2D, and the size is therefore only shown for the case.

Mixed global-read and small tile This version mixes the small tile version, where each thread load their own index into a shared tile (and where other threads use this if relevant to their own requirements), but where indices outside of the tile are loaded as needed. This version does have a poor degree of reuse for wide stencils, and the access patterns are poor for the indices outside of the tile for higher dimensional stencils.

5.2.2 Global read

This version simply reads and immediately consumes the values it fetches from its redirected view into the input array. This approach therefore reads $N^{flat} \cdot P$ elements from global memory, however all of these reads are coalesced so they are expected to be fast, even if we do a lot of them.

This variant has low overhead for each group, but has no sharing of loaded values. This means that it is competitive against the other variant of evaluation, but only for very short stencils, as the complete lack of reuse makes it scale poorly on anything but the smallest stencils. The not using shared memory can however be beneficial in the case where the tiled version would run out of shared memory, because for example of a very wide stencil, in which case the Global read would still run.

5.2.3 Small tile

This version has the read-tile being set to the shape of the thread-group (line/square/cube), and the write-tile is then a sub-shape containing the indices that have their requirements fulfilled. A graphical illustration can be

seen in figure (12).

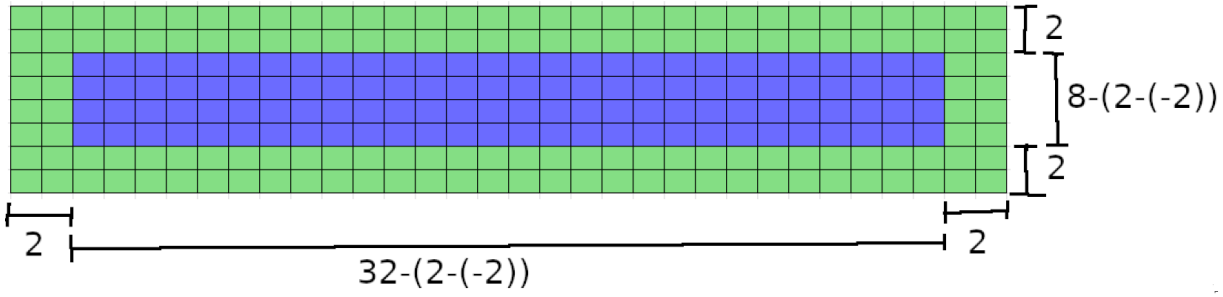


Figure 12: Illustration of the Small-tile design, with how the per-thread-group read-tile (green and blue) and write-tile (blue) is laid out with 256 threads per thread-group yielding a 8x32 groupsize, when the stencil has axisMins=[-2,-2] and axisMaxs=[2,2].

This variant only needs 1 load from global memory per thread per thread-group, leading to a medium amount of overhead per thread-group. The consuming threads within a group only read from shared memory so they run fast. The downsides of this version are however twofold. Firstly we need to spawn additional groups as the size of the stencil increases, as the write-tile become smaller and smaller than the thread-group size, and this number of additional groups increase exponentially the closer the width of the stencil is to the thread-group size. Secondly each thread-group will have non-running threads after the loading phase as there are threads that do not have their requirements fulfilled, which is damaging to the performance as there is only so many groups that can be active at a time irrelevant of how many threads inside them are running.

For this reason performance of this version is very reliant on the width of the stencil, and will perform well on narrow stencils, but poorly on wide stencil. One the off-change that stencil fusion will one day be implemented, this base version of small-tile is unsuitable for handling this, as fusion will produce a wider stencil for each fusion, meaning a smaller and smaller write-tile.

It should be said that the read-tile may be larger than the requirements of the write-tile, such as with cases of diagonal neighbours. In this case there are unused reads in the read-tile.

5.2.4 Big tile

This version has the write-tile being to the shape of the thread-group (line/square/cube), and the read-tile is then a super-shape (line/square/cube) containing the indices/values such that have their requirements fulfilled. A graphical illustration can be seen in figure (13).

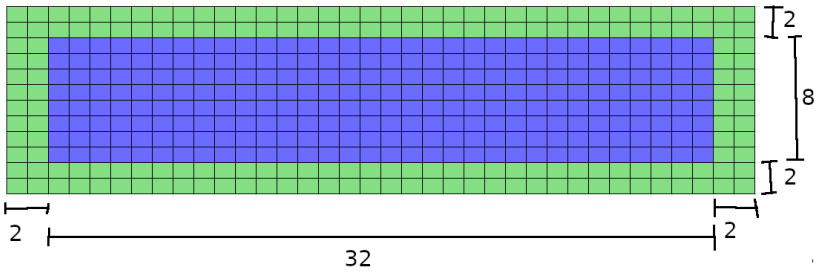


Figure 13: Illustration of the Big-tile design, with how the per-thread-group read-tile (green and blue) and write-tile (blue) is laid out with 256 threads per thread-group yielding a 8x32 groupsize, when the stencil has axisMins=[-2,-2] and axisMaxs=[2,2].

This variant requires (for non-trivial stencils) more than 1 read from global memory per thread per group. The synchronisation of threads need however only before any thread reads from the tile, so no additional synchronisation is required (compared to small-tile). The number of thread-groups needed to cover the full input array is therefore the same as with global-read

It should be said that the read-tile may be larger than the requirements of the write-tile, such as with cases of diagonal neighbours. In this case there are unused reads in the read-tile.

Weaknesses of the initial big tile implementation There are a number of weaknesses to the design, although some of them can be justified with it being a simpler pattern:

1. The design does potentially load too much as it loads the smallest square containing all the indices. There may however be elements in the corners of the square which are not used depending on the stencil, fx. the index list [(-1,0),(0,-1),(0,0),(0,1),(0,1)] where the corners of the loaded square would not be used (and this is even worse in 3D). This reduces the reuse factor, but is accounted for in the formula.

2. The design only works on limited sizes of tiles, as the is a hardware cap on how much memory there can be allocated in shared memory per thread-group. On a some GPUs the upper limit is 48KB, so thread-group-size of (z,y,z)=(4,8,32) with the stencil [(-6...6, -6...6, -6...6)] would load in a tile of $(4+12)\cdot(8+12)\cdot(32+12) = 14080$ elements. For a element of float32, this would then require $4 \cdot 14080 = 56320$ Bytes, which would exceed the limit of 48KB.

5.3 Improving basic designs

The initial designs were used as a guideline for which direction we should take our later designs. The designs *Global Read* and *Big Tile* were taken further while the *Small Tile* was dropped (although some parts were reused).

5.3.1 Common parts between multiple designs

The indices The designs have in common that they would have their indices inlined (by using a simple pattern that can be evaluated at compile time when the constant length loops are unrolled).

The lambda function All designs were benchmarked with a lambda function which took the mean of the neighbors which were in the stencil indices. This is because it is a computationally lightweight function that still does some meaningful work, which makes the most amount of sense when testing memory loading designs.

Handling of tuples A thing that should be noted is that the prototyping has no cases of arrays of tuples (which would be handled as a tuple of arrays). For the tiled design this would be handled for k-length tuples requiring each a n-length tile, by it would be split this up into k tiles of n-length, where each tile handles one array of the tuple.

The preferable loading tactic for tuples is however not tested. One can use a loader strategy to traverse the read-tile space once, and for each index of the read-tile load the corresponding element from all k inputs arrays before continuing onto the next index. This tactic means we only need to iterate the read-tile-iteration space once, but is bad for the caches as we require a lot more space in them for the same performance to be achieved. Alternatively one can iterate over the read-tile-iteration space k times (once for each array in the tuple), and load the array elements one at a time. The tactic would increase the temporal factor for the caches, but would need to iterate the read-tile k times.

Virtualization Virtualization is a technique where each spawned group in the grid will be tasked with sequentially running multiple independent groups from a virtual grid. The number of spawned groups should correspond to the number the groups which is run simultaneously by the hardware. The virtual grid should be the same grid as before.

This technique is usually used to avoid having allocate more memory than what the hardware can actually use at any time, which in turn result in a much smaller memory footprint. This is however not relevant to the stencil evaluation as the stencil kernel does not itself do any allocation to global

memory. It was however still attempted for multiple reasons. Firstly it was done as there is some potential reuse of computations (depending on where one places certain calculations). Another reason is that if the stencil construct in the future needs to support virtualization, as to be better integrated into the rest of the compiler, then it would be relevant to test what impact it would have on the performance. It should be noted that the virtualized kernels require slightly more variables than their counterparts, which will increase the required number of registers.

The difference in performance in this version compared to other common designs can be seen in figure (14). The take away from this is the the penalty of running with virutalization is not great.

Multi-Write / Stripmining We have implemented a version that does multiple writes per thread, we refer to this as *multi-write big tile* or maybe in a few places as *stripmined big tile*. We also refer to this version as stripmined, since it is formed by stripmining the group tiles of the big tile version into sequential loops. To make a high-level description of this version we only need to apply group-tiling of the two innermost loops below:

```
computeStencil(T input[N,M], int64 inds[D,2], (T[D] -> T) f)
  for (ii = 0; ii < (divUp(N,8*2)); ii+=8*2) // parallel
    for (jj = 0; jj < (divUp(M,32*2)); jj+=32*2) // parallel
      for(i=ii; i < min((ii + 8*2),N); i++) // parallel
        for(j=jj; j < min((jj + 32*2),M); j++) // parallel
          T stencilValues[D]
          for (k = 0; k < D; k++) // sequential
            int64 x = bound(i+inds[k,0],N-1);
            int64 y = bound(j+inds[k,1],M-1);
            stencilValues[k] = input[x,y];
          out[i,j] = f(stencilValues);
```

This results in the following transformation

```
computeStencil(T input[N,M], int64 inds[D,2], (T[D] -> T) f)
  T output[N,M];
  for (ii = 0; ii < (divUp(N,8*2)); ii+=8*2)
    for (jj = 0; jj < (divUp(M,32*2)); jj+=32*2)
      for(i=ii; i < min((ii + 8*2),N); i+=2)
        for(j=jj; j < min((jj + 32*2),M); j+=2)
          for(strip_y=i; strip_y < min((i + 2),N); strip_y++)
            for(strip_x=j; strip_x < min((j + 2),M); strip_x++)
              T stencilValues[D]
              for (k = 0; k < D; k++) // sequential
                int64 x = bound(strip_y+inds[k,0],N-1);
                int64 y = bound(strip_x+inds[k,1],M-1);
                stencilValues[k] = input[x,y];
              out[strip_y,strip_x] = f(stencilValues);
```

The algorithm above is easily translated into a multi-write version of the global read kernel. A potential improvement to our report would to implement multi-write global read versions in addition to the multi-write big-tile version. The idea is that the two innermost loops over `strip_x` and `strip_y` will be executed sequentially by a thread. However, our multi-write big-tile implementation also uses shared memory. The algorithm loads from global memory into shared memory using the flat loader (add/carry) approach mentioned in section 5.3.4. In this example, we have a 2D stencil that is stripmined with tile size 2 on both loops parallel loops. Therefore we multiply the group size $(8, 32)$ by 2 on both dimensions, such that the total number of parallel iterations are reduced by a factor 2 in both loops. Alternatively, one could also say that four output elements per thread are computed, rather than one. We will further evaluate the efficiency with respect to runtime of this approach in section 9.

Lambda function representation For the prototyping the 'lambda function' was simply a inlined function that took the mean of an array that it was given as input. This could be unrolled as the length of the array was know at compile time, however it relied on the compiler to do this.

5.3.2 Variants of the Global read design

The global read does not use any shared memory, but the indexing pattern is coalesced with regards to a warp.

The different variants with a name with 'span' in them are different ways of partitioning the work into a grid.

The performance of these models is reliant on the cache performing well instead of taking manual control using shared memory. The models may therefore not scale well when the cache starts becoming full as it will need to evict chunks that may still require additional reads.

Lengths span This variant is similar to simply doing one flat map over the indices of the flat length of the input and assigning flat thread-groups to this grid. A thread-group will in this case often need perform write on 2 adjacent rows (or more in the case that the x-axis is tiny). An example of this is if the input is of length 384x384 then the first 2 groups would be assigned the write-tile of the ranges $[0, 0..256]$ and $([0, 256..384] + [1, 0..128])$ respectively when the thread-group-size is of size 256.

The advantages of this method is firstly that it could be written as a map and would therefore not require any special constructs. Secondly the stencil would not access many different rows (this depends on the width of the outer dimensions of the stencil), and access these rows multiple times, meaning it will have good cache behaviour for small stencils. Thirdly, all thread-groups except the last one in the grid will have all threads running all the time.

Grid span This variant uses a multidimensional thread-group matching the dimension of the stencil, an uses this to write/read in a rectangle (when in 2d) shape from the input to output.

The advantage of this method is that there is better reuse between the rows, although this is handled by the cache and is therefore unreliable.

Multi-write A multi-write version was attempted in 1D and 2D of the lens-span variant, where each of the thread-groups was responsible for handling 4 successive groups (of the single-write variant). This was attempted to increase the reuse from the caches, as successive groups would likely already have been loaded and reside the cache.

5.3.3 Variants of the big tile design

Grid dimensions The reason for multiple variant of this, is that CUDA has some arbitrarily limitation on grid sizes. These variants concerns the grid used when launching the CUDA kernel:
`<<< grid, group_size >>>someKernel(...)`
There are 2 design variants:

MultiDim When working on multidimensional input (at max 3 dimensions), the most intuitive approach to partition the input into groups is to create a multidimensional grid. Inside the kernel, a thread will then use group index for each axis to find out were it is in the grid. This variant does however have a practical issue, which is that in CUDA the maximum size of the y- and z-dimensions of the grid are $2^{16} - 1 = 65535$, while the x-dimension can have at most $2^{31} - 1$ groups. The limit of $2^{16} - 1$ can be problematic for multidimensional arrays skewed heavily towards either the z- or y-dimension.

SingleDim A more robust method (and this is what is used in general in the Futhark compiler), when considering this limitation in CUDA, is to flatten then grid, launch that, and unflatten the group-index inside the kernel. Example of conversion for 2d:

```
__global__ void someKernel(...){
    int group_id_y = blockIdx.y;
    int group_id_x = blockIdx.x;
    ...
}
...
    dim3 grid(x_length, y_length, 1);
    <<< grid, group_size >>>someKernel(...);
...
```

becomes

```
__global__ void someKernel(..., int2 grid){
    int group_id_flat = blockIdx.x;
    int group_id_y = group_id_flat / grid.x;
    int group_id_x = group_id_flat % grid.x;
    ...
}
...
    int group_size_flat = group_size.x * group_size.y;
    int2 grid = { x_length, y_length };
    int grid_flat = grid.x * grid.y;
    <<< grid_flat, group_size_flat >>>someKernel(..., grid
   );
...
```

The transformation does however add overhead in the form of introduced division and remainder operations for unflattening the index. This is only really matters for the group-index, and not so much for the thread index as the tread index is unflattened using the group_sizes, which are picked to be powers of 2, hence can be replaced with bit-wise operations. The higher the dimension of the iteration space, the higher the overhead from unflattening.

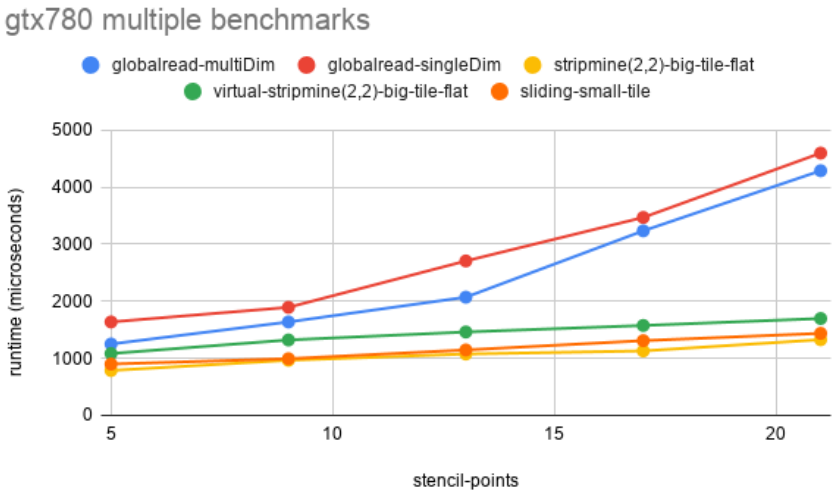
For the purpose of robustness we did choose to go with the SingleDim version although some speedups can be achieved if one want to go with the MultiDim version as can be seen in figure (14).

5.3.4 Tile Loaders

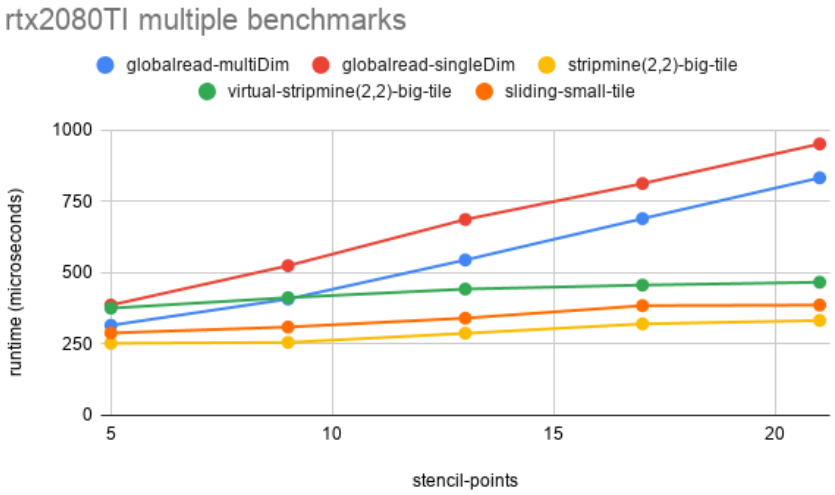
The big tile strategy requires the tile of shared memory to be loaded somehow.

For the purpose of simplicity the descriptions here will not cover handling of tuples in arrays. The tile is a 1D array (irrelevant of the dimensions of the stencil) and multidimensional indices will therefore need to be flattened for both writes and reads.

Cube loader This variant loads the data as a cube in 3d (and as a rectangle in 2d, and a line in 1d) of the same dimensions as the thread-group. This then needs to create a grid over the tile, and iterate over this in order for it to load the full tile. A illustration showing this loader strategy can be seen in 15. This illustration (of a 2d case) should highlight that while all of the reads are coalesced within a warp, there are a lot of threads doing nothing for several iterations, and that the threads in one area of the group (upper-left) will need to do several loads, while in the other end (bottom-right) they need only do 1 load. This leads to an unbalanced amount of work throughout the thread-group which is of importance as there is a synchronisation-point after the loader has run.



(a) GTX780



(b) RTX2080TI

Figure 14: Benchmarks of multiple designs in 2D on a [4100][4098] sized input with Jacobi programs of several sizes [5,9,13,17,21] points for them respectively. The 'multi-write' variants have the same work-multipliers.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	32	33	34	35
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	64	65	66	67
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	96	97	98	99
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	128	129	130	131
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	160	161	162	163
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	192	193	194	195
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	224	225	226	227
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	0	1	2	3
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	32	33	34	35
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	64	65	66	67
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	96	97	98	99

Figure 15: Illustration of how the cube/square loader partitions the work among the threads of the thread-group. The illustration is where there are 256 threads per thread-group. The numbers in the tiles represent which thread-id is responsible for writing to that index in the tile. The colours are to differentiate the different iterations. The read-tile iterated over is of size 12x36, which corresponds to a Big-Tile read-tile when groupsize=[8,32] and the stencil has axisMins=[-2,-2] and axisMaxs=[2,2]

The performance difference between this loader an the flat-loader (add/-carry) was measured for the gtx950, on a 3D stencil (Jacobi-like) of increasing width. This can be seen in figure (16). A 3D stencil is the most ideal case for the cube-loader as the outer axis's of the loader will be small and have less wasted threads on average, compared to a 1D version where most of the threads in the last iteration would be doing nothing. From the benchmark we can see that only on the largest of the examples will it be competitive.

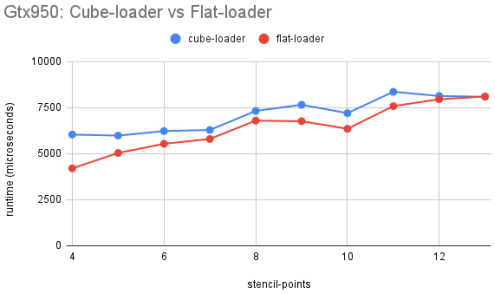


Figure 16: Benchmark comparing the cube-loader vs the flat-loader (add/-carry) for a Jacobi-like stencil in shape, but is increased in size along the z- then y- then x-axis and the repeat for each time we increase the size.

Flat loader (div/rem) The flat-Loader variants iterates over the flat indices of the tile, and then unflatten them to find the local indices, and then using a offset for each dimension will find the index into the input array. This approach uses a flat length of the tile ($\frac{\text{flat-tile-length}}{\text{group-size-flat}}$), and iterates over this to load the tile. An illustration of this variant can be seen in figure (17).

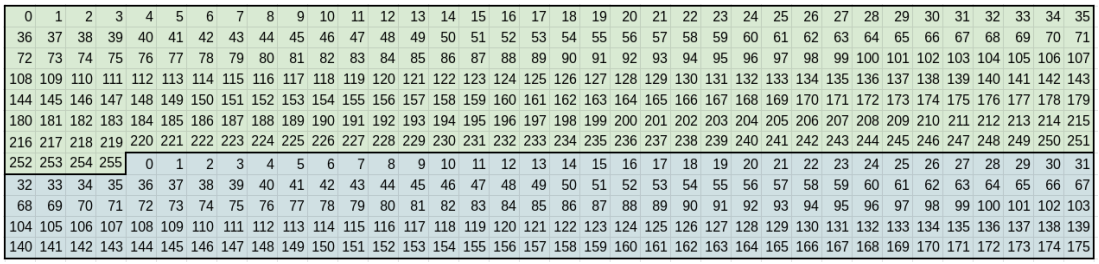


Figure 17: Illustration of how the flat loader partitions the work among the threads of the thread-group. The illustration is where there are 256 threads per thread-group. The numbers in the tiles represent which thread-id is responsible for writing to that index in the tile. The colours are to differentiate the different iterations. The read-tile iterated over is of size 12x36, which corresponds to a Big-Tile read-tile when groupsize=[8,32] and the stencil has axisMins=[-2,-2] and axisMaxs=[2,2]

The advantage of the version is that the work-load is equally distributed among the threads (except for the last iteration on the tile). A suboptimal advantage is that for many iterations of the warps the loaded segment is split into 2 internally coalesced pieces. This reason for this is that if the front end of a warp is filling in one row of the tile and the tail end is filling in the next row, then the corresponding segments in the input array are (in most cases) nowhere near each other. This leads to 2 internally coalesced segments of the input array being read.

The (div/rem) part of the name comes from how it unflattens its indices:

```
template<int groupDimFlat, int tile_size_x, int tile_size_flat>
void flat_loader_div_rem(int groupIdxFlat, ...):
int loaderGridSize = divUp(tile_size_flat, groupDimFlat);
int tile_span_y = tile_size_x;
for(int i=0; i < loaderGridSize; i++){
    int local_flat = (i*groupDimFlat) + groupIdxFlat;
    int local_y = local_flat / tile_span_y;
    int local_x = local_flat % tile_span_y;
    // insert the part of the loader that uses the local
   indices.
}
```

Note that for a k-dimensional unflatten we would require $(k-1)$ division/remainder pairs. So the div/rem version has $(k-1)\cdot$loaderGridSize division/remainder pairs.

A different approach which requires less division/remainder pair is however possible.

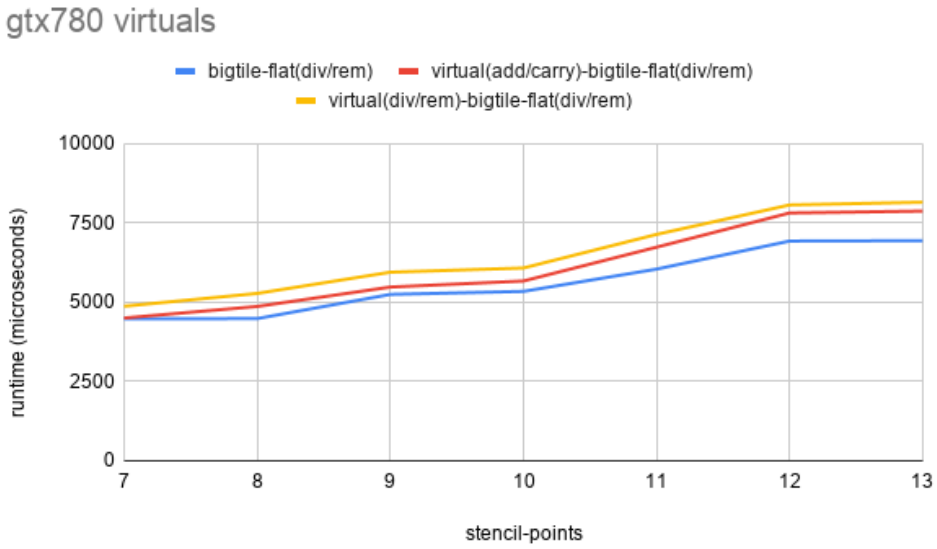
Flat loader (add/carry) This version is similar to the version above, except it unflattens the indices in a different fashion. This version has $(k-1)\cdot 2$

division/remainder pairs, and $(k+1)$ additions, and $(k-1)\cdot 2$ conditional additions.

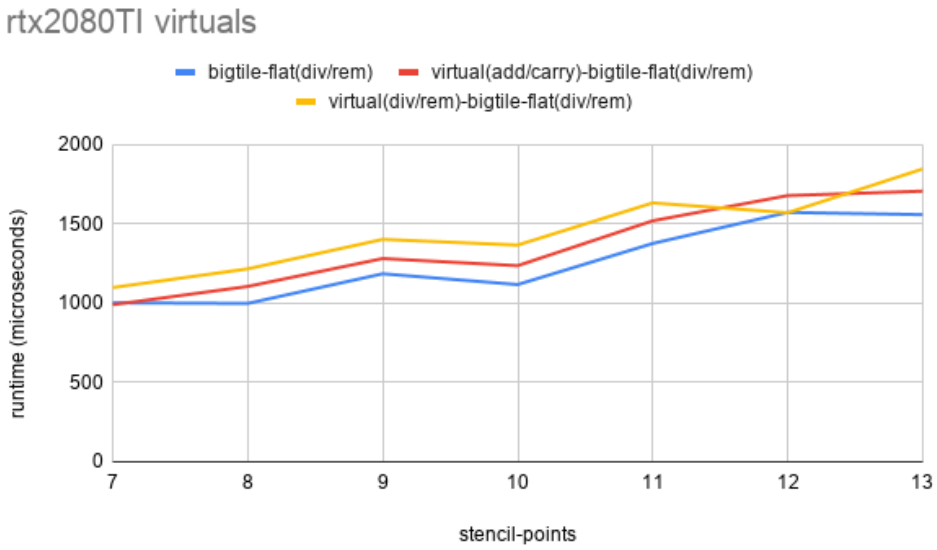
It is referred to as `(add/carry)` is shown for 2d below. The variant unflattens the starting index and the increase after each iteration, in order for it to turn the loop increase into additions and carry. This is done as to avoid having to do any more of the expensive instructions `(division/remainder)` than necessary. It does however require more variables and likely more registers. Furthermore this trades 2 unflattens (plus a bit extra work) for the `(div/rem)`s `[loaderGridSize]` unflattens, which is disadvantages when loaderGridSize ≤ 2, but advantages when strictly larger that 2.

```
template<int groupDimFlat, int tile_size_x, int tile_size_flat>
void flat_loader_div_rem(int groupIdxFlat, ...):
int loaderGridSize = divUp(tile_size_flat, groupDimFlat);
int tile_span_y = tile_size_x; // must be >= 1
int local_flat = groupIdxFlat;
int local_y = local_flat / tile_span_y;
int local_x = local_flat % tile_span_y;

int added_flat = groupDimFlat; // must be >= 1
int added_y = added_flat / tile_span_y;
int added_x = added_flat % tile_span_y;

for(int i=0; i < loaderGridSize; i++){
    // insert the part of the loader that uses the local
   indices.

    // add index increase
    local_flat += added_flat;
    local_y += added_y;
    local_x += added_x;
    // handle carries between axis
    if(local_x >= tile_size_x){
        local_x -= tile_size_x;
        local_y += 1;
    }
}
```

The difference performance between these 2 sub-variants can be seen in figure (18). This shows that the overhead off virtual-(add/carry) is lower than that of virtual-(div/rem). It should however be said that this outer loops run a lot more than just 2 or 3 iterations, so the (add/carry) should always win.



(a) GTX780



(b) RTX2080TI

Figure 18: Benchmarks of the 2 different virtualisation designs and the corresponding non-virtual design. The only difference between the 2 virtual designs is how the outermost loop is handled.

5.4 Sliding Tile versions

2 similar sliding-Tile versions was attempted in 2D, both a flat-groupsize (1 x groupsizeFlat) and a 2D-shaped group with the same shape as groupsize. The designs took inspiration from the Small-tile design in that the read-tile has a constant width, but is a sliding-tile on the y-axis. The size is like the Big-tile design for the y-axis, in that it is wide enough to supply the write-tile of the window. Aside from an initial number of loads, it will, for each iteration of the window, load groupsize elements and then do groupsize writes, and then move the window along the y-axis and repeat for a certain constant number of iterations. For a graphical illustration of this see figure (19).

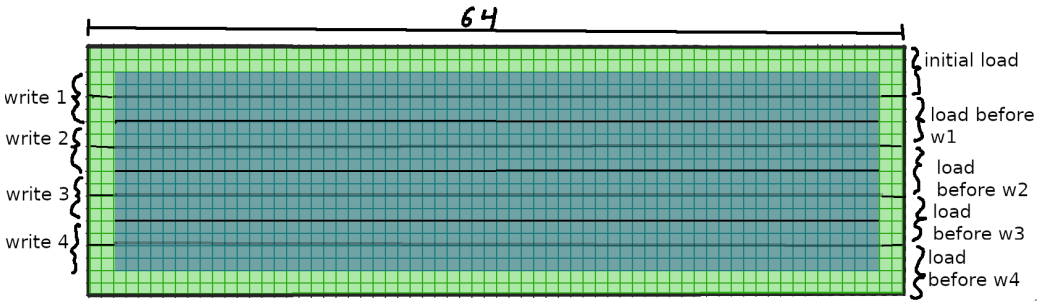


Figure 19: Illustration of the Sliding-tile design, with how the per-thread-group read-tile (green and blue) and write-tile (blue) is laid out with 256 threads per thread-group, with a 4x64 groupsize, windowIters=4, where the stencil has axisMins=[-2,-2] and axisMaxs=[2,2]

When it performs loads of the read-tile, it wraps around the tile such that fresh loads replace the values that are not longer needed. This does however mean that it requires many uses of the arithmetically heavy function `remainder` to do the wrap around the y-axis of the tile. One can often get away with less costly methods, such as using conditional updates, but the computations are still required in some form or another, which adds some constant arithmetic overhead compared to all the versions without any wrap-around. The sliding tile also has many more group-syncronization points, 2 per iteration of the window, as it needs 1 between the loads and read from the tile, and 1 to handle the cross-iteration dependency on the tile (so that we don't start writing to the tile before all threads are done reading from the tile). This latter part could be solved by increasing the size of the tile to have separate reading and writing sections per iteration.

The number of iterations for the window is a constant that requires tuning, as just increasing it as high as possible is not a good idea as can be seen in figure (20). The reuse increases as the y-size of the write-tile increases, but

The sliding-tile design was attempted for 2 reasons.

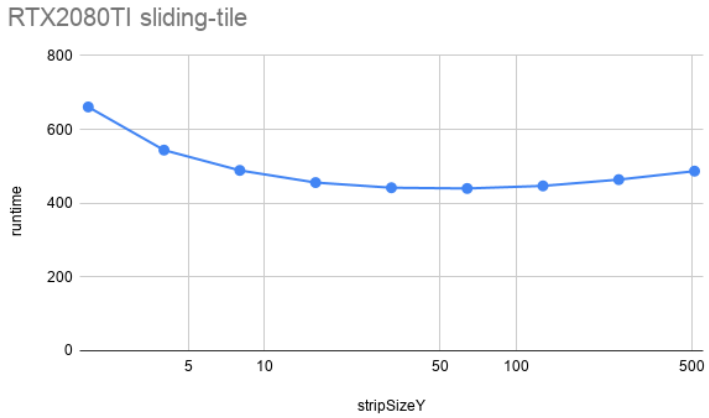


Figure 20: Runtime vs y-length of the write-tile.

1. It does in general uses less shared memory than the multi-write big-tile version to cover the same area. If we had a sliding-Big-Tile (just to make the calculations easier) then to cover a 16x64 area, we could be use a 4x64 groupsize and 4 window-iterations. Which with a stencil having axisMins=[-2,-2] and axisMaxs=[2,2], the tile/window would need to be of size (4+4)x(4+64)=544 elements, while a multi-write big-tile version would need to store (4+4*4)x(4+64)=1360 elements.

2. It can increase the reuse factor to a much degree higher degree than the multi-write big-tile as it can simply increase the number of iterations of the window to increase reuse.

However as was shown in figure (19), it clearly shows that reuse is not the only factor that is relevant when optimizing.

5.5 Additionally Stripmined / multi-write versions

These tiled multi-write versions process multiple adjacent chunks, by loading in their combined read-tiles and the evaluating each of the chunks. This is done as to be able to have better re-use of loaded elements. So while a normal chunk would be of the same size as the group-size, then the `multi-write` version would be a multiple of the group-size on each axis. E.g in 3d a normal chunk and group-size could be of size $(32, 8, 4)$ for (x, y, z)-axis respectively, while a chunk could be of size $(32 \cdot 2, 8 \cdot 2, 4 \cdot 4)$, which we refer to as the write-tile. The numbers that we multiply the group size with are referred to as work-multipliers e.g. (2,2,4). For an graphical illustration of this, see figure (21).

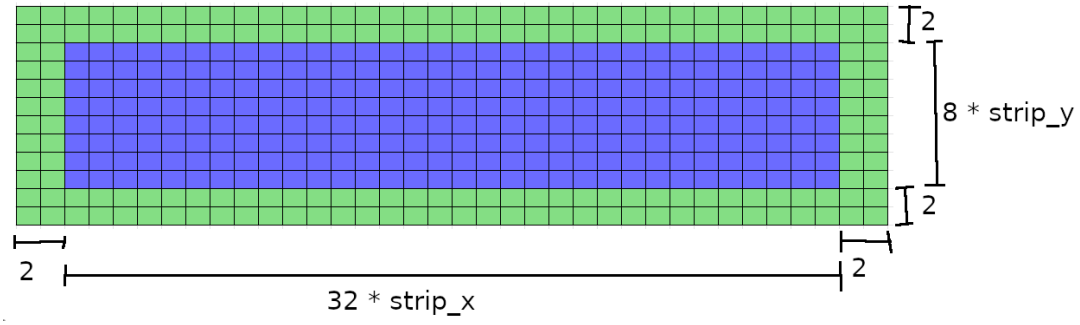


Figure 21: Illustration of the multi-write big-tile design, with how the per-thread-group read-tile (green and blue) and write-tile (blue) is laid out with 256 threads per thread-group yielding a 8x32 group size, work-multipliers is [1,1], where the stencil has axisMins=[-2,-2] and axisMaxs=[2,2].

For a 3d 7-point stencil of the point itself and the 6 direct neighbours, we can calculate the difference in reuse. For the standard version we have a mean reuse per thread-group of (see 5.1) of

$$\frac{7 \cdot (32 \cdot 8 \cdot 4)}{(32+2) \cdot (8+2) \cdot (4+2)} = 3.514$$

While the multi-write version (with work-multiples $(1, 2, 2)$ for (x,y,z)-axis) has a mean reuse per thread-group of

$$\frac{7 \cdot (32 \cdot (8 \cdot 2) \cdot (4 \cdot 2))}{(32+2) \cdot ((8 \cdot 2)+2) \cdot ((4 \cdot 2)+2)} = 4.685$$

There is however an important restriction for this, which is based on a hardware constraint. The tiles where all the loaded elements reside is in shared memory, and total amount of shared memory which can be allocated per group (not even taking occupancy into consideration) depends on the hardware. Some of the benchmarks are done on hardware where this limit is 48KiB, but other hardware have a limit in the same magnitude of size.

In relation to details of the Futhark Compiler Given what one has seen so far one may think that maxing that amount of used memory is a good idea, and simply to increase the work-multipliers as much as possible. However the benchmarks from the 2D sliding window experiments should show that reuse is not everything. Furthermore there is the problem in the current version of version of the Futhark Compiler (Ver. 20) that when it quarries the hardware for the amount of shared memory it can allocate, it is only provided the maximum possible for a single group. Using this variable is however not necessarily the best idea, as this number deviates from the amount one should

use to achieve maximum occupancy (percentage of active groups/threads per core), see table 4. Source [7]. The source does however state that the RTX 2080TI has a limit of 64KiB shared memory per group, but also states this only is the case when using dynamic shared memory, and only 48KiB if static shared memory is used. However when one quarry the hardware, then it says 48KiB

GPU	GTX 780	GTX 950	RTX 2080 TI
max shared-memory per thread-group	48 KiB	48 KiB	48 KiB (64 KiB dynamic)
max threads per SM	2048	2048	1024
max shared-memory per SM	48 KiB	96 KiB	64 KiB
max no. 1024-thread-groups per SM	2	2	1
(100% occupancy) max sh-cap for 1024-group	24 KiB	48 KiB	64 KiB

Table 4: Table of the shared memory restrictions for the different graphics cards we have used in the benchmarks.
Note that for the RTX 2080TI, that (*max shared-memory per thread-group*) is 48KiB when one quarry the hardware, and this is the limit for static shared-memory, but the limit for dynamic shared-memory is 64KiB.
(*max no. 256 thread groups per SM*)=(*max threads per SM*/256) and (*suggested shared-memory usage per 256-thread-group*)=(*max shared-memory per SM*/max no. 256-thread-groups per SM).
(*(100% occupancy) max sh-cap for 256-group*) is based on how much shared memory per thread-group we can use, without reducing the occupancy. Note that just because it uses less than this number does not guarantee that it will have 100% occupancy as there are other factors.

From table 4 we can see that (*max shared-memory per thread-group*) is not a good measurement of how much shared-memory is practically available per group, as we need an exact number as this can have severe performance implication if one uses too much or too little. This is a problem is this is the only variable available inside the Futhark compiler (as per version V.20) for measuring how much shared-memory is available.

Furthermore there is the issue that when the Futhark Compiler compiles a program that uses this variable *max shared-memory per thread-group*, then the variable is provided as a runtime variable (as opposed to the current setup where the work-multipliers are compile-time constants), meaning that all variables derived from it are also runtime variables (as they are derived from runtime variables). This would mean that the following list (of lists) of variables become runtime variables instead of compile-time constants: *work-multiples, shared-memory-sizes, shared-memory-spans, loader-iterations, loader-caps*. In

[7] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications`

the 2D multi-write version this adds up to 12 variables (and more for 3D) that used to be compile-time constants (and hence could be placed in assembly-instructions) which are turned into runtime variables (and hence need to compete for register space). The *loader-iterations* is a variable that defines how many iterations the loader has, where the loop can only be unrolled when it is a compile-time constant. The *work-multiples* are chosen to be powers of 2 as this means that when they are compile-time constant the compiler will handle they are turned into bit-shifting operations.

Based on what has previously been mentioned, it should be clear that making the kernel-code depend at runtime on the maximum amount of shared-memory per thread-group is no viable. An alternative to this is to create a number of versions of the kernel-code, each with different compile-time constants for this amount, and then at runtime pick which of these kernels to run based on the maximum amount of shared-memory per thread-group.

5.6 Prototyping Conclusion

We found that the multi-write big tile version had the best performance. It is combined with the flat loader approach. The flat loader approach generally performed better on newer GPUs, therefore we decided to use it.

6 Description of the GPU algorithm implemented in the compiler

Based on the prototyping done, and some experiments with the Futhark compiler, we decided to implement a combination of the designs that we call *multi-write big-tile* (presented in section 5.5) with the `flat-loader` variant (5.3.4), together with the `Global-read` design. The `Global-read` design is primarily used as a fall-back method in case the multi-write big-tile should not be run.

The `Global-read` design is described in section 5.2.2 and is nearly identical to the generated code of the reference implementation seen in section 9.2.1. A pseudocode of the implemented multi-write big-tile can be seen below for a 2D example. For simplicity a number of things will be be presented in a ways slightly different from actual implementation.

1. The input and output types are not tuples. Additionally the input types are assumed to be of size 4 bytes per element, so that we don't have to show how the work-multiplier are dependent on this.

2. We are normally working on C-style arrays, where multi-dimensional arrays are presented as 1D arrays with additional parameters used to describe the dimensions of the 3D array.

3. Normally we use the SingleDim approach for groups-ids (see section 5.3.3), but here show the MultiDim approach is used, since it is more simple.

.

Helper functions These are the helper functions used.

```
1  function divUp(x, y){
2    return (x + y - 1) / y
3  }
4  function __syncthreads();
5  function unflatten_2d(T index, T dim_x){
6    T y = index / dim_x;
7    T x = index % dim_x;
8    return (x, y);
9  }
10 function flatten_2d(T x, T y, T dim_x){
11   return y * dim_x + x;
12 }
```

line 1-3: Function for integer division rounding up.
line 4: Barrier. All threads in the thread-group must have reached this point before any can continue.
line 5-9: unflattens a flat index from the flat representation of a multi-dimensional array into the multi-dimensional indices.
line 10-12: Opposite way of unflattening.

Compile-time constants There are a lot of compile-time constants. Most of these have been presented in the table ref.

```
1  // These names are types: a, b, c.
2  int D;
3  (c -> [D]a -> b) lambda_function;
4  int indexes[D,2];
5  int a_min_x = minimum(indexes[:,0]);
6  int a_min_y = minimum(indexes[:,1]);
7  int a_max_x = maximum(indexes[:,0]);
8  int a_max_y = maximum(indexes[:,1]);
9  (int,int) Mul = (2, 2); //work multipliers
10 (int,int) B = (32, 8) //multi-dimensional group shape
11 int groupSize_flat = B.x * B.y
12 int S_x = B.x * Mul.x; //write tile x shape
13 int S_y = B.y * Mul.y; //write tile y shape
14 int shared_size_x = S_x + (a_max_x - amin_x);
15 int shared_size_y = S_y + (a_max_y - amin_y);
16 int shared_size_flat = shared_size_x * shared_size_y;
```

For ease of reading the variables will be presented again

line 2: The number of point of the stencil.
line 3: The lambda function.
line 4: The neighbourhood indices.
lines 5-6: Minimum value of the neighbourhood indices on each axis.
lines 7-8: Maximum value of the neighbourhood indices on each axis.
line 9: The work-multipliers on each axis.
line 10: The multi-dimensional group shape.
line 11: The flat group size.
lines 12-13: The write-tile.
lines 14-15: The read-tile.
lines 14-15: The amount of shared memory used in the kernel.

CPU side setup These variables are setup host-side/on the CPU, and are passed on to the kernel.

```
1  function multiWriteBigTile(
2    c invariant[M,N], a variant[M,N], b output[M,N]):
3    int64 grid_x = divUp(N, S_x);
4    int64 grid_y = divUp(M, S_y);
5    kernel<<<(grid_x,grid_y), (B.x, B.y)>>>
6          (invariant, variant, output)
```

The `<<< grid , group >>>` syntax is CUDA-style notation for launching a kernel using `group` as the thread-group size, and grid being the multidimensional area on which we run a thread-group for each index.

GPU side This contains the general kernel structure, but where `flatLoader` and `writeFromTile` are described in the paragraphs below this one.

```
1  __global__ kernel(
2      c invariant[M,N], a variant[M,N], b output[M,N]):
3      __shared__ a sharedMemArr[shared_size_y][shared_size_x];
4      (int64,int64) groupIdx;
5      (int,int) threadIdx;
6      (int64,int64) group_offset
7          = (groupIdx.x * S_x, groupIdx.y * S_y);
8      // Reading Phase
9      flatLoader(sharedMemArr,variants, group_offset, threadIdx);
10     __syncthreads();
11     // Writing Phase
12     writeFromSharedMem(invariants, sharedMemArr, output,
     group_offset, threadIdx);
```

The variables `groupIdx` and `threadIdx` are implicitly provided inside the kernel. The `group_offset` are the offsets into the arrays, from which reading/writing is to take place.

Reading phase / flatLoader The reading phase loads the smallest rectangle containing all the required elements from the variants array into the tile. The loader used is the flatLoader design (see section 5.3.4).

```
inline flatLoader(
  a sharedMemArr[shared_size_y][shared_size_x],
  a variants[M,N],
  (int,int) group_offsets,
  (int,int) threadIdx):
    int local_flat = flatten_2d(threadIdx.x, threadIdx.y, B.x);
    (int,int) adds
        = unflatten_2d(groupSize_flat, tile_size_flat);
    (int,int) locals
        = unflatten_2d(local_flat, tile_size_flat);
    for(int i = 0; i < iterations; i++):
        int64 gid_y = max(0, min(M-1,
            locals.y + a_min_y + group_offsets.y));
        int64 gid_x = max(0, min(N-1,
            locals.x + a_min_x + group_offsets.x));
        if(local_flat < shared_size_flat):
            sharedMemArr[local_y, local_x] = variants[gid_y,
   gid_x];
        // prepare next iteration.
        local_flat += groupSize_flat;
        local.y += adds.y;
        local.x += adds.x;
        if( local.x >= tile_size_x):
            local.x -= tile_size_x;
            local.y += 1;
```

Writing phase / writeFromSharedMem The writing phase uses the elements loaded from shared memory, together with the invariant array, to evaluate the lambda function for each of the indices of the output area which is covered by the work-tile.

```
writeFromSharedMem(
  c invariants[M,N],
  a sharedMemArr[shared_size_y][shared_size_x],
  b output[M,N],
  (int64,int64) group_offset,
  (int,int) threadIdx):
    (int64,int64) base_write_gid
        = ( group_offset.y + threadIdx.y
          , group_offset.x + threadIdx.x;
    for(int j = 0; j < Mul.y; j += 1):
      for(int k = 0; k < Mul.x; k += 1):
        int tile_off_y = j * B.y;
        int tile_off_x = k * B.x;
        int64 gid_y = base_write_gid.y + tile_off.y;
        int64 gid_x = base_write_gid.x + tile_off.x;
        if(gid_y < M && gid_x < N):
```

```
17            a var_points[D];
18            b invariant = invariants[gid_y, gid_x];
19            for (d = 0; d < D; d++):
20              int x = indexes[d,0]-a_min_x;
21              int y = indexes[d,1]-a_min_y;
22              int tile_ix_y = y + tile_off_y + threadIdx.y;
23              int tile_ix_y = x + tile_off_x + threadIdx.x;
24              var_points[d] = sharedMemArr[tile_ix_y, tile_ix_y];
25            output[gid_y, gid_x] =
26                lambda_function(invariant, var_points);
```

Multiple `B.x * B.y` sized areas are evaluated per thread-group, hence the 2 outer for-loops are used to iterate over these areas. The body of the for-loops is simply to load the parameters of the lambda-function, evaluate it, and write the result to the output.

7 Implementation of GPU code generation

We extend the pass in the GPU compilation pipeline that produce imperative intermediate representation. We use the example in Figure 2 to generate CUDA code. We could also have generated OpenCL code for the examples, however, the code would have been very similar for most parts. Some of the generated CUDA code examples might have some details omitted since they are not relevant for presenting the implementation. In this and following sections, the terminology *local id* is used to refer to the thread index as it is known in CUDA. The terminology *group id* is used to refer to the block index as it is known in CUDA.

7.1 The setup of the kernel

Initially we compute the size of the shared memory array and allocate the corresponding amount of shared memory as a flat array. The shared memory array size is determined by the unflattened group sizes, how many elements per thread are computed, and the width of the stencil (i.e. the distance between the minimum point and the maximum point for each dimension in the stencil). The flat size of the shared memory array is computed using the formula presented in table 3 for the read-tile. The compiler implementation allocates the shared memory array based on the types of the input array. There can be multiple shared memory arrays based on the different types, since the input array of the source Futhark code can contain tuples with elements of different types. However, for the CUDA generated code shown throughout this section only has a single shared memory array.

We need to unflatten the flat group into a multi-dimensional group shape, in order to get the unflattened group sizes. We refer to the unflattened group

sizes as `group_sizzes`. E.g. if we have a flat group size of 256, then we compute an unflattened group with dimensions ($y = 8$,$x = 32$), which equals $y \cdot x = 256$. The flat group size is by default 256 in the Futhark compiler, however, users can change the flat group size if they wish to. Therefore we have determined a formula for automatically computing the dimensions of the unflattened group inside the kernel based on the flat group size.

```
1  let = group_size_flat_subexp = unCount $
2          segGroupSize lvl
3        group_size_flat_exp = TPrimExp .
4          toExp' int32 $
5            group_size_flat_subexp
6        flNumT2 x = 1 +
7          sMin32 1
8            (group_size_flat_exp `quot` x)
9        fl256t2 = flNumT2 256
10       fl512t2 = flNumT2 512
11       fl1024t2 = flNumT2 1024
12       group_sizes_exp ::
13         [Imp.TExp Int32]
14       group_sizes_exp =
15         case dimentionality of
16           1 -> [group_size_flat_exp]
17           2 -> [fl64t2 *fl128t2
18                 *fl256t2*fl512t2
19                 *fl1024t2, 32]
20           3 -> [ fl256t2*fl1024t2
21                  ,fl128t2*fl256t2
22                   *fl512t2, 32]
23           _ -> error
24                  "not valid dimensions"
25 group_sizes <-
26   mapM (dPrimVE "group_sizes")
27     group_sizes_exp
```

Listing 14: Compiler implementation

```
1  #define segstencil_group_sizze_4569 (
       mainzisegstencil_group_sizze_4568)
2
3  int32_t group_sizes_4613;
4
5  group_sizzes_4612 =
6    (1 + smin32(1,
7         squot32(segstencil_group_sizze_4569,
8           64)))
9    * (1 + smin32(1,
10        squot32(segstencil_group_sizze_4569,
11          128)))
12   * (1 + smin32(1,
13        squot32(segstencil_group_sizze_4569,
14          256)))
15   * (1 + smin32(1,
16        squot32(segstencil_group_sizze_4569,
17          512)))
18   * (1 + smin32(1,
19        squot32(segstencil_group_sizze_4569,
20          1024)));
21
22 int32_t group_sizes_4614;
23
24 group_sizzes_4614 = 32;
```

Listing 15: generated CUDA code

Figure 22: Unflattening of the local id

On lines 1-2 of the compiler implementation, the flat group size is fetched from `lvl`. When using the flat group size, a macro `segstencil_group_sizze` is implicitly generated as a compile-time constant. On line 1 of the generated CUDA code we see the macro. On lines 6-8 of the compiler, we declare a function `flNumT2` which is used in the computation of the unflattened group. The function `flNumT2` will return `1` if the flat group size is smaller than `x`, or return `2` if the flat group size is larger than or equal to `x`. Then on lines 14-24 of the compiler we determine the shape of the unflattened group, which depends on the `dimensionality` of the input (i.e. if we have a 1D, 2D, or 3D stencil). The function allows the user to use group sizes of 32, 64, 128, 256, 512, and 1024. For 2D stencils we see on lines 17-19 in the compiler implementation, that the unflattened groups will be of the form (1,32), (2,32), (4,32), (8,32), (16,32), or (32,32). We finally bind the `group_sizes_exp` containing the shape of the unflattened group on lines 25-27 in the compiler. This produces the CUDA code on lines 3-24.

Since we now have computed the multi-dimensional group shape, we can compute the shape of the multi-dimensional write-tile, and other values which will be used during execution of the kernel.

```
let work_multiples :: [Integer]
    work_multiples =
      case dimentionality of
        1 -> [rescale_on_byte_size 4]
        2 -> [2, rescale_on_byte_size 2]
        3 -> [4, 2,
              rescale_on_byte_size 1]
        _ -> error
               "not valid dimensions"
    work_multiples_exp ::
      [Imp.TExp Int32]
    work_multiples_exp =
      map fromInteger work_multiples

group_spans <-
  mapM (dPrimVE "group_spans") $
    createSpans group_sizes

group_size_flat <-
  dPrimVE "group_size_flat" $
    product group_sizes

work_sizes <-
  mapM (dPrimVE "work_sizes")
    $ zipWith (*) work_multiples_exp
        group_sizes

shared_sizes <-
  mapM (dPrimVE "shared_sizes") $
    zipWith (+) work_sizes $
      zipWith (-) a_maxs a_mins

shared_size_flat <-
  dPrimVE "shared_size_flat" $
    product shared_sizes
```

Listing 16: Compiler implementation

```
int32_t group_spans_4614 = group_sizzes_4613
    ;
int32_t group_sizze_flat_4615 =
    group_sizzes_4612
  * group_sizzes_4613;
int32_t work_sizzes_4616 = 2 *
    group_sizzes_4612;
int32_t work_sizzes_4617 = 2 *
    group_sizzes_4613;
int32_t shared_sizzes_4618 =
    work_sizzes_4616 + 2;
int32_t shared_sizzes_4619 =
    work_sizzes_4617 + 2;
int32_t shared_sizze_flat_4620 =
    shared_sizzes_4618 * shared_sizzes_4619;
```

Listing 17: generated CUDA code

Figure 23: Computing the multi-dimensional tile shapes and flat tile size

On lines 15-17 of the compiler we compute `group_spans`. The variable `group_spans` is used for unflattening the flat local id for each thread. It is computed using `createSpans` which applies a built-in `scanr1` Haskell function on the tail of the multi-dimensional group shape `group_sizes`.

```
createSpans :: Num a => [a] -> [a]
createSpans = scanr1 (*) . tail
```

E.g. in 2D we only have two dimensions, therefore the result will be a list [32]. However, in 3D for a multi-dimensional group shape of [4,8,32] we will have a result of [256,32] which produces two `group_spans` variables. In 1D we do not have any unflattening of the local id, therefore the variable `group_spans` will not be generated for 1D stencils.

In order to compute the write-tile shape we first multiply each dimension of the multi-dimensional group `group_sizes` by some factor `work_multiples_exp`. These products are computed on lines 23-26 of the compiler and bound to `work_sizes`, which results in the generated CUDA code on lines 10-16. One could also say that `work_sizes` corresponds to how many output values are computed by some group. The factors `work_multiples_exp` are represented as a list of compile-time constants at lines 1-13 of the compiler. For input arrays with 4-byte elements the factors will be [2,2], which corresponds to increasing the shape of a multi-dimensional group from [8,32] to a larger write-tile shape of [16,64]. The write-tile size varies on the innermost dimension based on the element type of the input array.

Finally, we compute the shape of the read-tile by adding the $haloWidths = axisMaxs - axisMins$ to the `work_sizes` on lines 28-31 of the compiler. This could also be referred to as the as the number of elements that are read by a group. The corresponding CUDA code can be seen on lines 18-26.

Performance It should be mentioned that all computations in the CUDA generated code shown in this section, are performed on CUDA compile-time constants. Therefore all operations are computed at compile-time by the CUDA compiler, which makes the code very efficient. However, this approach does come at the cost of not being able to dynamically at run-time determine the write-tile size. Another important optimisation is that many of the compile-time constants are a power of two. If x is a run-time constant and y is a compile-time constant power of 2, then any operation $x \cdot y$, $\frac{x}{y}$, and x 'rem' y will be computed with more efficient instructions (bit-shifting and bit-wise-and). Here, the 'rem' operator represents the remainder operator. However, we cannot guarantee that this optimisation occurs for the shared memory (read-tile) shape, since the length of each dimension depends on the $haloWidths$. However, the write-tile shape and the multi-dimensional group shape dimensions are always a power of two. In theory one could arbitrarily increase the amount of shared memory used, in order to ensure that the shape dimensions are a power of two. However, this might significantly increase the amount of shared memory required for a kernel, which might impact the occupancy of the GPU and thereby reduce performance.

7.2 Computing the local ids and group ids

The algorithm works on a single-dimensional grid for 1D, 2D and 3D stencils. In other words, we only have a single flat local id and group id for each thread in each group. However, for 2D and 3D stencils we compute the stencil in multi-dimensional tiles. This requires the local id and group id to be unflattened with respect to the shape of the multi-dimensional group. An alternative to using single-dimensional grids would be to use multi-dimensional grids. If we used a multi-dimensional grid, we could avoid the unflattening computa-

tions as we would have a precomputed local id and group id for each thread. The unflattening computations use computationally heavy operators such as division and modulus. However, CUDA has special restrictions on the number of groups allowed in a grid for the second and third dimensions. Therefore, for the sake of robustness we decided to use a single-dimensional grid.

We implemented a function `unflattenIx` that computes unflattened indices based on some shape. Specifically, the function takes a `name` for the binding of the unflattened indices, a list of dimensions (e.g. the dimensions of a two-dimensional group could be ($y = 8$,$x = 32$)), and the flat index `i` to be unflattened.

```
1  unflattenIx ::
2    IntExp t =>
3    String ->
4    [Imp.TExp t] ->
5    Imp.TExp t ->
6    ImpM lore r op [TV t]
7  unflattenIx name [] i = (: []) <$> dPrimV name i
8  unflattenIx name (x : xs) i = do
9    dimIx <- dPrimV name $ i `quot` x
10   rem_val <- dPrimV "rem_val" $ i `rem` x
11   (dimIx :) <$> unflattenIx name xs (tvExp rem_val)
```

As the function recursively unflattens `i` with respect to the length `x` of a dimension, it also binds results `dimIx` and `rem_val` with `dPrimV` such that these division and modulo expressions are executed only once. As an example for 2D stencils with two-dimensional groups, the unflattening of the local id produces the following CUDA code:

```
1  local_id_flat <- dPrimVE "local_id_flat"
2    . kernelLocalThreadId $ constants
3  local_ids <- map tvExp <$>
4    unflattenIx "local_id"
5      group_spans local_id_flat
```

Listing 18: Compiler implementation

```
1  int32_t local_id_flat_4491 = local_tid_4476;
2  int32_t local_id_4492 = squot32(
3      local_id_flat_4491,
4    group_spans_4484);
5  int32_t rem_val_4493 = srem32(
6      local_id_flat_4491,
7    group_spans_4484);
8  int32_t local_id_4494 = rem_val_4493;
```

Listing 19: generated CUDA code

Figure 24: Unflattening of the local id

On lines 1-2 in the compiler implementation we query and bind the flat local id. The variable name for the flat local id is bound to `local_id_flat`. The corresponded generated code can be seen in Listing 2 on lines 1-3. Then we use the `unflattenIx` function on lines 3-5 in the compiler implementation to unflatten the flat local id. The result is a list of variable names bound to `local_ids`. For the 2D stencil we get a list of two local id variables for the first

and second dimension, respectively. The generated code from `unflattenIx` can be seen on lines 5-17 in the generated CUDA code.

In terms of unflattening the group id, we refer to an unflattened group id as `work_id`. Since we handle multiple elements per thread, we also have a fewer number of total groups in the grid. The shape of the grid is referred to as the `work_grid_spans`, which is a variable computed before launching the CUDA/OpenCL kernel. Extending upon the example of the 2D stencil, we compute the unflattened group id using the code in Figure 25.

```
1  work_id_flat <- dPrimVE "work_id_flat"
2    . kernelGroupId $ constants
3  work_ids <- map tvExp <$>
4    unflattenIx "work_id"
5      work_grid_spans work_id_flat
6  let bound_idxs =
7    zipWith sMin64 max_ixs .
8                    map (sMax64 0)
9  writeSet_offsets <-
10   mapM (dPrimVE "writeSet_offset") $
11     zipWith (*) (map sExt64 work_ids)
12                 (map sExt64 work_sizes)
```

Listing 20: Compiler implementation

```
1  int32_t work_id_flat_4495 = group_tid_4477;
2  int32_t work_id_4496 = squot32(
       work_id_flat_4495,
3    work_grid_spans_4470);
4  int32_t rem_val_4497 = srem32(
       work_id_flat_4495,
5    work_grid_spans_4470);
6  int32_t work_id_4498 = rem_val_4497;
7  int64_t writeSet_offset_4629 =
8    sext_i32_i64(work_id_4626) *
9      sext_i32_i64(work_sizes_4616);
10 int64_t writeSet_offset_4630 =
11   sext_i32_i64(work_id_4628) *
12     sext_i32_i64(work_sizes_4617);
```

Listing 21: generated CUDA code

Figure 25: Unflattening of the group id

The generated code for unflattening the flat group id is symmetrical to the previous example in Figure 24. The only difference is that we now unflatten `work_id_flat` which is based on the flat group id, using the shape of `work_grid_spans`. On lines 9-12 of the compiler we also compute and bind `writeSet_offset`. We generate a `writeSet_offset` variable for each dimension in the stencil. The `writeSet_offset` is an index offset that is used for all threads in the group to read from the input array and write to the output array. The corresponding CUDA generated code can be seen on lines 27-29.

Performance

The unflattening computation of the local id is very efficient, since the division and remainder operations are converted into efficient instructions. This is due to the fact that `group_spans` is a compile-time constant and a power of two. However, the unflattening of group id is not as efficient. The variable `work_grid_spans` is not a compile-time constant, since it is passed as a parameter to the kernel (i.e. computed before launching the kernel). A potential optimisation would be to compute `work_grid_spans` inside the kernel when the input array size is known at compile-time. However, it will not increase efficiency unless the input array size is also a power of two. We decided to omit this optimisation, since it seems only relevant for edge-cases.

7.3 Loading from global memory to shared memory

A central part to the kernel is reading values of the input array from global memory and writing those values into shared memory. Each thread will read and write more than one value using a loop which is generated on line 1 of the CUDA code. The loop is generated using a function `sForUnflatten` on line 1 of the compiler.

```
1  sForUnflatten shared_sizes local_id_flat
2    group_size_flat
3      $ \(loader_ids,
4          loader_ids_flat,
5          isNotLastIter) -> do
6
7    loader_gids <-
8      mapM (dPrimVE "loader_gid") $
9        bound_idxs $
10         zipWith (+) writeSet_offsets
11           $ map sExt64 $
12             zipWith (+) loader_ids
13               a_mins
14
15   sWhen (isNotLastIter .||.
16          loader_ids_flat .<.
17          shared_size_flat) $
18     forM_ (zip tiles (stencilArrays op))
19     $ \(tile, input_arr) ->
20       copyDWIMFix tile
21         [sExt64 loader_ids_flat]
22         (Var input_arr)
23         loader_gids
24 sOp $ Imp.Barrier Imp.FenceLocal
```

Listing 22: Compiler implementation

```
1  ... // extra code from sForUnflatten
2  for (...) {
3    int64_t loader_gid_4526
4      = smin64(max_ixs_4483, smax64(
5          (int64_t) 0,
6          writeSet_offset_4510
7            + sext_i32_i64(start_4515 + -1)));
8    int64_t loader_gid_4527
9      = smin64(max_ixs_4484, smax64(
10         (int64_t) 0,
11         writeSet_offset_4511
12           + sext_i32_i64(start_4517 + -1)));
13   if (1) {
14     ((__local float *) tile_mem_4491)
15     [sext_i32_i64(start_flat_var_4524)]
16     = ((__global float *) as_mem_4460)
17     [loader_gid_4526 * n_4434
18     +loader_gid_4527];
19   }
20    ... // extra code from sForUnflatten
21 }
22 // last iteration is unrolled
23 int64_t loader_gid_4529
24   = smin64(max_ixs_4483, smax64(
25       (int64_t) 0,
26       writeSet_offset_4510
27         + sext_i32_i64(start_4515 + -1)));
28 int64_t loader_gid_4530
29   = smin64(max_ixs_4484, smax64(
30       (int64_t) 0,
31       writeSet_offset_4511
32         + sext_i32_i64(start_4517 + -1)));
33 if (slt32(start_flat_var_4524
34           ,shared_sizze_flat_4501)) {
35     ((__local float *) tile_mem_4491)
36     [sext_i32_i64(start_flat_var_4524)]
37     = ((__global float *) as_mem_4460)
38     [loader_gid_4529 * n_4434
39     +loader_gid_4530];
40 }
41 barrier(CLK_LOCAL_MEM_FENCE);
```

Listing 23: generated CUDA code

Figure 26: Loading from global memory to shared memory

Note that `sForUnflatten` is described in section 5.3.4 (as design, not implementation), and that the last iteration of the loop it generates is unrolled (which is not shown in the design).

1-4 L The function `sForUnflatten` uses the read-tile shape, the flat local id, the flat group size and a function to generate the loop.

7-13 L : 3-12(23-32) R: The unflattened indices used for loading from global memory `loader_gids` are generated. The number of generated indices correspond to the dimensionality of the stencil. In this example we get two `loader_gid` in the generated CUDA code, since we are generating a 2D stencil.

15-17 L : 13(33-34) R: Checks for out-of-bounds of the shared array. We know that this can only happen on the last (unrolled) iteration, hence the `(1)` which is a result of constant folding of `(True || ...)`.

18-23 L: 14-18(35-39)R: For each shared array, fetch an element from the invariant array, and store it in the corresponding shared array.

24 L: 41 R: Synchronise the groups before we start the phase where we read from the shared array.

7.4 Preparing the offsets for the multi-write loop nest

```
1  base_write_gid <-
2    mapM (dPrimVE "base_write_gid")
3      $ zipWith (+) writeSet_offsets
4        $ map sExt64 local_ids
5
6  local_id_shared_flat <-
7    dPrimVE "local_id_shared_flat"
8      $ flattenIndex shared_sizes
9                     local_ids
10
11 let nest_shape =
12   map ( Constant . IntValue
13       . intValue Int32) work_multiples
14
15 sLoopNestSE nest_shape $ \local_work_ids ->
       do
16   tile_ids_offs <-
17     mapM (dPrimVE "tile_ids_offs")
18       $ zipWith (*) group_sizes
       local_work_ids
19
20   tile_ids_offs_flat <- dPrimVE "
       tile_ids_offs_flat" $ flattenIndex
       shared_sizes tile_ids_offs
21   zipWithM_ dPrimV_ gids_vn $ zipWith (+)
       base_write_gid $ map sExt64
       tile_ids_offs
22   let gids = map (toInt64Exp . Var) gids_vn
```

Listing 24: Compiler implementation

```
1  int64_t base_write_gid_4531
2    = writeSet_offset_4510
3    + sext_i32_i64(local_id_4503);
4  int64_t base_write_gid_4532
5    = writeSet_offset_4511
6    + sext_i32_i64(local_id_4505);
7  int32_t local_id_shared_flat_4533
8    = local_id_4503 * shared_sizzes_4500
9    + local_id_4505;
10 for (int32_t nest_i_4534 = 0;
11      nest_i_4534 < 2;
12      nest_i_4534++) {
13   for (int32_t nest_i_4535 = 0;
14        nest_i_4535 < 2;
15        nest_i_4535++) {
16     int32_t tile_ids_offs_4536
17       = group_sizzes_4493 * nest_i_4534;
18     int32_t tile_ids_offs_4537
19       = group_sizzes_4494 * nest_i_4535;
20     int32_t tile_ids_offs_flat_4538
21       = tile_ids_offs_4536
22       * shared_sizzes_4500
23       + tile_ids_offs_4537;
24     int64_t gtid_4446
25       = base_write_gid_4531
26       + sext_i32_i64(tile_ids_offs_4536);
27     int64_t gtid_4447
28       = base_write_gid_4532
29       + sext_i32_i64(tile_ids_offs_4537);
```

Listing 25: generated CUDA code

Figure 27: Preparing the offsets for multi-write loop nest

1-4 L : 1-6 R: The per group-index offset to the output array.

6-9 L : 7-9 R: The per thread offset to the shared array.

11-15 L : 10-15 R: These are the loop-nests over each of the subsections of the area (write-tile) of the output array that we write to.

16-18 L : 16-19 R: The per thread index into the shared array, and what needs to be added onto the base-write-gid.

20 L : 20-23 R: This is the flattened offset into the shared array.

21-22 L : 24-29 R: These are the axis-wise elements of the index of the output array that we will write to.

7.5 Loading of elements for the lambda-function for each work-multiplier

Since the values from the input array that are to be used for the entire group has been loaded into shared memory, we can now compute the stencil function and write to the output array. Initially we need indices for writing to the output array and reading from the shared memory array.

```
1  let tile_offsets = map (flattenIndex
     shared_sizes) $ transpose $ zipWith (mapM
      (-)) stencil_ixss a_mins
2      variant_params_tuples = transpose $
     chunksOf n_point_stencil variantParams
3
4  tile_ixs <- mapM (dPrimVE "tile_ixs" . (+
     local_id_shared_flat) . (+
     tile_ids_offs_flat)) tile_offsets
5
6  sWhen (foldl1 (.&&.) $ zipWith (.<=.) gids
     max_ixs) $ do
7    compileStms mempty (kernelBodyStms kbody)
     $ pure ()
8
9    zipWithM_ dPrimV_ (map paramName
     invariantParams) . map TPrimExp
10     =<< mapM toExp invarElems
11
12   dLParams variantParams
13   forM_ (zip tile_ixs variant_params_tuples)
      $ \(tile_ix, pars) ->
14     forM_ (zip pars tiles) $ \(par, tile) ->
15       copyDWIMFix (paramName par) [] (Var
     tile) [sExt64 tile_ix]
```

Listing 26: Compiler implementation

```
1    int32_t tile_ixs_4539
2      = tile_ids_offs_flat_4538
3      + local_id_shared_flat_4533;
4    int32_t tile_ixs_4540
5      = shared_sizzes_4500 + 1
6      + tile_ids_offs_flat_4538
7      + local_id_shared_flat_4533;
8    int32_t tile_ixs_4541
9      = 2 * shared_sizzes_4500 + 2
10     + tile_ids_offs_flat_4538
11     + local_id_shared_flat_4533;
12   if (sle64(gtid_4446, max_ixs_4483) &&
13       sle64(gtid_4447, max_ixs_4484)) {
14       float stencil_inv_4457
15         = ((__global float *) cs_mem_4459)
16         [gtid_4446 * n_4434 + gtid_4447];
17       float x_4438 = stencil_inv_4457;
18       x_elem_4454
19         =((__local float *) tile_mem_4491)
20         [sext_i32_i64(tile_ixs_4539)];
21       x_elem_4455
22         =((__local float *) tile_mem_4491)
23         [sext_i32_i64(tile_ixs_4540)];
24       x_elem_4456
25         =((__local float *) tile_mem_4491)
26         [sext_i32_i64(tile_ixs_4541)];
```

Listing 27: generated CUDA code

Figure 28: Loading of elements for the lambda-function for each work-multiplier

1-4 L : 1-11 R: These are the neighbourhood indices to be read from the shared array.
6 L : 12-13 R: Checks if we are in bounds of the output array.
7 L : 14-16 R: The invariant element loaded from the invariant array.
9-15 L : 18-26 R: Declares and binds the parameters of the lambda-function to neighbours in the shared array.

7.6 Running the lambda-function and writing to output array

```
1 compileStms mempty (bodyStms lamBody) $
2   zipWithM_ (compileThreadResult space)
      (patternElements pat) $
3     map (Returns ResultMaySimplify) $
      bodyResult lamBody
```

Listing 28: Compiler implementation

```
1         float x_4442 = x_elem_4454
2                      + x_elem_4455;
3         float x_4444 = x_4442
4                      + x_elem_4456;
5         float defunc_1_f_res_4445
6           = x_4444 / x_4438;
7         ((__global float *) mem_4464)
8         [gtid_4446 * n_4434 + gtid_4447]
9         = defunc_1_f_res_4445;
10      }
11    }
12  }
13  error_0:
14    return;
15    #undef segstencil_group_sizze_4452
16 }
```

Listing 29: generated CUDA code

Figure 29: Running the lambda-function and writing to output array

1-3 L : 1-9 R: Inlines the lambda-function and writes the result to the output array.
10-16 R: Finishes the loop nest and returns.

8 Validation of the generated code and a comparison to the prototypes

This section concerns showing that the generated C code, and GPU code does indeed run correctly. Additionally it we want to compare the performance of our implementation to the of our prototypes.

8.1 Validation of the CPU and GPU code

The test suite contain a number of unit-tests, some for each combination of dimensions, static/dynamic stencils, tupled/single element input arrays. An additional test was provided by our supervisor Troels, which tested that the stencils construct is actually internalised. We have 12 unit-tests, and 8 random-data fixed-size tests, and 1 structural test (the one provided to us).

The test validated for the C back-end as shown per figure (30).

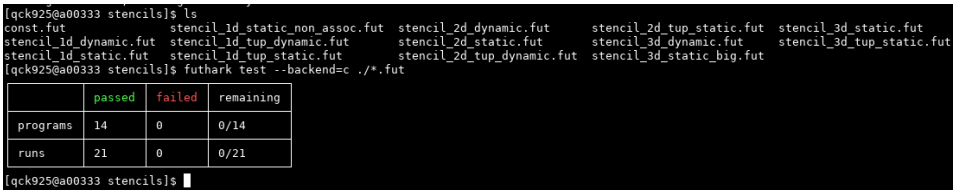


Figure 30: C code validation run in terminal in the test directory in the Futhark repository on the stencil branch.

The GPU code generator is a bit more tricky, as there are checks in the OpenCL/CUDa code generated for the stencil construct, which only runs the multi-write big-tile kernel if the input is sufficiently large. This is for performance reasons, but it does mean that the unit-tests only test the fallback-kernel *GlobalRead*. Most of the fixed-size random-data tests are sufficiently large to make the multi-write big-tile run. This is known to be the case as it for a thread-group-size of 256 (which is the default) needs to be strictly larger than $[256 \cdot 4]$ in 1D, $[8 \cdot 2, 32 \cdot 2]$ in 2D, and $[2 \cdot 2, 4 \cdot 2, 32 \cdot 1]$ in 3D (with the current configuration of work-multipliers). These random-tests are for all dimensions and for tupled static stencils of multiple data-types, and the result is compared to the C back-end (which could be argued to be a problem).

The test validated for both the OpenCL back-end and the CUDA back-end, as can be seen in figure 31.

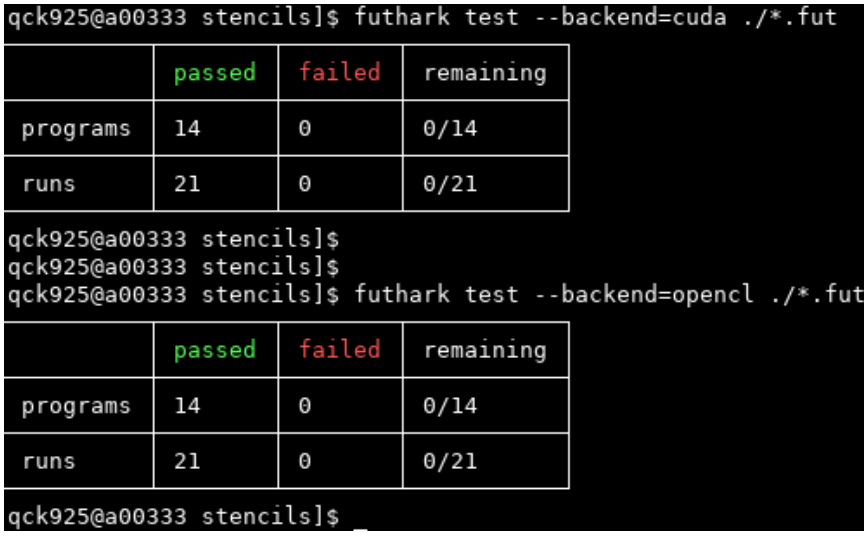


Figure 31: OpenCL and CUDA code validation run in terminal in the test directory in the Futhark repository on the stencil branch.

There are occasionally some issues with the tests such that they are within a margin-of-error. This happens on floating point examples where we use

division. This could be due to how certain floating point operations behave slightly different on NVidia GPUs compared to common CPUs.

8.2 Comparison between prototype and code generation

In Figure 32 we verify that our Futhark code generated multi-write big-tile version performs similar to our prototype of the multi-write big-tile version.

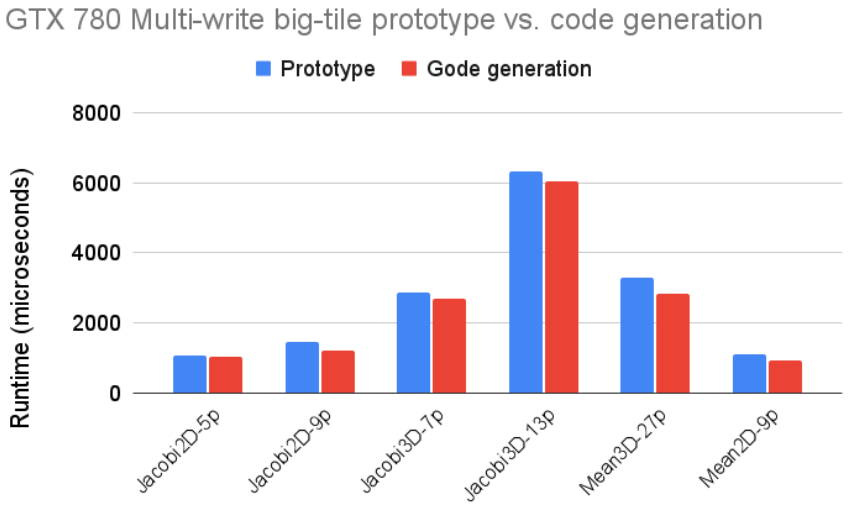


Figure 32: The Futhark code-generation version of multi-write big-tile vs. the prototype of multi-write big-tile, using the same work-multiplier. The Jacobi stencils only perform one iteration. The Mean2D and Mean3D compute the mean of the neighbourhoods $\{-1,0,1\}\times\{-1,0,1\}$ and $\{-1,0,1\}\times\{-1,0,1\}\times\{-1,0,1\}$ for 2D and 3D, respectively. The 2D and 3D input arrays have the sizes 4100×4098 and $264 \times 260 \times 258$, respectively.

The Futhark-generated version actually performs slightly better than our prototype. We have not attempted to figure out exactly why this is the case.

9 Empirical evaluation

In this section we will evaluate our design decisions and the performance of the multi-write big-tile on commonly used stencils. In section 9.1 we focus on evaluating the design decisions. In section 9.2 we perform benchmarks on common stencils such as Hotspot3D, Heat3D, and Jacobi3D.

9.1 Evaluation of the design decisions

We made some design decisions with regards to our multi-write big-tile implementation. We also want to validate that we made sensible design decision, based on a proper evaluation of the efficiency of our implementation. Some of our most significant design decisions were:

1. Setting the block/group size.
2. Increasing the number of output elements computed per thread to increase performance.
3. Setting the work-multipliers of the sequential loops.
4. How to configure the number of elements per thread and the group size based on different input element data types.

For stencils one would wish to maximize reuse. The potential reuse within a group is increased when you increase the group size. Therefore setting the group size to less than the maximum possible group size is not a trivial design choice. In section 9.1.1 we will explain why a group size of 256 is often better than 1024 for many common stencils. Another interesting design choice is why we would wish to reduce the amount of parallel work on a GPU. We will delve into this in section 9.1.2. After validation of our design decisions we will evaluate our implementation on a number of common stencil applications in section 9.2.

Many of the metrics shown in the following sections 9.1.1 and 9.1.2 are derived from the Nvidia CUDA profiler. Furthermore, the profiler was executed on our prototype of the multi-write big-tile version running on Nvidia GTX 780 GPU[8]. Therefore, one should not expect to see the exact same numbers on a newer or older Nvidia GPU. However, the overall tendencies shown are likely to be generalised to other contemporary Nvidia GPUs. A weakness of our evaluation and project is that we only did benchmarks and evaluation on Nvidia GPUs. This is due to the fact that we only had access to Nvidia GPUs.

9.1.1 Group size

The amount of shared memory we use in our kernels have a direct correspondence with how large the group size is. Therefore, if we increase the group size, then we also increase the reuse. However, empirical data shows that it is not always the case that more reuse provides better performance. This is due to a trade-off between latency hiding and reuse when setting the group size. Based on the CUDA programming guideline we should:

[8]The CUDA profiler require special access rights to the operative system, therefore we had to use our personal computer for profiling.

> "*Use several smaller thread blocks rather than one large thread block per multiprocessor if latency affects performance. This is particularly beneficial to kernels that frequently call __syncthreads().*"[9]

Here, a thread block is Cuda terminology for what in OpenCL is referred to as a group. A multi-processor has a limited potential amount of active groups. If the groups are large, then the number of active groups must usually be small. If the number of active groups are too small, the hardware cannot be fully utilised. We can also see in Figure 33, that the instructions per cycle increases as the group size decreases on a 1D 9-point stencil that computes a mean of its neighbourhood.

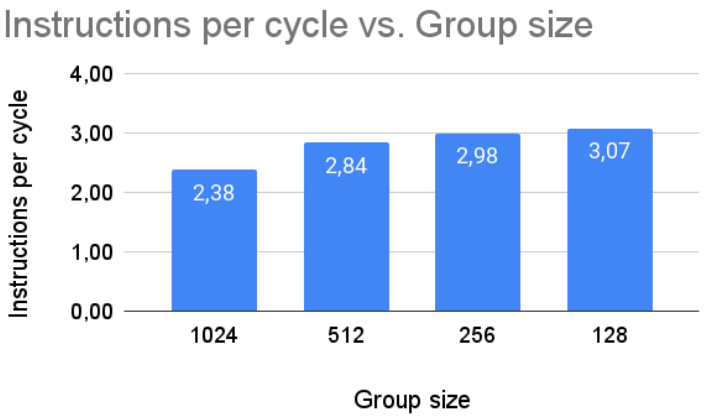


Figure 33: The instructions per cycle on the Nvidia GTX 780 based on running the Nvidia CUDA profiler on our big-tile prototype. The maximum possible instructions per cycle is four on this particular GPU.

As previously mentioned, reuse is also a factor to consider. When measuring the runtime of the 1D 9-point stencil, we see that the version using a 256 group size achieves the best performance

[9] https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#thread-and-block-heuristics

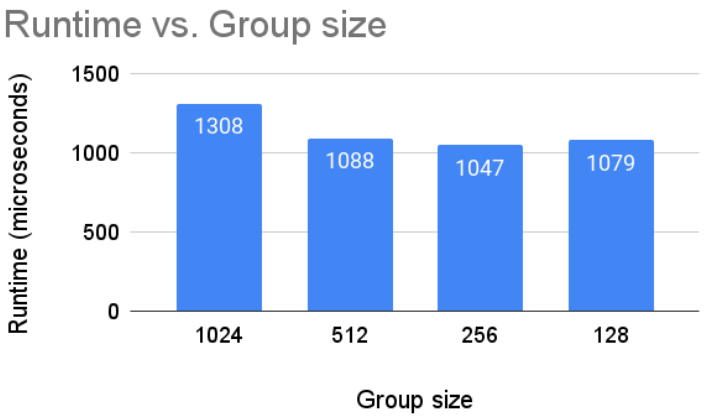


Figure 34: The runtime (measured in microseconds) versus the group size. The runtime is measured on the Nvidia GTX 780 GPU using the same stencil and prototype from Figure 33.

Figure 34 shows that even if the kernel using a smaller group size of 128 performs more instructions per cycle, it does not directly translate into better performance in terms of runtime. It shows that a trade-off exists between reuse and the benefits of a smaller group size. This example does not illustrate that a group size of 256 is always the best option in 1D. However, it shows the impact of reuse for this particular 9-point stencil in 1D. The figure shows that even if a large group size of 1024 maximises the reuse it is not necessarily the best option for this stencil. If the stencil had many more points than this example, then the results could have been different. However, as we will see in the following figures, a group size of 1024 is not the ideal option in general for stencils with a number of points between 2 and 25 for the GPUs that we benchmarked with. Stencils with a number of points between 2 and 25 corresponds to what most common stencils have. As mentioned previously, we are primarily concerned with common stencils in this thesis. All benchmarks in the following figures were performed on our multi-write big-bile prototype, where the number of writes per thread is one. Therefore this configuration of the multi-write big-bile version is equivalent to the regular big tile prototype.

Points	Shape	Points	Shape
2	[0,1]	11	[-5,...,5]
3	[-1,...,1]	13	[-6,...,6]
5	[-2,...,2]	15	[-7,...,7]
7	[-3,...,3]	17	[-8,...,8]
9	[-4,...,4]	25	[-12,...,12]

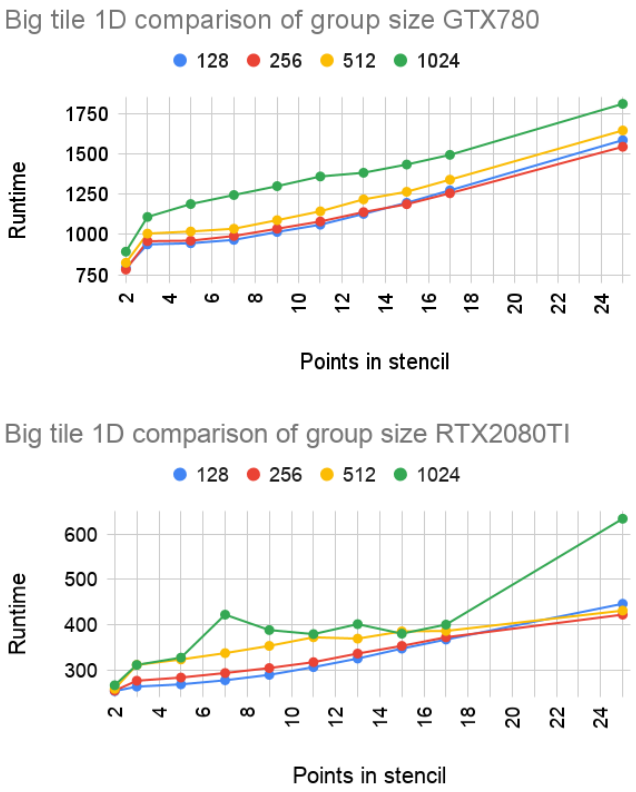


Figure 35: Benchmarks with various group sizes, where we plot the runtime versus the number of points in the 1D stencil. The randomly generated input array has length $2^{24} + 2$. The runtime is the average runtime of 100 runs measured in microseconds. The labels with colours indicate the group size.

On both a newer and an older GPU, we see that a smaller group size of 128, and 256 gives better performance for the 1D stencils in Figure 35. It is relevant to consider the shape of the stencil neighbourhood, since the shape affects how we should interpret the results. As the range of the minimum and maximum points of the stencil increase, the number of points also increase. Therefore larger stencils are expected to increase reuse and increase performance of the big tile versions using larger groups. We see that on smaller stencils, the smaller group size of 128 performs better, while on larger stencils the group

size of 256 performs better. However, the performance is very similar for both of the smallest group sizes.

Points	Shape
4	$\{0,1\} \times \{0,1\}$
6	$\{-1,0,1\} \times \{0,1\}$
9	$\{-1,0,1\} \times \{-1,0,1\}$

Points	Shape
12	$\{-1,0,1,2\} \times \{-1,0,1\}$
16	$\{-1,0,1,2\} \times \{-1,0,1,2\}$
20	$\{-2,-1,0,1,2\} \times \{-1,0,1,2\}$
25	$\{-2,-1,0,1,2\} \times \{-2,-1,0,1,2\}$

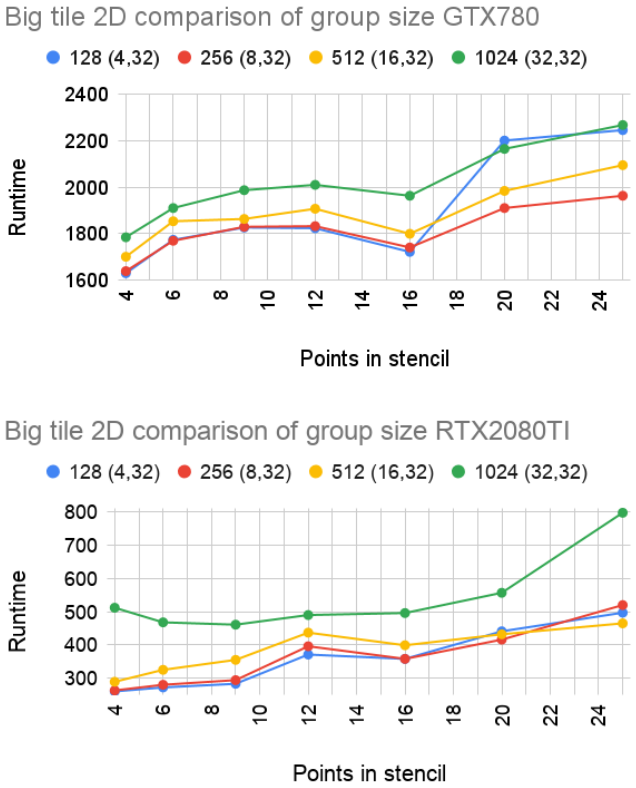


Figure 36: Benchmarks with various group sizes, where we plot the runtime versus the number of points in the 2D stencil. The randomly generated $n \times m$ input array has $n = 2^{12}+4$ and $m = 2^{12}+2$. The runtime is the average runtime of 100 runs measured in microseconds. The labels with colours indicate the flat group size, and the shape of the multi-dimensional group.

In Figure 36, we see that the smaller groups are more efficient for the 2D stencils. The performance using a 128 group size is worse compared to using a 256 or 512 group size on larger stencils (more than 16 points) on the older GPU. This occurs due to the cache of the 780 GPU. On large stencils the cache of the older GPU has worse performance, and the reuse of the stencil points

become more significant. On the newer GPU we see no significant performance decrease of the 128 group size on larger stencils, since the cache on the newer GPU is better compared to the older GPU. However, even on the newer 2080TI GPU, reuse of the stencil points does play a role. The slope of the runtime has a relatively small increase when using a 512 group size on larger stencils (16 or more points), compared to using group sizes of 128 or 256.

Points	Shape
8	$\{0,1\} \times \{0,1\} \times \{0,1\}$
12	$\{-1,0,1\} \times \{0,1\} \times \{0,1\}$
18	$\{-1,0,1\} \times \{-1,0,1\} \times \{0,1\}$
27	$\{-1,0,1\} \times \{-1,0,1\} \times \{-1,0,1\}$

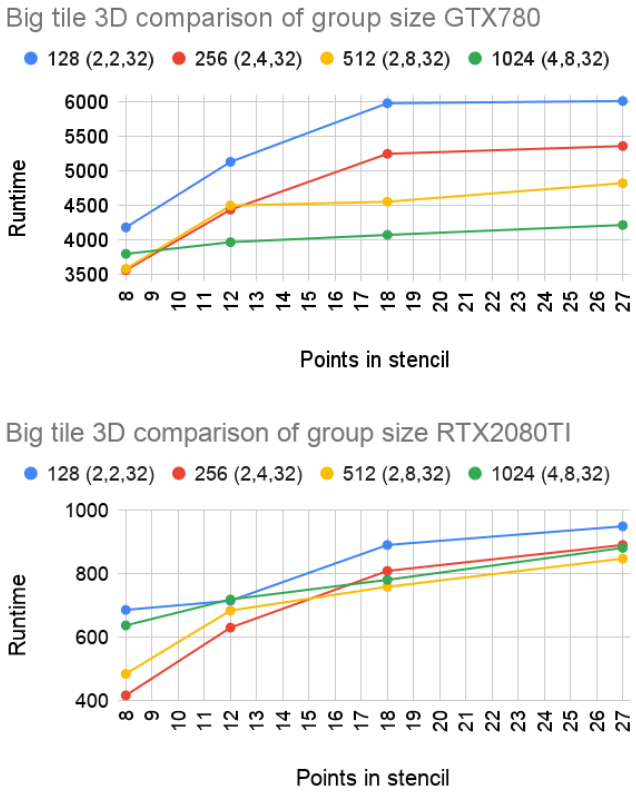


Figure 37: Benchmarks with various group sizes, where we plot the runtime versus the number of points in the 3D stencil. The randomly generated $n \times m \times k$ input array has $n = 2^8 + 8$, $m = 2^8 + 4$, and $k = 2^8 + 2$. The runtime is the average runtime of 100 runs measured in microseconds. The labels with colours indicate the flat group size, and the shape of the multi-dimensional group.

In Figure 37, we see that a larger group size increases the performance on the older GPU 780. The slope of the runtime has a relatively small increase when using a 1024 group regardless of the number of points in the stencil. This could be due to the fact that performance is largely affected by the reuse of the stencil points in 3D when the cache system is not as effective. For both GPUs we see similar tendencies, but also that other factors affect the performance. However, on the newer GPU, a group size of 1024 does not seem to be as effective.

Summary and evaluation of the group size benchmarks

When selecting the group size, we must consider the trade-off between

1. using a smaller group size,
2. the efficiency of the cache system,
3. and the reuse of stencil points.

When using a smaller group size, we effectively increase the potential for latency hiding since the GPU is more saturated with more potential active groups. This means that components of the GPU can be more utilised, including the cache system. However, even if the cache system is more utilised, it does not necessarily lead to higher efficiency than having more reuse of stencil points through shared memory. Especially for 3D stencils, one can expect that many points will be evicted from the cache between different warp executions. For 1D and 2D stencils it seems that the cache system on both the older and the newer GPU is efficient enough such that the trade-off leans toward using smaller group sizes rather than increasing the reuse of stencil points. For 3D stencils the trade-off is more dependant on the efficiency of the cache system. Using a smaller group size for the newer GPU seems to be the most viable option. However, using a larger group size for the older GPU would be better. With all this in mind, we cannot find a group size that fits all GPUs or stencils, but we can lean towards optimising for the newer GPU rather than the older one. Therefore, the default group size when running a stencil in Futhark is 256. However, the user is able to specify a different group size when running a Futhark program with[10]

```
$ ./FutharkProgram --default-group-size=1024
```

An ideal option would be to allow for autotuning of the group size, such that any hardware and any stencil can be optimised in terms of group size. However, we did not have time to implement any options for autotuning stencils through Futhark. A potential improvement to this analysis and evaluation, would be to investigate the effect of the multi-dimensional group shape which is based on the flat group size. The shape of the multi-dimensional group shape

[10]When running a program using the Terminal of the operative system Ubuntu.

can alter the reuse. The shape can also change the pattern in which the stencil is computed for a group, which will effect the efficiency of the cache system. Another potential improvement to this analysis and evaluation would be to investigate the impact of using different data types for the input array elements. All of the benchmarks in this section are performed on an input array with elements of single-precision floating-point numbers. We will investigate the performance when using different data types in section 9.1.4.

9.1.2 Increasing the number of output elements computed per thread

The stencil used for analysis and profiling throughout this section is a 2D 4-point stencil with the points $[(-1,-1),(-1,0),(0,-1),(0,0)]$ on a $n \times m$ input array size with $n = 2^{12} + 4$ and $m = 2^{12} + 2$. With this method we stripmine the parallel loops into sequential loops on a 4-point stencil in 2D. This leads to fewer groups being spawned, since each thread will handle multiple elements sequentially. A benefit of this approach, is that the program will contain fewer instructions. This includes instructions that are only computed once per thread, regardless of the number of output elements per thread. In this example the input and output array contains single-precision floating-point numbers, which require floating point instructions when computing the stencil function. We refer to these instructions as *FP instructions*. We also need to read from the input array and write to the output array with memory load and store instructions. We will refer to these instructions as *Load/Store instructions*. For each thread we need thread synchronisation barriers, index offsets, and bit-conversions which we will refer to as *other instructions*. Below we can see an illustration of how many instructions from each category that are produced, depending on how many output elements we compute per thread.

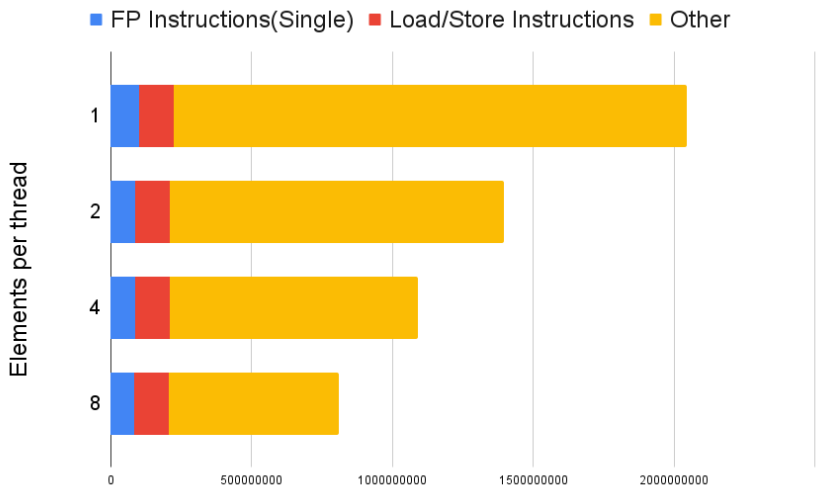


Figure 38: The total number of instructions based on the number of output elements per thread using group size 256. FP instructions are single-precision floating-point instructions, load/store instructions are loads and stores from and to memory (including shared memory). The other instructions consists of synchronisation barrier instructions, integer arithmetic instructions, control flow instructions and more. Credits to Vasily Volkov for the inspiration to this type of diagram. https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf

As seen in Figure 38, the number of "other instructions" decrease significantly when computing more than one output element per thread. The number of FP instructions and Load/Store instructions stays almost the same, regardless of how many elements are computed sequentially. An important category among the other instructions are related to barrier synchronisations. Since we have fewer instructions due to thread synchronisation, we have a relatively larger ratio of parallel instructions such as load and stores. This will increase the capability of the GPU to hide latency. Therefore we reduce the total amount of other instructions that must be computed only once by a thread, in order for that thread to compute multiple elements.

We also increase the reuse when computing multiple elements, since the read-tile shape become larger such that we store more elements in shared memory. Below in Figure 39 we can see a comparison of how the number of output elements per thread affects the runtime.

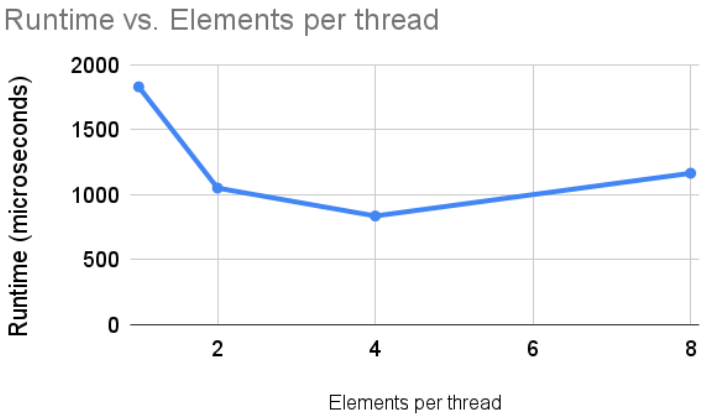


Figure 39: The runtime of a 4-point 2D stencil versus the number of output elements per thread. Using group size of 256 and input array $n \times m$, with $n = 2^{12} + 4$ and $m = 2^{12} + 2$.

As seen in Figure 39, the performance increases significantly with respect to runtime, when computing multiple elements per thread. However, there is also a trade-off, as we see that the performance start decreasing at eight or more elements per thread. In Figure 40, we compare the instructions per cycle depending on how many output elements per thread we compute.

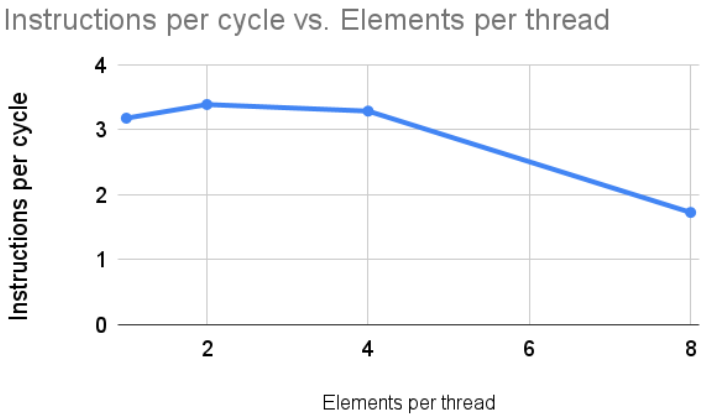


Figure 40: The instructions per cycle of a 4-point 2D stencil versus the number of output elements per thread. Using group size of 256 and input array $n \times m$, with $n = 2^{12} + 4$ and $m = 2^{12} + 2$.

In Figure 40, we see that the instructions per cycle start decreasing when processing four or more output elements per thread. However, in Figure 39, we

see that the runtime increases even when the instructions per cycle decrease. This could be due to the fact that we have significantly fewer instructions when computing four output elements per thread, rather than two. However, we cannot keep increasing the number of elements per thread, since we also increase the amount of shared memory we use and how many registers we use. the GPU has limits on important resources such as shared memory and number of registers per thread and group. Below we have a table showing how many resources we use depending on the number of output elements per thread.

Elements per thread	Shared memory per group	Registers per thread	Total shared memory with 4 groups
1	1.1602KB	19	6.402KB
2	2.28KB	21	9.12KB
4	4.3164KB	29	17.27KB
8	16.629KB	32	66.5KB

Figure 41: The GPU resource usage of the 2D 4-point stencil depending on the number of output elements per thread. Using group size of 256.

In Figure 41, we see that the amount of required shared memory increases significantly as the number of elements per thread increases. Even for a 4-point stencil with a small range, the required amount of shared memory (66,5KB) exceeds the available shared memory with four groups of size 256 and 8 elements per thread. The amount of available shared memory is (48KB[11]) per streaming multiprocessor of the Nvidia GTX 780 GPU. It should be noted that this GPU is relatively old, and therefore has a lower amount of available shared memory than contemporary Nvidia GPUs.

Summary and evaluation of increasing the amount of output elements per thread Processing more elements per thread does increase the efficiency of the implementation significantly. The increase in efficiency is caused by an increase in reuse per group and reducing the number of instructions that are executed once for each thread[12]. It is unclear how much of the performance benefits are related to increased potential for latency hiding (due to instruction-level parallelism), or the overall reduction in the number of instructions. E.g. using a larger number of elements per thread, we potentially achieve a larger occupancy, compared to processing one element per thread. Occupancy is the ratio between the number of actual active warps per streaming multi-processor (SM) vs. the maximum number of active warps per SM.

[11] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications

[12] https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#effects-of-shared-memory

Elements per thread	Write-tile	Occupancy
1	$[8, 32]$	0.948435
4	$[16, 64]$	0.967696

Table 5: 9-point 2D stencil, showing an increased occupancy with more elements per thread. The shape of the stencil and the input array shape is similar to the 9-point stencil benchmark from Figure 36.

However, during this process we are at risk of decreasing the potential for occupancy of the GPU, which can compromise the performance. We also reduce the total number of groups. For smaller input sizes we might reduce the number of groups such that we have too few groups to saturate and fully utilise the GPU. In these cases we cannot meaningfully execute this approach. Therefore we need to implement some safeguards to ensure that the multi-write big-bile implementation cannot be run under certain conditions such as

1. when the required amount of shared memory is too large,
2. when the number of required registers is too large,
3. or when the number of groups will be too low.

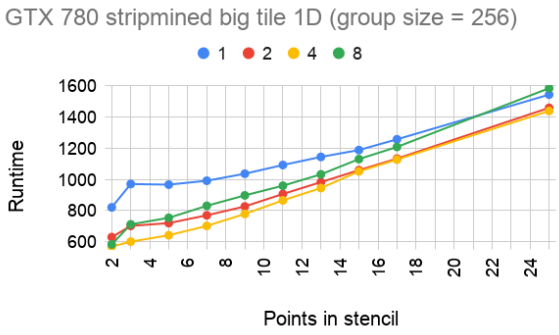
We attempted to make safeguards that would check these conditions, however, they will almost never work as intended. Most of our safeguards are based on some constants, these constants have nothing to do with the actual properties of the GPU that will run the Futhark program. Furthermore, our safeguards do not take into account the required registers, or if the number of groups will be too low. The CUDA programming guide recommends to have at least more than one active group for each streaming multi-processor[13]. We only have a safeguard for shared memory. However, the shared memory safeguard does not take the desired number of active groups into account. Instead it conservatively only allows using half of the total shared memory per SM. An improvement would be to extend the Futhark compiler such that the properties of the GPU running the Futhark program are used for the safeguards. Instead we need conservative estimates of how much we will allow in terms of resource usage.

9.1.3 Setting the number of elements per thread

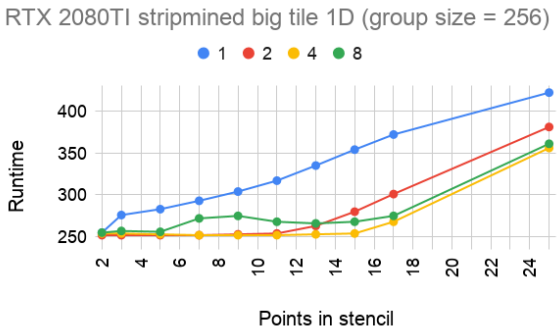
Given the implementation considerations at section 5.5 regarding the details of the futhark compiler, we decided to set the work-multipliers to constant values (with some exceptions for different input data types). Since the values are constant for any GPU configuration and stencil, the values chosen must be

[13] https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#thread-and-block-heuristics

robust in terms of handling common stencils, while still providing increased performance. If we choose a work-multiplier too large, then we might achieve suboptimal occupancy for many common stencils. However, if we choose a work-multiplier too small, we might be loosing out on a significant performance increase. The optimal choice can be very GPU and even stencil dependent, therefore we chose a work-multiplier based on empirical evidence. If the work-multipliers causes too large resource requirements for some configuration of GPU and stencil, then the Global Read version should be executed instead. We benchmark dense stencils of increasing size with a number of different work-multipliers. The stencils used for the benchmarks are similar to those in section 9.1.1 for 1D, 2D and 3D unless otherwise specified. All benchmarks were carried out on input array elements of single-precision floating-point numbers, unless otherwise specified. Figure 42 shows that a work-multiplier of four is most optimal on both the GTX 780 and RTX 2080TI for these 1D stencils.



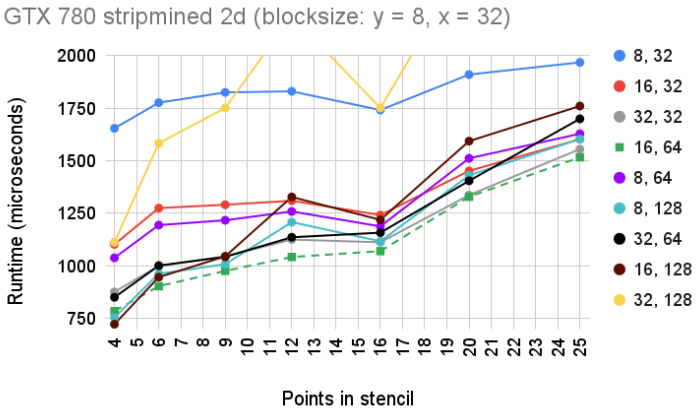
(a) GTX780



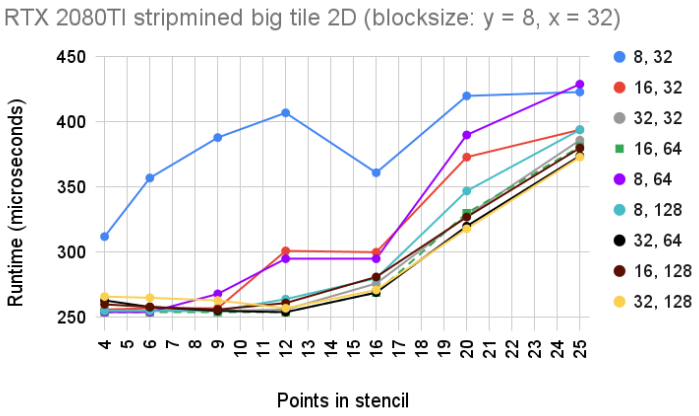
(b) RTX2080TI

Figure 42: Benchmarks for finding the most optimal work-multipliers for dense 1D stencils of increasing size. The labels show the number of output elements per thread. The randomly generated input array has length $2^{24} + 2$. The runtime is the average runtime of 100 runs measured in microseconds.

Figure 43 contains the benchmarks for 2D stencils with various write-tile shapes. We use the benchmarks for deciding which work-multipliers are most appropriate when using a group size of 256, with a multi-dimensional group shape of ($y = 8$,$x = 32$). From this figure one should be able to see that the version with a write-tile shape of ($y = 16$,$x = 64$) - meaning work-multipliers of [2,2], is the most robust option between the older and the newer GPU.



(a) GTX780

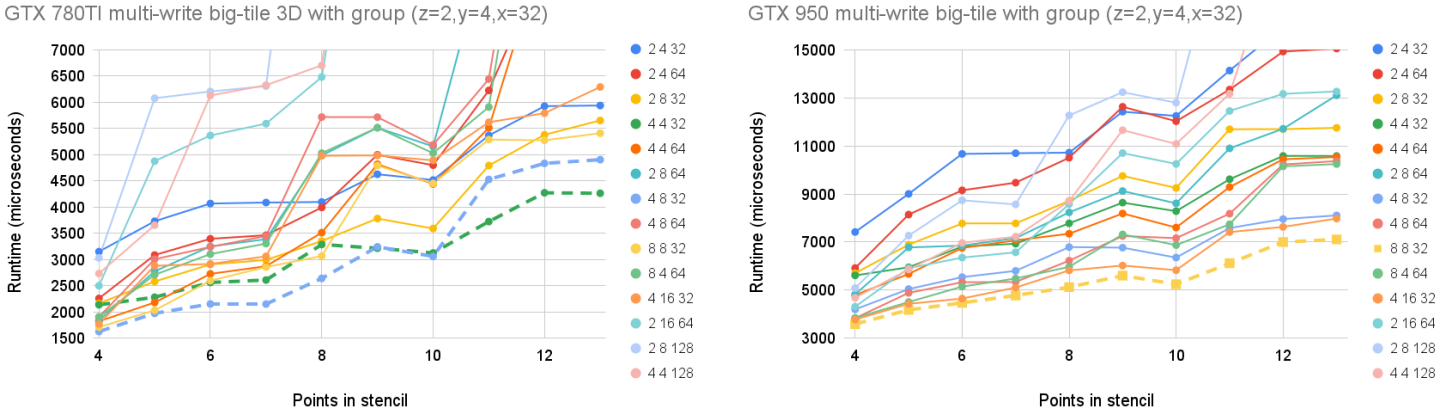


(b) RTX2080TI

Figure 43: Benchmarks for finding the most optimal work-multipliers for dense 2D stencils of increasing size. The labels show the write-tile size after multiplying the multi-dimensional group with the work-multipliers for the sequential loops. The randomly generated $n \times m$ input array has $n = 2^{12} + 4$ and $m = 2^{12} + 2$. The runtime is the average runtime of 100 runs measured in microseconds.

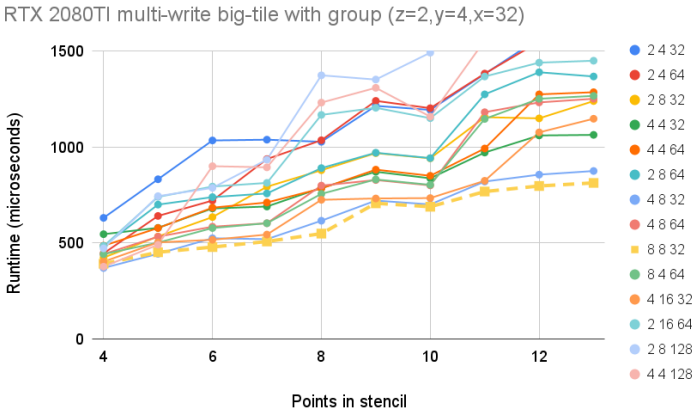
Figure 44 contains the benchmarks for 3D stencils with various write-tile

shapes. We use a group size of 256, with a multi-dimensional group shape of ($z = 2$,$y = 4$,$x = 32$). Based on Subfigures 44b and 44c one would chose work-multipliers [4,2,1] for the (z,y,x) axis respectively. Based on Subfigure 44a one would chose [2,2,1] or [2,1,1], depending on the number of points in the stencil.



(a) GTX780TI

(b) GTX950



(c) RTX2080TI

Figure 44: Benchmarks with stencils that calculate a mean of the von Neumann neighbourhood with increasing sizes. The labels show the write-tile size after multiplying the multi-dimensional group with the work-multipliers for the sequential loops. The randomly generated $n \times m \times k$ input array has $n = 2^8 + 8$, $m = 2^8 + 4$, and $k = 2^8 + 2$. The runtime is the average runtime of 100 runs measured in microseconds.

Summary and evaluation of setting the work-multiplier

Regardless of using an older or newer GPU, the optimal number of output elements per thread seems to be four with some exceptions. For the 3D stencils the GTX 780 GPU seems to have relatively worse performance with four elements per thread, probably due to using too many resources. However, newer GPUs are most efficient when computing eight output elements per thread. The GTX 780 GPU has smaller amount of shared memory available. For robustness and simplicity we compute four elements per thread for 1D, 2D and 3D stencils. Four elements per thread performs relatively well in all cases except one. We also have a potential benefit from choosing four rather than eight elements per thread for 3D stencils. We provide better support for stencils that use more registers and shared memory, than the stencils used for our benchmarks. A weakness of this analysis and evaluation is that we use von-Neumann style neighbourhoods for stencils for 2D and 3D examples. One could have made stencils with a fewer number of points, but in a larger range, e.g. in 2D we could have a stencil with shape $[(-4,-3),(-2,-1),(2,1),(4,3)]$. Even though the stencil has a fewer number of points it still contains a smaller and larger minimum and maximum index. Therefore it will use more shared memory than the stencils we benchmarked with. It might be that setting the number of output elements per thread to four is too large for other stencils. Therefore we have a safegaurd that ensures that we have at least half of the shared memory available for the stencil kernel. Another potential improvement to this section is to perform benchmarks on different input array element types. This is done in the following section.

9.1.4 Configurations based on different input element data types

Throughout this section we will only focus on the RTX 2080TI GPU, since our goal here is to find potential optimisations that are targeted contemporary GPUs. The data types that we will focus on are 8-bit integers and double-precision floating-point numbers.

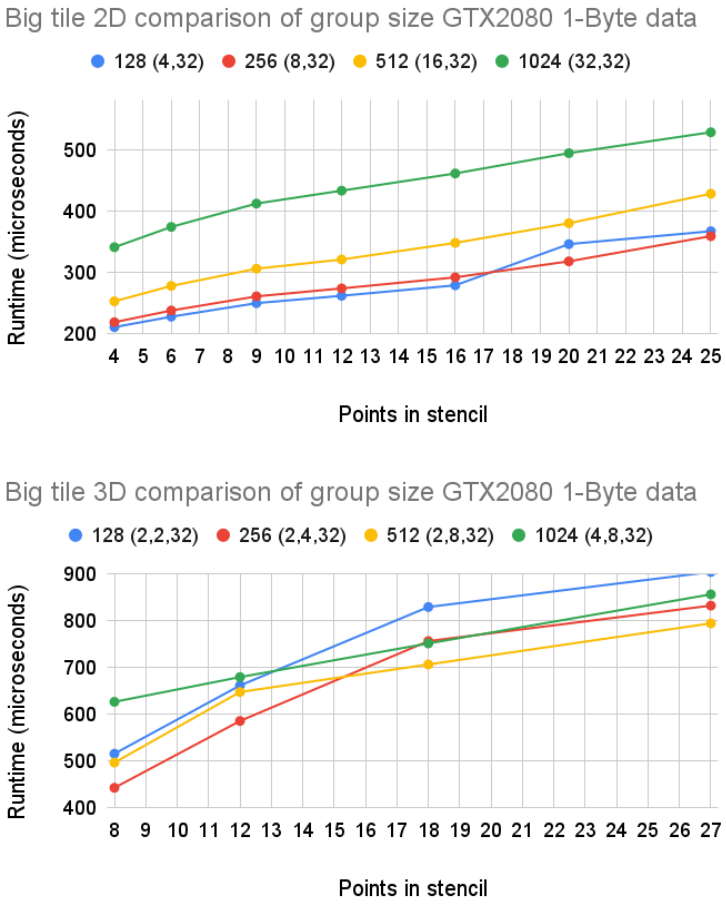


Figure 45: Benchmarks with various group sizes, where we plot the runtime versus the number of points in the 3D stencil for 1-byte data elements. The randomly generated input arrays have the same shapes, and the same stencil neighbourhoods as in Figure 36 and in Figure 37. The runtime is the average runtime of 1000 runs measured in microseconds. The labels with colours indicate the flat group size, and the shape of the multi-dimensional group.

As seen in Figure 45, the performance achieved when varying the group size for 1-byte data elements follows the same principles as with the 4-byte data elements for both 2D and 3D stencils. One could make minor optimisations by using a group size of 128 on 1-byte data elements for 2D stencils, rather than using a group size of 256. However, we do not desire to further complicate the implementation, based on a relatively small optimisation. Also for 3D one could attempt to switch between the group size of 256 and 512 for 1-byte data elements. Again, it is a relatively small optimisation, therefore we decided not to implement it.

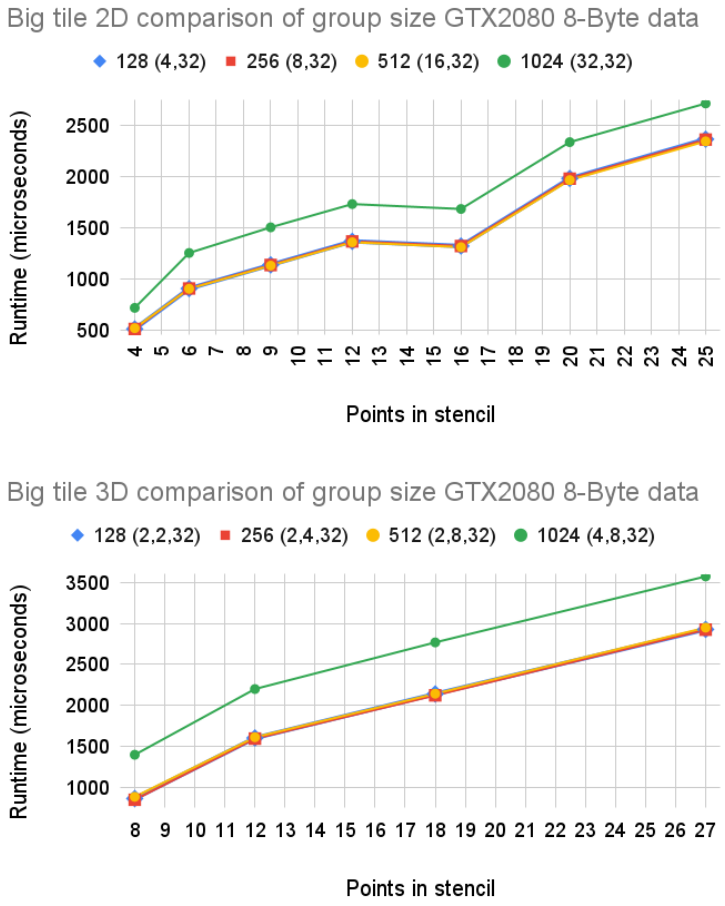


Figure 46: Benchmarks with various group sizes, where we plot the runtime versus the number of points in the 3D stencil for 8-byte data elements. The randomly generated input arrays have the same shapes, and the same stencil neighbourhoods as in Figure 36 and Figure 37. The runtime is the average runtime of 1000 runs measured in microseconds. The labels with colours indicate the flat group size, and the shape of the multi-dimensional group.

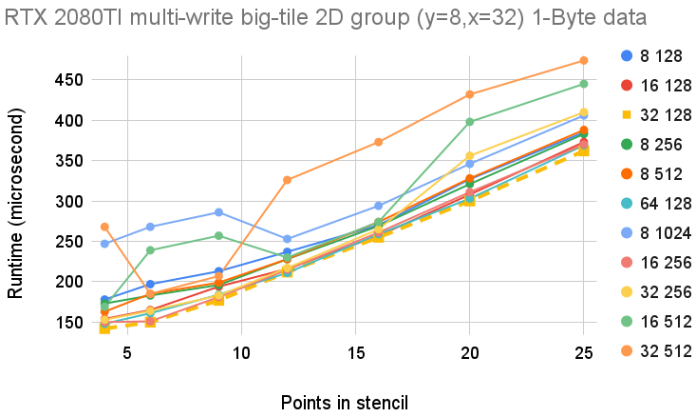
Surprisingly, the group sizes 128, 256, and 512 have almost identical runtime measurements, regardless of the number of points in the 2D or 3D stencil. Usually, we would expect that smaller group sizes would be more efficient on smaller stencils, due to the increased potential for latency hiding. However, it seems that two or more active groups are enough to reach a point where the RTX 2080TI is not limited by a lack of memory latency hiding. One would also expect to see increased performance on stencils with relatively many points when using larger group sizes due to reuse. However, the runtime is almost identical for the group sizes below 1024. This suggests that the double-

precision floating-point arithmetic instructions are the bottlenecks that limit the runtime of this kernel. Based on the CUDA programming guide, the RTX 2080TI can compute two double-precision floating point number operations per clock cycle[14], which is relatively low compared to other GPUs.

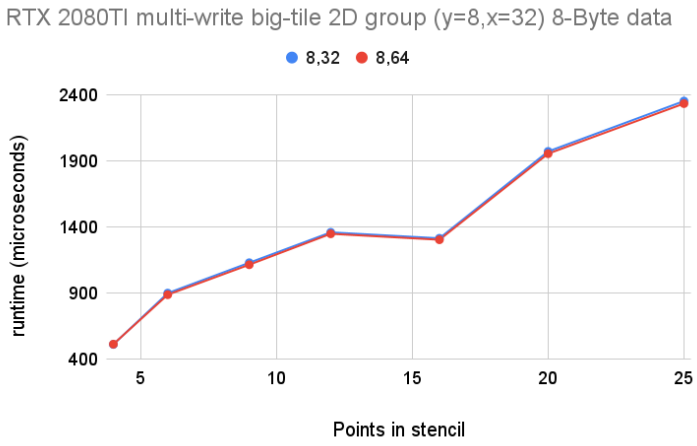
Number of elements per thread for different data sizes

The benchmarked data sizes are 1-Byte (integers) and 8-Bytes data (double-precision floating-point numbers) on the RTX 2080 TI GPU. In Figure 47 we see the runtime for write-tiles.

[14] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions



(a) 1 Byte data.



(b) 8 Byte data - floating point.

Figure 47: Running time for different work-multipliers on the RTX2080TI for different data sizes.

The optimal elements per thread for these 2D stencils is 16 elements per thread, such that we multiply the multi-dimensional group size with $[4, 4]$. The result suggests that we should scale the elements per thread according to the byte size of the element. Then if we use $[2, 2]$ multipliers for the multi-write big-tile version on 4-byte elements, we should scale by a factor of four, when using 1-byte elements. On 8-byte elements we see that processing more elements per thread does not have an impact. This is similar to what we saw with regards to the group size. That is, on the RTX 2080TI GPU the performance is limited by the throughput of double-precision floating-point numbers. However, as we will see in section 9.2, this is not the case for all GPUs. Therefore we use the

same strategy of scaling the elements per thread according to the byte size of the input array elements. Then if we use $[2, 2]$ for 4-byte elements, we should use $[1, 2]$ or $[2, 1]$ for 8-byte elements, which is scaled by a factor of two. For the simplicity of our implementation we always scale the elements per thread of the innermost dimension, such that we choose $[2, 1]$ for 8-bytes. Choosing whether or not to decrease or increase the elements per thread for the inner- or outer-dimension can depend on both the GPU and the stencil. This can be seen in the figures of section 9.1.3. A potential improvement of this analysis and evaluation would be to make benchmarks with different data types for more GPUs and stencils. However, we have evaluated our implementation, such that we can reason about benchmark results of more common stencils in the following section. A significant weakness of our evaluation, is that we have not considered input arrays containing tuples.

9.2 The benchmarks programs

In this section we show the performance that can be expected from the current GPU code generation implementation of stencils in the Futhark compiler. The benchmark programs were implemented based on the LIFT-stencil paper [3], where some of these originate from the Rodinia benchmark suite [2]. The actual benchmark implementations for Futhark can be found in a Github repository[15].

[15] https://github.com/Quartzinin/futhark-stencil-benchmarks

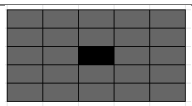
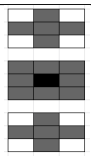
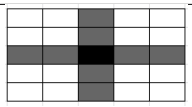
Name	Dimensionality	Points	Neighbourhood pattern
Gaussian-blur	2D	25	
Poisson-blur	3D	19	 The 3 layers of the 3D neighbourhood
Gradient	2D	5	
SRADv2	2D	5,5	
Heat3D	3D	7	Shaped like Jacobi-2D-5point but in 3D
Hotspot2D	2D	5	
Hotspot3D	3D	7	Shaped like Jacobi-2D-5point but in 3D
Jacobi2D-5p	2D	5	
Jacobi2D-9p	2D	9	
Jacobi3D-7p	3D	7	Shaped like Jacobi-2D-5point but in 3D
Jacobi3D-13p	3D	13	Shaped like Jacobi-2D-9point but in 3D
type-f64	3D	7	Shaped like Jacobi-2D-5point but in 3D
type-i8	3D	7	Shaped like Jacobi-2D-5point but in 3D

Table 6: Table of benchmarks programs. The table shows the number of dimensions in the stencil and the number of points in the stencil neighbourhood. For the pattern, the black cells are the centre-point of the neighbourhood, while grey cells are the surrounding neighbours. The white cells are unused for that particular computation.

Some of these programs in table (6) are iterative stencils. An iterative stencil is a stencil inside of a loop that repeats the stencil computation a number of times.
Descriptions of the stencils:

Gaussian-blur: This is a Gaussian-blur with $\sigma = 1.5$ as a compile-time constant.

Poisson-blur: This is similar to the Gaussian-blur, except it uses the discrete Poisson distribution (with $\lambda = 0.25$ as a compile-time constant) to create the weights. Furthermore the corners of the neighbourhood are not used for whatever reason (though the weight of those corners would also be very

small). It is based on the Poisson benchmark from the a LIFT paper [3], however it is not clear if the Poisson benchmark in their benchmarks is in fact a Poisson blur. In the LIFT code the weights are hard-coded for the neighbourhood, and it uses subtraction of the weighted neighbours from the centre-index. This is different from our implementation, which simply computes a weighted mean.

Gradient: This is a weighted mean of the forward and backward finite-differences on each axis. This is based on the Gradient benchmark from a LIFT paper [3].

SRADv2: This is based off of the Rodinia benchmark with the same name [2]. Some minor modifications were made, but it should still calculate the same result. Correctness of this benchmark program assumes that the original program was correct. This example is interesting since it works with an input array containing tuples, which is something the stencil-construct supports.

Heat3D: This is a heat transfer calculation. Our implementation is based on the LIFT-stencil benchmarks [3].

Hotspot2D/3D: These are based on the Rodinia benchmarks of the same name.

Jacobi: These are a weighted mean of the von Neumann neighbourhood.

Type-i8: This example is used to test different data sizes (here signed 8-bit integer) rather than the f32 (single-precision floating-point number) type we have been using so far. This example is simply a sum of all elements in the stencil neighbourhood.

Type-f64: Similar to the example above, except input array element type is of f64 (double-precision floating-point numbers).

9.2.1 The reference implementation

The reference implementation that we benchmark the stencil construct against, is made by using nested maps with some basic optimisations. We need to compare the new stencil construct to what can already be done using pre-existing features of Futhark.

1. The indices that are read from the input are sorted (e.g. stencil indices [(1,1),(-1,-1),(1,0)] would be moved around to be [(-1,-1),(1,0),(1,0)]), such that we can have the best possible spatial-locality.

2. The Futhark compiler will insert runtime bounds checking for array accesses. The `#[unsafe]` keyword will disables the insertion of bounds

checking. This will reduce some amount of redundant computations for the reference implementation.

3. When using maps, we know that the global-id that we are working with at any given index is in fact in-bounds. Therefore we can remove some of the min/max computations. Furthermore the implementation of maps in the compiler seem to do common sub-expression elimination on the indices into arrays.

```
-- index a neighbour in 2D (-1,1) on [M][N]f32 input.
-- note: tabulate_2d M N f
--       === map (\gidy -> map (\gidx -> f gidy gidx)
   [0..<N]) [0..<M]
tabulate_2d M N (\gidy gidx ->
    bound_y = max(0, min(M-1, gidy-1))
    bound_x = max(0, min(N-1, gidx+1))
    element = input[bound_y, bound_x]
    -- ... load other elements and call function
)
-- the above can be replaced with
tabulate_2d M N (\gidy gidx ->
    bound_y = max(0, gidy-1)
    bound_x = min(N-1, gidx+1)
    element = input[bound_y, bound_x]
    -- ... load other elements and call function
)
```

9.2.2 The benchmarks

The following sizes were used for the benchmarks:

Name	small-size	large-size	very-large-size	stencil-iterations
Gaussian-blur	[4095][4095]f32	[8191][8191]f32	[16383][16383]f32	5
Poisson-blur	[255][255][255]f32	[511][511][511]f32	[511][1023][1023]f32	5
Gradient	[4095][4095]f32	[8191][8191]f32	[16383][16383]f32	1 (not iterative)
SRAD-v2	[2047][2047]f32	[4095][4095]f32	[8191][8191]f32	10
Heat-3D	[255][255][255]f32	[511][511][511]f32	[511][1023][1023]f32	5
Hotspot-2D	[4095][4095]f32	[8191][8191]f32	[16383][16383]f32	5
Hotspot-3D	[255][255][255]f32	[511][511][511]f32	[511][1023][1023]f32	5
Jacobi-2D-5p	[4095][4095]f32	[8191][8191]f32	[16383][16383]f32	5
Jacobi-2D-9p	[4095][4095]f32	[8191][8191]f32	[16383][16383]f32	5
Jacobi-3D-7p	[255][255][255]f32	[511][511][511]f32	[511][1023][1023]f32	5
Jacobi-3D-13p	[255][255][255]f32	[511][511][511]f32	[511][1023][1023]f32	5
Type f64	[255][255][255]f64	%	%	1
Type i8	[255][255][255]i8	%	%	1

Table 7: Table of benchmarks programs. The sizes are shown using [depth][column_length][row_length](element-type) format, where f32 is a 32-bit floating point number, and i8 is a 8-bit signed integer. The reason all of the sizes of the dimension are a power of 2 minus 1 is to avoid having transaction boundaries being at the same row-indices across all rows, as this would make designs that hit these exactly could run much better than what they would in general. The "Type f64" and "Type i8" have no large dataset. Note that the very large datasets are only for use with the RTX 2080TI as the other GPUs do not have space for this. Additionally note the the very large datasets are not cubes in 3D like the smaller sizes.

Additional information about the benchmarks:

- The benchmarks are performed using the setup shown in Table 7. For the GTX 780TI and GTX 950 GPUs we compute the average runtime of 80 runs for each stencil. For the RTX 2080 TI we compute the average runtime of 1000 runs for the small and large datasets. For the very large datasets we compute the average runtime of 300 runs. We omit the initial 30 runtimes for all cases, since the runtime is usually more unstable at the initial runs.
- The group sizes used for benchmarking are 256 and 1024 for the GTX 780TI and GTX 950, where we perform 80 runs for each. We only show the best results between these two group sizes, and this is for both the stencil-construct and the reference nested-maps construct. For the RTX 2080TI we only benchmark with a group size of 256.

- The Rodinia benchmarks actually have datasets associated with them, but we chose to generate our own as we wanted multiple dataset sizes. The data for the datasets are all generated with the same seed (1337). As an example, we generate the dataset with dimensions `[255][255][255]f32` with the command below. The dataset is generated into binary code, as this takes noticeably less space and time to process. However if one wishes to get the datasets in text format then replace the option `-b` with `--text`.

```
$ futhark dataset -s 1337 -b -g '[255][255][255]f32' \
    > datasets/255x255x255xf32.bin
```

- The benchmark programs are compiled (into CUDA) using the command below. The `.execuda` suffix has no special meaning, as it is simply used to create distinct names for the `OpenCL` executables and the `CUDA` executables. The `program.fut` is a placeholder for the program one wishes to run and `dataset/data` is a placeholder for the dataset of choice.

```
$ futhark cuda -o program.execuda program.fut
```

  The benchmark can then be run using the following command[16]:

```
$ cat datasets/data | ./program.execuda --default-group-
   size=256 -b -r 80 -t program_data.txt > /dev/null
```

  The executables will then write the measurements to a file `program_data.txt` and discard the result of the program into `/dev/null`.

- The sizes of these datasets leads to a relatively small runtime for the RTX 2080TI. With a 3D dataset (255x255x255=16,581,375 elements) the fast GPU can compute stencils (gradient and type-i8) in 271 and 342 microseconds, respectively. Therefore to reduce the runtime variance of the RTX 2080TI, we ran a large number of runs. The large data-set is (511x511x511=133432831) elements, which for the f32 data type is a size of 0.5GB per array and we need at least 2 (input/output). Perhaps even more if we the compiler introduces a larger memory requirement. The older GPUs have a noticeably smaller amount of memory available (3GB and 2GB for the GTX 780TI and GTX 950, respectively) therefore we use a relatively small dataset size compared to what the RTX 2080TI could handle.

- For the benchmarks on types `f64` and `i8`, the default shared memory behaviour in CUDA is that each successive 4 bytes of shared memory are mapped to different memory banks. Some Nvidia GPUs have the option of changing this to every 8 successive bytes. The performance of

[16]For the CUDA executable and group size of 256

the memory banks is more efficient when each thread in a warp writes to a different bank. This is however an issue for i8 (and f64 if one does not change the settings), since four (or two for f64) threads in a warp will access the same bank. This means that the shared memory performance of these types are sub-optimal. However, if one could change the setting to every 8 successive bytes, then one can go without this issue for the f64 type.

The plots Below we see the results of the benchmarks for the Futhark stencil construct vs the reference implementation (Futhark nested maps):

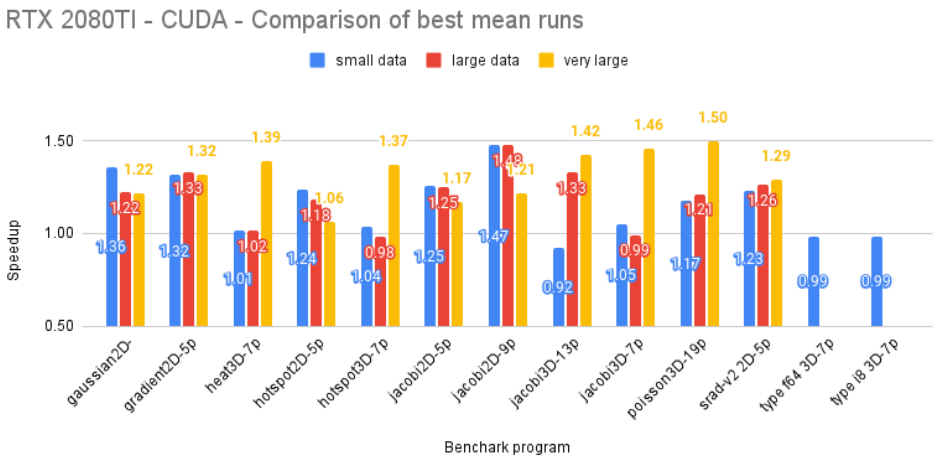


Figure 48: The speedups for the stencil construct vs. reference. The speedups are shown for various common stencils, based on the best average runtime between using a group size of 256. These benchmarks are performed on the GPU RTX 2080TI using the Futhark CUDA back-end

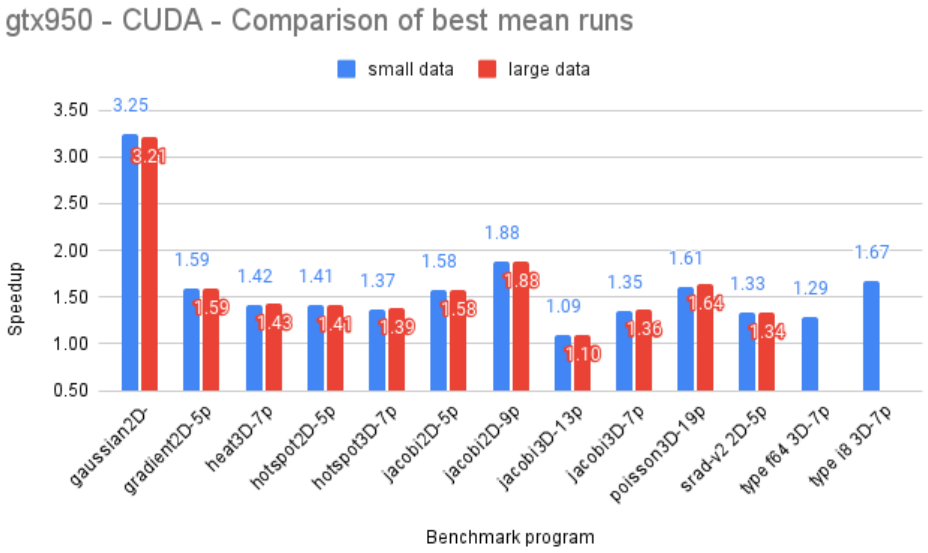


Figure 49: The speedups for the stencil construct vs. reference. The speedups are shown for various common stencils, based on the best average runtime between using a group size of 256 or 1024. These benchmarks are performed on the GPU GTX 950TI using the Futhark CUDA back-end

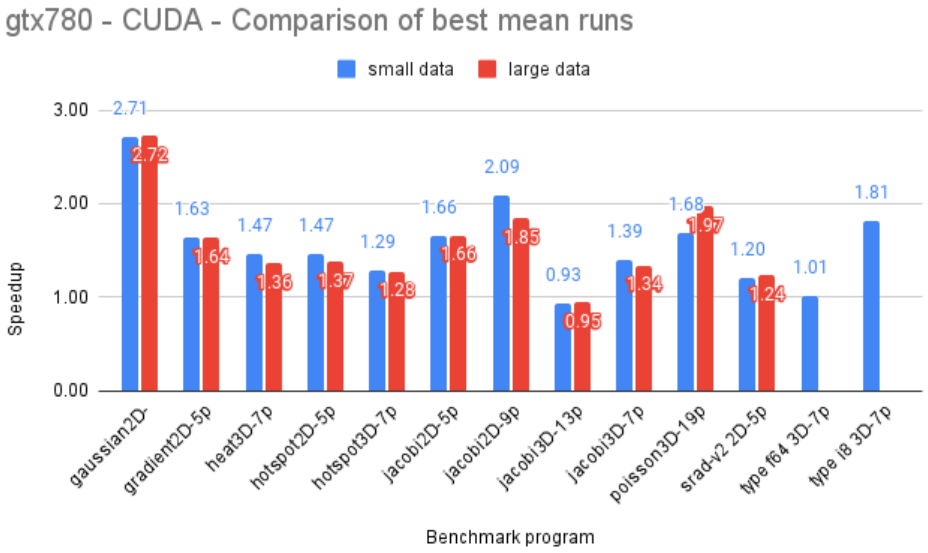


Figure 50: The speedups for the stencil construct vs. reference. The speedups are shown for various common stencils, based on the best average runtime between using a group size of 256 or 1024. These benchmarks are performed on the GPU GTX 780TI using the Futhark CUDA back-end

Discussion The Futhark benchmarks in Figures 50, 49, and 48 show a number of general tendencies, as well as some specific things. In general, the older GPUs have a larger speedup compared to the newest GPU (RTX 2080TI). In some cases we also have worse performance than the reference solution. When it comes to specific details of the individual plots, we can see a number of things.

1. The plots in figures (48 and 50) have speedups factors for certain benchmarks which are below one, meaning these are slowdowns. This means that the threshold, for when we should fall back to the *GlobalRead* design, is not conservative enough.

2. The very large dataset with the RTX 2080TI has good speedups on all of the benchmarks, which shows that the design will scale well on datasets of a very large size (here $511 \cdot 1023 \cdot 1023 = 534,776,319$ elements). This dataset was not run on the older GPUs since the total size of the input- and output-arrays are $(534,776,319 \cdot 4) = 2.1GB$ per array.

3. Based on the Futhark benchmarks of the older GPUs (GTX780 and GTX950 in Figures 50 and 49) the 2D stencils with many points (Gaussian2D with 25 points, and Jacobi2D with 9 points) perform well since there is a lot of potential for reuse (due to the large number of points).

4. The GTX 780 is limited by GPU resources as seen in Figure 50 for the Jacobi3D-13p stencil. This suggests that we need better safeguards to ensure that multi-write big-tile will not be executed under such conditions.

5. The RTX 2080TI only get speedups for some stencils when the input dataset is large enough as seen in Figure 48. This could be due to the fact that the cache system is efficient enough, such that using shared memory only adds overhead. An interesting point for future work would be to implement a multi-write global-read version. Then one could see the effects of an increased instruction level parallelism without any tiling with shared memory.

6. The type f64 stencil have a slight slowdown compared to the reference implementation on the RTX 2080TI. This is not a surprise since we discovered that the f64 arithmetic throughput per cycle is quite low.

7. It is not immediately obvious why the performance decreases as the dataset grows larger for the Hotspot2D-5p stencil. However, as we have seen before, the performance can be impacted by the arithmetic throughput. The Hotspot stencil involve many heavy computations such as multiplications and divisions. However, that does not explain why the runtime increases for Hotspot3D.

10 Conclusion

10.1 Related work

There have been other research with regards to optimising stencils. Cecilia et al. [1] has optimised Jacobi2D stencils for CUDA, by loading from global memory into shared memory, where they also use smaller group sizes to increase performance. This is similar to some of the things, that we have done with regards to increasing reuse, but also improving latency hiding of the GPU. Schäfer and Fey [10] investigates various GPUs with respect to memory bandwidth and floating point throughput, and states that floating point arithmetic can be a limiting factor if the floating point throughput is not high enough. We saw similar results with respect to our benchmarks of double-precision floating-point numbers on the RTX 2080TI. Furthermore, they implement a method for computing Jacobi3D stencils. As an important factor to increase throughput they suggest to increase instruction level parallelism (ILP). They increase instruction level parallelism by computing multiple elements per thread, similar to what we have done with our multi-write big-tile implementation. As an alternative to our method, they iterate across the outermost z-dimension (for 3D) with a sequential loop to optimise for cache performance. Then registers

are used to store some of the dependencies of the stencil computation, and shared memory is used for updating values. Values that are not in registers, but are used for computations are loaded directly from global memory. They optimise for computing multiple steps of an iterative stencil, which is a form of stencil fusion, where each iteration of the Jacobi3D stencil is fused. However, our implementation only optimises for a single iteration. We also avoid using more registers than necessary for storing dependencies used in the stencil computations. A Futhark programmer can make arbitrary stencils, and we do not want to exceed the maximum number of registers available. The field of generating GPU code for stencils have been researched by others and has led to some domain-specific languages (DSLs). There exists a DSL called LIFT. Hagedorn et al. [3] states that LIFT can generate high-performance stencils for back-ends such as CUDA and OpenCL, which are also the back-ends we target. The LIFT stencils does also utilise tiling, multiple elements per thread, and adjustment of the group size. Their approach is similar to our code generation. An advantage of LIFT, would be that all the parameters can be autotuned which is not yet possible for our stencil construct. However, we did an extensive evaluation in order to set some sensible default parameters. We attempted to download and install LIFT, in order to compare our Futhark stencil performance to the LIFT stencils. However, the installation guide and build scripts were not up-to-date with newer Linux-based systems. Another DSL presented by Ragan-Kelley et al. [8] is a DSL which is concerned with image processing that consist of stencil pipelines. In Halide, one can specify a pipeline for various stencils and a schedule specifying how the pipeline should be executed. This can further complicate the process of designing a program, however, it can provide significant improved performance. Then, if a programmer is only concerned with performing a pipeline of stencils, one might have better results using Halide. In relation to this thesis we also have the Futhark compiler, which is still in development. Among many things, the Futhark compiler has been extended with incremental flattening for nested data parallelism by Henriksen et al. [5]. The feature adapts nested-parallel applications written in Futhark to hardware specifications and dataset characteristics. The compiler essentially generates all possible mappings of parallelism for a nested-parallel program and chooses the best mapping based on hardware and dataset.

10.2 Future work

- To implement autotuning, such that a Futhark programmer can autotune their program. The autotuner should find the optimal number of output elements per thread and group size for some combination of GPU and stencil. An important aspect is that the elements per thread for each dimension should still be represented as CUDA/OpenCL compile-time constants.

- To investigate the performance impacts of using different shapes for the multi-dimensional groups when computing 2D and 3D stencils. The shape of a multi-dimensional group could also be part of an autotuning extension. Instead of simply searching for the optimal group size, it could also try different shapes for the multi-dimensional group.
- Many of our design decisions and evaluations are based on using Nvidia GPUs. However, we might have made different decisions if we had also used e.g. AMD GPUs for our thesis. It would be interesting to further investigate our implementation with different hardware architectures in mind, and benchmark Futhark stencils on AMD GPUs.
- In the tiled designs, considering conditioning the max/min of whether we are in one of the groups on the edge of the grid, or rather if any 1 thread in the group need to do any bounding.
- Further investigation of how to optimise stencils that are computed on input arrays where the elements consists of double-precision floating-point numbers. Perhaps it would not be realistic to make optimisations for code generation when limited by floating point arithmetic throughput.
- To implement more sophisticated methods and safeguards for selecting which generated stencil kernel to run, based on GPU hardware properties.
- Stencil fusion on either iterative stencils, or distinct stencils. Both would move a lot of reads/writes from global memory to shared-memory, which is a lot faster. Implementing these designs is however a difficult exercise, which would require a noticeable amount of tuning for the stencil fusion operation.
- Implement a multi-write global-read version, that processes multiple elements per thread and does not use tiling or shared memory. Then investigate the performance. If the results are promising, one could also implement autotuning for setting the optimal elements per thread for this version. Hagedorn et al. [3] refers to a non-tiled version that processes multiple elements per thread, which outperforms a multi-write tiled version using shared memory for a heat3D stencil.

10.3 Final remarks

We have implemented the stencil construct such that it is supported in multiple areas of the Futhark compiler. This includes function type-specifications (for 2D and 3D), the Futhark interpreter (for 2D and 3D), sequential C code generation, and OpenCL/CUDA code generation. The implementation of the

OpenCL/CUDA code generation is based on a prototype we deemed most robust and efficient for common stencils. The prototype has some weaknesses in terms of resource usage. It might use too many registers for some stencils, however, this is not the case for the common stencils that we benchmarked our implementation against. Using our new stencil construct will provide speedups compared to the old way of computing stencils in Futhark. However, there are some exceptions. If the cache system of the hardware is large and efficient enough, then the input size to the stencil has to be relatively large in order to see a speedup. Another limiting factor for speedup can be the GPUs that are bounded by the arithmetic throughput for some data types.

The implementation of the stencil construct was validated for correctness with tests for both the generated C code and the generated OpenCL/CUDA code.

References

[1] J. Cecilia, J. Garcia, and M. Ujaldon. Cuda 2d stencil computations for the jacobi method. volume I, pages 173–183, 06 2010. doi: 10.1007/978-3-642-28151-8_17.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. 2009.

[3] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 100–112, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356176. doi: 10.1145/3168824. URL `https://doi.org/10.1145/3168824`.

[4] T. Henriksen. *Design and Implementation of theFuthark Programming Language*. PhD thesis, University of Copenhagen, Datalogisk Institut, 2017.

[5] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 53–67, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295707. URL `https://doi.org/10.1145/3293883.3295707`.

[6] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, page 51–60, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595935789. doi: 10.1145/1178597.1178605. URL `https://doi.org/10.1145/1178597.1178605`.

[7] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 235–244, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250761. URL `https://doi.org/10.1145/1250734.1250761`.

[8] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN*

Not., 48(6):519–530, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462176. URL https://doi.org/10.1145/2499370.2462176.

[9] G. Roth, G. Roth, J. Mellor-crummey, J. Mellor-crummey, K. Kennedy, K. Kennedy, R. G. Brickner, and R. G. Brickner. Compiling stencils in high performance fortran. In *In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM*, pages 1–20. ACM Press, 1997.

[10] A. Schäfer and D. Fey. High performance stencil code algorithms for gpgpus. *Procedia CS*, 4:2027–2036, 12 2011. doi: 10.1016/j.procs.2011.04.221.