Utrecht University

Independently Computed Regions in a Data Parallel Array Language

Martijn Fleuren ICA-5666163

January 25, 2020

"It took an hour to write, i thought it would take an hour to read!" — Philip J. Fry

Abstract

Programmers who are involved in large numeric computations such as simulations, image- and signal processing and machine learning are in dire need for the highest performance. The most often used functions represent map, reduce, multiply-accumulate, stencil computations or other functions that are cheap on their own. Industry has responded with various specialised hardware to execute these functions using 'single-instruction multiple data' parallelism. Users of this hardware often have to resort to libraries written in low-level programming languages, provided by the manufacturers, to obtain the performance they seek. Without proper abstraction it becomes very easy to make mistakes. Accelerate is an embedded domain specific language in Haskell that provides the right abstractions for high-performance, shape-polymorphic array computations. This work will explore a possible solution for a shortcoming in the expressivity of Accelerate when implementing a simulation consisting of partial differential equations with initial conditions that depend on boundary specifics e.g. the index. We achieve this by adding expressivity to subdivide an array into tiled, independently executed *regions*. The subdivision in regions studied in this thesis had its original motivation in the treatment of boundary conditions, but the technique can be fruitfully applied to a variety of other scenarios.

Contents

Chapter 1

Introduction

Programmers who demand the best performance of their programs may find themselves trapped between two worlds.

The first is the world of abstraction, where programs are written in a way that clearly separates the inherent algorithm from the schedule i.e. *what* is computed and *how* it is computed. Languages that reside here are most functional languages such as Haskell, and Futhark. This world comes with disadvantages e.g. control over the schedules and direct memory management are often impossible. On the other hand, we get intelligibility of the code, better maintainability and properties that aid equational reasoning such as referential transparency.

The second is the world of absolute control, where programs and schedules are interleaved and where hardware specific functions are used directly. This is the realm of many imperative languages such as C and Fortran, languages that can be compared to a Swiss army knife: while being amazing and versatile tools when used properly, you can also accidentally cut yourself. We get direct access to (virtual) memory and input/output (I/O) via peeking and poking pointers, and can use the libraries provided by manufacturers to interface with their hardware directly.

Interfacing with hardware requires expertise and intimate knowledge about specific architecture. We know that we can make hardware agnostic high performance computing more accessible by providing the right abstraction[13, 2]. This is desirable, because not everyone that wants a very high performance, also has the time or the background to become and expert in their own hardware.

Accelerate is an embedded domain specific language (EDSL) in Haskell that provides this abstraction. By specialising the language to the single domain of parallel array programming, we can translate assumptions into high performance. Accelerate is a deeply embedded language, which allows for certain optimisations and a modular code-generator backend of which we have two, both based on LLVM [9].

Accelerate provides specialised patterns such as `map` and `fold`, as well as a more general `generate`. While specialised patterns can never be complete. We aim to provide those that are used frequently because they can be better optimised. We address the case of a system in which different computations are performed on different sections of an array. Currently, we have to resort to checking the index in order to select the computation we need. This implementation of chained conditional expressions is not only inelegant but is also inefficient: if there is a dependency between different cells in

the array that we cannot satisfy directly, we have to do a second pass over the array to fix all values that were incorrectly computed.

We will introduce a new concept to Accelerate called *regions* with which we attempt to solve the problems of inelegance and inefficiency simultaneously. Regions are independent, hyper-rectangular subsets of an array that are treated different from the rest. This way, we attempt to clearly separate what is being computed and where from how regions are set up.

Accompanying the regions we will add another new concept to Accelerate to increase performance when using regions. *unguarded stencils* carry the assumption that all accesses into the array will stay within bounds, so we can omit all boundary checks and conditions.

We will run benchmarks to verify whether these extensions make a difference and present 'readability benchmarks' to see whether code with the new extensions is clearer than code without.

Chapter 2

Preliminaries

2.1 Accelerate

Accelerate is a domain specific, data-parallel and collection oriented language embedded in Haskell. Each of these concepts will be explained in the following section.

2.1.1 Domain Specific Languages

A domain specific language (DSL) is a constrained language that is tailored to a specific domain. In the case of Accelerate, this domain is parallel array programming. Examples of other languages that are domain specific are *SQL* for querying data from relational databases, *SVG* for drawing vector graphics and *dot* for describing graphs. Examples of constraints in these languages are that *SQL* has no user-definable functions, *SVG* has no loops and *dot* cannot do arithmetic. These languages facilitate intuitive reasoning and problem solving within their domains, but not necessarily outside their domains.

Accelerate's domain is parallel array computations. This is an often painful domain to work with in imperative languages because one needs to take care of low-level details that are irrelevant to what one is trying to compute. For example, an implementation of any image processing algorithm in an imperative language is littered with indices, bound checks, accumulators and nested loops, often heavily dependent on the target hardware architecture. These considerations are necessary in languages such as C, but have little to do with the computation itself and can be derived by a compiler instead. This is the core of what Accelerate attempts to provide: let the programmer worry about *what* is computed and derive *how* a result is computed for a given hardware platform.

Using facilities from the host language is also known as *piggy-backing*

A DSL can be embedded in another programming language. This has several benefits, such as allowing us to use facilities — such as variable assignment — from the host language. For example, Accelerate piggy-backs on Haskell's type-checker to achieve type-safety. Embeddings may be *shallow* or *deep*. A shallow embedding provides a set of functions, usually in the form of a library, that directly manipulate entities of the domain.

In contrast to a shallow embedding, which typically only has a single meaning, a deep embedding can have multiple meanings or *semantics*. This is facilitated by leaving the interface the same, but instead of working with the semantics directly, an abstract syntax tree (AST) is built up which can then be evaluated, transformed and manipulated

in any way we deem fit. This freedom allows us to perform different sorts of analyses over the DSL code, such as optimizations and security-related analyses, but comes at the cost of a more complex compiler.

2.1.2 Parallelism

Parallel programs perform the same task in a shorter amount of time, and being faster is always desirable. Writing parallel programs is hard work, because no benefit comes without the cost of extra problems to overcome. These problems can be inherent to parallelism — such as competing for resources — or investing into specialised hardware. Parallelism also comes in different flavours. Accelerate embraces data-parallelism, which contrasts to task-parallelism. We will discuss both flavours briefly.

In the **data-parallel** model we run the same program on on a collection of data synchronously. In practice, the data is often split equally between the available processors and merged after the computation is complete[1]. Performance increases in this model are larger than in the task-parallel model, with performance gain factors approaching the number of available processors. The performance gain is in practice diminished by overhead resulting from splitting and merging.

Splitting data efficiently often relies on the data being laid out in memory in some optimal way e.g. for two dimensional arrays this often means row-by-row or *row major order*. If data is scattered throughout memory instead, splitting overhead will grow. Some architectures have extensions that can efficiently read data from arbitrary locations in memory to mitigate this problem.

row-major order is "zig-zag from the top left to the bottom right"

One example of a type of data-parallel computer is the single instruction, multiple data (SIMD) computer. All processing units execute the same instruction on a given clock cycle, and each processing unit operates on different elements of data. It is well suited for specialised problems that are very regular, such as image processing.[1] SIMD devices often have large register files which allows for cheaper context switching.

context switching is saving the state of a thread A and switching to another thread B, so that A can be resumed later.

In the **task-parallel** model, different tasks may be executed on the same or on different data elements. If the tasks are different then execution must be asynchronous. For example: with data-parallelism, we can share the program counter over n processors and amortise the cost of setting up the execution threads. Asymptotically we will get a performance gain factor of n. In the task parallel model we also have to pay the overhead, but cannot share the program counter, causing a performance gain factor of less than n.

Accelerate is data-parallel because (i) we can achieve a higher performance gain; (ii) we are working with arrays where each cell is essentially independent and computations are regular; and (iii) modern computers almost always have multiple processors with SIMD extensions available to them.

2.1.3 Language Features

Collection-oriented languages natively support composite data structures — lists and arrays — and the functions that manipulate them e.g. `concat` and `map`. Indexed languages manipulate structures element-wise by specifying indices. C is a good example of an indexed language because it has native support for arrays, but manipulation of arrays,

other than accessing and assigning, is not included in the language. Instead, programmers will have to write these functions themselves.

Accelerate is a collection oriented language because it allows us to reason about our code on a higher level of abstraction. Another good example of a comparable collection based language is Futhark. Note that in both languages, we do have the option to explicitly index arrays, but this is avoidable in many cases.

As an example we will demonstrate how to implement the summation of a vector of numbers. In Futhark, this is implemented with

```
let sum (xs : [f32]) = reduce (+) 0 xs
```

and can be similarly constructed from primitives in Accelerate as follows

```
sum :: Acc (Array (sh:.Int) Float) -> Acc (Array sh Float)
sum = fold (+) 0
```

In C, we would write this function with a single loop and an accumulator. Every time we use a slightly different data structure e.g. lists, we would have to provide a re-implementation. Higher order functions and type classes make this easier in a language like Haskell.

Being collection oriented and automatically data-parallel are desirable properties for a language that attempts to take control over the details of a computation, but it all comes at the cost of assumptions about the execution model, the hardware and a complex compiler. For example, abstraction over the target hardware platform is achieved by different code-generator backends. Two code-generation backends, based on LLVM are currently available: one for multi-core CPUs, and another for GPUs.

Accelerate represents arrays as *delayed arrays*, which contrast with *manifest arrays*. Manifest arrays are existing blocks of memory. Elements of manifest arrays are directly accessed by specifying an index. An example of a language with manifested arrays is C. We can allocate an array with `malloc`, which gives us the pointer to the block containing memory artefacts, assign cells with `A[x] = y` and read cells with `a = A[b]`.

Delayed arrays are conceptual. Instead of being manifested in the memory, they are represented by an access function

```
! :: ix -> a
```

where `ix` is the index type and `a` the element type. A commonly chosen type for `ix` is `Int`, but this is not necessary. Delayed arrays have the benefit that the access function can be composed with other functions, that change how the array is accessed. This has the advantage that it enables the fusion transformation, i.e. the combination of array-transforming and array-consuming functions to avoid intermediate data copying and save on storage. An example of this is when a large array is transposed, we can push the index transformation function into the consumer function and then use the original array[7, 3].

2.1.4 Writing a program in Accelerate

Let us consider an example program to illustrate how to use Accelerate. We will select the dot product for this purpose because it is a simple, well known algorithm. Recall from linear algebra that

$$\text{dotp}(\vec{p}, \vec{q}) = \vec{p} \cdot \vec{q} = \sum_{i=0}^{N-1} p_i q_i, \tag{2.1}$$

for two vectors $\vec{p}$ and $\vec{q}$ of length N. In Accelerate the dot product is written as shown in Listing 1. The function `dotp` takes two vectors and returns a scalar. `Vector` and `Scalar` are type synonyms for `Array DIM1` and `Array DIM0` respectively. `Acc` indicates that the inputs and outputs are already embedded in Accelerate, this means that they are not evaluated in Haskell but on the target backend.

```
1 dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Vector Float)
2 dotp xs ys = fold (+) (zipWith (*) xs ys)
```

Listing 1: The dot product between two vectors expressed in Accelerate.

In regular Haskell we would write this function in almost the same way as in Accelerate i.e. with `foldl` and `zipwith`. Writing the dot product like this is definitely more compact than writing the same in C, or by using intrinsics, but we are still able to compile efficient code[2, 10, 9]. `fold` and `zipWith` have highly parallel semantics and support as many threads as there is data.

Intrinsics are hardware specific instructions and functions.

To illustrate the meaning behind different types and classes we will discuss the type of `fold` in Listing 2.

```
1 fold
2   :: (Shape sh, Elt e)
3   => (Exp e -> Exp e -> Exp e)
4   -> Exp e
5   -> Acc (Array (sh :. Int) e)
6   -> Acc (Array sh e)
```

Listing 2: The type of `fold` in Accelerate

With the `Elt` and `Shape` type classes, we ensure that the type variables `sh` and `e` can be used to denote the shape and an element respectively. Shapes are lists of `Int`, formed with `(:.)` and terminated with a nil value of `Z`. For example, `Z :. Int` denotes a one dimensional shape.

The zero dimensional base case is `type DIM0 = Z`. A scalar is the only array of this type that can be constructed with this shape. In higher dimensions we can specify the length along that dimension by specifying a value for the `Int`[2, 7]. In contrast to the dimension, this value may not be statically known.

For example, functions such as

```
sum :: Shape sh => Acc (Array (sh :. Int) Float) -> Acc (Array sh Float)
```

are constrained by the type to only accept arrays that are at least one dimensional. The type checker of the Glasgow Haskell Compiler (GHC) will statically ensure this property

because `Z` does not unify with `Z :. Int` or any higher dimension. The `Shape` type class further constrains this function by enforcing that `sh` must be a shape type, but it also allows *shape polymorphism*. The example we just gave will work for any array that has a dimensionality of at least one. Shape polymorphism further increases the re-usability of the code.

2.2 Scientific Computation

Scientific computations are often sets of partial differential equations, which may not have analytical solutions. A commonly used method to approximate solutions to these is the finite element method (FEM). With this method we iterate a function from a known initial state until a stop condition is reached. Efficiency of these functions is of great importance, because both the number of elements and the number of iterations are typically large.

For example consider a rigid, rectangular steel beam intended to be fixed at both ends and mounting a load at the centre point. We are interested in the microscopic static properties — nothing is in motion — of this beam with a given load to see how it will bend or when it will break. Instead of actually testing the situation and wasting many beams, we simulate it. To approximate the microscopic forces in the beam we assume that initially, the beam is in equilibrium and the external force is only applied to a single cell in the centre. In the next step, this force propagates to the adjacent cells because they have to deliver the force to maintain the equilibrium by Newtons third law of motion $F_a = -F_r$. We continue this process until we no longer observe significant changes to the internal forces between iterations.

With this example we want to illustrate that 'scientific computation' commonly translates to iterating a state transformer function over an array. The beam can easily be mapped to a three-dimensional array with a resolution in cells per millimetre. The initial conditions translate directly to initial values for the array, and changes to cells between iterations only depend on the values of that cell and its neighbours. Accelerate has a native feature that is very well suited for the latter requirement: stencil computations.

2.2.1 Stencil Computations

Stencil computations are computations that allow the programmer to use the values of neighbouring cells, or a *local context*, to compute the next value. A special stencil computation is the weighed sum of a local context or a *convolution*, denoted by an asterisk $I * \mathbf{M}$. These weights are typically constant for the computation and define the entire operation. Two examples of convolution filters are shown in Equation 2.2. The identity function returns the image unchanged, while mean blur averages the nine neighbours. More on convolutions can be found in Appendix A.

$$\text{identity}(I) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} * I \quad \text{meanblur}(I) = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} * I \tag{2.2}$$

This allows us to express simulations such as the steel beam example more easily but also has a couple of caveats:

1. When we perform a stencil computation at the boundary, it always occurs that the stencil needs to access a cell with a non-existing index. An example of this situation is shown in Figure 2.1. To get around this issue we facilitate a variety of *boundary conditions*. Every time a stencil computation indexes an array, we have to test whether the index is in-bounds. Accelerate supports all common boundary conditions:

   (a) Simply returning a **constant**.

   (b) Assuming an array of size L, when a cell at index i is accessed and not in range then return index $L - i$. It is as if we are **wrap**ping around. This boundary condition is also known as tile.

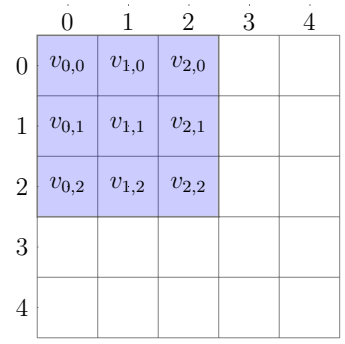
   (c) Assuming the same array and index, return $-i$, as if a **mirror** has been placed on the edges.

   (d) Assuming the same array and index, return 0 if $i < 0$ and $L - 1$ if $i \geq L$, **clamp**ing the values to the boundaries. Also known as "extending the boundary cells towards infinity"

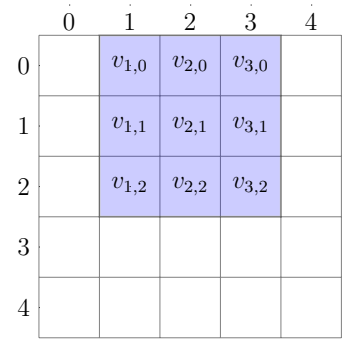
   (e) Assuming a source array of type `Array sh a`, perform some arbitrary **function** `f :: Exp sh -> Exp a`. All the aforementioned boundary conditions can be implemented using this one.

2. The same boundary condition is run on all boundaries and can only *produce* values that the stencil function can consume. There is no way to bypass the boundary function and produce values for the result immediately.

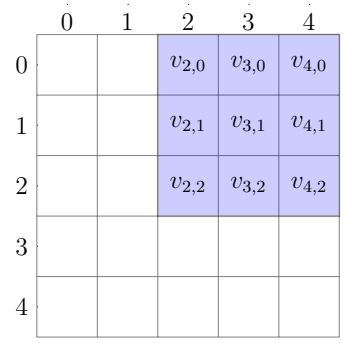
3. Stencils run on every cell. Because the value of a cell has — loosely speaking — been in every position in the stencil once finished (e.g. in a 3×3 stencil we expect a cell to be accessed 9 times), we are sensitive to having the right data in the cache at the right time to avoid expensive memory traffic i.e. we need *cache hits*.



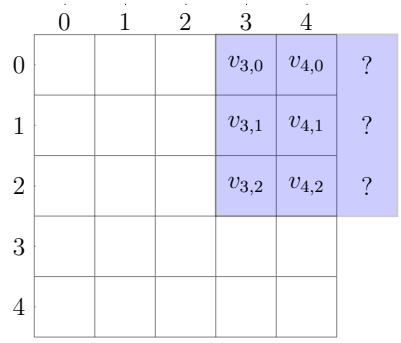
(a) Iteration 1, focus on $v_{1,1}$



(b) Iteration 2, focus on $v_{2,1}$



(c) Iteration 3, focus on $v_{3,1}$



(d) Iteration 4, focus on $v_{3,1}$, now the right side of the stencil is out of bounds.

Figure 2.1: Why boundary conditions are required. The question marks denote the positions for which we have to synthesise a value.

We will refer to cells that belong to the boundary with *"shell"* while the converse will be called *"core"*

Although testing whether an index belongs to the boundary does not change the complexity of the computation, it is relatively expensive. Most cells do not belong to the boundary i.e. the number of cells belonging to the boundary is insignificant to the total amount of cells. **Proof**: in a hyper-rectangular array of D dimensions where D_i is the size of the ith dimension, the total number of cells $T = \prod_{i=1}^{|D|} D_i$. The number of cells in the core, assuming a boundary size of s, is $I = \prod_{i=1}^{|D|} (D_i - 2s)$. Because $T = I + S$ where S is the number of cells in the shell, we can state that the relative amount of boundary cells is $\frac{B}{T} = \frac{T-I}{T} = 1 - \frac{I}{T}$, and can alternatively be written as

$$\frac{S}{T} = 1 - \frac{I}{T} = 1 - \prod_{i=1}^{|D|} \frac{D_i - 2s}{D_i}, \tag{2.3}$$

The shell becomes an insignificant part of each dimension as the dimension grows, for any finite shell size s we have that

$$\lim_{\forall D_i \in D : D_i \to \infty} 1 - \prod_{i=1}^{|D|} \frac{D_i - 2s}{D_i} \to 0. \tag{2.4}$$

Almost no cells belong to the shell, and so every time an index belonging to the core is tested for shell membership, it is a waste of time. For example, if we have a 4k image

```
generate    :: Exp sh -> (Exp sh -> Exp a)
            -> Acc (Array sh a)
fold        :: (Exp a -> Exp a -> Exp a) -> Exp a -> Array (sh:.Int) a
            -> Acc (Array sh a)
map         :: (Exp a -> Exp b) -> Acc (Array sh a)
            -> Acc (Array sh b)
zipWith     :: (Exp a -> Exp b -> Exp c) -> Acc (Array sh a)
            -> Acc (Array sh b) -> Acc (Array sh c)
permute     :: (Exp a -> Exp a -> Exp a) -> Acc (Array sh' a)
            -> (Exp sh -> Exp sh') -> Acc (Array sh a)
            -> Acc (Array sh' a)
backpermute :: Exp sh' -> (Exp sh' -> Exp sh) -> Acc (Array sh a)
            -> Acc (Array sh' a)
stencil     :: stencil -> Boundary (Array sh a) -> Acc (Array sh a)
            -> Acc (Array sh b)
stencil2    :: stencil -> Boundary (Array sh a) -> Acc (Array sh a)
            -> Boundary (Array sh b) -> Acc (Array sh b) -> Acc (Array sh c)
```

Listing 3: Summary of the types of our most commonly used functions.

that we want to blur with a 3×3 mean blur (shell size of 1 cell) then only 0.14% belongs to the shell, but we still have to test the other 99.86%.

2.3 Summary of the Accelerate interface

In Section 2.1.4 we have provided a small example which sufficiently demonstrates how to write a basic program and what different types mean. The example was however insufficient to demonstrate every function that Accelerate has to offer. Because we do not want to provide detailed examples for each, we will provide a list of our most commonly used concepts in Listing 3.

- To generate a new array we can use `generate`. The first argument is the desired shape, and the second argument specifies how a value has to be created from the index. For example, creating a zero-filled vector of length 10 is done with `generate (Z_ ::. 10) (\_ -> 0)`.
- Reducing an array along its outermost dimension can be achieved with `fold`, by using a function specified by the first argument and a default value specified by the second. For example, a sum can be expressed with `fold (+) 0`.
- Unary functions can be applied to each value in an array with `map`. For example, adding one to every value is done with `map (+1)`.
- For binary functions there is a special version of `map`: The function `zipWith` takes a binary function specified by its first argument. Arrays should be the same size in all dimensions, for if they are not, the larger dimension is truncated. For example, taking the pairwise sum of two arrays is done with `zipWith (+)`.
- Scattering of arrays is done with `permute`. The result is initialised with values from the second argument and any values that are permuted into the result by the

third argument are combined with the current value in a way that is specified by the function in the first argument.

- Gathering of arrays is done with `backpermute`. Its first argument specifies the size of the result, and the second a function that maps indices to the original array from indices in the target array. For example, if we assume a vector `a = [1,2,3,4]` then we can flip the vector with

```
let Z_ ::. len   = A.shape a
    f (Z_ ::. x) = Z_ ::. (len - x)
 in backpermute (A.shape a) f a
```

  and get `[4,3,2,1]`.

- A special purpose version of map, which includes a local context, is `stencil`. Its first argument, `stencil`, is instantiated to a unary function of nested tuples to a single value, depending on the size and the dimensions of the stencil. The second argument specifies a boundary condition that is used to synthesise values if we access indices that are out of bounds. For example, a local sum in one dimension is implemented with

```
let f (a, b, c) = a + b + c
 in stencil f (Constant 0)
```

- For binary maps with local context we have `stencil2`. It compares to `stencil` in the same way that `zipWith` compares to `map`. For example, a pairwise sum (same as `zipWith`) can be implemented with

```
let f (_, x, _) (_, y, _) = x + y
 in stencil2 f (Constant 0) arr1 (Constant 0) arr2
```

  Note that we use a stencil function, but simply disregard the local context in this example.

2.4 Related Work

We will discuss a range of languages which provide a similar programming model as Accelerate to the user. To make the conceptual and syntactical differences more clear, we have provided a mean blur example to each language. An implementation in Accelerate is shown in Listing 4.

2.4.1 Futhark

Futhark is a purely functional and data-parallel array language that offers a hardware independent programming model and optimising compiler that generates code for graphical processing unit (GPU)s[6]. Futhark combines the advantage of functional and imperative features. The type system supports equational reasoning and guarantees the safety of in-place updates, with conservation of referential transparency. Futhark is similar to

```
blur :: Num a => Acc (Matrix a) -> Acc (Matrix a)
blur = stencil f clamp
  where
    f ((nw, n, ne)
      ,( w, m,  e)
      ,(sw, s, se)) = (nw + n + ne + w + m + e + sw + s + se) / 9
```

Listing 4: 3×3 mean blur implementation in Accelerate for comparison to other languages

```
let stencil[rows][cols]
             (image: [rows][cols]f32) (row: i32) (col: i32): f32 =
  unsafe
  let sum =
    image[row-1,col-1] + image[row-1,col] + image[row-1,col+1] +
    image[row,  col-1] + image[row,  col] + image[row,  col+1] +
    image[row+1,col-1] + image[row+1,col] + image[row+1,col+1]
  in sum / 9f32

let meanBlur [rows][cols] (channel: [rows][cols]f32): [rows][cols]f32 =
  map (\row ->
    map (\col ->
       if 0 < row && row < rows - 1 && 0 < col && col < col - 1
       then stencil channel row col
       else channel[row,col])
    (iota cols))
  (iota rows)
```

Listing 5: 3×3 mean blur implementation in Futhark, notice the lack of a stencil construct and how we get around this by using map over indices. The edges are clamped to the edge by using min and max.

Accelerate in that it is strictly evaluated, supports static shape and is monomorphic. With Accelerate we can write polymorphic functions using Haskell's type system.

The mean blur example in Futhark is shown in Listing 5. Because Futhark does not support stencil, we have to omit the collection-oriented approach and use indices instead. The indices are obtained by using the `iota` function, which returns the indices into the array it has been provided with. The `unsafe` elides all safety checks and assertions that occur during execution of the expression that follows it. This is useful if the compiler is otherwise unable to avoid bounds checks[5]. Using `unsafe` can lead to memory corruption. To protect the memory, we guard the boundaries with a conditional expression, and simply ignore the boundaries. Conditional expressions such as these are examples of precisely the ones we are trying to avoid.

2.4.2 Halide

Halide is an EDSL in C++ that focuses on very high performance and platform independence.

Manually fine-tuned C programs perform up to an order of magnitude faster than the equivalent vanilla C program and are often memory bandwidth limited even if they consist of many data-parallel stages. Manually fine-tuning C programs comes at a price of pain to the programmer, because it requires intimate knowledge about the target hardware and implementations can differ greatly across different architectures[12].

Halide's approach is to bias programs with knowledge about the target architecture. For example, a program written for two-dimensional image processing that will run on SIMD hardware may differ vastly from the same program that will run on a mobile phone.

This flexibility is facilitated by custom optimisation passes, modular code-generator backends and by exposing an application programming interface (API) to write a *schedule* – choices about storage and ordering of a computation – to the user. Accelerate does not offer a way to control the schedule.

Functions in Halide are written with a simplified functional style and are composable into 'image processing pipelines' that essentially form a directed acyclic graph (DAG) of interdependent computations that is later optimised. The representation of arrays in Halide is with delayed arrays that are similar to Accelerate[3], except that their index functions are variatric — one extra argument per dimension.

Halide demonstrates impressive results with this approach such as equal performance between two implementations of a Laplacian filter, one implemented by an x86 expert in months, another by an intern in one day using Halide[13].

```
1  int main(int argc, char **argv) {
2    Buffer<uint32_t> img = load_image("some_image.png");
3
4    Func blurx, blur;
5    Var x, y;
6
7    blurx(x, y) = img(x-1, y) + img(x, y) + img(x+1, y);
8    blur (x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/9;
9
10   Buffer<uint32_t> out = blur.realize(img.width(), img.height());
11 }
```

Listing 6: Implementation of a mean blur in Halide. We provide no schedule so it is inferred for us by the compiler. The function `blur` is automatically computed for all coordinates.

The mean blur example in Halide is shown in Listing 6. Note that there are two different functions: `blurx` and `blur`. The former blurs only in the horizontal dimension, by taking the sum of a value and its immediate neighbours. This function is then used from three distinct vertical indices to get a total of nine values, which are then averaged.

If we write the the mean blur in this way we actual have a two-pass function, but Halide is able to fuse this automatically into one single pass using the DAG. The call to `realize` computes this function on the specified domain, with the lower bounds left implicit.

2.4.3 Single Assignment C

Single Assignment C (SAC) is a functional language with C-like syntax aimed at array-based numerical computation with a run-time performance comparable to hand optimised Fortran programs. Accelerate and SAC have a very similar spirit: in both languages arrays are the only data type and functions are shape-polymorphic. In contrast to Accelerate, SAC comes with a stand-alone compiler. By default, this compiler does a source-to-source translation from SAC to C, but other backends such as a GPU backend are available.

SAC is designed to be a 'functional subset of C'. Stripping C of non-functional features essentially comes down to the elimination of side effects, most notably global variables, reference arguments and pointers[14]. SAC tries to exclude typical functional features – such as partial application and lazy evaluation – that make functional languages slow, and keep those that are desirable for parallel array processing e.g. referential transparency so that the result of evaluating expressions is insensitive to its context, and the Church-Rosser property so that these evaluations are also insensitive to the computation order. Accelerate follows a similar trend, borrowing only the features that are desirable in the context of parallel computation from Haskell, and working around those that are undesirable.

If we want to express a binary relation on two vectors of a different size without truncation, then we have to explicitly concatenate elements to the smaller vector to make them the same size. In Accelerate we can do this with (++) In SAC, you can get around explicit concatenation by using their WITH-loop[4]. For example, if two vectors of numbers, of different length are added. We can give a detailed specification of the indices we want to include — with lower and upper bounds, strides, and step size — and simply ignore the excluded indices. WITH-loops can be considered a shape polymorphic version of array-comprehensions.

```
1 int blurx(int[.] row, int x) {
2   return (row[x-1] + row[x] + row[x+1])
3 }
4 int [y,x] meanblur(int[y,x] img) {
5   return (with ( 0 < row < y - 1)
6           with ( 0 < x   < x - 1)
7           modarray(img[row], x,
8             (blurx(img[y], x - 1) + blurx(img, x) + blurx(img, x+1)) / 9));
9 }
```

Listing 7: 3×3 mean blur implementation in SAC

The mean blur example in SAC is shown in Listing 7. In this example, we use a similar approach as with Halide: we combine two single-pass functions into a single,

multi-pass function that can be fused. In the `blurx` function we take the sum of three adjacent values in the same row. In the body of the two nested WITH-loops we modify a specific index of the row to to be the mean of all neighbours of that index. `modarray` will perform in-place updates whenever possible, or a copy otherwise. In this case a copy will occur because there is a data dependency between the cells. We ignore the boundaries by tightening the bounds on the WITH-loop, boundary values will be effectively copied.

2.4.4 Repa

Regular **P**arallel **A**rrays (Repa) is another language that is built on the premise that purely functional array algorithms are easier to comprehend than their imperative counterparts[3]. Haskell lacks a good way of expressing these algorithms and then evaluate them efficiently and in parallel.

Writing programs in Haskell that approach the performance of hand written C is possible by sacrificing purity and resorting to index based methods in the IO-monad. This way, the program is obscured and becomes much more difficult to comprehend. Repa gets around this by using an interface based on collective operations to emphasise the algorithm.

Repa is a shallow EDSL in Haskell. Because of this there is no AST to transform and all optimisations take place on the library level i.e. how primitive functions are implemented in Haskell. Like SAC and Accelerate, it is a shape polymorphic language that makes it easy to reuse code for arrays with arbitrary dimensions. This shape polymorphism is achieved with a type class, in the same way as Accelerate (Section 2.1.4) which allows Repa to also piggy-back on the type-checker of GHC to statically ensure the dimensions of arrays.

The mean blur examples in Repa is shown in Listing 8. This example shows some parallels with Accelerate because the same patterns are used for describing shapes. Two examples are shown: the first is implemented with an update function that explicitly accesses an array using its index, similar to our approach in Futhark. Here we also ignore the shell. The second example uses template Haskell to generate a two-dimensional stencil for us with the weights we specified. In this case we also have a boundary condition that clamps values that are out of bounds onto the boundary.

2.4.5 Масси́в

Масси́в (Massiv) is another shallow EDSL for parallel array computations in Haskell. Масси́в has a single back end and always runs on the host central processing unit (CPU) via Haskell's runtime system.

Масси́в has support for schedules that allow the user to specify how a computation has to be run, but this support is limited to `Seq` and `ParOn`: `Seq` is simple sequential computation on one core, `ParOn` and variants fix worker threads to specific cores and distribute the work to those workers. Their approach is task-parallel because these workers do not synchronise.

To compare Масси́в to Accelerate, we turn to their GitHub page[8] to see how they see how they compare Масси́в to Repa on the points where Accelerate differs:

```haskell
-- Explicit version
meanBlur :: Array DIM2 Double -> Array DIM2 Double
meanBlur arr = traverse arr id update
  where
    _ :. h :. w = extent arr

    update get d@(sh :. i :. j)
      = let g ox oy = get (sh :. (i + ox) :. (j + oy))
        in if isBoundary i j
           then get d
           else (g (-1) (-1) + g 0 (-1) + g  1 (-1)
                 g (-1)   0  + g 0   0  + g  1   0
                 g (-1)   1  + g 0   1  + g  1   1 )/9

    isBoundary i j
      =  (i == 0) || (i >= w - 1)
      || (j == 0) || (j >= h - 1)

-- Fancy version
meanBlur' :: Monad m => Array U DIM2 Double -> m (Array U DIM2 Double)
meanBlur' arr =
  let stencil = [stencil2| 1 1 1
                           1 1 1
                           1 1 1 |]
   in computeP .  A.smap (/9) $ forStencil2 BoundClamp arr stencil
```

Listing 8: Implementation of the mean blur in Repa in two ways.

1. *"Better scheduler that is capable of handling nested parallel computation."* Accelerate does not support nested parallel computation but as Массúв is task parallel rather than data-parallel, this can be done.

2. *"Shape polymorphic but with improved default indexing data types."* Roughly equal to what Accelerate has, they use `snoc`-lists of `KnownNat` instead.

3. *"Safe stencils for arbitrary dimensions, not only two dimensional convolution. Stencils are composable through an instance of Applicative"* In principle Accelerate supports stencils for arbitrary dimensions although stencils are not composable.

Массúв gives the user control over the underlying representation of arrays and their elements, supporting both manifested and delayed arrays. Elements can be represented as primitive Haskell values, boxed values (either in normal form or weak-head normal form) unboxed or fixed in a specific memory location for marshalling pointers to a foreign-function interface.

The mean blur examples in Массúв is shown in Listing 9. Массúв , like Repa and Accelerate, uses a familiar pattern for accessing a particular index. Similar to what we

```
blur :: Array D Ix2 Double -> Array DW Ix2 Double
blur = mapStencil Edge blurStencil
  where
    blurStencil = makeStencil (Sz (3 :. 3)) (1 :. 1)
      $ \ get ->
        (/9) . sum $ fmap (\ix -> get ix)
          [(-1 :. -1), (-1 :. 0), (-1 :. 1)
          ,( 0 :. -1), ( 0 :. 0), ( 0 :. 1)
          ,( 1 :. -1), ( 1 :. 0), ( 1 :. 1)
          ]
```

Listing 9: Implementation of the mean blur in Massiv

have seen with Futhark and Repa, Масси́в does have a stencil primitive, but it is based on index-based methods and an offset for the anchor point rather than assuming the anchor point and providing the values immediately.

Chapter 3

Motivation

3.1 Problem Description

We have a simulation of the sediment layer in a salt marsh, which is an instance of a simulation with a complicated stencil computation and complex boundary conditions. The two most widely used boundary conditions for simulations like these are the Dirichlet condition and the Von Neumann condition. The Dirichlet condition forces the value of a cell to be constant and the Von Neumann conditions forces the derivative of a cell to be constant. The Dirichlet condition has a straight forward implementation, but the Von Neumann condition depends on the local context of a cell and has dependency to its neighbours.

The salt marsh represents a physical system, and we have to implement physically sensible behaviour at the boundaries. In this case, not all boundaries behave in the same way. The configuration of this simulation is shown in Figure 3.1. We have reflection of the flow, i.e. the medium hits a wall and is reflected according to the reflection principle following from Fresnel's equations, at the red and green boundaries. Water enters the system at a constant rate on the red boundary, and flows towards the green boundary. This is done by setting the sediment height to 0 on the green boundary, which is an example of the Dirichlet condition. We resort to multi-pass computations and conditional expressions on the index to achieve this complex behaviour in Accelerate.

Angle of incidence equals the the angle of reflection.

The code computes a single step in this simulation is shown in Listing 10. A step is performed with a binary stencil — line 1. We can also implement this step with a unary stencil by pairing the constant array `b` with the variable array `arr`, but we have chosen for a binary stencil to keep different kinds of data separate. After we have computed a single step we have to fix the boundaries, because the boundary cells have been recomputed in the `simulateUV` step and now contain invalid results.

`simulateUV` is the function for the white region, externally defined.

To fix the boundaries, we check the index and choose our behaviour depending on which boundary this index belongs to. (i) If the index belongs to the yellow, blue or green boundaries then the velocity of the direction pointing towards the boundary we just 'hit', is negated — lines 14-17 for green, 18-21 for yellow, 22-25 for blue. All other values are copied from the previous iteration. (ii) If the index belongs to the red boundary then we compute a value for the water flow — lines 8-12. (iii) Otherwise, the value is copied as is. — lines 26-27.

We have to treat each boundary separately to reflect our model of reality, but the

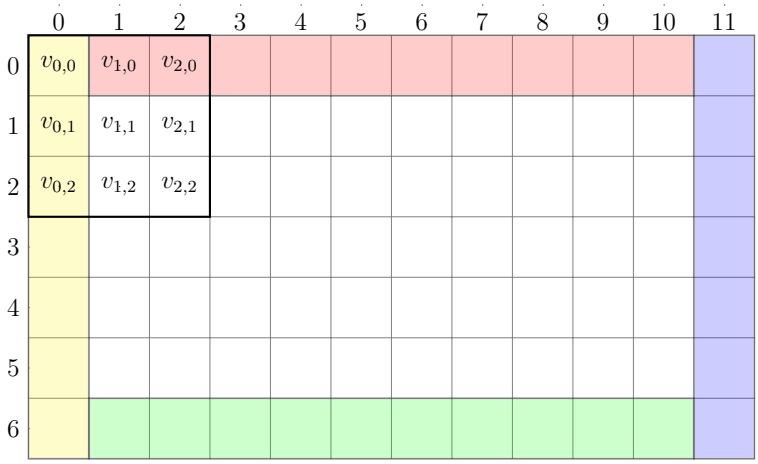


Figure 3.1: Configuration of the salt marsh simulation. There is a source at the yellow boundary and a sink at the blue boundary. The yellow and blue boundaries are *walls* where the medium is reflected. In the centre (white) we run the simulation with a stencil. An overlaying stencil is shown as an example.

way in we distinguish them is convoluted and causes redundant steps i.e. copying and conditionals on the core.

In summary, this code is awkward to write and difficult to read, understand and maintain. The branch that is most often chosen is the last, because there are relatively few cells in the shell (Section 2.2.1) And although this approach is data-parallel, the SIMD hardware will be occupied performing a copy on many cells which is essentially unnecessary memory traffic.

Our ideal solution prevents the user from having to escape to explicit, index-based methods such as IF-expressions inside generates. From now on, we will refer to this solution as 'regions of independent computations', or *regions* in short.

3.2 Applications

Regions will allow us to write more maintainable code for a variety of problems. In this section we will demonstrate a couple of applications that benefit from the region notation in order to motivate them further.

3.2.1 Salt Marsh

If regions were available to us then we can describe each of the branches in the conditional expression of Listing 10 as a separate region.

No boundaries would have to be fixed because the colored boundaries can each be generated by a separate `generate`. This will still copy most values at these boundaries but avoids having to copy cells from the centre, which is where we win out.

The body of the simulation can be done with a stencil computation in a single pass. If we have a version of stencils that omits bound-checking, we can further speed up this computation

3.2.2 Heat Transfer

An example of a simulation that can demonstrate clearly what regions should be able to achieve is the simulation of heat transfer in two dimensions. The configuration of of what a heat transfer simulation would look like is shown in Figure 3.2. The red region is an infinite heat reservoir of high temperature and the blue region is an infinite heat reservoir of low temperature. The white centre represents the material we want to simulate.

The derivation of the entire simulation can be found in Appendix B

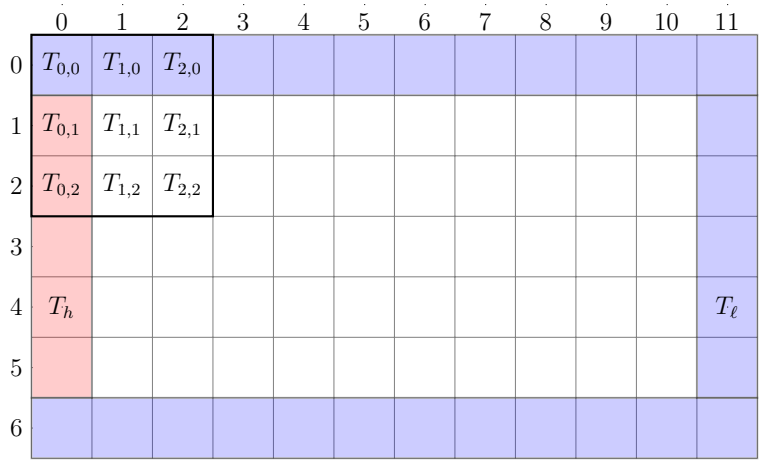


Figure 3.2: Configuration of what our heat transfer simulation would look like. The red and blue regions have a constant high (T_h) respectively low (T_ℓ) temperature. The white section is computed with a stencil, the focus is currently on $T_{1,1}$.

We will have to compose three rectangular regions to obtain the blue region, because the blue region is not rectangular and this is difficult to work with The red and white regions are already rectangular. Each rectangle is assigned a different computation. The blue and red regions can simply generate an array with constant values and the white centre can be simulated with a stencil computation.

3.2.3 Local Time Resolution

Explicit time marching simulations such as the salt marsh and heat transfer are only stable if the Courant-Friedrichs-Lewy (CFL) condition is met

$$C = \Delta t \left(\sum_{i=1}^{n} \frac{u_{x_i}}{\Delta x_i} \right) \leq C_{\max}. \tag{3.1}$$

Where x_i is the velocity of a phenomenon along the x_i dimension and Δx_i is the spatial resolution in that dimension. In practice $C_{\max}$ is often set to 1. In our two dimensional case we have that

$$\Delta t \left(\frac{u_x}{\Delta x} + \frac{u_y}{\Delta y} \right) \leq 1, \tag{3.2}$$

In a more difficult simulation such as one that has variable spatial resolution or a body that consists of multiple materials, we might have that on some points we do not meet the CFL condition, but only locally. In this case we can reduce the time resolution (Equation 3.1) of the simulation from Δt to $\Delta t'$ so that the condition is met and iterate more often on this region. The rest of the array can then be updated every n steps where $\Delta t > n\Delta t' \implies n = \lceil \frac{\Delta t}{\Delta t'} \rceil$.

This is a speculative example because we would need to be able to identify groups of cells in an array that do not satisfy the CFL condition, and put them in a separate region. For example, if we have two small regions of a single cell, diagonally opposite in an array, we would have to split the array in five regions. With more of these regions, the amount of regions may grow quickly and partitions might not be easy to find.

What we do want to illustrate with this example is that even when there is no direct use for regions because there are no different computations in *space*, it might still be useful to have them because of different computation in *time*.

3.3 Research Questions

The previous section clearly illustrates the problem: we want to be able to express regions of independent computations inside an array. The research questions boil down to: (i) How can we facilitate regions of independent computations in Accelerate? (ii) How can these be used to express boundary conditions for both stencil computations and partial differential equations efficiently? (iii) Do we need different evaluation strategies depending on the size of the regions? (iv) If so, what are these strategies?

```
simulationStep arr = fixBoundaries $ stencil2 simulateUV clamp b clamp arr
  where
    fixBoundaries arr = generate (shape arr) boundaryFn
      where
        (uarr, varr, harr, sarr, darr) = A.unzip5 arr
        boundaryFn i =
         -- Red
         cond (r A.== 0)
           (lift $ ( 2 * uarr A.! row1 - uarr A.! row2 :: Exp Float
                   , 2 * varr A.! row1 - varr A.! row2 :: Exp Float
                   , harr A.! neumannTop               :: Exp Float
                   , 0                                 :: Exp Float
                   , darr A.! neumannTop               :: Exp Float
                   ))(
         -- Green
         cond (r A.== grid_Height - 1)
            (lift $ ( uarr A.! neumannBot    , - varr A.! neumannBot
                    , harr A.! neumannBot    ,   sarr A.! neumannBot
                    , darr A.! neumannBot))(
         -- Yellow
         cond (c A.== 0)
            (lift $ (-uarr A.! neumannLeft   ,   varr A.! neumannLeft
                    , harr A.! neumannLeft   ,   sarr A.! neumannLeft
                    , darr A.! neumannLeft))(
         -- Blue
         cond (c A.== grid_Width - 1)
             (lift $ (-uarr A.! neumannRight,   varr A.! neumannRight
                     , harr A.! neumannRight,   sarr A.! neumannRight
                     , darr A.! neumannRight))(
         -- White
         lift $ ( uarr A.! i, varr A.! i
                , harr A.! i, sarr A.! i , darr A.! i
                )))))
         where
            (_ :. r :. c) = unlift i :: (Exp Z :.  Exp Int :. Exp Int)
            row1 :: Exp DIM2
            row1 = lift (Z :. (r+1) :. (0 :: Exp Int))
            row2 :: Exp DIM2
            row2 = lift (Z :. (r+2) :. (0 :: Exp Int))
            neumannTop   = lift (Z :. (1               :: Exp Int) :. c)
            neumannBot   = lift (Z :. (grid_Height - 2 :: Exp Int) :. c)
            neumannLeft  = lift (Z :. r :. (1              :: Exp Int))
            neumannRight = lift (Z :. r :. (grid_Width - 2 :: Exp Int))
```

Listing 10: Hot swapping functionality at the boundaries in Accelerate.

Chapter 4

Integrating the Regions

We will try to find a solution by placing ourselves out of the functional- and into the imperative context, where we will look for a similar problem. For if we manage to find one, we might also find a solution.

We are running an IF-statement inside a loop, which result in two different program paths through this loop. Instead of having the IF-statement on the inside, we can 'hoist' this IF-statement from the loop, and convert the program into two sequential loops: one for all THEN-branches, and another for all ELSE-branches. This is similar to loop invariant code motion. It is initially known where our boundaries are, and these boundaries do not depend on loop variables. The only problem is that the compiler does not have this information, because we have no way to provide it.

Example: If we consider a single IF-statement or equivalently, just two sections: a top row and the rest. Any reasonably proficient programmer would write at least two explicit loops that have different bodies as shown in Listing 11, one for the THEN-branch, and another for the ELSE-branch.

```
size_t j = 0;
for (size_t i = 0; i < width); ++i) {
  ... // body for computing the top row (boundary size = 1)
}

j = 1;
for (; j < height; ++j) {
  for (size_t i = 0; i < width; ++i) {
    ... // body for computing the rest
  }
}
```

Listing 11: C implementation of different regions, a visual representation is shown in Figure 4.1

For some special cases such as the Dirichlet condition, we are able to write the same computation with a single traversal T_1. This is done by initialising the shell — analogous to the first loop in Listing 11 — and then constraining the domain of T_1 to exclude the

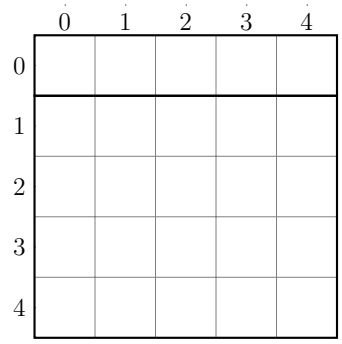


Figure 4.1: Partition of an array in two regions where the top row has to be treated in some special way. Also see Listing 11

shell e.g. in a vector of size n and a shell size of 1, we constrain the bounds of T_1 from $[0..n-1]$ to $[1..n-2]$. The computation then relies on mutability of the array which is a side-effect. We also require a context independent computation such as `map` because if we were to run a stencil in row-major order, then values that gravitate to the north-west are updated first and so we mix new with old values in our stencil. A small modification, namely an additional buffer, solves this problem: we can run T_1 over the core and have no dependency constraints, but then have to run a second traversal T_2 to with different bounds to copy the shell.

We can attempt a similar idea in Accelerate. We run some computation over the core, and run different computations on the shell. The shell consists of contiguous regions of indices, which can be fully specified by two hyper-dimensional vectors: the offset and the extent. For example, assuming a two-dimensional array of size $n \times m$ with a shell size of s. From a dimensionality of $\mathbb{N}^2$ we know that each vector needs two points. The core of the array will be at $[s;\ s]$ and the extent will be $[n-s;\ m-s]$.

We will first add a new datatype that holds the information we need for a single region. The offset and the extent will be expressions of shapes because we want to be able to dynamically compute ranges of arbitrary dimension *using* Accelerate. Furthermore, the extent will be implicit in the shape of the computation and there is no need to add a field for it. Lastly we specify the computation to be run at this offset. The resulting type is as follows

The `Exp` type denotes an Accelerate *runtime* value.

```
data RegionSpec acc exp a where
  RegionSpec
    :: exp sh             -- offset
    -> acc (Array sh t)   -- computation
    -> RegionSpec acc exp (Array sh t)
```

We add this type to the AST by referencing it from a new constructor that carries some additional information such as the total size of all the regions combined, to aid in allocation of the arrays. This is not strictly necessary as we could have computed the total size with an analysis over the AST, but we specify it because it is a rather small price to pay in comparison to implementing the analysis.

```
data PreAcc acc exp as where
  ...
  Region :: (Shape sh, Elt b)
         => exp sh                              -- Total size
         -> [RegionSpec acc exp (Array sh b)] -- region specifications
         -> PreAcc acc exp (Array sh b)
```

The consequence of using this representation is that we cannot ensure anything statically. The total size, partitioning and computations are all runtime values. It also means that we have to assume that the region specification is complete i.e. all cells belong to at least one region. In Section 5 we will talk about possible extensions to- and shortcomings of this implementation in detail.

Now that we can express independent regions, we can also add a special stencil constructor to the AST in which we embed the assumption that no boundary conditions are required because all the indices will be in range. Apart from the lack of a boundary condition, it will be equal to the `Stencil` constructor as follows

```
data PreAcc acc exp a where
  ...
  Stencil  :: Stencil sh a stencil  -- Already supported
           => stencil
           -> Boundary (Array sh a)
           -> acc (Array sh a)
           -> acc (Array sh b)

  RStencil :: Stencil sh a stencil -- New
           => stencil
           -> acc (Array sh a)
           -> acc (Array sh b)
```

The benefit of `RStencil` is that it can be entirely implemented from existing parts. For example, the interpreter can support `RStencil` by abstracting the interpretation function for regular stencils. The evaluation function in the backend now takes an additional argument of `(sh -> sh)` that specifies how an index should be accessed. For our new unguarded stencil this is simply `id`, for the regular stencil we provide the guards that were already in place.

4.1 Evaluation

Intra-region the semantics are data-parallel like all other Accelerate computations. Inter-region this is not necessarily so. When multiple regions carry the same function, those regions could in principle be merged and then be executed in parallel without violating the data-parallel property of Accelerate.

Accelerate hashes sub-trees of the AST to determine whether they have already been compiled and can be re-used. However, this is based on a shallow hash e.g. we only hash the function in argument of `map`, and this is not enough to determine whether entire computations are equal. Determining whether functions are equal could be done based on a deep hash of the AST, is in undecidable in general.

We can get around this in practice, by letting the user specify multiple offsets per computation instead of pairing an offset with an embedded computation. The implementation needs to be severely changed to facilitate this however. We will discuss this problem in more detail in Section 5.

Running different functions on different threads can in principle be done if our target architecture is a CPU. For a GPU this is not useful because these devices are optimised for data-parallelism, and only sequential execution of regions with different functions remains.

```
1  -- Evaluation of regions, assuming that implementations for all other AST nodes
2  -- exist.
3  eval :: Acc a -> IO a
4  eval (Region _  []) =
5    do
6      res <- allocate undefined Z
7      return res
8  eval (Region sz rs) =
9    do
10     res <- allocate undefined sz
11     return $
12       map (\(offs, reg) -> copyRegion res (eval reg) offs) rs
13
14 -- Copy the contents of an array @reg@ to another @arr@ at @offs@. @arr@ needs
15 -- to be large enough to contain @reg@
16 copyRegion :: Array sh t -> Array sh t -> sh -> Array sh t
17 copyRegion arr reg offs =
18   let f sh = let sh' = sh + offs
19              in update arr sh' (r ! sh)
20    in map f (indices reg)
21
22 -- The implementation of these functions is assumed
23 allocate :: (Shape sh, Elt t) => t -> sh -> IO (Array sh t)
24 update   :: (Shape sh, Elt t) => Array sh t- > sh -> t -> Array sh t
25 indices  :: (Shape sh, Elt t) => Array sh t -> [sh]
```

Listing 12: Evaluation semantics of the `Region` node in the Accelerate AST

The current evaluation strategy of `Region` is given in Listing 12. We assume that semantics for every node in the AST exists with the exception of `Region` and `RStencil`. Additionally we will assume the existence of `allocate` which allocates an array of a specified size, `update` which updates a single value in an array at a specified index, and `indices` which returns a list of all the valid indices that an array has.

Here it is nice to have the shape of the result array at the top level, because we can immediately allocate an array that is large enough to accommodate all regions. The nested computations are recursively evaluated and their results are copied to the `res`

array at the specified offsets, in the order that they occur in the list. We could accomplish the same with scatters to avoid copying.

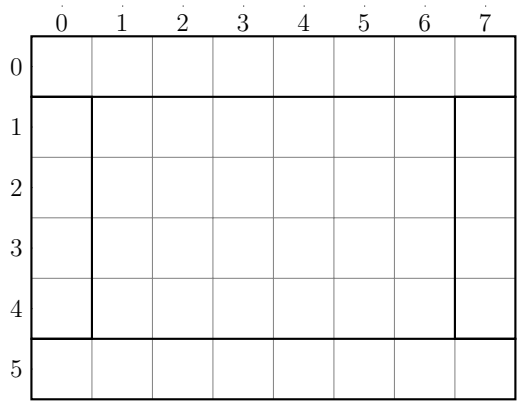


Figure 4.2: Different regions (subarrays) whose memory regions have to be scattered to avoid copying.

Evaluation for `RStencil` bypass the bound-check, but are otherwise equal to the semantics of `Stencil`. Because of this similarity, we will not show the code here. Evaluation of `RStencil` is safe because an efficient runtime check can ensure from the size of the source array, offset of the region, size of the region, and size of the stencil size, that no out-of-bound indices will be accessed. All of these properties are available to us so we can check this in $O(1)$. If we do not include this runtime check, we have undefined behaviour if the user makes a mistake.

The conditions we have to test for each region that contains an unguarded stencil are that the 'left' must stay in bounds: $0 < o - \lfloor \frac{w_s}{2} \rfloor$ where o is the offset and w_s is the width of the stencil, and that the 'right' must stay in bounds $o + w_r + \lfloor \frac{w_s}{2} \rfloor \leq w$. where w_r is the width of the region and w is the width of the source array. Left and right refer to an example in one dimension dimensions here. For a multidimensional array, this condition has to hold for each dimension.

4.2 The User Language

The interface we have exposed to the user is rather compact. A summary is shown in Listing 13

A single function, `regions`, is used to specify a list of computations and the offsets at which they have to be placed. This allows the user to specify separate concepts into different sections of code e.g. modules, and join them later.

Unguarded stencils are also directly exposed to the user via `rstencil`. This way, users are free to mark any region in their computation as safe to access without bound-checks. The runtime check will catch any mistakes and prevent undefined behaviour.

As an auxiliary helper function we have added `stencil'`. The semantics are exactly the same as `stencil` but an array is automatically split up into different regions, depending on the size of the stencil. For the regions that belong to the shell, we simply execute the `stencil`. For the core, we run `rstencil` instead, omitting the boundary checks.

```
regions  :: Shape sh
         => Exp sh                  -- size
         -> [(Exp sh, Acc a)]       -- (offset, computation)
         -> Acc a

rstencil :: (Stencil sh a stencil, Shape sh, Elt a, Elt b)
         => (stencil -> Exp b)      -- stencil function
         -> Acc (Array sh a)        -- source array
         -> Acc (Array sh b)

stencil' :: (Stencil sh a stencil, Shape sh, Elt a, Elt b)
         => (stencil -> Exp b)      -- stencil function
         -> Boundary (Array sh a)   -- boundary condition
         -> Acc (Array sh a)        -- source array
         -> Acc (Array sh b)
```

Listing 13: A summary of the interface to regions.

4.3 Results

In Section 3.1 we have argued that conditional expressions make the code difficult to understand and less maintainable. In this section we will present some examples to strengthen this claim.

4.3.1 Salt Marsh

We will present our re-implementation of the salt marsh example from Section 3.2.1. Recall the original implementation from Listing 10, our new version is shown in Listing 14.

With regions and unguarded stencils we are indeed better at separating concepts. However, the re-implementation is longer than the original by 5 lines and is littered with offsets and extents. This is an unfortunate outcome in this specific case. The good news is that we do not have fix the boundaries in a second step, because we do not break them. Each region is defined by three variables representing the offset, size and the function to execute. Boundaries are generated at every step — lines 1-4 — and the actual simulation step is run with an unguarded stencil in the core — lines 5,43.

For this specific case we will conclude that the clarity of the code has not improved, but we managed to separate concepts so that no errors have corrected in a second pass.

4.3.2 Heat Transfer

An implementation of a heat simulation is shown in Listing 15. The functions that are executed on these boundaries are simply `generate`s, they are filled in once and then remain constant. The function that is executed on the core is a stencil that takes the weighed sum of the differences to all orthogonal neighbours. This formula can be easily derived from the law of conservation of energy, a detailed derivation for this model can

be found in Appendix B. The rest of the code are offsets and sizes, which is unfortunate boilerplate code we have to write.

From the implementation of the heat transfer simulation we will conclude that the code is clear and concise and can be run efficiently. We consider this a success.

4.3.3 Benchmarks

Jizos specifications can be found in Appendix D

To test the performance of regions we ran benchmarks on our test system *'Jizo'*.

We were unable to test our own implementation because we have only implemented an interpreter backend at this time. Instead, we have approximated our implementation of regions with with the unextended Accelerate v1.4. To achieve this, we represent regions as tuples of arrays e.g. an array with two regions is represented by

```
Acc (Array sh a, Array sh a)
```

We cannot approximate unguarded stencils without modifying the code generator in both the CPU and GPU backends. Hence, unguarded stencils have not been benchmarked.

The only software we have used for benchmarking is the Haskell library Criterion[11]. The settings we benchmarked with are a confidence interval $\alpha = 0.95$ and $N = 1000$ resamples. Criterion supports *environments* that are provided to the functions, which are fully evaluated before the benchmarks start. We have forced evaluation of all input arrays using this method. All of the different settings we have benchmarked are

- **Array size**: 1M, 2M, 5M, 10M, 50M and 100M cells
- **Boundary size**: 1, 2, 5, 10, 50 and 100 px
- **Backend**: LLVM-native and LLVM-PTX, CPU and GPU respectively.
- **Functions**: `trivial`, `if-inside-trivial`, `if-inside-hard` `if-outside-trivial`, `if-outside-hard`

The names of the functions were inspired by how regions are separated in different loops in an imperative language, as described in the beginning of this chapter:

1. `trivial` simply adds one to the entire array. We have included this to show how well a simple, highly optimisable function performs.
2. `if-inside` means that the conditional is part of the computation that takes place on the hardware, in contrast to `if-outside` where we have 'split the regions' before running anything, saving us the conditionals on the hardware.
3. The difference between `if-inside-hard` and `if-inside` is that the former does a 'hard', computationally expensive function everywhere. The latter does a trivial computation on the core, and a hard computation on the boundaries. A loose naming scheme is `if-<where we split>-<what is done in the core>`.

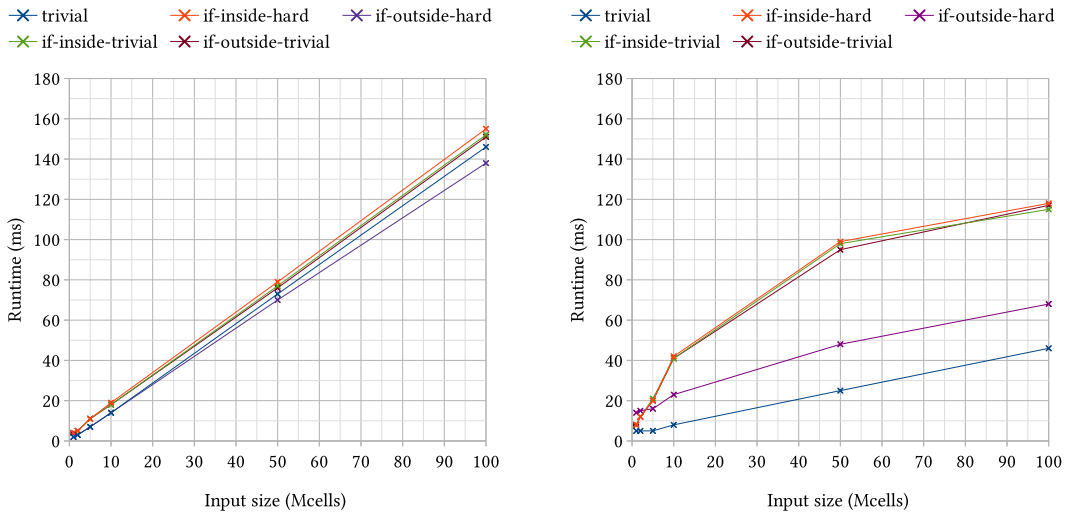


Figure 4.3: Runtimes vs input sizes on GPU (left) and CPU (right). The same function for all regions with a shell size of 1. Lower is better.

The 'hard' function has to equal or exceed the complexity of an unguarded stencil function. To make a function sufficiently difficult we used an iterated function of the form $f(x) = \mod(ax+b, p)$ where a, b and p are large primes, repeated for 1000 iterations. A table of numeric results can be found in Appendix C.

In Figure 4.3 we show the runtime of different functions on both the GPU and the CPU as a function of the input size with a shell size of 1, so that they impact the measurement as little as possible.

The GPU scales linearly for all input sizes and the runtime for all functions is very close together. The reason for this is that the shell is made irrelevant by its tiny width, therefore the GPU is mostly doing the same function for all inputs i.e. exactly what a glsgpu is specialised in. Splitting up the regions helps a little for the `if-outside-hard` function, which comes out 17 milliseconds faster than `if-inside-hard`.

Results for the CPU have a shape that may be explained by the caches. We expect the graph to continue linearly for input sizes larger than 50 MiB because the largest cache that our CPU has (L_3) is 32 MiB and so no entire input will fit in the highest cache. The other bending point appears roughly around the sizes of the lower L_2 cache of 8 MiB, although this is hypothetical.

For the trivial and if-outside-hard cases, we do see a reasonably linear trend in runtime over different input sizes, which at least partly disproves the cache hypothesis. These two particular benchmarks share that the intra-region function is equal and does not contain a conditional, therefore these variants may be much easier to vectorise.

In all other cases the intra-region function changes, either dynamically or statically, and so vectorisation is more difficult. We also have to pay the overhead of setting up different programs on the same processor multiple times.

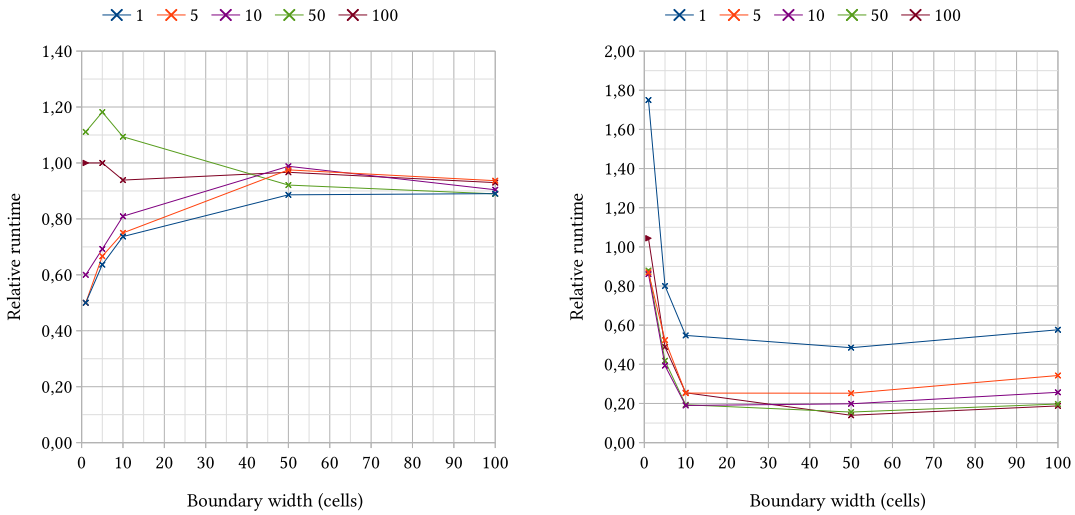


Figure 4.4: Runtime vs boundary width on GPU (left) and CPU (right). For different input sizes in Mcells, using only the function if-outside-hard. Lower is better.

In Figure 4.4 we present the relative runtime of `if-outside-hard` to `if-inside-hard`. On a CPU, small boundary widths do not seem to benefit from regions as much as when the boundary width is larger. The inverse holds for a GPU, but only when the arrays are large, and not by a large margin. With the exception of five cases, all benchmarks performed faster with regions than without. For GPU we observe improvements of around 10% for larger regions and 40% for smaller regions. For CPU we observe improvements of around 45% to 80%.

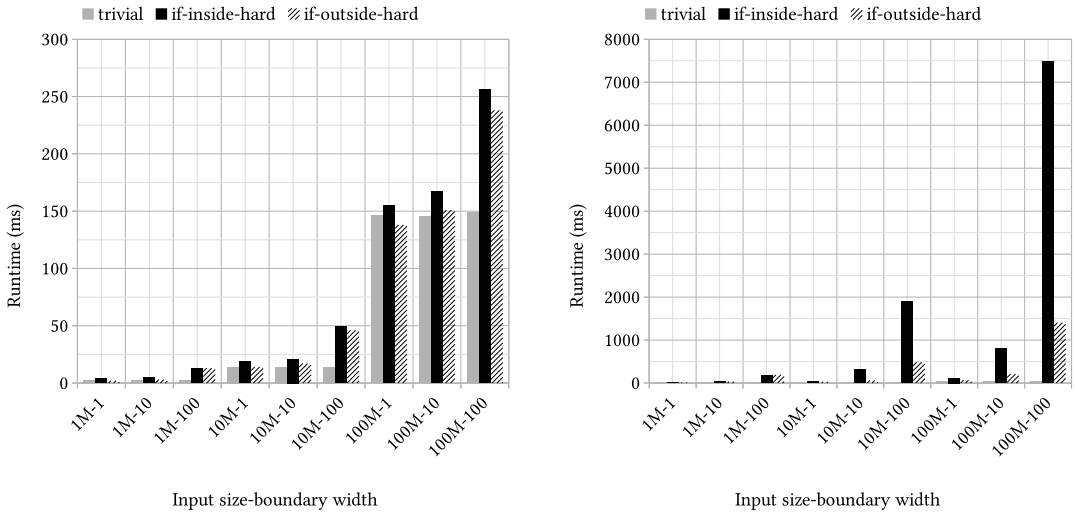


Figure 4.5: Runtimes for different shell sizes on GPU (left) and CPU (right). Only trivial, if-inside-hard and if-outside-hard. Lower is better.

Absolute runtimes for a selection of input- and shell sizes is shown in Figure 4.5. A GPU is clearly much more suited for these kind of computations than the CPU: large

inputs with large regions are brutally punished by a CPU while runtimes on a GPU still scale well. The light-grey bar shows the runtime of a trivial function with no regions. This function sets an approximate baseline for the performance — the fastest we could achieve. An exception is the 100M-1 scenario on the GPU, where the `if-outside-hard` is faster than the map. We attribute this artifact to noise in the measurement.

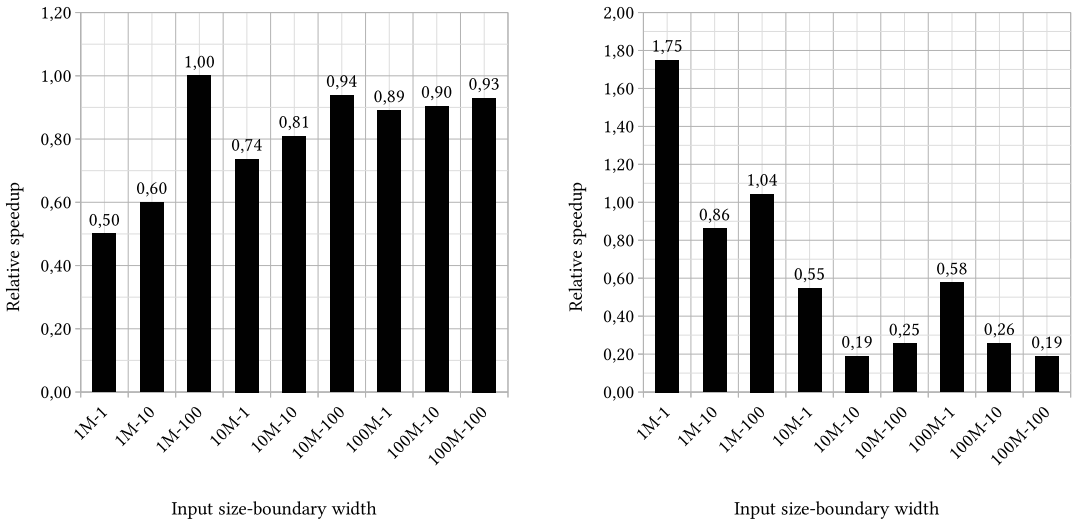


Figure 4.6: Relative runtime for different input/shell-size combinations on GPU (left) and CPU (right). The graph displays $\frac{T_{\text{if-outside-hard}}}{T_{\text{if-inside-hard}}}$. Lower value means that our extension performs better.

A selection of relative results of Figure 4.5 are presented in Figure 4.6. In almost all cases — with the exception of 1M-1 on the CPU — regions perform equal to or better than conditionals in the computation with a good margin. Especially when the target hardware is a CPU it makes a big difference to use regions instead of a conditional expression.

4.3.4 Adaptive Evaluation Strategy

The functions `if-inside-hard` and `if-outside-hard` are the most representative of a real-world scenario that might use regions. In Figure 4.5 and Figure 4.6 a relative comparison of those functions is presented. Relative runtimes larger than 1 imply that running an IF-expression inside a computation is cheaper to execute than splitting the computation in different regions.

If we know the relative shell size for which embedded IF-expressions are faster than regions, we can implement an adaptive evaluation strategy. When IF-expressions are faster, we can simply generate a chain of conditional expressions similar to how it is done in the salt-marsh implementation (Listing 10).

In theory, the cost C of running n IF-expressions must be less than the overhead cost of setting up m computations, $nC_{\text{IF}} < mC_{\text{oh}}$, with n representing the total number of cells and m representing the number of distinct regions. This is difficult to determine theoretically because we have no cost model. From our measurements it is also difficult do draw a clear line because our benchmarks are not fine-grained enough for small

inputs, or for very large shell sizes.

In the case of 1M-1 on CPU, where roughly 0.4% of the cells belong to the shell we observe a relative performance increase of $1.75\times$. In other words, regions are 75% cent slower than simply executing the IF-statements. On the other hand, in the case of 10M-1 where roughly 0.13% of the cells belongs to the shell, is $5.3\times$ faster with regions than with IF-statements. On GPU we have a few other cases — 1M-50, 5M-50, and 10M-50 — from Figure 4.6 where evaluation of IF-expressions is faster than using regions. The same cases for a either a larger or smaller input size show the same or better performance. Clearly relative shell size is not the most important factor here.

More research on the interplay between region- and array size is required before we can claim whether an adaptive evaluation strategy is sensible to implement.

```
step arr = regions (A.shape arr) [ (n_offs, A.generate n_sz n_f)
                                 , (e_offs, A.generate e_sz e_f)
                                 , (s_offs, A.generate s_sz s_f)
                                 , (w_offs, A.generate w_sz w_f)
                                 , (c_offs, A.rstencil c_f c_sz c_offs arr)]
  where
    (w, h, bw) = (grid_Width, grid_Height, 1)
    (uarr, varr, harr, sarr, darr) = A.unzip5 arr
    n_offs = Z_ ::. 0 ::. 0
    n_sz   = Z_ ::_ w ::. bw
    n_f sh = let Z_ ::. r ::_ w = unlift sh
                 row1           = Z ::. r + 1 ::. 0
                 row2           = Z ::. r + 2 ::. 0
                 neumannNorth   = Z_ ::. 1 ::. c
             in ( 2 * uarr A.! row1 - uarr A.! row2
                , 2 * varr A.! row1 - varr A.! row2
                , harr A.! neumannTop
                , 0
                , darr A.! neumannTop)
    e_offs = Z_ ::. w - bw ::. bw
    e_sz   = Z_ ::. bw ::. (h-2*bw)
    e_f sh = let Z_ ::. r ::. c = unlift sh
                 neumannEast     = (Z_ ::. h - 2 ::. c)
             in (-uarr A.! neumannEast, varr A.! neumannEast
                , harr A.! neumannEast, sarr A.! neumannEast
                , darr A.! neumannEast)
    s_offs = Z_ ::. 0 ::. h - bw
    s_sz   = Z_ ::. w ::. bw
    s_f sh = let Z_ ::. r ::. c = unlift sh
                 neumannSouth     = (Z_ ::. h - 2 ::. c)
             in ( uarr A.! neumannSouth,-varr A.! neumannSouth
                , harr A.! neumannSouth, sarr A.! neumannSouth
                , darr A.! neumannSouth)
    w_offs = Z_ ::. 0 ::. bw
    w_sz   = Z_ ::. bw ::. (h-2*bw)
    w_f sh = let Z_ ::. r ::. c = unlift sh
                 neumannWest     = (Z_ ::. h - 2 ::. c)
             in (-uarr A.! neumannWest, varr A.! neumannWest
                , harr A.! neumannWest, sarr A.! neumannWest
                , darr A.! neumannWest)
    c_offs = Z ::. bw ::. bw
    c_sz   = Z ::. w - bw ::. h - bw
    c_f    = simulateUv
```

Listing 14: Hot swapping functionality at the boundaries in Accelerate with the new region construct

```haskell
step prev =
  regions
    (A.shape prev)
    [ (n_offs, A.generate n_sz f_low)
    , (e_offs, A.generate n_sz f_low)
    , (s_offs, A.generate n_sz f_low)
    , (w_offs, A.generate n_sz (\_ -> A.constant t_high))
    , (c_offs, rstencil c_f c_sz c_offs prev) ]
  where
    Z_ ::. w ::. h = A.shape prev
    t_high   = 373.14
    t_low    = 273.14

    f_low  _ = A.constant t_low

    c_sz     = Z_ ::. w - 2 ::. h - 2
    c_f ( ( _, n, _)
        , ( w, m, e)
        , ( _, s, _)) = alpha * (n + e + s + w - 4 * m)

    n_offs   = Z_ ::. 1     ::. 0
    n_sz     = Z_ ::. w - 2 ::. 1
    s_offs   = Z_ ::. 1     ::. h - 1
    s_sz     = Z_ ::. w - 2 ::. 1
    e_offs   = Z_ ::. w - 1 ::. 0
    e_sz     = Z_ ::. 1     ::. h
    w_offs   = Z_ ::. 0 ::. 0
    w_sz     = Z_ ::. 1 ::. h
    c_offs   = Z_ ::. 1     ::. 1
```

Listing 15: implementation of heat transfer with the regions

Chapter 5

Discussion

5.1 Future Work

The current implementation of regions just scratch the surface of what we can do with them. In this section we will discuss some ways in which they can be extended.

5.1.1 Cover Analysis

The first extension we want to discuss is one that verifies whether all cells in an array are *covered* by regions. An array A is covered if the union of all regions is the same as the entire array: $\cup_{r \in R} r = A$.

Whenever there are cells that are *orphaned*, we have to assume some default behaviour for them. New regions will have to be made that contain these orphaned cells. We can efficiently find all orphaned cells and generate regions that contain them by using a hill climber algorithm that iteratively produces the largest region that wraps orphans, until there are no more orphans. A small example of a situation with orphans is shown in Figure 5.1. The blue and red regions are specified while the white region is not.

If the computation can be done in-place then the cells will remain untouched, which is sensible. Assuming that the regions are not done in-place, then the white region either contains artefacts from whenever the block was allocated, a default value, or a copied value from the source array. We argue that copying is the most viable option because it has the same behaviour as the in-place version.

This analysis will have to live in the runtime system of Accelerate because array bounds in Accelerate are not statically known. In the current implementation of regions we even use `Exp sh`, which is an *explicit* runtime type. Generalising the shape type to describe both static and dynamic shapes might open up the possibility to move the analysis from run- to compile time.

5.1.2 Overlap and Ordering

Another analysis that is related to a binary set operation is **overlap**. This analysis is analogous to set intersection with $\forall (r_1, r_2) \in R^2 : (r_1 \cap r_2 \neq \emptyset)$ and can detect whether

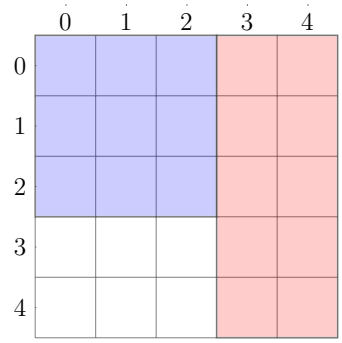


Figure 5.1: Array where two regions are specified (blue and red) and a third region that is orphaned.

there are overlapping regions. If regions overlap, there is potential for further optimisation or it may have consequences for the execution order.

Example: If there is a region that is entirely contained in another, and the outer region does not use the result of the inner region, then the inner region can be optimised away. This optimisation is difficult to implement in Accelerate, because now we need to introduce a dependency DAG, similar to what Halide has, in order to find and eliminate the inner region.

Currently, the execution order is implicitly fixed by the list of regions in the `Region` constructor. When code-generation is implemented for regions, the order of execution determines what a result will be. If regions are executed in task-parallel, then our functions may be unsafe because the overlapping part will be a seemingly random interleaving of the two.

Example: If two overlapping regions r_1 and r_2 execute the same function and we do not update the array in-place, then the order will be irrelevant to the result. In any other case, the result of the overlapping part of the region will be determined by unspecified behaviour due to race-conditions. There are multiple solutions to this. The first is to never run regions in task-parallel if there is overlap. The second is to have the user explicitly prioritise the execution order of regions.

5.1.3 Code generation

We have implemented an interpreter backend to experiment with the syntax of regions. In our benchmarks we have approximated what regions would be like with generated code by using tuples of arrays. These benchmarks show that there is performance to be gained when we generate code for distinct regions: 10% for GPU and $45\% - 80\%$ for CPU.

Ideally we generate a single code section for each distinct computation, and re-use it for each region that the computation is executed on. As stated before, it is problematic to determine whether two computations are the same. Equality of code sections will have to be based on a deep hash value of the AST.

Automatically splitting arrays into regions, as is done by the `stencil'` function, may generate a lot of regions and hence, code sections. In fact, the amount of code sections scales linearly with $2d + 1$ where d is the dimensionality of the array. This becomes 3^d if corners, edges, faces, etc have to all be treated in a different way.

5.2 Caveats

As we have shown, expressivity of regions can be useful in a variety of situations. There are however a couple of caveats that need to be discussed.

5.2.1 Representation of Regions

The current `Region` constructor contains a list of function-region tuples. Every function is paired with a single region, which is problematic when generating code for separate regions that share the same function because we have to assume that every function is unique and generate a separate code section for each. We can test if two functions are equal by testing on a deep hash value — assuming no hash-collisions, but this is undecidable in general. To solve this problem we need to change the type to explicitly contain the function that is being used. Intuitively it is a relatively simple problem to overcome: instead of function-region pairs, we can have a pair of a function with a list of offset-extent pairs. For example a map could be

```
data PreAcc acc exp t where
  ...
  RegionMap
    :: (exp a -> exp b)    -- function?
    -> [(exp sh, exp sh)] -- list of regions to map over
    -> acc (Array sh a)    -- source
    -> PreAcc acc exp (Array sh b).
```

In contrast to how regions are currently implemented, this approach has to carry extents for each region for each type of function. Here we only show `map`, but we ultimately want to embed any arbitrary Accelerate computation in a region. It will be quite difficult to change the compiler to handle this construct because it does not use any existing parts of the compiler. Although it is intuitively simple, working around the problems and actually implementing it will probably require a lot of time.

5.2.2 Nested Regions

The `Region` constructor holds a list of complete Accelerate computations. These complete computations might again be regions, hence regions can be nested. An artificial example of a nested region can be made by using `region` as follows

```
region sz [region sz [region sz ... [x]].
```

When we do this we get a list-like tree structure which, in this case, does not carry any meaning. It is best to flatten this tree and simply have `x` instead. However, we stumble on the same problem again: we cannot easily determine whether computations are equal, and we have no dependency DAG. This makes flattening of these arbitrary, list-like trees impossible in the current implementation.

There is no clear use case for nested regions. One could argue that a boundary region could be split up again into three other regions: the two corners and the middle, but this is a similarly synthetic situation. It is best to avoid nested regions. We have to demand this from the user because in the current implementation there is no way to statically prevent users from constructing them.

Chapter 6

Conclusion

In this work we have explored how regions of independent computations can be added to Accelerate.

We have enabled the expression of these regions by adding a new constructor to the Accelerate's AST. This constructor contains a collection of pairs with complete Accelerate computations and their offsets. Stencil boundary conditions can be expressed using these regions by converting the old boundary conditions to functions that produce their values and wrapping those in a `generate`, the exact same can be done for initial conditions of partial differential equations.

Code is not always more clear when regions are used. In fact, the code size increases linearly with the amount of regions that are added. Most of these extra lines are offsets and extents that cannot be avoided, but they can be written inline. We argue that clear separation of concepts weighs in heavier than having the least amount of code, because separation aids in intelligibility and maintainability.

To make regions even more useful, we have added another constructor to the AST that represents a stencil with the assumption that all accesses will be within the bounds of the source array. This way, we can omit specifying a boundary condition, and it also saves us the time from having to perform a bound-check while executing the stencil function.

Runtime performance depends on region- and array size, although not on the ratio between them. Another factor that makes a difference is target architecture, with relative gains in performance of around $2.2 - 5.3\times$ on CPU, while we only observe around $1.1\times$ for the GPU.

Although we do have multiple evaluation strategies, we cannot provide conclusive evidence of when to switch between them. From our measurements we observe only five cases that are faster without regions, two on CPU, where one of those is just slightly worse (4 per cent), and 3 on GPU with input sizes of 50M while input sizes of 100M are faster. None of our results show that we should adaptively change the evaluation strategy depending on the target architecture or input size.

Additional research is required in order to find what the interplay between region- and array size is, and whether it makes sense to implement an adaptive evaluation strategy.

In conclusion, we believe that we have demonstrated that regions are a sensible and desirable feature for Accelerate.

Appendix A

Matrix convolution

A.1 Definition

Matrix convolution is a special stencil operation that is the basis for many image processing filters. It is related to continuous convolution which is a higher order mathematical operator that is typically denoted with an asterisk ($*$). The type of this operator is $(*) : (\alpha \to \beta) \to (\alpha \to \beta) \to \alpha \to \beta$ showing that it consumes two functions and produces another. A common way that convolution is explained is that the resulting function shows how the shape of a function is changed by another.

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,\mathrm{d}\tau \tag{A.1}$$

Note that g is *reversed* around t. We will use a discrete, localised form that is related to the continuous form in Equation A.1 called *matrix convolution*. Convolution of a kernel over an image is defined as

$$\begin{bmatrix} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mn} \end{bmatrix} * \begin{bmatrix} y_{11} & y_{12} & \dots & y_{1n} \\ y_{21} & y_{22} & \dots & y_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \dots & y_{mn} \end{bmatrix} = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} x_{(1+i)(1+j)}y_{(m-i)(n-j)}. \tag{A.2}$$

We are using two sums because we are operating in two dimensions and that y is reversed like g in the continuous case. If we assume that the kernel and the region of interest (ROI) of the image are represented by one-dimensional vectors in row-major order then we can compute the convolution in Haskell with

```
convolve xs ys = sum (zipWith (*) (reverse xs) ys)
```

In practice we have a level of indirection and have to use indirect addressing with e.g. gathers and scatters instead.

Convolution kernels are data parallel computations, that GPUs are optimised for. Many stencil computation e.g. the mean blur, sharpen filter and sobel edge detection, can be written as a convolution. If the kernel is *separable* e.g. the kernel is the product of a row- and a column vector, then we can compute the convolution more efficiently

than when it is inseparable. Discussing whether kernels are separable or not is however not in the scope of this work.

The anchor of a convolution matrix is its center. Running a convolution with a kernel of size $p \times q$ on an image with dimensions $n \times m$ requires reading from cells with horizontal coordinates from the interval $\lfloor -\frac{p}{2} \rfloor \leq x \leq \lfloor n + \frac{p}{2} \rfloor$ and vertical coordinates from the interval $\lfloor -\frac{q}{2} \rfloor \leq y \leq \lfloor m + \frac{q}{2} \rfloor$. The bounds of the intervals are larger than the height and width of the matrix, hence it is required to handle this gracefully i.e. a boundary condition is required.

A.2 Two Dimensional Example

As a small proof that the general two-dimensional matrix convolution works, consider two 3×3 matrices. Recall from Equation A.2 that the convolution operation is defined as

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} * \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{bmatrix} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} x_{(m-i)(n-j)} y_{(1+i)(1+j)}. \tag{A.3}$$

Expanding the definition for this example yields Equation A.4. This is consistent with what can be found in other literature modulo UCAI. An implementation of this equation in C is shown in Listing 16.

$$\mathbf{X} * \mathbf{Y} = x_{11}y_{33} + x_{12}y_{32} + x_{13}y_{31} + x_{21}y_{23} + x_{22}y_{22} + x_{23}y_{21} + x_{31}y_{13} + x_{32}y_{12} + x_{33}y_{11} \tag{A.4}$$

```
int convolve(void ** A, void ** B, size_t m, size_t n) {
  int res = 0;
  for (size_t i = 0; i < m; ++i) {
    for (size_t j = 0; j < n; ++j) {
      res += A[i][j] *  B[m - i - 1][n - j - 1];
  } }
  return res;
}
```

Listing 16: Convolution example in C. The implementation differs from the mathematical example due to zero-based arrays.

Appendix B

Heat transfer derivation

The 'heat equation' is the popular name for a partial differential equation that describes a collection of phenomena that depend on their initial value. We will focus on the temperature of a body depends on its initial temperature. The heat equation for our particular case is as follows

$$\frac{\partial Q}{\partial t} = -k\nabla^2 Q. \tag{B.1}$$

Where Q is the heat, depending on the constant this could resemble temperature, energy content or a similar physical quantity. k is a constant that depends on the material the heat is flowing through. The *Laplace operator*, ∇, is a shorthand notation for a gradient of a function in multidimensional space. Simply put

$$\nabla^n Q = \frac{\partial^n Q}{\partial x^n} + \frac{\partial^n Q}{\partial y^n} + \frac{\partial^n Q}{\partial z^n}. \tag{B.2}$$

This equation states that the difference in heat at a single point in space, with respect to space, depends on some constant k and the second derivative of heat with respect to position. In other words: where there is curvature in space, there is change in time.

To compute the next state of temperature T_{n+1}, we simply add a gradient to the current state of the temperature. We can describe it with the recurrent relation

$$\begin{aligned} T_0 &= T_a \\ T_n &= T_{n-1} + \Delta T. \end{aligned} \tag{B.3}$$

Where T_a is the ambient temperature. The equation for one-dimensional, time independent heat flow is $\Delta Q = mc\Delta T$. We will divide this equation by Δt to make it time-dependent

$$\frac{\Delta Q}{\Delta t} = \frac{mc\Delta T}{\Delta t}. \tag{B.4}$$

The gradient can be computed from an equation similar to Equation B.1, instead of taking the second derivative with respect to position we will take the difference to the neighbouring cells,

$$\begin{aligned} \frac{\Delta Q}{\Delta t} &= \alpha \cdot \left(\frac{\Delta Q}{\Delta x} + \frac{\Delta Q}{\Delta y} \right). \\ \text{With } \alpha &= \frac{k}{\rho c_p d} \end{aligned} \tag{B.5}$$

where k is the same constant as in Equation B.1, ρ is the density of the material, c_p is the specific heat capacity at constant pressure and d is the thickness of the plate we are simulating, furthermore $[\alpha] = \text{m s}^{-1}$. We will substitute $\Delta Q = mc\Delta T$ for every ΔQ and factor out mc. This yields

$$mc\frac{\Delta T}{\Delta t} = mc\alpha \cdot \left(\frac{\Delta T}{\Delta x} + \frac{\Delta T}{\Delta y} \right), \tag{B.6}$$

or

$$\frac{\Delta T}{\Delta t} = \alpha \cdot \left(\frac{\Delta T}{\Delta x} + \frac{\Delta T}{\Delta y} \right). \tag{B.7}$$

Multiplying by Δt and substituting the result in Equation B.3 we get the recurrent relation

$$\begin{aligned} T_0 &= T_a \\ T_n &= T_{n-1} + \alpha\Delta t \cdot \left(\frac{\Delta T}{\Delta x} + \frac{\Delta T}{\Delta y} \right). \end{aligned} \tag{B.8}$$

We will approximate $\frac{\Delta T}{\Delta x}$ by simply computing the difference in temperature to the neighbouring cells. For the horizontal dimension we compute, in terms of T_{22} from the temperature distribution in Equation B.9, $\Delta T_{22,x} = T_{21} - T_{22} + T_{23} - T_{22} = T_{21} + T_{23} - 2T_{22}$. Analogously for $\Delta T_{22,y} = T_{12} + T_{32} - 2T_{22}$ and $\Delta T_{22} = \Delta T_{22,x} + \Delta T_{22,y}$. We divide the result by Δx because our spatial resolution is the same for both dimensions and it remains constant during the simulation.

$$\begin{bmatrix} T_{11} & T_{21} & T_{31} \\ T_{12} & T_{22} & T_{32} \\ T_{13} & T_{23} & T_{33} \end{bmatrix} \tag{B.9}$$

Appendix C

Full table of results

In the next two tables we present the benchmark results rounded to milliseconds. Rounding was done because sub-millisecond timeresolution is likely noise on computations of this scale. The two functions that have been tested are: a trivial function $x + 1$, and a hard function which is an iterated function of the form $f(x) = \mod (ax + b, p)$ where a, b, and c are large primes. For a more detailed explanation of what the functions mean we refer to Section 4.3. Benchmarks were run using the Criterion[11] package with a confidence interval of $\alpha = 0.95$ and $N = 1000$ resamples.

I	B	trivial	inside-hard	outside-hard	inside-trivial	outside-trivial
1	1	5	8	14	8	8
	2	4	12	16	12	12
	5	5	23	20	22	23
	10	5	36	31	36	34
	50	5	124	109	123	122
	100	4	182	190	181	181
2	1	5	12	15	12	12
	2	5	19	16	19	18
	5	5	33	26	32	34
	10	5	55	35	55	55
	50	5	215	144	218	221
	100	5	307	263	302	307
5	1	5	20	16	21	20
	2	6	31	20	30	30
	5	6	63	33	61	61
	10	7	117	46	115	114
	50	6	500	209	504	500
	100	6	720	352	727	729
10	1	8	42	23	41	41
	2	8	71	26	72	70
	5	8	162	41	163	164
	10	8	316	60	311	313
	50	8	1499	290	1481	1485
	100	10	1892	482	1900	1904
50	1	25	99	48	98	95
	2	25	171	54	171	169
	5	26	396	100	396	387
	10	26	761	151	757	762
	50	26	3726	582	3710	3696
	100	26	7338	1026	7360	7325
100	1	46	118	68	115	117
	2	44	188	81	188	191
	5	45	420	144	420	415
	10	45	813	209	808	821
	50	45	3894	768	3889	3889
	100	48	7493	1406	7510	7489

Table C.1: Measured results for different functions, input sizes and boundary widths on the CPU. All number are in ms, I is input size, B is boundary width. Measurements are all within $\alpha = 0.95$

I	B	trivial	inside-hard	outside-hard	inside-trivial	outside-trivial
1	1	2	4	2	4	4
	2	2	4	2	4	4
	5	2	4	2	4	4
	10	2	5	3	5	5
	50	2	9	10	9	9
	100	2	13	13	13	13
2	1	3	5	3	5	5
	2	3	6	3	5	5
	5	3	6	4	6	6
	10	3	6	4	6	6
	50	3	13	17	13	12
	100	3	19	22	19	19
5	1	7	11	7	11	11
	2	7	12	8	11	11
	5	7	12	8	12	12
	10	7	13	9	13	13
	50	7	22	26	22	22
	100	7	32	32	32	32
10	1	14	19	14	18	18
	2	14	19	15	19	19
	5	14	20	15	19	19
	10	14	21	17	20	20
	50	14	32	35	32	32
	100	14	49	46	49	49
50	1	73	79	70	77	76
	2	69	79	71	78	77
	5	69	81	79	79	79
	10	70	84	83	82	81
	50	68	114	105	114	114
	100	69	150	145	148	148
100	1	146	155	138	152	151
	2	137	155	140	153	152
	5	137	158	148	163	160
	10	145	167	151	164	164
	50	146	208	185	209	208
	100	149	256	238	254	253

Table C.2: Measured results for different functions, input sizes and boundary widths on the GPU. All number are in ms, I is input size, B is boundary width. Measurements are all within $\alpha = 0.95$

Appendix D

Specifications of Jizo

All programs and benchmarks were run on *Jizo*, our personal test system. We provide a list of specification so that when the benchmarks are re-run on another system they can still be compared if the specification are equal.

Part	Specification
CPU	AMD ThreadRipper 2950X (16 cores @ 3.5GHz), zenv1 architecture Caches: L_{1d}: 512 KiB, L_{1i}: 1 MiB, L_2: 8 MiB, L_3: 32 MiB
GPU	NVidia RTX 2080, turing architecture
RAM	64 GB DDR4, 2833 MHz
Disks	2×8TB 'spinning rust' and 2×1TB NVME drives, both pooled with LVM.
OS	GNU/Linux, distribution Ubuntu 19.10 Eoan

Table D.1: Specifications of Jizo.

List of Figures

List of Listings

List of Tables

Bibliography

[1] Lawrence Livermore National Laboratory Blaise Barney. Introduction to Parallel Computing. https://computing.llnl.gov/tutorials/parallel_comp/#ModelsData. [Online; accessed 20-01-2020].

[2] Manuel M T Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming*. ACM, January 2011.

[3] Robert Clifton-Everest, Trevor L. McDonell, Manuel M T Chakravarty, and Gabriele Keller. Embedding Foreign Code. In *PADL '14: The 16th International Symposium on Practical Aspects of Declarative Languages*, LNCS. Springer-Verlag, January 2014.

[4] Sven-Bodo Scholz et al. Single Assignment C Tutorial, version 1.2. http://www.sac-home.org/lib/exe/fetch.php?media=docs:tutorial.pdf. [Online; accessed 20-01-2020].

[5] Troels Hendriksen et al. Futhark documentation 0.14.0. https://futhark.readthedocs.io/en/latest/language-reference.html. [Online; accessed 22-01-2020].

[6] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[7] Gabriele Keller, Manuel Chakravarty, Roman Leshchinskiy, and Simon Peyton Jones. Regular, shape-polymorphic, parallel arrays in haskell. In *ICFP'10*, January 2010.

[8] lehins. massiv README. https://github.com/lehins/massiv/blob/master/README.md, 2017-2019. [Online; accessed 06-01-2020].

[9] Trevor L. McDonell, Manuel M T Chakravarty, Vinod Grover, and Ryan R Newton. Type-safe Runtime Code Generation: Accelerate to LLVM. In *Haskell '15: The 8th ACM SIGPLAN Symposium on Haskell*, pages 201–212. ACM, September 2015.

[10] Trevor L. McDonell, Manuel M T Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP '13: The 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2013.

[11] Bryan O'Sullivan et al. Criterion Robust, reliable performance measurement and analysis. `https://hackage.haskell.org/package/criterion`, 2009-2016. [Online; accessed 20-01-2020].

[12] Jonathan Ragan-Kelley. Decoupling algorithms from the organization of computation for high performance image processing, June 2014.

[13] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[14] Sven-Bodo Scholz. *Single Assignment C – Entwurf Und Implementierung Einer Funktionalen C-variante Mit Spezieller Unterstützung Shape-invarianter Array-operationen.* PhD thesis, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität, Kiel, Germany, 1996. Shaker Verlag, Aachen, 1997.

:wq