Universiteit Utrecht

Faculty of Science
Dept. of Information and Computing Sciences

Accelerating Nested Data Parallelism: Preserving Regularity

Author

Lars van den Haak

Supervisor

Dr. Trevor L. McDonell
Prof. dr. Gabriele K. Keller

August 31, 2019

Abstract

In the world of parallel programming, the data parallel style is well suited for the GPUs. However nested data parallelism forms a difficult problem to solve, under the condition that we produce fast code. However, we need nested data parallelism for expressiveness in algorithms.

In this thesis, we use Accelerate, a domain specific language embedded in Haskell, to investigate this problem. We look at the *gridding* algorithm, which is related to data processing for radio telescopes but also contains nested data parallelism. A vital component of those calculations is performing fast Fourier transformations, so we look specifically at different ways to speeds these up. We benchmarked these and found that, if we can stick to arrays that have the same shape called regular arrays, we obtain a good speed-up. We discuss the cases of when we can stay regular, also in intermediate calculations. We present two analysis compilers can use to remain regular. We call these the *Independence* and *Shape* analysis. We also implemented these in the Accelerate compiler. Finally, we show benchmarks of the gridding algorithm, that can use these analyses to stay regular. We conclude that this a step in the right direction for solving nested data parallelism, but that there are still other factors to consider.

Keywords: **Arrays, Data parallelism, Functional programming, GPGPU, Haskell, Nested data parallelism**

Contents

1 Introduction

In modern systems, it is getting harder to utilise the full power of our machine because hardware moved from sequential to parallel processing. Graphics Processing Units (GPUs) have more raw processing power than most Central Processing Units (CPUs), and most CPUs nowadays are built with multiple (virtual) cores. Writing fast and correct parallel code is not an easy task: manually vectorising a program[1] is difficult, as is using low-level GPU frameworks, such as CUDA and OpenCL.

There are two main ways to tackle parallel computing: task parallelism and data parallelism. We consider the latter. This approach means we apply the same function, a *vectorised* function, to elements in a collection (data) in parallel. This approach is naturally suited for GPUs and SIMD (Single Instructions Multiple Data) instructions on the CPU[2]. For this research, we use Accelerate [6], which offers a convenient, abstract, high-level and safe interface for data parallel computations. However, Accelerate restricts itself mostly to flat data parallelism, with some experimental support for one level deep nested data parallelism [8]. Nested data parallelism occurs when the elements on which we apply our functions are collections itself again. For instance, when doing a particle simulation, we loop over all the particles, but each particle needs to check the forces that all the other particles apply on it. Many problems are easier to express with this nesting. The goal of this research is to investigate how we can extend this experimental work, and more importantly, how we can achieve fast code with nested data parallelism.

As a case study, we have the Square Kilometre Array (SKA), which is a radio telescope that is being built right now and will be the world biggest telescope. It is based on aperture synthesis to make an image of the sky in the radio bandwidth. The data that is processed exceeds the daily global internet traffic. Because of these huge amounts of data to be processed, it is essential to utilise the computing power to the maximum. The gridding algorithm is one of the computations that the SKA has to perform. We implemented this algorithm and found that to utilise the available hardware fully, we needed to express the program with nested parallelism. More specifically, the algorithm spent most of its time doing multiple convolutions, which we implemented via Fast Fourier Transforms (FFTs). Thus, our focus is on investigating how to speed up multiple FFTs.

With nested parallelism, we can make the distinction between regular and irregular data, which is processed[4]. Data is regular if all elements of a (nested) collection have the same shape. For example, a collection of vectors where each one has length 5. If the data is not regular, we call it irregular. Since some functions can alter the size of data, the compiler cannot always assume that the data remains regular, thus must switch to an irregular representation. Which means that the functions we apply must work with irregular data. We investigate the performance difference between these data representations and what ideas the compiler can use to remain regular.

1.1 Research Contributions

We conducted this research in the context of Accelerate; the main contributions are:

1. We give two analyses, which allow the compiler to preserve regularity.
2. We show that doing multiple FFTs is faster when expressed with nested parallelism, than with a loop statement.
3. We show that when doing FFTs on nested data, that being regular is significantly faster than being irregular.
4. We show that nested data parallelism is also faster than flat data parallelism in a more complex setting, namely the gridding algorithm.

We split this thesis into three parts. In the first part, we work towards preserving regularity in nested data parallelism. First we explain more about data parallel programming in general (Section 2) and more specifically about Accelerate and its limitations (Section 2.2). The process of making a function work on nested data is called *vectorising.* We explain how the compiler does this and discuss the differences between regular and irregular nested data (Section 3). Furthermore, we discuss how to determine statically whether we can preserve regularity, give our two analysis that helps the compiler in this, and how we implemented this (Section 4). The second part is a case study, about the gridding algorithm for the SKA in which we encounter nested data parallelism. Here

[1]Using SSE instructions
[2]Like the SSE and AVX instructions for the x86 architecture.

we can use the results of the last part, and we can see how this positively impacts performance. First we give some background on the gridding algorithm (Section 5). Next, we go into detail how to vectorise FFTs and show proof for contributions 2 and 3 (Section 6). After that, we give benchmarks of the gridding algorithm (Section 7), which serves as proof for our last contribution.

The final part is the remainder of our thesis. We elaborate on some other recent research done on data parallel languages (Section 8) and how this compares to our work (Section 8.3). Finally, we discuss how one could extend on this work and use it in other cases (Section 10).

Part I

Preserving Regularity with Nested Data Parallelism

2 Data Parallel Programming

First, we briefly review parallel computing in a general setting. Next, we discuss the data parallel language Accelerate, which we used to investigate the problem of nested data parallelism.

2.1 Parallel Computing

Parallel computing means that we compute things at the same time. In contrast to sequential computing, where we compute things one after another. We have two major programming styles to address parallel computing:

1. With *task parallelism* we have several tasks, which the computer divides over the processing units. Each task can be completely different.

2. With *data parallelism*, we apply the same function on different data. Thus each processing unit performs the same functions.

Parallel computing brings some new problems and opportunities. In sequential computing, we cannot use all the available hardware, since we must calculate each part of the code must in a certain order. This does mean that the results are predictable; things happen in the order we write them down. When we are doing parallel computations, this order can be different each time we run the program. Thus, if we try to print two words to the computer screen, they can come in any particular order.

Central Processing Units (CPUs) are suited for both forms of parallelism. They can consist of multiple (virtual) cores, and each core can do a task separately. There are also vectorized instructions available on CPUs, for example, SSE, and we can only use them in a data parallel programming style. A typical CPU nowadays has 4 or 6 hardware cores, for example, the Intel(R) Core(TM) i7-6800K CPU has 6 cores, and a performance of 55.14 GFLOPs (floating point operations per second)[3].

In recent years Graphical Processing Units (GPUs) have become more important for parallel computations[21]. They are mostly suited for data parallelism. A GPU consists of 100s of cores and is designed to handle 10 000s of threads at once. These can be used to hide latency in the calculations; the GPU can switch to another thread while waiting for data. The GPU also has dedicated memory, where we must explicitly transfer our data to. This means that data sent to the GPU takes some time to arrive. We can send this data asynchronous, during which the GPU can perform other calculations. So a GPU needs to have enough parallel actions, for a decent performance. Albeit, it also has a lot more computation power compared to a CPU. For instance, the NVIDIA GeForce GTX 1080 Ti, which is a high-end GPU, has a theoretical peak performance of 11.34 TFLOPS[4] for floating numbers of 32 bit.

Hence, there are different programming styles in parallel computing. The architectures on which they work behave quite different, which needs to be taken into account if we want to achieve the maximum performance.

2.2 Accelerate

Accelerate [6, 8, 19, 20] is a domain-specific high-level language of array computations embedded in Haskell that mainly uses SIMD parallelism. It relies on collective array operations (parallel actions), based on the scan-vector model [7], to do the computations. See Figure 1 for two examples of these actions. These operations build on algorithmic skeletons to produce code. The code is generated dynamically, which causes some overhead, but can also exploit runtime information to optimize the code. We could even generate new functions at runtime. The overhead is minimized by caching binaries of previously generated skeleton instantiations and running this

[3] `https://asteroidsathome.net/boinc/cpu_list.php`

[4] This is theoretical of course, but it shows an order of magnitude. `https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877`

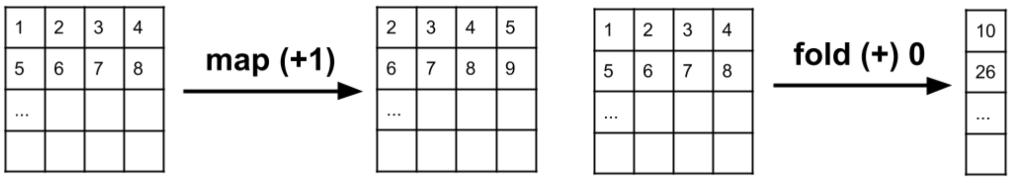


Figure 1: The higher dimensionality support of Accelerate allows the `map` and `fold` functions to work on all the dimensions at the same time and the `fold` can process the dimensions separately.

step in parallel. Although there is also a possibility to compile the language, during the compilation of Haskell.[5] Accelerate has two back-ends: one compiles to CPUs, the other to GPUs.

Accelerate works with arrays that are typed like `Array sh e`, where `sh` is the shape and `e` the type of the elements. The shape is an inductive type and defined by:

```
data Z = Z
data tail :. head = tail :. head
```

The type `Z` is the rank-0 array (Scalar), and for example, the shape `Z:.5` has type `Z:.Int` and is a one-dimensional array (Vector) of length 5. The types are not explicitly set to `Int`, because this allows us to define slices of arrays. For instance, the shape `Z:.All:.1` can be used to specify row number 1 from a 2-dimensional array. The data type and constructor `Any` is used to match an arbitrary outer dimension. Common type synonyms for some shapes and arrays are:

```
type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
type DIM3 = DIM2 :. Int

type Scalar e = Array DIM0 e
type Vector e = Array DIM1 e
type Matrix e = Array DIM2 e
```

The elements of arrays can be shapes, primitive types like integers, floating points types, chars, and Booleans or (nested) tuples of these types. We cannot store functions, other arrays or recursive data types as elements. An array with type `Array sh (Int, Int)` is stored as two contiguous blocks in memory, one block for each tuple index. We can add wwn user types, but the same restrictions apply.

We mark arrays to be used by computations are explicitly by type constructor `Acc`, we use the `use` function for this:

```
use :: (Shape sh, Elt e) ⇒ Array sh e → Acc (Array sh e)
```

For the GPU back-end, this means that the compiler transfers these arrays to the device memory.

Accelerate marks embedded scalar computations with the `Exp` type constructor. They are just like the `Acc` computations and Accelerate will execute them. These calculations can be all the standard arithmetic operations, some basic form of iteration and flow control, but there is no form of recursion. Specifically, the scalar computations cannot contain collective operations; thus this excludes general nested data-parallelism.

The fold pattern reduces the inner dimension of an array.

```
fold :: (Shape sh, Elt a)              -- It must have a valid shape, and valid elements
     ⇒ (Exp a → Exp a → Exp a)      -- A combinator function, that must be commutative
     → Exp a                           -- An initial value
```

[5]This is done via Template Haskell, which via metaprogramming. This is only available in the newer versions of Accelerate, not in the branch most of our work takes place unfortunately.

```
  → Acc (Array (sh :. Int) a)   -- The array for which the outermost dimension is reduced
  → Acc (Array sh a)            -- The resulting array,
```

For example see Figure 1, the code `fold (+) 0 xs` where `xs` is a two-dimensional array, calculates the sum of each row and produces a 1-dimensional array with the sums as a result. This is a rank polymorphic function, meaning that it can work on an array of arbitrary rank (dimensionality). This rank polymorphism allows some form of nested data parallelism. For instance, we just applied a reduce on a list of vectors (matrix). However, note that this limited to a few functions, so this is not a general solution to nested data parallelism.

The standard collective array operations include `map`, `fold`, `scan` and `stencil`. Indexing arrays and zipping them is also possible. We can modify shapes, extract sub-arrays, replicate dimensions, and we can use `gather` and `scatter` operations. Conditional and loop statements are also available on the array level.

The embedded language does not execute code immediately, instead it builds a term tree to represent the embedded computations. The surface language uses a higher-order abstract syntax where we preserve type information. We call this tree an AST (Abstract Syntax Tree). Accelerate enforces sharing, such that terms will not get calculated multiple times if explicitly used again. Thereupon the AST is made nameless via de Bruijn indices. After these transformations, optimizations and analyses that are independent of the back-end take place. For example `map f (map g xs)` can get turned into `map (f ∘ g) xs`, which we call array fusion. The first builds an intermediate array and effectively traverses the data twice, while the latter only passes over the data once. The front end makes sure we can work with tuple types; it turns the array of tuples into a tuple of arrays. This is better for the memory access for SIMD architectures. [6]

The back-end consists of actually generating code for the correct platform. This is done via LLVM[7], the compiler ports the AST to the LLVM intermediate language where it preserves types. There are currently two back-ends based on LLVM, one for multi-core CPUs and one for GPUs targeting CUDA. These back-ends use the same LLVM intermediate language, but use different algorithmic skeletons, since they have other ways in which they perform optimally. For CPUs, a task can be divided over the available cores, while GPUs have more threads and thus requires a more fine-grained division of work to perform optimally. Hence, for the different back-ends, the scalar computations are the same in the intermediate language, but the array operations have different implementations. A compiled function for these back-ends is called a kernel. The compiler hashes generated kernels, so they can be easily compared and recovered if used again.

Clifton-Everest et al. [8] made an effort to let Accelerate work with nested data parallelism. We go into more details of this in Section 3. Note that this work is still somewhat experimental and it did not work with all the Accelerate functionalities yet.

Accelerate is thus a data-parallel language that can compile to the CPU and GPU, and in which some form of nested data parallelism can be dealt with.

[6]This way of storing data is called Structure of Arrays (SoA). See this article for more information: `https://software.intel.com/en-us/articles/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture`

[7]The LLVM Project is a collection of modular and reusable compiler and toolchain technologies, commonly used for compiler back-ends. See `https://llvm.org/`

```
averageLiftedRegular :: Acc (Matrix Int) → Acc (Vector Int)
averageLiftedRegular xs = map (/n) sum
  where
    sum = fold1 (+) xs
    n   = (fromIntegral ∘ indexHead ∘ shape) xs
averageLiftedIrregular :: Acc (IrregularArray DIM1 Int) → Acc (IrregularArray DIM0 Int)
averageLiftedIrregular xs = (newSegs, newVals)          -- Stepping up
  where
    (segs, vals) = xs                                    -- Stepping down
    segsInt      = map indexHead segs                    -- Getting lengths of segments
    newSegs      = map indexTail segs                    -- The new segment descriptor
    sum          = fold1Seg (+) vals segsInt             -- Using the lifted version of fold1
    n            = map (fromIntegral ∘ indexHead) segs   -- Lifted (indexHead . shape) xs
    newVals      = zipWith (/) sum n                     -- Lifted map (/n) sum
```

Listing 1: Lifted average function, for a regular nested vector and an irregular one.

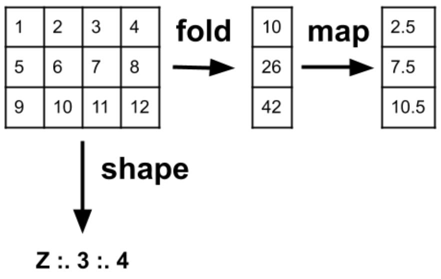


Figure 2: A visualisation of how Accelerate can calculate the average from multiple vectors (rows) at the same time.

3 Flattening Nested Data Parallelism

In this section, we will explain how the compiler can transform nested data parallelism into flat data parallelism. We call this transformation *vectorization* and the compiler can do this automatically. The ideas for this were first explored by Blelloch and Sabot [3]. We will explain this in more detail in Section 3.2. First, we give an intuition how we can do this; next we formalise this, and lastly, we compare the notion of regular and irregular nested data. Through this whole section, we will stick mostly to nested data parallelism of 1 level deep, since Accelerate supports this due to the work of Clifton-Everest et al. [8]. Although, we can use the vectorisation transformation for arbitrarily nested data parallelism. We base most of the information we give here on the two works we just mentioned.

3.1 An intuition of vectorising in the regular case

If we want to calculate the average of multiple vectors, this is in essence, nested data parallelism. We want to do a parallel action over a collection (list of vectors), of which the elements itself are arrays again. When calculating the average, we sum (a fold) all the elements and dividing them by the length of the vector. This is easy to do when all the vectors are the same shape, which we call a *regular* nested array. We can do a higher dimensional fold on the matrix that represents the list of vectors to calculate the average, see listing 1 for the code. Figure 2 gives a visual of how we calculate the average.

Note that we use the same code as for a single vector, but the shape types are one rank higher. This is possible because fold can handle each dimension in an array separately. Although in general, the vectors can be

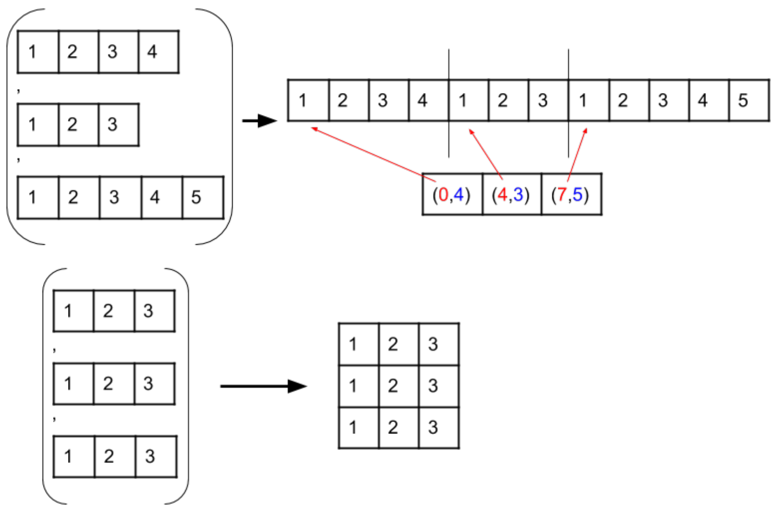


Figure 3: We can represent a list of arrays in two ways. In general as an irregular array, where we keep track of the shape dimensions (above), or, if all dimensions are the same, as a regular array (below).

of a different shape, which we then call an *irregular* nested array. To calculate the average on this list in parallel requires us to rewrite our program. In general this can be a lot of work, especially if we are using library functions since we would need to rewrite them as well.

3.2 Flattening Nested Data and Lifting Functions

We want to convert to flat data parallelism. Therefore the arrays that we work on must be flat, although they can be higher-dimensional. We call this *flattening* of the data. The functions we consider are normally written for flat data parallelism. To let them work on the flattened data, we need to adjust them. We call this *lifting*, and we will explain this after we have explained the flattening.

3.2.1 Flattening Nested Data

If we have a list of 5 vectors each of size 3, we can represent this a matrix that has dimension 5×3, or as an Accelerate shape `Z:.5:.3`. This is true in general, we can consider a regular nested array as an array of one dimension higher. The number of different arrays will be equal to the leftmost (outermost and slowest changing index) dimension of the new array. See Figure 3 for a visual representation of a regular and an irregular array.

Let's consider a list of three vectors, each with different sizes, e.g. $[[1,2],[3,4],[2]]$. This is an irregular nested array, but we want to express this as a flat data structure. We can do this using a *segment descriptors* and a *value vector*. The segment descriptors, describe how we divide the data. E.g. we convert the nested list above into:

```
segdes1 = [3]
segdes2 = [2,2,1]
value = [1,2,3,4,2]
```

Where `segdes1` describes that we have 1 list of lists and this lists has 3 elements; thus it describes `segdes2`. The descriptor `segdes2` tells us that the first list has 2 elements, the second also 2 and the last list has 1 element. For data that is nested deeper, we add more segment descriptors. Since Accelerate only supports one level deep nested parallelism, we can omit the first segment description, since it will always be one element that contains the length of the second segment descriptor.

Because the arrays can be multidimensional, the segment descriptor is a list of shapes. Therefore, if we have two matrices, the first with dimensions 5×3 and the second 2×4, the segment descriptor would be `[Z:.5:.3, Z:.2:.4]`. Additionally, we keep track of the starting index position of each array for performance reasons, but we ignore that for now. In Accelerate we name this the `IrregularArray` type:

```
type IrregularArray sh e = (Segments sh, Vector e)
```

3.2.2 Lifting Functions

To let functions work with nested data parallelism, we consider four parts. The first part is that we need to make a serial and a parallel version of each function. The parallel one is used for nesting and is called the *lifted* function. For example, for the multiply operation, we need one that multiplies only one element, and a lifted one that can multiply whole lists.[8] When the compiler makes a parallel version, it will replace function calls within the lifted function by their parallel counterparts and most things can stay the same, but some statements like if-then-else constructs require special care.

The second part is about the map function. Again, the compiler makes two versions. Then any free variables (not captured in the body of the function), are replicated, to have the same dimensions as the list mapped over. E.g. for **let** b = 3 **in** map (+b) [1, 2, 3] the variable b is free, thus we make b into [3, 3, 3]. These variables can be scalars, but can also be list themselves. In the work of Clifton-Everest et al. [8] an effort is made to avoid unnecessary array structures that arises from this step and is named vectorisation *avoidance* [16].

The next step are the *stepping-down* and *stepping-up* manipulations. Consider the nested list again, where we want to do a double map over:

```
let xss = [[1, 2], [3, 4], [2]]
in map (λxs → map (λx → x + x)) xss
```

Where xss has the internal description of ([3],([2,2,1],[1,2,3,4,2])). We first do a stepping down by removing the segment descriptors, so we can apply our vectorized functions which work on only flat (or regular) data structures. Hence, we get:

```
let xs = [1.2, 3, 4, 2]
in xs+_V xs
```

Where we have a lifted add $+_V$. After the calculation, we step up again, adding the segment description. This will eventually yield the answer [[2, 4], [6, 8], [4]] where we do the underlying calculations in parallel. This can be done for arbitrary complex instructions within the map, although we should take more care for fold like instructions, where the segment descriptor is also needed. In Accelerate the vectorised add $+_V$ would be equal to `zipWith` (+).

In our example above, we gave the lifted version of average in the regular case; we give the irregular one in listing 1 as well. Getting and returning the segments and values are the stepping-up and down manipulations, `fold1Seg` is the lifted version of `fold1`, which takes an extra segment argument. The lifted version of `indexHead` is now a map of this function and `shape xs` is replaced by the segment descriptor. There were no free variables to consider. Since we are using a fold, the new segment descriptor shapes have the innermost dimension removed — this idea correspondents with taking the tail of the shapes. The actual lifting would look slightly different, but the idea is the same.

3.3 Sequences in Accelerate

In Accelerate some of these ideas are implemented [8], and we call a nested list of arrays a *sequence*, captured by the Seq data type. Since each list is completely independent in this representation, we can also do *streaming* calculations. This means that we do not process all input at once but in parts. This can help in two ways, firstly if the input is too large for the memory to process at once, and secondly for fine-tuning how much parallel work the computer does at the same time.

On a high-level view there happen three things when processing sequences. First, a sequence is *produced*, then it is *traversed* (sequence transducers) and finally it is consumed again, making it an array level term again. A typical traverse is mapping an array function to each element of the sequence:

```
mapSeq :: (Arrays a, Arrays b) ⇒ (Acc a → Acc b) → Seq [a] → Seq [b]
```

In Accelerate, the type relation between an array and its lifted representation is given by $\mathcal{V}$ `t t'`. This encodes if we have a regular array, an irregular one, or if we avoid it. It will also do the same for tuples of arrays.

[8]On the CPU this can be made into vectorised multiply, which is a SIMD instruction.

```
data 𝒱 t t' where
  AvoidedT   :: 𝒱 (Array sh e) (Array sh e)              -- avoid vectorisation
  RegularT   :: 𝒱 (Array sh e) (RegularArray sh e)       -- regular context
  IrregularT :: 𝒱 (Array sh e) (IrregularArray sh e)     -- irregular context
  TupleT     :: (𝒱 t₁ t_1', 𝒱 t₂ t_2', ..., 𝒱 t_n ·, t_n')
             → 𝒱 (t₁, t₂, ..., t_n) (t_1', t_2', ..., t_n')
```

Accelerate deals with the nested parallelism as we explained above, but Clifton-Everest et al. [8] made two improvements:

- Say we have two vectors, and we want to calculate the average (`zipWith` (λx y → (x + y) / 2) `xs ys`). If we do not take special care, the 2 will be lifted to an array containing only 2's, which is unnecessary. If we do not do this, we call this *vectorisation avoidance.*
- Treat the case of regular nested arrays as a special case.

The first is an improvement because otherwise, it creates more structures in memory. Also, it could have been a calculation instead of a constant. Which means each computation is performed again for the lifted function, although the result of this variable would always be the same.

The result of the vectorisation is scheduled to the available machine hardware dynamically. It will start with one chunk of the sequences, and doubles it until the average executing time of a chunk will decline. Note that this work is still experimental and it did not implement all the Accelerate functionalities yet. For example, the lifting of `awhile` construct was not implemented yet. We will now go into more detail why it is a good idea to have regular nested arrays and how this improves performance.

3.4 Irregular vs Regular

Regular nested arrays have several advantages over irregular ones. First off, since the dimensions are all the same for regular arrays, we have less data to keep track off, consequently, the memory footprint is smaller. Moreover, when we change the array shape, it takes additional work to update the segment descriptor in the irregular case. If we look at the lifted average function in listing 1, the updating of the segment descriptor takes an additional parallel action.

Secondly, most irregular lifted functions take extra (parallel) steps. In the average function the lifted `fold1` needs an additional step, to get the correct part of the segments and the `indexHead` is turned into a parallel action. Whereas in the regular case, we do not need all of these steps. Hence, in the irregular case, we will execute some steps as a parallel step instead of a scalar step. Although the longest sequential path of executions that needs to the computer does stay the same in that case, the total parallel work does increase. So it might not always matter in execution time, but a scalar step will never be slower.

Returning to our average example, in the regular lifted version, we have 2 parallel steps (`fold1`, `map`) and 1 scalar step. In the irregular version, we have 5 parallel steps, 4 steps after the fusion optimisation, and 2 scalar steps (stepping up and down).

The overhead of irregular functions grows fast, since each function the compiler needs to lift, needs irregular versions of their used functions again. Thus the longer we can stay regular, the better. This is especially true for computations that are at the beginning of our pipeline. Since if these must work on an irregular array, the following functions must assume irregularity as well. In Section 6, we will give some proof for the claims that calculations on regular nested arrays are indeed faster, with the example of FFTs.

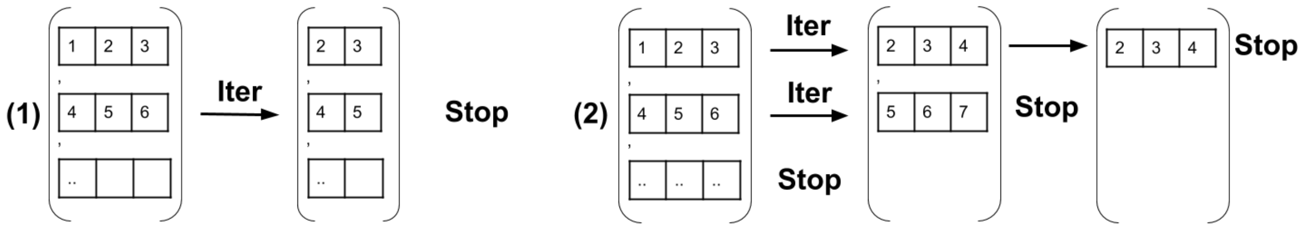


Figure 4: There are two ways to preserve regularity in an `awhile` function. (1) If the iteration will always stop at the same time. (2) If the iteration function always keeps the same shape. This is under the condition that the input is and the iteration function stays regular.

4 Preserving Regularity

In the last section, we argue that an irregular lifted version is slower than a regular one. Moreover, we will quantify this in the case of FFT calculations in Section 6. This leads to the question, when is it safe to keep calculating with regular nested data? Also, can we determine this statically? In theory, we could choose to check dynamically at run-time when we are regular, which we will briefly discuss in the future work (Section 10). However, for now, we consider the static case. In the next section, we talk about a few ways to determine when we can remain regular, which gave ideas for two static analyses: the *independent* and *shape* analysis. We implemented both of them in the Accelerate compiler. We talk more about these in the remainder of this section.

4.1 Statically Determine Regularity

We can determine statically if the compiler can preserve regularity in several ways. As an example, take a regular list of matrices, where each has dimensions 5×2. If we apply a `map` function to each matrix, they will keep their shape. The result is still a regular list of matrices. If we apply fold, we reduce each matrix to a vector, but in the exact same way. So for functions that do not change the shape, or alter it predictable, we preserve regularity as long as the input is regular. Furthermore, if we are dealing with scalars (dimension 0) or the results turns into a scalar (e.g. by a `fold`), the result is regular since a scalar can only be one shape.

Now take `generate` (`shape xs`) (`xs` !! 0), which is a new array that has the same shape as `xs` and fills it with the first element of `xs`. If we want to vectorise this, it will generate a list of arrays. However, this is only regular, if all the shapes are the same. So this is only the case if the `xs` we are pointing at is either a normal array or a regular list of arrays. If the array is normal, we can even *avoid* vectorisation, since all the elements are the same. These kinds of functions (`generate`, `reshape`, etc.) get their shape as an argument. If this expression is *independent*, meaning constants or shapes of regular arrays only determine it, the expressions are the same in the sequence calculation, and we stay regular.

4.1.1 Control flow functions

The last category we consider is control flow functions. In Accelerate's AST there are two present on array level, being `acond` and `awhile`. About the conditional (`acond` condition true-branch false-branch) we can determine regularity in two ways. If the condition (which is an expression) is independent, then the result is the same for all the arrays in the sequence. Thus if both the 'if' and 'else' branches are regular (in general we cannot yet determine which branch we take), the result is regular. The other approach is when again, both the branches are regular, but also have exactly the same shape. If we vectorise this, each part of the sequence can pick its branch. It does not matter since it is the same shape.

If we look at `awhile`, we have a predicate function, returning true or false denoting whether the loop stops, the iteration function and the initial array. There are again two ways to preserve regularity, see also Figure 4. The first: we want to determine that the predicate is independent, thus only dependent on constants or array shapes with regular shapes. Then the predicate returns the same result for each iteration. Thus all the elements require the same number of iterations. If the input is regular and the iteration function preserves regularity, then we can guarantee that the output is regular as well. In the split radix FFT algorithm that was implemented in

Accelerate (which we use in the gridding algorithm, Section 5.3), the latter was the case. Therefore, we can use this to prove regularity.

The second way to stay regular is if we look at the shapes of the arrays that the iteration function returns. If it returns exactly the same shape, as what was put in, then it does not matter where each loop stops in the sequence of arrays. The output will eventually have the same shape. However, we require that the initial array is at least regular, the iteration function should be regular as well since we already determined it returns the same shape.

These observations lead to two analysis, the *independent analysis* and the *shape analysis.* We will discuss them both in the remainder of this section, but first we will give some definitions.

4.2 Definitions for analysis

We need a grammar to give our analyses in, so we define it here. It is partly based on the nested data-parallel core language used in Clifton-Everest et al. [8, See Figure 1.]. But in contrast to that work, we will make a distinction between expressions and arrays, because we apply our analysis on a flat data parallel language. Actually it is very similar to Accelerates core AST, but we left out some things for simplicity.

$$
\begin{array}{llll}
\texttt{expression variables} & \texttt{x} & ::= \texttt{x}_0 \mid \texttt{x}_1 \mid \dots & \\
\texttt{array variables} & \texttt{xs} & ::= \texttt{xs}_0 \mid \texttt{xs}_1 \mid \dots & \\
\texttt{literals} & \texttt{l} & ::= 0 \mid 1 \mid 2 \mid \dots & \\
\texttt{booleans} & \texttt{b} & ::= \texttt{True} \mid \texttt{False} & \\
\texttt{constants} & \texttt{c} & ::= \texttt{l} \mid \texttt{b} & \\
\texttt{shapes} & \texttt{sh} & ::= \texttt{Z} \mid \texttt{sh} :. \texttt{e} \mid \texttt{shape a} & \\
\texttt{expression tuples} & \texttt{t} & ::= (\texttt{e}_0, \dots, \texttt{e}_\texttt{n}) & \\
\texttt{array tuples} & \texttt{t}_\texttt{a} & ::= (\texttt{a}_0, \dots, \texttt{a}_\texttt{n}) & \\
\texttt{primitive} & \texttt{p} & ::= (+) \mid (*) \mid (-) \mid \texttt{indexHead} \mid \texttt{indexTail} \mid \texttt{shapeSize} \mid \dots & \\
\texttt{expression} & \texttt{e} & ::= \texttt{x} \mid \texttt{c} \mid \texttt{sh} \mid \texttt{t} & \\
 & & \mid \texttt{p e} & \textit{-- Apply primitive function} \\
 & & \mid \texttt{a ! sh} & \textit{-- Index array a at sh} \\
 & & \mid (\lambda \texttt{x}_0 \rightarrow \texttt{e}_1)\ \texttt{e}_2 & \textit{-- let x0 = e2 in e1} \\
 & & \mid \texttt{prj l e} & \textit{-- Take the lth element of the tupl} \\
\texttt{arrays} & \texttt{a} & ::= \texttt{xs} \mid \texttt{t}_\texttt{a} & \\
 & & \mid (\lambda \texttt{xs}_0 \rightarrow \texttt{a}_1)\ \texttt{a}_2 & \textit{-- let xs0 = a2 in a1} \\
 & & \mid \texttt{map}\ (\lambda \texttt{x}_0 \rightarrow \texttt{e})\ \texttt{a} & \textit{-- Apply a function to each element} \\
 & & \mid \texttt{fold}\ (\lambda \texttt{x}_1\ \texttt{x}_0 \rightarrow \texttt{e}_1)\ \texttt{e}_2\ \texttt{a} & \textit{-- Reduce the innermost dimension} \\
 & & \mid \texttt{zip}\ \texttt{a}_1\ \texttt{a}_2 & \textit{-- Combine as array of tuples} \\
 & & \mid \texttt{aprj l a} & \textit{-- Take the lth element of the tupple} \\
 & & \mid \texttt{unit e} & \textit{-- Make a scalar array from an expression} \\
 & & \mid \texttt{generate sh}\ (\lambda \texttt{x}_0 \rightarrow \texttt{e}) & \textit{-- Make an array of shape sh} \\
 & & \mid \texttt{reshape sh a} & \textit{-- Reshape the array to shape sh} \\
 & & \mid \texttt{acond e}\ \texttt{a}_1\ \texttt{a}_2 & \textit{-- if e then a1 else a2} \\
 & & \mid \texttt{awhile}\ (\lambda \texttt{xs}_0 \rightarrow \texttt{a}_1)\ (\lambda \texttt{xs}_0 \rightarrow \texttt{a}_2)\ \texttt{a}_3 & \textit{-- while a1==true do a2}
\end{array}
$$

When we are doing our analyses, we store the values of the variables (xs and x) in an environment (Γ). We will denote this as $\Gamma \vdash \texttt{a}$ or $\Gamma \vdash \texttt{e}$. When we push value a (e) to variable $\texttt{xs}_0$ ($\texttt{x}_0$) in the environment, this is denoted as $\texttt{a} : \Gamma$ ($\texttt{e} : \Gamma$). So $\texttt{xs}_0$ will point to the top value of the environment, $\texttt{xs}_1$ to the one after that, and so on. Although in our analyses, we will store our analysis results for that specific variable.

4.3 Independent Analysis

We want to determine that a predicate function in an awhile application, always gives the same answer when we are vectorising (The first case of Figure 4) .This can be the case for the following function[9]:

```
predEx :: Acc (Vector Int) → Acc (Scalar Bool)
predEx xs = let sh = shape xs
```

[9]Yes this is not the grammar we just defined, but we can easily translate it, and now we can name our variables.

```
        n   = indexHead sh
     in unit (n > 1)
```

There are three situations to consider, the input array is: a regular arrray, an irregular array, or an array that is not vectorised at all (avoided). The last case we ignore for now, since we could just skip vectorisation. Now if `xs` is regular, then `shape xs` will be the same for each application of this predicate function, thus so is `sh`. Now `n` must be the same as well, 1 is a constant and $>$ is a primitive function, thus so is $\texttt{n} > 1$. Now the result $\texttt{unit}\ (\texttt{n} > 1)$ is the same for all the applications of this function. Thus it always gives the same answer. If we have an irregular array at the start, the only thing we can determine is that the result always has the same shape, since we end up with a scalar (which we could see immediately with the type it has).

4.3.1 Formalization

We formalize this in the independent analysis. We use it to determine if a term is only dependent on constants factors or shapes of regular arrays. So more specifically it is not dependent on array elements, or on randomly generated factors (e.g. an RNG that we can acces via a foreign function). As we mentioned, this can help to stay regular in the `awhile` function. The compiler can use this analysis for `generate` (& others) as well, but this is implicitly already done. During vectorisation, it will mark certain arrays or expressions that can avoid vectorisation. We can only avoid, when it is fully independent of all the sequences computations, meaning we would only need to calculate it once (instead of once for each array in the sequence list). So it does the same. However, the `awhile` function is special since the predicate function it uses is on array level. Thus some adjustments had to be made.[10] So if the independence of an array level function needs to be checked, this analysis can be used. We keep track of two things in the analysis. Firstly if the shape of an array is dependent, and secondly if the elements of an array or an expression itself are dependent. This leads to an ordered ranking (we can view it as a simple lattice):

1. *Not Independent* (`NotI`), the shape and elements could be dependent.
2. *Shape Independent* (`ShapeI`), the shape must be independent.
3. *Totally Independent* (`TotalI`), the shape and elements must be independent.

We could think of a fourth rank, only independent elements, which would lie on the same level as `ShapeI`. However, we can capture almost all the information necessary with these three already, so we skipped this for now.

If we have a `TotalI` predicate function, we can preserve regularity. We use the ordering, to always guarantee a safe use, and we err on the safe side. We will denote our Independence Analysis with $\mathcal{I}_A[\Gamma \vdash \texttt{a}]$, where Γ is the environment so that we can look up the dependence of variables. We can join two results with the $\wedge$ (join) operator, which will give a safe guarantee. We can also combine them with the $\vee$(meet) operator if we are sure we can give a better guarantee. For example, $\texttt{TotalI} \wedge \texttt{ShapeI} = \texttt{ShapeI}$ and $\texttt{TotalI} \vee \texttt{NotI} = \texttt{TotalI}$.

The predicate at the beginning of this subsection would give us:

```
𝓘_A[Γ ⊢ predEx xs] = case xs of
  TotalI → TotalI
  ShapeI → TotalI
  NotI → ShapeI
```

As we already discussed above, we are independent when the input is either regular or avoided. Actually, there is a close relationship between the vectorised arrays. An array that the compiler did not vectorise (avoided) is `TotalI`, the shape and elements are the same for each call of the variable in the sequence. A regular array will always have the same shape, thus is `ShapeI`. Moreover, from an irregular arrays we really cannot determine anything, so it is `NotI`.

In Listing, 2, we give most of the rules that we use for our analysis. We apply the rules on all the terms that are present in the grammar, that was presented in Section 4.2.

[10] Actually, we made the analysis completely independent of this vectorisation avoidance, but probably parts can be combined.

$$
\begin{aligned}
&\mathcal{I}_A[\Gamma \vdash (\lambda \mathtt{xs}_0 \rightarrow \mathtt{e}_1)\ \mathtt{e}_2] &&= \textbf{let}\ \Gamma' = (\mathcal{I}_A[\Gamma \vdash \mathtt{e}_2]) : \Gamma \\
& && \quad\ \textbf{in}\ \mathcal{I}_A[\Gamma' \vdash \mathtt{e}_1] \\
&\mathcal{I}_A[\Gamma \vdash \mathtt{xs_i}] &&= \mathtt{lookup}\ \Gamma\ \mathtt{xs_i} \\
&\mathcal{I}_A[\Gamma \vdash \mathtt{map}\ (\lambda \mathtt{x}_0 \rightarrow \mathtt{e})\ \mathtt{a}] &&= \textbf{let}\ \Gamma' = (\mathcal{I}_A[\Gamma \vdash \mathtt{a}]) : \Gamma \\
& && \quad\ \textbf{in}\ \mathcal{I}_A[\Gamma' \vdash \mathtt{e}] \wedge \mathcal{I}_A[\Gamma \vdash \mathtt{a}] \\
&\mathcal{I}_A[\Gamma \vdash \mathtt{fold}\ (\lambda \mathtt{x}_1\ \mathtt{x}_0 \rightarrow \mathtt{e}_1)\ \mathtt{e}_2\ \mathtt{a}] = \\
&\quad \textbf{let}\ \mathtt{dep} = \mathcal{I}_A[\Gamma \vdash \mathtt{a}] \wedge \mathcal{I}_E[\Gamma \vdash \mathtt{e}_2] \\
&\qquad\quad \Gamma' = \mathtt{dep} : (\mathtt{dep} : \Gamma)) \\
&\quad \textbf{in}\ \mathcal{I}_A[\Gamma \vdash \mathtt{a}] \wedge \mathcal{I}_A[\Gamma' \vdash \mathtt{e}_1] \\
&\mathcal{I}_A[\Gamma \vdash \mathtt{zip}\ \mathtt{a}_1\ \mathtt{a}_2] &&= \mathcal{I}_A[\Gamma \vdash \mathtt{a1}] \wedge \mathcal{I}_A[\Gamma \vdash \mathtt{a2}] \\
&\mathcal{I}_A[\Gamma \vdash (\mathtt{a}_0, \ldots, \mathtt{a_n})] &&= \\
&\quad \mathcal{I}_A[\Gamma \vdash \mathtt{a}_0] \wedge \ldots \wedge \mathcal{I}_A[\Gamma \vdash \mathtt{a_n}] \\
&\mathcal{I}_A[\Gamma \vdash \mathtt{aprj}\ \mathtt{l}\ \mathtt{a}] &&= \mathcal{I}_A[\Gamma \vdash \mathtt{a}] \\
&\mathcal{I}_A[\Gamma \vdash \mathtt{unit}\ \mathtt{e}] &&= \mathcal{I}_E[\Gamma \vdash \mathtt{e}] \\
&\mathcal{I}_A[\Gamma \vdash \mathtt{generate}\ \mathtt{sh}\ (\lambda \mathtt{x}_0 \rightarrow \mathtt{e})] = \\
&\quad \textbf{case}\ \mathcal{I}_E[\Gamma \vdash \mathtt{sh}]\ \textbf{of} \\
&\qquad \mathtt{TotalI} \rightarrow \textbf{let}\ \Gamma' = (\mathcal{I}_E[\Gamma \vdash \mathtt{sh}]) : \Gamma \\
&\qquad\qquad\qquad\ \ \textbf{in}\ \mathcal{I}_E[\Gamma' \vdash \mathtt{e}] \\
&\qquad \_ \qquad\quad\ \rightarrow \mathtt{NotI} \\
&\mathcal{I}_A[\Gamma \vdash \mathtt{reshape}\ \mathtt{sh}\ \mathtt{a}] &&= \\
&\quad \textbf{case}\ \mathcal{I}_E[\Gamma \vdash \mathtt{sh}]\ \textbf{of} \\
&\qquad \mathtt{TotalI} \rightarrow \mathcal{I}_A[\Gamma \vdash \mathtt{a}] \vee \mathtt{ShapeI} \\
&\qquad \_ \rightarrow \mathtt{NotI}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{I}_A[\Gamma \vdash \mathtt{acond}\ \mathtt{e}\ \mathtt{a}_1\ \mathtt{a}_2] &&= \\
&\quad \textbf{case}\ \mathcal{I}_E[\Gamma \vdash \mathtt{e}]\ \textbf{of} \\
&\qquad \mathtt{TotalI} \rightarrow \mathcal{I}_A[\Gamma \vdash \mathtt{a}_1] \wedge \mathcal{I}_A[\Gamma \vdash \mathtt{a}_2] \\
&\qquad \_ \qquad\quad \rightarrow \mathtt{NotI} \\
&\mathcal{I}_A[\Gamma \vdash \mathtt{awhile}\ (\lambda \mathtt{xs}_0 \rightarrow \mathtt{a}_1)\ (\lambda \mathtt{xs}_0 \rightarrow \mathtt{a}_2)\ \mathtt{a}_3] \\
& &&= \\
&\quad \textbf{let}\ \Gamma' = (\mathcal{I}_A[\Gamma \vdash \mathtt{a}_3]) : \Gamma \\
&\qquad\quad \Gamma'' = (\mathcal{I}_A[\Gamma \vdash \mathtt{a}_3] \wedge \mathcal{I}_A[\Gamma' \vdash \mathtt{a}_2]) : \Gamma \\
&\quad \textbf{in case}\ \mathcal{I}_A[\Gamma'' \vdash \mathtt{a}_1]\ \textbf{of} \\
&\qquad \mathtt{TotalI} \rightarrow \mathcal{I}_A[\Gamma \vdash \mathtt{a}_3] \wedge \mathcal{I}_A[\Gamma' \vdash \mathtt{a}_2] \\
&\qquad \_ \qquad\quad \rightarrow \mathtt{NotI} \\
&\mathcal{I}_E[\Gamma \vdash (\lambda \mathtt{x}_0 \rightarrow \mathtt{e}_1)\ \mathtt{e}_2] &&= \textbf{let}\ \Gamma' = \mathtt{e}_2 : \Gamma \\
& && \quad\ \textbf{in}\ \mathcal{I}_E[\Gamma' \vdash \mathtt{e}_1] \\
&\mathcal{I}_E[\Gamma \vdash \mathtt{x}_0] &&= \mathtt{lookup}\ \Gamma\ \mathtt{xs_i} \vee \mathtt{ShapeI} \\
&\mathcal{I}_E[\Gamma \vdash \mathtt{c}] &&= \mathtt{TotalI} \\
&\mathcal{I}_E[\Gamma \vdash \mathtt{Z}] &&= \mathtt{TotalI} \\
&\mathcal{I}_E[\Gamma \vdash \mathtt{sh} :. \mathtt{e}] &&= \mathcal{I}_E[\Gamma \vdash \mathtt{sh}] \wedge \mathcal{I}_E[\Gamma \vdash \mathtt{e}] \\
&\mathcal{I}_A[\Gamma \vdash (\mathtt{e}_0, \ldots, \mathtt{e_n})] &&= \\
&\quad \mathcal{I}_E[\Gamma \vdash \mathtt{e}_0] \wedge \ldots \wedge \mathcal{I}_E[\Gamma \vdash \mathtt{e_n}] \\
&\mathcal{I}_A[\Gamma \vdash \mathtt{prj}\ \mathtt{l}\ \mathtt{e}] &&= \mathcal{I}_E[\Gamma \vdash \mathtt{e}] \\
&\mathcal{I}_E[\Gamma \vdash \mathtt{p}\ \mathtt{e}] &&= \mathcal{I}_E[\Gamma \vdash \mathtt{e}] \\
&\mathcal{I}_E[\Gamma \vdash \mathtt{a}\ !\ \mathtt{sh}] &&= \\
&\quad (\mathcal{I}_A[\Gamma \vdash \Gamma]\ \mathtt{a} \wedge \mathcal{I}_E[\Gamma \vdash \mathtt{sh}]) \vee \mathtt{ShapeI} \\
&\mathcal{I}_E[\Gamma \vdash \mathtt{shape}\ \mathtt{a}] &&= \textbf{case}\ \mathcal{I}_A[\Gamma \vdash \mathtt{a}]\ \textbf{of} \\
& && \quad \mathtt{NotI} \rightarrow \mathtt{ShapeI} \\
& && \quad \_ \qquad \rightarrow \mathtt{TotalI}
\end{aligned}
$$

Listing 2: The rules that we use for the independent analysis applied on the array terms of the simple grammar we defined. The analysis indicates that if arrays or expressions are totally independent (`TotalI`), shape independent (`ShapeI`) or not independent (`NotI`). The function $\mathcal{I}_A[\Gamma \vdash \mathtt{a}]$ represents the independence analysis, with environment Γ, used on term $\mathtt{a}$. This is on array level, on expression level $\mathcal{I}_E[\Gamma \vdash \mathtt{e}]$ is used.

4.3.2 Examples

We will now give some examples, to show the rules practice. First we look at

```
generateEx :: Acc (Vector Int) → Acc (Scalar Bool)
generateEx _ = let ys = generate (Z :. 10) indexHead
                   ys' = fold (+) 0 ys
               in map (>10) ys'
```

Which will always be TotalI, since well, it is only dependent on constants. Not even on the input array. This would be a very bad predicate for an awhile, since it would either always produce false (the awhile is unnecessary) or true, making it loop forever.

We get a more interesting example if we consider that the function is not necessarily a closed term, it can call other arrays then the array the awhile is working on. Take this example (where we still explicitly gave the argument ys):

```
openF :: Acc (Vector Int) → Acc (Vector Int) → Acc (Scalar Bool)
openF ys xs = let sh = shape xs
                  n = indexHead sh
              in unit (n > ys !! 0)
```

We see that it accesses the array ys[11], thus in the case of a regular or irregular ys, this would give ShapeI as an answer. However, ys could also be avoided (not the input of the awhile), thus being the same for each call of this function. This means that ys !! 0 is also TotalI, and if xs is regular, we still end up being TotalI.

There are of course predicate functions that have the same property, stopping at the same time for each array in the sequence, but for which we cannot determine this. For example, if we have the function:

```
indexing:: → Acc (Scalar Int, Vector Int) → Acc (Scalar Bool)
indexing xs = let n = afst xs
              in unit (n !! 0 < 10)
```

And the iteration function will always add 1 to the first part of the tuple of xs and each array starts with the same number (e.g. 0). This is essentially a for loop, that will iterate a fixed amount of times (e.g. 10 times, if we started with 0). However, this is only the case if we have this particular iteration function and initial array. Something that we cannot determine in general. Although if we would use the size of the array as the iteration number, e.g. we start with a size 0 array, and each iteration we increase the size with 1, and we stop at size 10, this has the same effect and is something we could determine with this analysis.

4.3.3 Implementation

We implemented this analysis in Accelerate. An important aspect of Accelerate is that the AST of the language is still typed[20]. This type safety prevents run time errors when implemented something incorrectly. E.g. if we work with a zip we can provide two arrays with different types to it, and will use a function to combine these. In the implementation of this analysis, we do the analysis on the arrays. However, if we swap the arrays in the environment, this could lead to wrong results. We can encode this in the data type, but we chose to encode this in the environment. Encoding it in the data type would lead to a lot of extra rules for retyping the independences, and it is easiest to do this wrong in the when pushing values to the environment. We encoded the data type like:

```
data Independence = NotInd | ShapeInd | TotalI
```

The environment looks like:

```
data IndEnv aenv where
  BaseIE :: IndEnv aenv
  PushIE :: IndEnv aenv
            -- Added the proxy type, so we have extra type check safety if
```

[11]!! is a linear index of the array

```
  -- we add the right independence value to the environment.
→ t {-proxy -}
→ Independence
→ IndEnv (aenv, t)
```

We coded the environment as a nested pair of tuples. An empty environment looks like (), and if the first variable is a vector of integers, we get ((), `Vector Int`). We added a proxy type, thus when we need to push a variable, we need to proof that we are adding an independence value of the correct array. Note that we do not force the base environment to be empty. Because we get the environment from the vectorisation step. So this keeps track of the lifted types, but the vectorisation step could also have started with some free variables. These variables are then outside of the vectorisation scope, thus can be seen as independent variables. So if we ever lookup a variable that is not present, then it is `TotalI`. Note we can still always fool the type checker since we can provide any proxy type we want, but it is a tool for the programmer to express exactly which array he would want to push to the environment, and we get feedback if this is correct. If we had included the type of the expression or array in the data type, we could have forced the type safety.

In our analysis above we pretended that there was one environment, but this is not true. We have separate ones, one for array terms and one for expressions. We implemented everything in the way presented in Listing 2, although not all terms present there. Furthermore, we made one simplification for the `Stencil` constructor, which is still safe, but could be smarter. For the foreign functions that can be added to Accelerate, we want to err on the safe side, and we always make them `NotI`. However, we could argue that the fallback implementation should have the same behaviour as the foreign one, thus analysing that. However, this means we have to trust the user never to implement a wrong fallback function for a foreign function. Which in turn, could lead to astonishing results that are hard for the user to debug. One last thing is tuples, a part of a tuple can also be independent, while the rest is not. This currently is not encoded, and to do this, we would need to add a tuple representation for independent types.

4.4 Shape Analysis

With the shape analysis, we want to determine if two arrays have the same shape. For example, the following three arrays have an equal shape

```
xs
map (+1) xs
generate (shape xs) f
```

This knowledge allows us to determine the following during compilation:

- When we vectorise a conditional `acond p t e`, and if `t` and `e` have the same shape, we can stay regular. Even if we are irregular, the lifted version is simpler, since the segment descriptor will stay the same.
- When we vectorise an `awhile p it i`, and if the iteration function `it` stays the same shape after each application, we can stay regular. Even if we are irregular, the segment descriptor can stay the same.
- If we are vectorising and using functions that can potentially change the shape. For example `zip`, `permute` and `stencil2`[12], we can avoid recalculating the segment descriptor in the irregular case.
- With compilation, in general, we can avoid certain bound checks, if we already know if the input arrays are the same shape, for example when zipping two arrays together.

The shape analysis consists of two parts:

1. Gathering the shape information ($\mathbb{S}\ \Gamma$ `a`) of a certain term, while keeping track of the right environment. We notate this as $\mathcal{S}_A[\Gamma \vdash \mathtt{a}]$, where Γ is the environment, `a` the term on which we work, and $\mathbb{S}\ \Gamma$ `a` is the data type for shape information. We can *push* shape information to the environment (($\mathbb{S}\ \Gamma$ `a`) : Γ) and *lookup* information (`lookup` Γ $\mathtt{xs_i}$).

[12]Although `zip` is already rewritten when we arrive at the vectorisation stage of the compilation and stencil operations are currently not supported in nested parallelism in Accelerate.

```
𝒮_A[Γ ⊢ xs_i]                = ShapeVar Γ xs_i
𝒮_A[Γ ⊢ (λxs_0 → a_1) a_2]   = let 𝕊_2 = 𝒮_A[Γ ⊢ a_2]
                               in 𝒮_A[(𝕊 : Γ) ⊢ a_1]
𝒮_A[Γ ⊢ map (λx_0 → e) a]    = Retype 𝒮_A[Γ ⊢ a]
𝒮_A[Γ ⊢ fold (λx_1 x_0) e_2 a]
                             = FoldedS 𝒮_A[Γ ⊢ a]
𝒮_A[Γ ⊢ zip a_1 a_2]         =
  ZippedS 𝒮_A[Γ ⊢ a1] 𝒮_A[Γ ⊢ a2]
𝒮_A[Γ ⊢ (a_0, ..., a_n)]     = (𝒮_A[Γ ⊢ a_0], ..., 𝒮_A[Γ ⊢ a_n])
𝒮_A[Γ ⊢ aprj l a]            = TupIdxS l 𝒮_A[Γ ⊢ a]
𝒮_A[Γ ⊢ unit e]              = Scalar

𝒮_A[Γ ⊢ generate sh (λx_0 → e)]
                             = ShapeExpr Γ sh
𝒮_A[Γ ⊢ reshape sh a]        = ShapeExpr Γ sh
𝒮_A[Γ ⊢ acond e a_1 a_2]     =
  if 𝒮_A[Γ ⊢ a_1] ≡ 𝒮_A[Γ ⊢ a_2]
    then 𝒮_A[Γ ⊢ a_1]
    else UndecidableS_j
𝒮_A[Γ ⊢ awhile (λxs_0 → a_1) (λxs_0 → a_2) a_3 =
  let 𝕊_3 = 𝒮_A[Γ ⊢ i]
  in if 𝕊_3 ≡ 𝒮_A[Γ′ ⊢ a_2]
    then 𝕊_3
    else UndecidableS_j]
```

Listing 3: The rules that we use for the shape analysis applied on the array terms of the simple grammar we defined. The analysis indicates what information we can determine of the shape of the array. The function $\mathcal{S}_A[\Gamma \vdash \mathtt{a}]$ represents the shape analysis, with environment Γ, used on term a.

2. Comparing the shape information of two terms, to determine if they are equal.

We must keep track of the environment since the terms that we are comparing might live in different parts of the program. This means that the variables which they refer to are different.

4.4.1 Gathering Shape Information

We use the $\mathbb{S}$ · · data type for shape information:

```
data 𝕊 Γ a where
  ShapeVar       :: Arrays arrs ⇒ ShapeEnv Γ Γ′ → Idx Γ′ arrs → 𝕊 Γ arrs
  ShapeExpr      :: Shape sh ⇒ ShapeEnv Γ Γ′ → Exp Γ′ sh → 𝕊 Γ (Array sh e)
  Scalar         :: 𝕊 Γ (Array Z e)
  FoldedS        :: Shape sh ⇒ 𝕊 Γ (Array (sh :. Int) e) → 𝕊 Γ (Array sh e)
  ZippedS        :: Shape sh ⇒ 𝕊 Γ (Array sh e_1) → 𝕊 Γ (Array sh e_2) → 𝕊 Γ (Array sh e_3)
  TupS           :: (𝕊 Γ a_1, 𝕊 Γ a_2, ..., 𝕊 Γ a_n) → 𝕊 Γ (a_1, a_2, ..., a_n)
  TupleIdx       :: Arrays arrs ⇒ TupleIdx arrs a → 𝕊 Γ arrs → 𝕊 Γ a
  UndecidableVar :: Int → 𝕊 Γ a
  Retype         :: 𝕊 Γ (Array sh e) → 𝕊 Γ (Array sh e')
```

We note a few things that aren't immediately obvious.

- We omitted a few data types, that we did implement, but are less interesting for the idea. They are very similar to the FoldedS type, since they capture shape information for a very specific use case.
- We keep all the type information of the original array term. Strictly speaking, only the shape of each term would be needed (the sh part of Array sh e). Moreover, since we do not throw away this information, we need the Retype constructor, to stay correct.
- We keep track of our shape environment, but this can have a different type than the original environment. We may add some more variables when we get to the place what determines the shape. Since we do not immediately solve variables that we encounter, we keep them in the new environment.
- Sometimes we cannot determine the shape of a term. However, if two terms point to this exact term, we are still sure that it has the same shape, so we keep track of a unique number in UndecidableVar.

Let us see how this works in practice. We want to determining the shape of the following:

```
reshape (Z :. shapeSize (shape a)) a
  where
    a = map (λe₀ → e₀ + 1) xs
```

Where xs has a random shape. We save this as an ShapeExpr, with the environment that we have at the moment (a has the same shape as xs). We can compare it with:

```
generate (Z :. shapeSize (shape a)) a
  where
    a = fold (λe₁ e₀ → e₀ + e₁) 0 xs
```

This is also a ShapeExpr, we expand on the expression. We see that the expression is the same; even the variable is the same. However, the environment is different (we can never assume it is the same), so we have to look up the value of a in both the environment. In the first one, we see that it is equal to the shape of xs, and in the second it is equal to the fold of xs. So we cannot assume they are the same.[13] They are actually not the same, but sometimes we have to err on the safe side and say things are not the same, even though they very well could be.

In Listing 3, we list the rules when used for gathering the shape information for our small grammar. In the actual implementation, we have rules for the complete AST of the core language of Accelerate. They follow the same ideas.

4.4.2 Comparing Shape Information

After we gather the shape information, we want to compare them, to decide if they are equal. In most cases, for two shapes to be equal, they must be build from the same constructor. However, we make a few improvements:

- We keep track of the dimension of the shape. If two shapes are a scalar, they are always equal. For example, a folded vector is a scalar but is probably encoded in a FoldedS constructor.
- The shape expression shape $\texttt{xs}_\texttt{i}$, will have the same shape as $\texttt{xs}_\texttt{i}$, so we make sure to rewrite this to a variable.
- When we index a tuple with (TupleIdx) and the second argument is a tuple (it could also be a variable), we will access the tuple and get out the right shape information.
- It is possible that a variable has no shape information. This means it was a free variable when we began our analyses. If two variables are both free and the same, the shape is also equal. However, if one is not free, we look up the shape information.
- When comparing expressions, they must be exactly equal, except for the array variables. If we index an array, we cannot determine equality, since the arrays might be different, we only store information about the shape. When we ask the shape of an array (e.g. shape $\texttt{xs}_0$), we can lookup the shape information in the environment. If this is equal for all the variables, then the shapes of these expressions are the same.

4.4.3 Implementation

Here we will discuss some implementation choices. We also choose to save the environment with the shape information, if this is needed. Another choice would be to fill in the shape information of a variable, as soon as we encounter it. However, in the AST of the Accelerate expressions can refer to shapes of arrays, which means that replacing a variable, is substituting it with a real array term. Then we would be forced to keep track of an environment that is full of real array terms. This seemed inefficient, so we chose to keep the environment together with the shape information, and only store shape information in the environment. This also allows us to only look up variables in the environment when we need them for comparison. For example, if we have a ShapeExpr and a ShapeVar inside a FoldedS constructor, we can already determine inequality.

[13]If we follow the real implementation, we would look up the shape of xs again in the environment, and then conclude that it is not a folded shape. Although if it is a folded shape, it would need to check further again.

In the implementation, we work with the state monad so that we can label `UndecidableVar` uniquely. This is the reason that the actual implementation differs slightly from the rules in 3, since the monad forces an order of execution.

We can still make some improvements. We can store a unique number for each array in the environment. This allows for quick comparison; equal unique number means we do not have to recurse deeper down. It also means that we could determine equality when we access an array in an expression. Also, we would have liked to save only the shape type in $\mathbb{S} \;\cdot\; \cdot$, but now we also save the type of the elements. It is thus forcing us to use the `Retype` constructor. We could solve this with type families in Haskell. However, tuples are also encoded with a type family in the compiler. Moreover, we encountered a problem, since we were unable to prove that the order of which we apply the type families are the same[14]. Furthermore, we only implemented this for vectorisation of `awhile` and `acond`, to check if we stay regular. However, as we mentioned at the beginning of this subsection, it has more uses.

The analyses we presented in this section allow us to stay regular more often, or have a better implementation for some irregular terms. The independent analysis allows us to do regular computations in the gridding algorithm of Section 5.3. Next, we will discuss the manual lifting of functions briefly.

4.5 Manual Lifting

Automatic vectorisation is the holy grail to obtain. However, the lifting gives overhead, and our lifting operations must be safe, and work on all program. This means that sometimes the compiler cannot determine properties, that we as programmer know are true. In this case, it would make sense to lift a function manually. We would not want to put the full responsibility of lifting everything on the programmer. Maybe he knows only a fast lifted algorithm in the regular case, as is the case with FFTs, but not in the irregular case. Therefore, we designed this in a way, that can fail and use a fallback function. Thus the programmer only has to implement the lifted version that it wants to. The vectoriser can then pick up this lifted version, but can always fall back on the normal version, and try to lift that one itself automatically. This is implemented in Accelerate as the following data type:

```
LiftedAFun :: (Arrays a, Arrays b)
   ⇒ (Acc a → Acc b))                                        -- Original function
   → (forall a' b'.𝒱 a a' → 𝒱 b b' → Maybe (Acc a' → Acc b'))   -- Maybe lifted function
   → Acc a                                                   -- Input array
   → Acc b
```

Here $\mathcal{V} \;\cdot\; \cdot$ is the type relation between the original type, and the lifted type of the array. The `Maybe` allows us to fail for certain type relations, thus we only have to implement the version that we would want.

It is important to note that the way this is constructed, we are unable to do sharing of terms within the lifted function. Sharing allows us to only compute the same term once. During the phase when sharing of terms is discovered, we do not know if we are going to use a manual lifted version yet. So this is a small price to pay for our own lifted functions.

In this section we discussed several ways to remain regular during vectorisation. This can be used in the case study of the gridding algorithm.

[14] We could not proof `TupleRepr` (`ShapeRepr` `t`) ≡ `ShapeRepr` (`TupleRepr` `t`), which we know is the case. However, the type families were both not injective, which made it unprovable for us at least. Some more thought should go into this, to determine if this a limitation of ourselves or GHC.

Part II
Case Study: Gridding for the Square Kilometre Array

5 Background on Gridding

This part of the thesis, is the practical case study of the gridding algorithm. First we explain some physics that is being used in radio astronomy, on which the gridding algorithm is based. Afterwards we explain this algorithm in more detail, how we implemented this in Accelerate, and what difficulties arose whilst programming this.

5.1 Radio Aperture Synthesis

This section is an introduction to radio aperture synthesis, mainly based on the book of Thompson et al. [27]. Radio interferometry telescopes use this technique to image the sky, and it forms the basis of the gridding algorithm.

Telescopes get radio signals (that is, electromagnetic radiation) from sources in the sky. Most of the signals are a form of continuum radiation, where the power varies gradual with frequency. The strength of a source is expressed as (spectral) flux density and is measured in watts per square meter per hertz, the unit of this is Jansky ($1\text{ Jy} = 10^{-26}\text{W m}^{-2}\text{Hz}^{-1}$). By integrating in frequency over a spectrum we receive what is called the power flux density.

When producing an image of the sky, we actually are just making an image of the surface of the celestial sphere (an imaginary sphere centred around an observer) on which the sources are projected, see Figure 5a. We want to know the intensity which is defined as the power flux density per solid angle. To demonstrate the idea we focus on the one dimensional case, which can be extended based on the same ideas.

In radio aperture synthesis we use pairs of antennas, as shown in Figure 5b, that both observe the same portion of the sky. Let's denote the angle to which they point θ. Since there is a distance between the pair of antennas, denote it by D, the signals are the same but have a time difference $\tau_g = \frac{D}{c}\sin\theta$ at which they arrive, where τ_g is called the geometric delay and c is the speed of light. The signals of the pair are combined by adding them normally and with a phase switcher, which effectively subtracts the two signals. What we measure is proportional to the square of the signal. Therefore, we measure $|V_1 + V_2|^2$ and $|V_1 - V_2|^2$ and when these results are subtracted we get $2V_1V_2$ which is the signal that is used.

Let us assume a point source as a signal. For a specific frequency ν the V is proportional to $\sin(2\pi\nu t)$. Hence, the output of $2V_1V_2$ is proportional to:

$$F = \sin(2\pi\nu t)\sin(2\pi\nu(t-\tau_g)) = 2\sin^2(2\pi\nu t)\cos(2\pi\nu\tau_g) - 2\sin(2\pi\nu t)\cos(2\pi\nu t)\sin(2\pi\nu\tau_g). \tag{1}$$

The geometrical delay τ_g changes with the rotation of the earth, but in the same time t changes a lot more (a few magnitudes of order more). Consequently, we can approximate (with time $T \gg \frac{1}{\nu}$) and are left with:

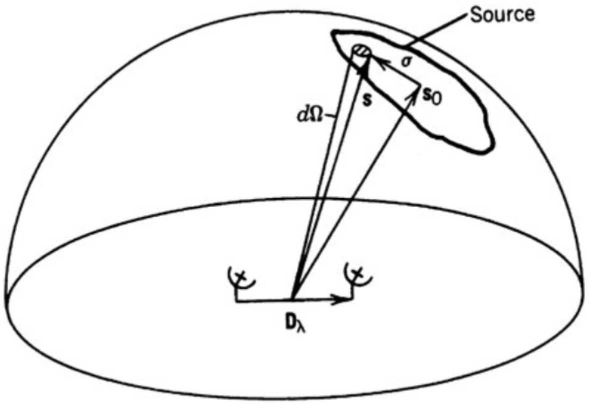
$$F = \cos(2\pi\nu\tau_g) = \cos(\frac{2\pi Dl}{\lambda}) \tag{2}$$

where $l = \sin\theta$ and $\lambda = \frac{c}{\nu}$. In general the interferometer also has a bandwidth filter and an integrator, as depicted in Figure 5b. The integrator sums the input for a time constant 2T. For a point source the output of the correlator is:

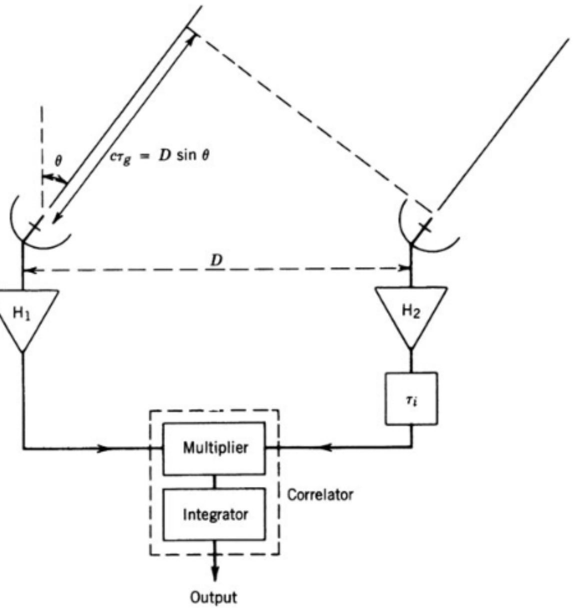
$$r = \frac{1}{2T}\int_{-T}^{T} V(t)V(t-\tau)dt \tag{3}$$

where system noise is ignored and we assume that the bandwidth filter has a finite bandwidth $\Delta\nu$. Since the integration time is much larger than $\Delta\nu^{-1}$, we can also write:

$$r(\tau) = \lim_{T\to\infty}\frac{1}{2T}\int_{-T}^{T} V(t)V(t-\tau)dt \tag{4}$$



(a) The source is represented by the outline on the celestial sphere. The baseline D_λ and position vector s_0 define the interferometer. Retrieved from [27] and used under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License `http://creativecommons.org/licenses/by-nc/4.0/`



(b) An interferometer of two antennas, pointing in the direction of θ with a distance between them of D, given rise to the geometric time delay τ_g. The signals pass band-pass amplifiers H and can be given an instrumental time delay τ_i. The correlator multiplies and integrates the signals. Retrieved from [27].

Figure 5

which is an autocorrelation function. The Wiener–Khinchin relation says that the power function of a signal is the Fourier transform of the autocorrelation function, which can be written as:

$$|H(\nu)|^2 = \int_{-\infty}^{\infty} r(\tau)e^{-j2\pi\nu\tau}dt \tag{5}$$

where H (\lambda nu) is the amplitude response of the signal. This shows how the outcome of the interferometer is linked with the Fourier space of the signal. The method seems cumbersome, but it's very useful to filter out noise in the signal.

Modern radio interferometers can track a certain part of the sky, by introducing an instrumental time delay τ_i, such that the total delay between the two signals is $\tau = \tau_g - \tau_i$. If we set the reference position of the sky we want to track as θ_0, the instrumental delay is set equal to the geometric delay for this positions, thus $\tau_i = \frac{D}{c}\sin\theta_0$. For radiation from direction $\theta_0 - \Delta\theta$, where $\Delta\theta$ is a small angle, using small angle approximations on equation 2 with the added instrumental time delay leads to:

$$F = \cos(2\pi\nu\tau) \simeq \cos[2\pi\nu\frac{D}{c}\sin\Delta\theta\cos\theta_0] \tag{6}$$

We introduce quantity u, that reflects how much of the distance between the antenna pair is normal to the incoming waves from the reference source, measured in wavelengths. We call this quantity the baseline. Thus:

$$u = \frac{D\cos\theta_0}{\lambda} = \frac{\nu D\cos\theta_0}{c} \tag{7}$$

Combining this we get that the response of radiation from direction $\theta_0 - \Delta\theta$ is proportional to:

$$F(l) = \cos(2\pi\nu\tau) \simeq \cos(2\pi ul) \tag{8}$$

where $l = \sin\Delta\theta$.

The total response of the interferometer is in the end also determined by the reception pattern of the antennas, represented as $A(l)$, the bandwidth pattern which depends on what frequencies are passed in the signal, denoted by $F_B(l)$ and the actual intensity profile of the source $I_l(l)$ in which we are interested. The response can then be written as:

$$R(l) = \int_{source} \cos[2\pi u(l-l')]A(l')F_B(l')I_l(l')dl' \tag{9}$$

Or as convolution (with $*$):

$$R(l) = \cos[2\pi ul] * [A(l)F_B(l)I_l(l)] \tag{10}$$

Generally $A(l)$ and $F_B(l)$ are instrumental characteristics and $I_l(l)$ can be recovered from this product. We can use the convolution theorem of Fourier transforms that says that a convolution of two functions is the same as a multiplication in Fourier space:

$$f * g \longleftrightarrow FG \tag{11}$$

where $f \longleftrightarrow F$ and $g \longleftrightarrow G$ indicate a Fourier transformation, with F and G being functions in the Fourier space. In our case, we have $r(u) \longleftrightarrow R(l)$, and the Fourier transform of the interferometer for a specific baseline u_0 is:

$$\cos[2\pi ul] \longleftrightarrow \frac{1}{2}[\delta(u+u_0) + \delta(u-u_0)] \tag{12}$$

Let $\mathcal{V}$ be the Fourier transform of $I_l(l)$, we then get:

$$r(u) = \frac{1}{2}[\mathcal{V}(-u_0)\delta(u+u_0) + \mathcal{V}(u_0)\delta(u-u_0)] \tag{13}$$

Now $\mathcal{V}(u)$ represents the amplitude and phase of the sinusoidal component of the intensity profile with spatial frequency u. So we see that interferometer response is just the Fourier component u_0 (and $-u_0$) of the intensity in the sky. And $r(u)$ is what actually is measured from the fringe patterns we get from the interferometer.

If we combine several antennas as a pair, we have different values of u, thus getting different baselines. These baselines are the Fourier terms of the image we want to make. In practice we can't have every baseline constructed, but we have finitely many. (But it is possible to get the real image from these baselines, with some other assumptions.) For this section a lot of details where omitted, and we focused only on the 1 dimensional case, but a general gist of the underlying idea is given. For further reference see [27].

Radio aperture synthesis forms a core component of the gridding algorithm, which we discuss in the next section.

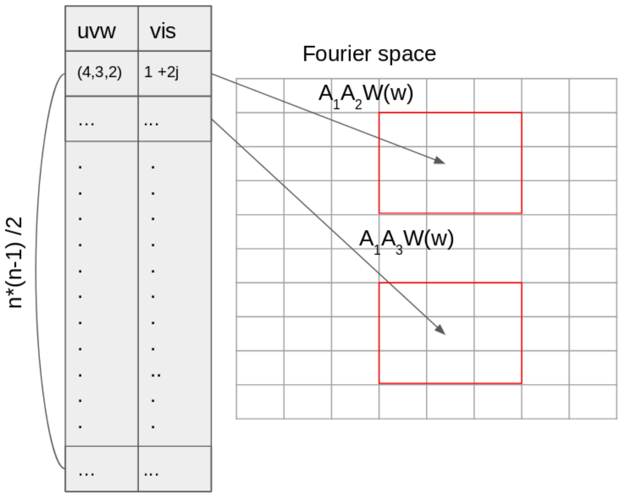


Figure 6: A simple visualization of gridding, where each visibility has it's own convolution kernel and adds its contribution to the Fourier space.

5.2 Gridding for the Square Kilometre Array

In the last section we discussed aperture synthesis in the one dimensional case. Here we give the formulas for the three dimensional case, and how this translates to the actual gridding algorithm.

In the three dimensional case, we have visibilities from our baselines (antenna pairs) u, v, w from the interference patterns. They are related to the image space as:

$$V_{a_1,a_2,t,f}(u,v,w) = \int \frac{A_{a_1,t,f}(l,m)A_{a_2,t,f}(l,m)G_w(l,m)I(l,m)}{\sqrt{1-l^2-m^2}} e^{-2\pi i(ul+vm)} \mathrm{d}l\mathrm{d}m \tag{14}$$

The G_w term is needed because of the fact that we make a 2D image, but we are using baselines that can be 3 dimensional. This form can be used for small w, l and m values. The A_a terms are as before per antenna, because of nonuniform antenna reception patterns. We want to retrieve the image, thus we can write:

$$\frac{I(l,m)}{\sqrt{1-l^2-m^2}} = \sum_{a_1,a_2,t,f} \int C_{a_1,a_2,t,f}(u,v) e^{2\pi i(ul+vm)} \mathrm{d}u\mathrm{d}v \tag{15}$$

where $C_{a_1,a_2,t,f} = V_{a_1,a_2} * \tilde{A}^{-1}_{a_1,t,f} * \tilde{A}^{-1}_{a_2,t,f} * \tilde{G}^{-1}_w$. Where the functions are the Fourier transforms of the original functions convolved with each other and the visibility. Together these functions form the grid convolution kernel. The convolution quickly goes to zero for non-zero u, which means each visibility only contributes a little to the Fourier space.

The gridding algorithm is processing the gathered visibilities with the appropriate convolution kernels into the Fourier space. Each baseline, which is an antenna pair, gives one visibility. This is distributed over the Fourier image, via the convolution kernel, centred around the u and v baseline of this antenna pair. We visualize this in Figure 6. After processing all the visibilities a Fourier transform is executed to get the image.

We discuss the implementation of this algorithm in the next section.

5.3 Gridding Algorithm in Accelerate

The gridding algorithm we implemented, as explained in the previous section, is based on the SKA Science Data Processor reference library[15]. We encounter two forms of nested parallelism in the algorithm. One can be solved in Accelerate whilst utilizing all the available parallelism, but for the other we had to improve Accelerate.

[15] `https://github.com/SKA-ScienceDataProcessor/algorithm-reference-library`

```
convgrid :: Acc (Array DIM2 Visibility)          -- The convolution kernel
         → Acc (Matrix Visibility)               -- Destination grid N x N of complex numbers
         → Acc (Vector (BaseLine, BaseLine))     -- The uvw baselines
         → Acc (Vector Visibility)               -- The visibilities
         → Acc (Matrix Visibility)               -- The resulting Fourier space
convgrid gcf a coords v =
  let Z :. gh     :. gw    = unlift (shape gcf) :: Z :. Exp Int :. Exp Int
      Z :. height :. width = unlift (shape a)   :: Z :. Exp Int :. Exp Int
      (cx, cy) = unzip2 coords
      newcx   = map (λx → x − gw `div` 2) cx
      newcy   = map (λy → y − gh `div` 2) cy

      coordsAndVis    = zip3 newcx newcy v
      coordsAndVisRep = replicate (lift (Z :. All :. gh :. gw)) coordsAndVis
      withKern        = imap getComplex coordsAndVisRep
      (x, y, val)     = unzip3 withKern

      getComplex :: Exp DIM3 → Exp (Int, Int, Visibility) → Exp (Int, Int, Visibility)
      getComplex
        (unlift ∘ unindex3 → (_, i, j) :: (Exp Int, Exp Int, Exp Int))
        (unlift → (x, y, vis) :: (Exp Int, Exp Int, Exp Visibility)) =
        lift (x + j
             , y + i
             , vis ∗ gcf ! lift (Z :. i :. j))

      indexer i = let y' = y ! i
                      x' = x ! i
                  in index2 y' x'
  in permute (+) a indexer val
```

Listing 4: The first nested parallelism problem in the gridding code of Accelerate, but fixed with replicating the input.

5.3.1 Problem # 1: using pre-calculated convolutions kernels

The first problem is about placing all the visibilities in the Fourier image, see Listing 4 for the simplified code. For a kernel size of $n \times n$, we place n^2 values in the Fourier image for each visibility. Ideally we would apply a parallel action to each entry in the list. Fortunately the parallel action is regular and simple, so we can use the higher dimension support of Accelerate. We `replicate` each visibility $n \times n$ and multiply it with the kernel, and place it into the grid afterwards, as can be seen in Figure 6. For cases where the same convolution kernel is used for each visibility this is an efficient implementation in Accelerate. Note that the replication are not actual memory copies, because Accelerate fuses the computations and turns them into indexing operations. If each kernel is different and should be calculated beforehand, this can become a bit more troublesome, because more memory needs to be used for the kernels. For a typical kernel size of 15×15, 512 antennas, up to 300 frequency channels and up to 50 time steps, this could amount up to 6.42 TB for all the kernels if they are stored as doubles. This can be partly solved by streaming the data and kernels, so that there is still abundant parallelism and the memory size doesn't get to large.

5.3.2 Problem # 2: calculating different convolution kernels

The second problem, which is not easily solved, is calculating a convolution kernel for each individual baseline. The kernels depend on the w value of the baselines, antenna specifics, time, and frequency. They could all be calculated beforehand, but this takes up a lot of space as shown above. The approach that was suggested and used, is getting the $A_{a_1,t,f}$, $A_{a_2,t,f}$ G_w kernels separately and take the convolution between each other. A convolution can be calculated by a multiplication in Fourier space. So we first perform FFTs on the kernels, and multiply

```
proccesOne :: Acc (Array DIM3 Visibility)                 -- All the w-kernels
    → Acc (Array DIM3 Visibility)                          -- All the a-kernels
    → Acc (Scalar (Int, Int, Int, Visibility))             -- The indices of the kernels
    → Acc (Matrix Visibility)                              -- The resulting kernel
proccesOne wkerns akerns
  (unlift ∘ the → (wbin, a1index, a2index, vis) :: (Exp Int, Exp Int, Exp Int, Exp Visibility))
  = let Z :. _ :. gh :. gw = unlift (shape wkerns) :: Z :. Exp Int :. Exp Int :. Exp Int
            halfgh = gh `div` 2
            halfgw = gw `div` 2
              -- Get the right a and w kernels
            a1 = slice akerns (lift (Z :. a1index :. All :. All))
            a2 = slice akerns (lift (Z :. a2index :. All :. All))
            w  = slice wkerns (lift (Z :. wbin :. All :. All))
              -- Convolve them
            akern  = convolve2d a1 a2
            awkern = convolve2d w akern
              -- Let the visibility have the same dimensions as the aw-kernel
            allvis = replicate (lift (Z :. gh :. gw)) (unit vis)
    in zipWith (*) allvis awkern
```

Listing 5: The second nested parallelism problem in the gridding code of Accelerate.

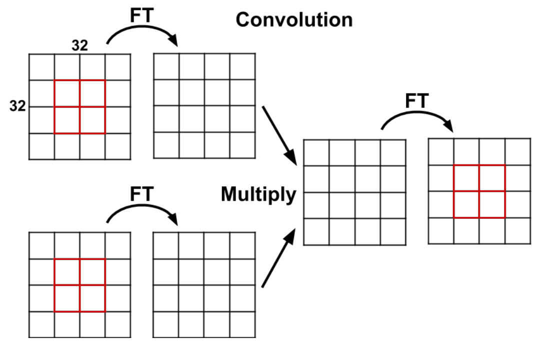


Figure 7: To convolve two arrays, you can use FFTs, since a convolution is the same as multiplication in Fourier space.

them afterwards. See Figure 7 for a visual representation of this. To calculate a convolution precisely, we need to zero pad the arrays, with at least twice the size of the original dimensions[25]. In our case the input kernels have size 15×15, we chose to pad them to 32×32 (powers of 2), because this was better suited for the FFT algorithm that we use.

The FFT is implemented in Accelerate as a parallel action. In the ideal case we can calculate the convolution kernels for all visibilities in parallel. But this is nested parallelism, since each visibility is again doing a parallel action. This time we cannot expose all the available parallelism easily, mainly because performing an FFT is no trivial action. We would need to make a vectorised version of the FFT by hand. Which is doable, but not something you would normally require from a programmer. And these FFT functions are wrapped around other functions, which also need to be vectorised. Thus, we decided to use the sequences work for this, since it is only one level deep nested parallelism. The code in Listing 5 shows the function we apply to each visibility, where we provide all the W and A-kernels to the functions as the first two arguments. If the visibilities are given as a sequence, we process it with `mapSeq :: (Acc a → Acc b) → Seq [a] → Seq [b]`. We still have to put the convolved visibilities in the right position in the Fourier space, but we left out that detail. In Section 6 we go into more details on how the sequences work is altered to allow the use of FFT. Another choice is to process the visibilities sequentially, where 1 FFTs is processed at the time, thus losing potential parallelism. We provide benchmarks in Section 6.4 comparing these approaches and show that the sequences approach is faster.

We solved both the nested parallel problems that we encountered by rewriting the computations, so they are no longer nested. This can be done by flattening them by hand (the first), calculating them sequentially and losing some parallelism or flattening them automatically. Flattening is the general solution to nested parallelism, although the main problem is to do this whilst still having good performance. In the next section we discuss the automatic flattening.

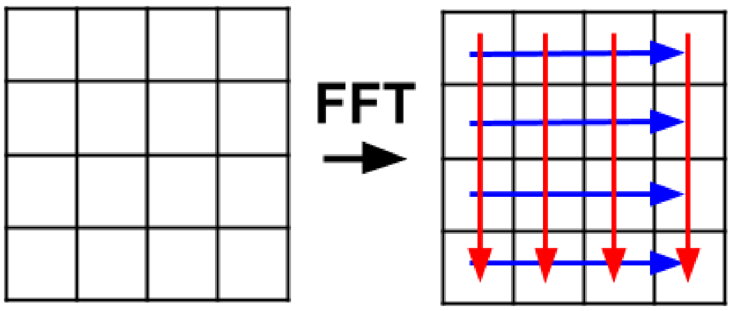


Figure 8: A two-dimensional FFT is performed, by first doing a one-dimensional FFT on the lower dimension, and performing a second FFT on the next dimension of the result.

6 Vectorising Fast Fourier Transforms

In this section we look at Fast Fourier Transforms (FFTs) since play a critical role in the gridding algorithm. It is spending most of its time doing convolutions, which are implemented via FFTs.We will consider four cases:

- Doing the FFTs one at the time, using a `awhile` construct
- Vectorising FFTs using foreign libraries
- Vectorising FFTs, where the `awhile` construct will work on an irregular data representation.
- Vectorising FFTs, where the `awhile` construct will stay regular.

In the following subsections we will explain a few of the details of these implementations, and lastly show some benchmarks comparing these implementations. But first we will explain how an FFT can be lifted manually.

6.1 Lifting FFTs

The structure of multidimensional FFT algorithms is special. We depicted this in Figure 8. A multidimensional FFT is first performing a one-dimensional FFT on a lower dimension, and uses the output of this as input for the next dimension. This means, in general one dimensional FFTs are implemented as being rank-polymorphic, they can work on an arbitrary dimension, and only transform the inner dimension. This means that lifting an FFT, in the case of regular nested data, is in general quite easy. We just use the FFT on all the dimensions, except of the last of the regular nested data.

But having irregular input isn't straightforward, since each array can have a different size. Consequently, we probably cannot use the rank-polymorphic in that case any more.

6.2 Foreign Libraries

6.2.1 FFTW and cuFFT

The FFTW[16] library currently provides the fastest implementation of Fourier transforms on the CPU. Therefore, we want to benchmark this library to get an indication of the maximum performance we can strive for. You'd expect that using this library would always be the best, since it is the fastest. This is true for the gridding algorithm, since almost all the work is spent doing Fourier transforms, this is not true in the general case. The strength of Accelerate is combining multiple functions and fuse them together, but this can only be done if all the functions are pure Accelerate functions.

On the GPU we will use cuFFT[17] which is a CUDA implementation modeled after FFTW.

[16]`http://www.fftw.org/`
[17]`https://docs.nvidia.com/cuda/cufft/index.html`

6.2.2 Execution plans

Both the FFTW and cuFFT library work with plans. The library can automatically test different setups for the best execution plan. This plan can be saved as a *wisdom* string, which can be used later to execute the same plan without the testing. It is also possible to estimate a plan without testing, this is how these libraries interface with Accelerate. This is the main use in Accelerate, since it is simple, and we do not want to require that the user knows how these libraries work.[18]

6.2.3 Foreign Function Interface

Is we mentioned in the last subsection, lifting FFTs in the regular case is most of the times straightforward. FFTW and cuFFT both have support for this, and is even only one function to call. Although, this is only the case for regular nested arrays. For irregular shapes, there is no simple support in these libraries.

In Accelerate foreign function can be added for a specific back-end. The back-end will handle the function call, but in the front-end of Accelerate a name and a fall-back function must be specified.

If we want a lifted version of a foreign function, we are responsible to add an implementation for each of the different types lifted we can get. This is being done via the `VectorisedForeign` data type:

```
data VectorisedForeign asm where
  VectorisedForeign :: (Foreign asm, Arrays a, Arrays b)
    ⇒ (forall b' a'.Arrays a' ⇒ 𝓥 a a' → 𝓥 b b' → asm (a' → b'))
    → VectorisedForeign (a → b)
```

Here `Foreign` is a class which has instances for each back-end, which will fully specify how this should execute.

For each manner in which we can lift the function (regular, irregular, avoid lifting) we must specify a different foreign function. But in case of the FFT we do not have a good implementation for irregular data. We cannot specify that at the moment, and we would rather fallback to the pure implementation if this is the case.

6.3 Vectorising Accelerate

The Accelerate implementation[19]. This package implements the FFT for all sizes, but we use the Split-Radix FFT algorithm [9, 29] that is only applicable on powers of 2. The main reason we chose to specialize is practical, the compiling of vectorised programs takes a long time[20] in Accelerate, which can be reduced if we choose a specialized version.

6.3.1 Counting parallel actions

We will sometimes count the parallel actions in the code. We do this either by hand for simple examples, or automatically by traversing the resulting AST after passing through the Accelerate compiler. This is an approximation of the real world, where we make some simple assumptions in the automatic counting. If something is let bound we assume it is always computed exactly once, we don't count variables uses. In a while loop, we only count the iteration function once and when we encounter a conditional, we take the branch with maximum parallel steps.[21]

6.3.2 Staying regular

After inspecting the code, we could deduce that when the input is regular, the intermediate forms and output arrays also stay regular. But the compiler couldn't come to the same conclusion, since the implementation uses the `awhile` function. The type of this function is:

```
awhile :: Arrays a
       ⇒ (Acc a → Acc (Scalar Bool))    -- evaluate while this returns true (predicate)
```

[18] We did not test the effect of adding wisdom hints.
[19] Version 1.2.0.0 of the accelerate-fft package, is used here `http://hackage.haskell.org/package/accelerate-fft`
[20] The compiler is working with an AST that is of over 1 million lines of code after vectorising the FFT function.
[21] The code can be found on `https://github.com/sakehl/accelerate/blob/thesis/Data/Array/Accelerate/Analysis/ActionsCount.hs`

```
      → (Acc a → Acc a)             -- keep applying this function (iteration)
      → Acc a                       -- initial value (initial)
      → Acc a
```

This function allows us to loop in the Accelerate language, for example we could write

```
increaseTillHundred :: Acc (Vector Int) → Acc (Vector Int)
increaseTillHundred xs = awhile pred iter xs
  where
    pred :: Acc (Vector Int) → Acc (Scalar Bool)
    pred xs = map (<100) (fold (+) 0 xs)
    iter :: Acc (Vector Int) → Acc (Vector Int)
    iter xs = map (+1) xs
```

This will increase the elements of vector by one, until the sum off them is more than a hundred.

But when we want to lift `awhile` we cannot freely assume regularity. For instance, we start with a list of vectors, each of size 5. The first vector will iterate, but the second vector will not. And although the iteration function might preserve regularity of the data, and changed all the vectors to a size of 4, the second one wasn't iterated. Consequently, we end up with irregular data. In general this means we cannot assume to stay in a regular, because the elements may require a different number of iterations and the shapes can still change.

In listing 6 the lifted version of `awhile` is shown. Conceptually we do the following. We keep track of the arrays that are being iterated, in a vector of Booleans we keep track which arrays are already finished iterating and lastly we keep track if all the arrays are done iterating. We keep iterating everything, until all the arrays are done. And we only place the new iteration result in the array, if that particular array wasn't already done. Therefore, if one array keeps iterating a lot more than another, a lot of additional useless work is being done. But this the cost of data parallel programming, and for good performance the number of iterations should be similar.

If we look closer, there doesn't seem to be a lot of extra cost for doing an irregular awhile. But most of the cost is hidden in the irregular lifted `acond` function. Since it needs to do a lot of bookkeeping to construct the new values. It has to compute the offsets of where to place the arrays, but this is dependent on the the choices of the earlier arrays, since the shapes don't need to be the same. See Figure 9 for a sketch of this idea. If we look at the following simple program with a conditional:

```
conditional :: Acc (Vector Double) → Acc (Vector Double)
conditional xs = acond p xs (map (+1) xs)
  where
    p = xs !! 0 > 1
```

When we run this trough the compiler whilst lifting it, we count 8 parallel actions present in the AST in the regular case and 46 in the irregular case. It should be noted that this is dependent on how `acond` is exactly lifted, but it does give an indication of the difference between irregular and regular lifting.

Let's get back to the FFT, we will do the same and count the parallel actions Accelerate's AST will contain for each of the different versions that are based on Split-Radix. The normal FFT contains 13 parallel actions, the regular lifted has 88 parallel actions and the irregular one has 663 actions when we start with irregular input. Thus with a bit more complicated program, the same effect is observed, that there are significant more parallel actions in the irregular case. In the next subsection we will give some benchmarks, comparing these different implementations. In Section 4 we will explain the cases when we can stay regular. But the general idea of why the implementation of this FFT stays regular, is because the `awhile` will do exactly the same number of iterations for each array, although the sizes of the intermediate shapes may change.

We want to note one last thing, the general solution for FFTs of arbitrary sizes is implemented with Bluestein's algorithm [5] (or chirp Z-transform) in the accelerate-fft package. We inspected this as well, and it stays regular as well. But this is harder to prove for Accelerate, since in the implementation one part is dependent on indexing a value of an array. We will go into a bit more detail of this in Section 4 why this is problematic.

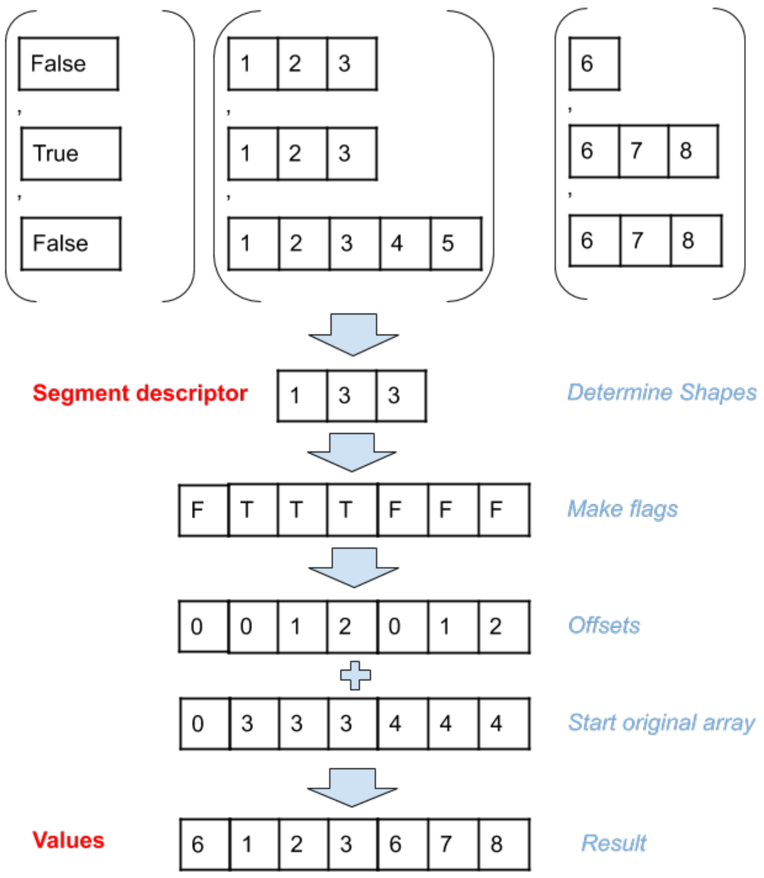


Figure 9: How a conditional, working with irregular nested arrays, should be lifted in a data parallel language. The list of are in the order: predicates, then-branch, and else-branch. The result (value vector), together with the segment descriptor, is the resulting irregular array.

```
liftedAwhile :: forall e sh.(Shape sh, Elt e)
   ⇒ (Acc (IrregularArray sh e) → Acc (Vector Bool))
   → (Acc (IrregularArray sh e) → Acc (IrregularArray sh e))
   → Acc (IrregularArray sh e)
   → Acc (IrregularArray sh e)
liftedAwhile pred iter init
   = let (a, _, _) = unlift (awhile pred' iter' init')
     in a
  where
    init' = let f = pred init in lift (init, f, or f)
    pred' :: Acc (IrregularArray sh e, Vector Bool, Scalar Bool) → Acc (Scalar Bool)
    pred' f = let (_, _, c) = unlift f in c
    iter' :: Acc (IrregularArray sh e, Vector Bool, Scalar Bool)
       → Acc (IrregularArray sh e, Vector Bool, Scalar Bool)
    iter' (unlift → (a, f, c))
     = let a' = liftedCond f (iter a) a
       f' = zipWith (∧) f (pred a')
       c' = or f'
       in lift (a', f', c')
```

Listing 6: The irregular lifted version of `ahwile`. The function `liftedCondIr` is the irregular lifted version of `acond`, the other functions or Accelerate Prelude functions. For simplicity we use a single irregular array as input type, but in practice this could also be a tuple type. The `pred iter` and `init` arguments are already lifted functions or values.

6.4 Benchmarks

6.4.1 Approaches

We want to compare six different approaches for calculating multiple FFTs at the same time, both for a CPU and GPU implementation.

- Vectorising FFTs using the **foreign** libraries (FFTW and cuFFT).
- **Lifting** the FFT ourselves, using the above **foreign** libraries without using the sequence work.
- Vectorising pure FFTs, where the `awhile` construct will stay **regular**.
- **Lifting** the pure FFT ourselves without using the sequence work, essentially working on **regular** arrays.
- Vectorising pure FFTs, where the `awhile` construct will do **irregular** computations.
- Doing the FFTs as a **normal** calculation one at the time, using a loop.

The point is to compare which approach works best in Accelerate and what the impact on performance is. The foreign functions are mainly here to see what the best achievable speed is, since we expect to be slower.[22] As we mentioned before, it is easy to lift an FFT yourself. A multidimensional FFT is simply doing a one dimensional FFT on each dimension in turn. The pure Accelerate implementation also works via this principle, it performs an FFT on one dimension at a time. So a lifted 2D FFT will only do FFTs on the first two dimensions and leave the last. We did this for the pure and the foreign implementations. Note that we don't need the sequences work for this. Which allows us to see what overhead is created in the sequences work, and what the performance is of the most ideal lifted function. Although this is easy for this particular problem, where we can lift the FFT ourselves, this is not something you want to do in general. For example in the gridding algorithm, we would need to create

[22]Which is to be expected, since people spend years optimizing those libraries for the specific problem of computing FFTs. The point of Accelerate isn't to beat highly hand-optimized implementations, but we do wish to be competitive.

Table 1: The total running times for the different versions of the FFT algorithm on the CPU. Times are in milliseconds (ms).

Version	**N = 1**	**N = 100**	**N = 1000**	**N = 5000**	**N = 10000**	**N = 20000**
Foreign	1.4(7)	26(3)	141(6)	273(15)	7.4(4) $\times 10^2$	1362(5)
LiftedForeign	0.28(4)	13.4(14)	38(7)	72(16)	1.7(5) $\times 10^2$	5.0(9) $\times 10^2$
Regular	41(3)	314(16)	1.34(5) $\times 10^3$	2.44(7) $\times 10^3$	5.86(4) $\times 10^3$	1.159(9) $\times 10^4$
LiftedRegular	8.8(8)	87(5)	3.7(4) $\times 10^2$	6.2(3) $\times 10^2$	1.36(4) $\times 10^3$	2698(12)
Irregular	129(13)	845(10)	4.25(10) $\times 10^3$	7.88(5) $\times 10^3$	1.950(18) $\times 10^4$	4.07(2) $\times 10^4$
Normal	5(3)	6.4(3) $\times 10^2$	1.04(2) $\times 10^4$			

a lifted version of all the calculations being done for one baseline. So we need to lift the code in Listing 5, which can be done, but isn't particularly easy.

We made the benchmarks using Criterion[23], a Haskell microbenchmarking library. Each FFT we calculate is a 2 dimensional with size 32×32. This is the same size we used in the Gridding algorithm, when calculating convolutions. We chose to do the generation of the test data in the benchmark itself (meaning it is not saved in main memory), we found that otherwise the garbage collector was having a big impact on the results. In the real world the data you are executing FFTs on is probably stored somewhere, but we're only interested in the actual performance of FFT and not how the data got there.

One last thing to note is that we're using a slightly older version of Accelerate, which uses a different representation for complex numbers than the FFTW and cuFFT libraries. Therefore, each time an FFT is calculated, there is some overhead to convert between these representations. This is different on the main branch of Accelerate, but this doesn't work yet with the sequences work, but will be added in the future.

6.4.2 Hardware

The benchmarks are performed on a node of the Lisa system of SURFsara[24]. The specs are of one node are:

Processor Type	**Clock**	**Memory**	**Sockets**	**Cache**	**Cores**	**GPUs**
Xeon Bronze 3104	1.70 GHz	256 GB	2	2 x 8.25 MB	2 x 6	4 x GeForce 1080Ti

Although there are 4 GPUs available, Accelerate has no support for multi GPU calculations, so we use a single GPU. The GeForce 1080Ti has 3584 CUDA cores in total divided over 28 Streaming Multiprocessors (SMs), running at 1582 Mhz. The maximum number of threads running at the same time is 2048 per SM for this architecture. But it depends on the code if this is achievable. But this gives an indication of how many threads that are needed to fully occupy this GPU.[25] Accelerate does use both the CPUs that are available.

We chose sizes between 1 and 20000 FFTs for the GPU and between 1 and 10000 for the CPU. Only the normal implementation is only tested up to 1000 FFTs. To get the maximum speed out of a GPU more threads are needed, than that can be executed at the same time. Because the GPU can hide latency in the threads, by switching them out. The 20000 FFTs should be sufficient for this.

’

6.4.3 CPU Results

The results can be found in Figures 12 and 10. In Figure 11, we made a linear ($y = ax + b$)) fit on these results since we expect that doing multiple FFTs scales linearly, and we can model the overhead as a constant. The linear

[23]Made by Bryan O'Sullivan, `http://www.serpentine.com/criterion/`.

[24]The Lisa system is a cluster computer consisting of several hundreds of multi-core nodes running the Linux operating system, `https://userinfo.surfsara.nl/systems/lisa/description`.

[25]If you're quick with arithmetic's this is indeed a maximum threads count of 57344, which would be equal to 56 FFTs of dimensions 32×32.

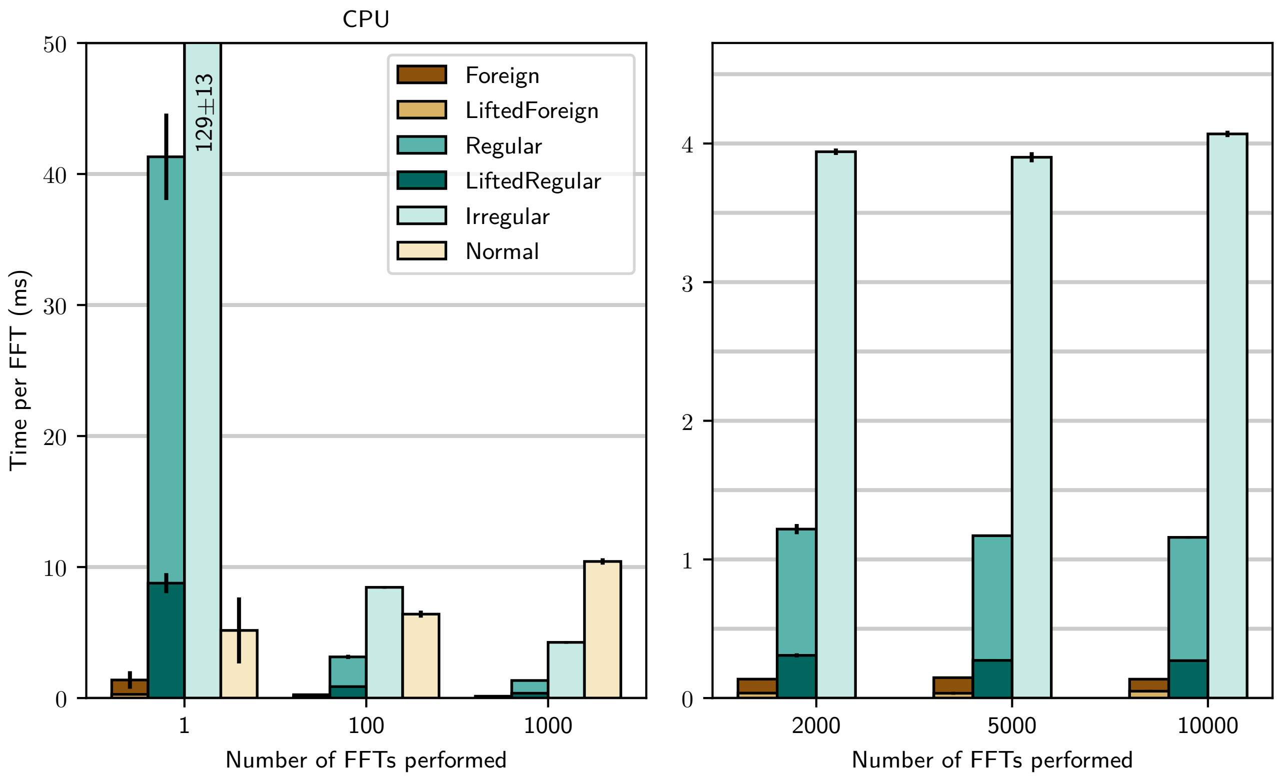


Figure 10: The benchmarks of running multiple FFTs on the CPU, with several approaches. **Foreign** is using a vectorized approach with the foreign FFTW library. **Regular** is an FFT in pure Accelerate, working on regular nested data, **Irregular** is the same but working on irregular nested data. The **Lifted** versions, are without the sequences work, and the lifting was done manually. The **Normal** version, calculates one FFT at a time using a loop.

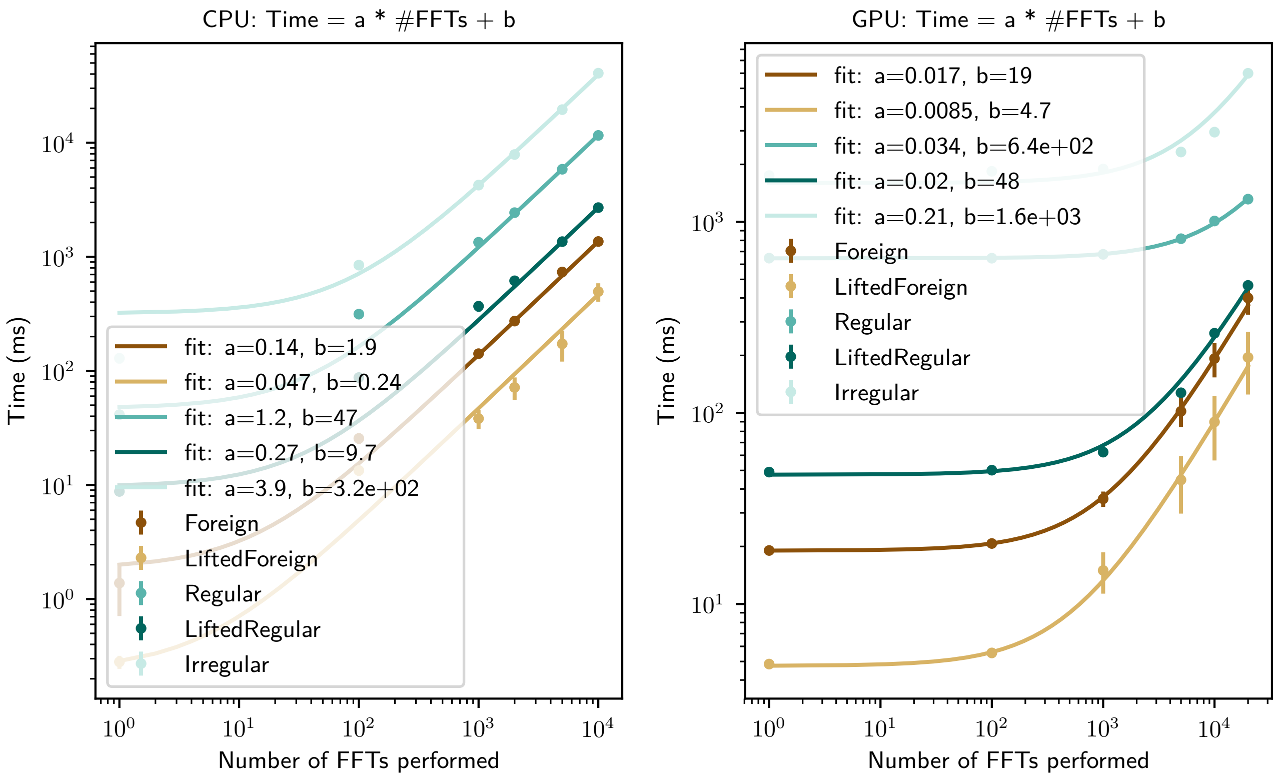


Figure 11: A linear fit made on the FFT benchmarks, where the total run times are plotted on y-axis, and the number of FFTs performed on the x-axis. The *a* parameter gives an indication of how much time one FFT takes, and the *b* parameter, what the start up time (overhead) is of the algorithm. **Foreign** is using a vectorized approach with the foreign FFTW (CPU) or cuFFT (GPU) library. **Regular** is an FFT in pure Accelerate, working on regular nested data, **Irregular** is the same but working on irregular nested data. The **Lifted** versions, are without the sequences work, and the lifting was done manually. The **Normal** version, calculates one FFT at a time using a loop.

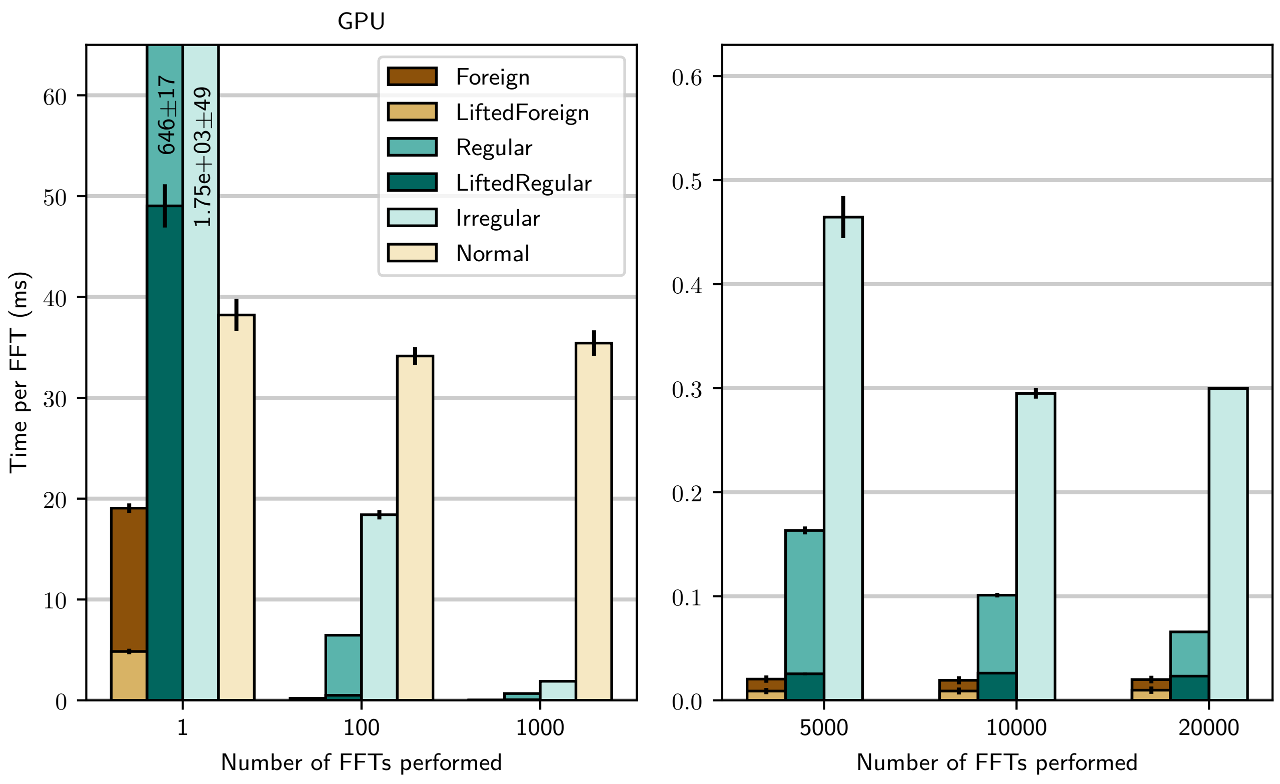


Figure 12: The benchmarks of running multiple FFTs on the GPU, with several approaches. **Foreign** is using a vectorized approach with the foreign cuFFT library. **Regular** is an FFT in pure Accelerate, working on regular nested data, **Irregular** is the same but working on irregular nested data. The **Lifted** versions, are without the sequences work, and the lifting was done manually. The **Normal** version, calculates one FFT at a time using a loop.

fit isn't made for the normal case, since it has only 3 points.

The CPU benchmark shows that the regular implementation is always faster than the irregular one by a large margin. The linear scaling is 3.3 times smaller, meaning each FFT takes approximately 1/3th of the time. The normal implementation is faster in the case of doing one FFT, but it is almost 5 times slower at 1000 FFTs. We didn't benchmark for a higher number of FFTs, since this demonstrates the scaling behaviour. We suspect that this is the case because the way it is constructed, it isn't possible to do more then 32×32 calculations at the same time (since the FFTs are made dependent via the `awhile`), and apparently it is more optimal for the CPU do more calculations at the same time.

The foreign FFTW library is always faster than the regular implementation, an 8.9 times better scaling if we look at the linear fit.

The scaling of the own lifted versions are as expected superior. Although surprisingly, this also holds true for the function. Which is essentially using the same function, but with the sequences work wrapped around it. We would expect a constant overhead, yet why the scaling is 2.8 times better is unclear. We suspect that the scheduler might induce this overhead, since it has to wait for the timing of each chunk, before deciding the size of the next chunk. The pure lifted function scales 4.3 times better, which can be partly attributed to the same effect as what is happening with the foreign function. However, it also shows that still a faster implementation could be obtained for the lifting. Although we are not sure if this is achievable, since the knowledge of FFTs allows us to come up with a faster version, which might not be something that the compiler could ever determine.

Table 2: The total running times for the different versions of the FFT algorithm on the GPU. Times are in milliseconds (ms).

Version	**N = 1**	**N = 100**	**N = 1000**
Foreign	19.1(5)	20.7(7)	36(3)
LiftedForeign	4.8(3)	5.54(19)	15(4)
Regular	646(17)	6.5(1) $\times 10^2$	677(14)
LiftedRegular	49(2)	50.2(13)	62(4)
Irregular	1.75(5) $\times 10^3$	1.84(5) $\times 10^3$	1.89(7) $\times 10^3$
Normal	38.2(16)	3.41(9) $\times 10^3$	3.54(13) $\times 10^4$

Version	**N = 5000**	**N = 10000**	**N = 20000**
Foreign	102(18)	1.9(4) $\times 10^2$	4.0(7) $\times 10^2$
LiftedForeign	45(15)	9(3) $\times 10^1$	2.0(7) $\times 10^2$
Regular	817(19)	1.01(2) $\times 10^3$	1.32(2) $\times 10^3$
LiftedRegular	128(7)	261(10)	465(13)
Irregular	2.32(10) $\times 10^3$	2.95(5) $\times 10^3$	6.00(3) $\times 10^3$

6.4.4 GPU Results

Let us look at the GPU benchmarks. The normal implementation scales the worst. It is 1.7 times slower than irregular at 100 iterations, but it is even worse at 1000, where it is 17 times slower. This was to be expected, since doing only 32λtimes 32 amount of work, would never satisfy the GPU. Next we see that there seems to be an overhead in the sequence implementations. The time difference between 1 and 1000 FFTs is significant, although it is still in a ±15% range of each other. This overhead seems to be caused by the sequences work, since the normal implementation has significantly less overhead. However, it also differs for the regular and irregular version. The foreign cuFFT implementation is again faster in all the cases. Although the scaling is only twice as fast compared to the regular version. In the aim for Accelerate to be competitive, the most important thing would be to reduce the overhead of the sequences work. Furthermore, we can see that the GPU does need enough data for it to scale well. The time difference between 1, 100 and 1000 FFTs is there, but is negligible compared to the overhead. And at a 1000 FFTs we have $32 \times 32 \times 2 \times 1000 = 2 \times 10^6$ data points (counting a complex number as two points), which is at about 35 times the maximum number of threads that can run on this specific GPU. And we need more threads than the maximum, to hide the latency on the GPU. The lifted versions behave the same as with the CPU lifted version, nevertheless there seems to be less difference in the scaling. The bigger overhead can probably be attributed to the scheduler again, since we will always start with a chunk of 1 FFT, and it takes $log_2 1000 \simeq 10$ steps (it doubles the size each time) before we get closer to what the GPU needs to get to full speed. And scheduling here is probably even worse for the GPU, since it takes more time to communicate between the CPU and the GPU (since the CPU is doing the scheduling). Although not everything can be attributed to the scheduler, since if you look at the lifted foreign, foreign pair the constant overhead difference is only 15 ms according to the linear fit. However, in the regular case this is 597 ms.

Thus, these benchmarks show that it is indeed better to solve nested data parallelism, with full flattening of the program, if the list of arrays is big enough. The other approach of doing it in a loop, falls of hard at already 100 FFTs, on the other hand it has less overhead to start with. Comparing the regular and irregular case, we see that the regular one is always better, as expected and by a large margin. Still, we also note that the lifting of the sequences work gives a lot of overhead in general. In the next section we will use these vectorised FFTs while benchmarking the gridding algorithm.

7 Benchmarks: Gridding Algorithm

7.0.1 Approaches

In Section 5.3, we explained the gridding algorithm and the nested parallelism in the problem. We recognized that FFTs are at the core of the gridding algorithm, so we made the independent analysis so that we can vectorise the Split-Radix FFT algorithm. In this section we will compare our work with the different versions of the gridding algorithm. These are:

- **Normal**: The original gridding algorithm, without nested parallelism. Here we process the baselines one at a time. It is using the foreign FFT libraries FFTW on the CPU or cuFFT on the GPU.
- **Reference Python**: The algorithmic reference library[26], on which we based our gridding implementation. This version is only implemented on the CPU, but it has a simple way parallelism by dividing the baselines over all the different cores. It uses the NumPy FFT version. It's important to note that this version isn't optimized for speed, but used as a clear way to describe the algorithm.
- **Foreign**: The gridding algorithm with nested parallelism, which is using the foreign libraries.
- **Regular**: The gridding algorithm with nested parallelism, where the FFT algorithm is a pure Accelerate one. Here we make sure to stay regular in our calculations.
- **Lifted Regular**: The gridding algorithm with nested parallelism, where the FFT algorithm is a pure Accelerate one. But here we did the lifting for the FFT ourselves.

We made a slight adjustment of the algorithm, where it doesn't perform a last Fourier transform, thus the resulting data remains in the Fourier space. But this shouldn't impact teh conclusions of our results, as this done only once and is not related to the nested data parallelism we encountered. We process a maximum of 130816 baselines, coming from 512 telescopes combinations ($512 * 511/2$). But we vary the number of baselines in the benchmarks, to compare the scaling of our different versions. We use a $Theta = 0.008$(field of view dimension) and $Lambda = 30000$ (uv-grid dimension). This leads to a grid resolution of 2400×2400. In the reference library normally $Theta = 0.08$ was used, but this lead to a grid resolution, which is about 4GB in memory. To solve this, you would want to represent this array as a sparse array, since the data is actually quite sparse. But for now we simply chose a lower resolution, since we are more interested in the performance of the processing of the baselines.

7.0.2 Hardware

The benchmarks are run on a different machine than was used for the FFTs results. It turned out that the compiling took a long time, so we needed a quicker CPU. The specs are:

Processor Type	**Clock**	**Memory**	**Sockets**	**Cache**	**Cores**	**GPUs**
Intel(R) Core(TM) i7-6800K CPU	3.40GHz	64 GB	1	15 MB	6	GeForce 1080Ti

The times we report is the time it takes starting it from the command line. Each reported time, is repeated at least 10 times for consistency. In the Accelerate implementation, we subtracted the compilation time. In theory the compilation of the Accelerate language, could already be done during compilation.[27]. The compilation times are reported in Table 5. Note that especially the regular implementation takes almost a minute to compile. In the Accelerate versions that are working with nested data parallelism, we fixed the chunk size of how many baselines to process at once. This is fixed to 20.000, which seemed to be the optimum after some manual checks. This is going to be system dependent, but chose to do this, to let the scheduler of the sequences have less of a negative impact on the performance.

[26] `https://github.com/SKA-ScienceDataProcessor/crocodile`

[27] This is supported in the main branch of Accelerate, with template Haskell. But the branch we are on, doesn't have this support yet

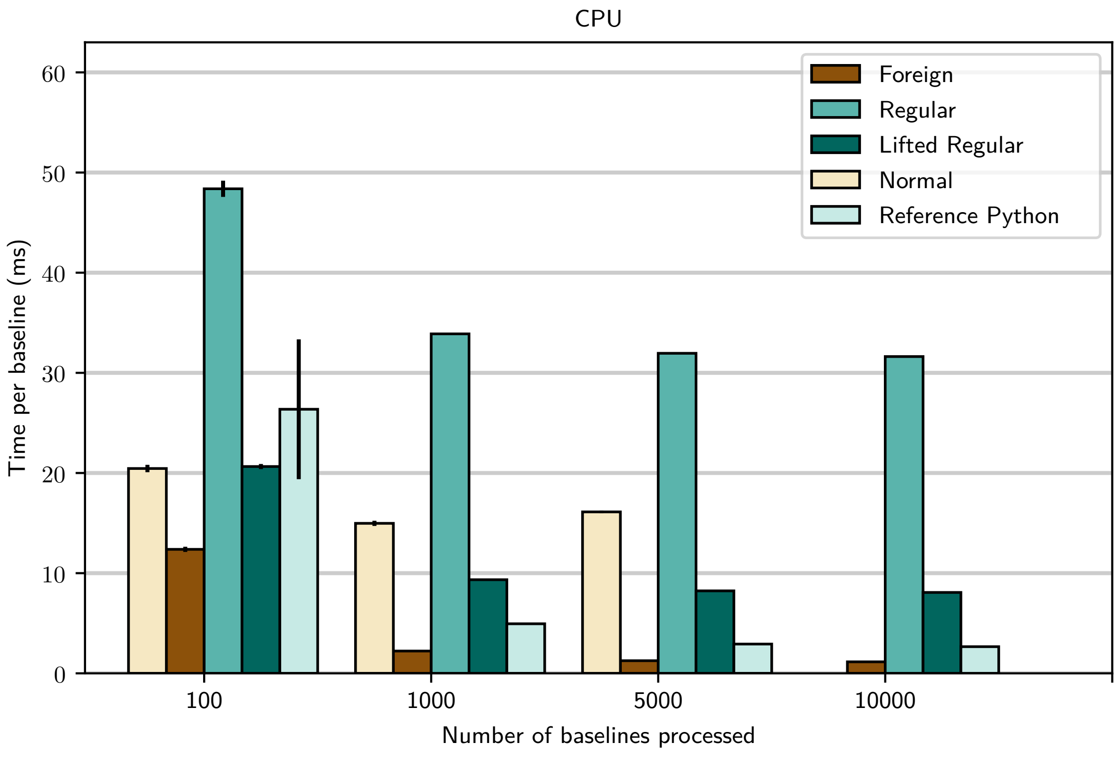


Figure 13: The CPU benchmarks of different versions of the gridding algorithm running, where we vary the number of baselines that are processed. We have the normal Accelerate version (Normal) that uses the FFTW and processes the baselines sequentially, the sequences version that uses the FFTW (Foreign), the sequences version that uses the pure FFT (Regular), and a sequences version where we manually lifted the pure function (Lifted Regular) and the reference Python implementation (Reference Python).

Table 3: The total running times for the different versions of the gridding algorithm on the CPU. Times are in seconds (s).

Version	**N = 100**	**N = 1000**	**N = 5000**	**N = 10000**
Foreign	1.238(26)	2.23(5)	6.30(5)	11.48(6)
Regular	4.84(8)	33.89(9)	159.76(16)	316.25(22)
Lifted Regular	2.065(25)	9.35(10)	41.17(4)	80.75(7)
Normal	2.04(4)	14.97(26)	80.6(6)	
Reference Python	2.6(7)	4.946(19)	14.606(18)	26.60(4)

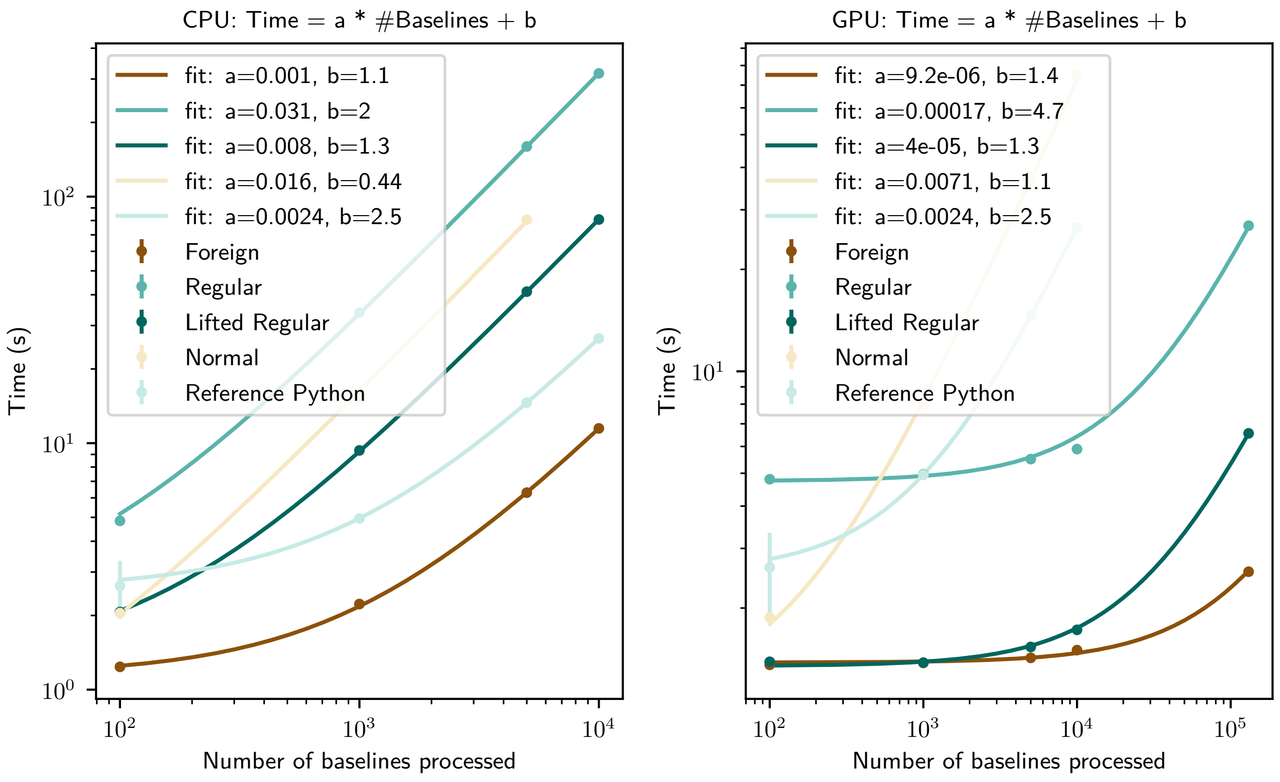


Figure 14: A linear fit made on the gridding benchmarks, where the total run times are plotted on the y-axis, and the number of baselines processed on the x-axis. The a parameter gives an indication of how much time one baseline takes, and the b parameter, what the start up time (overhead) is of the algorithm.

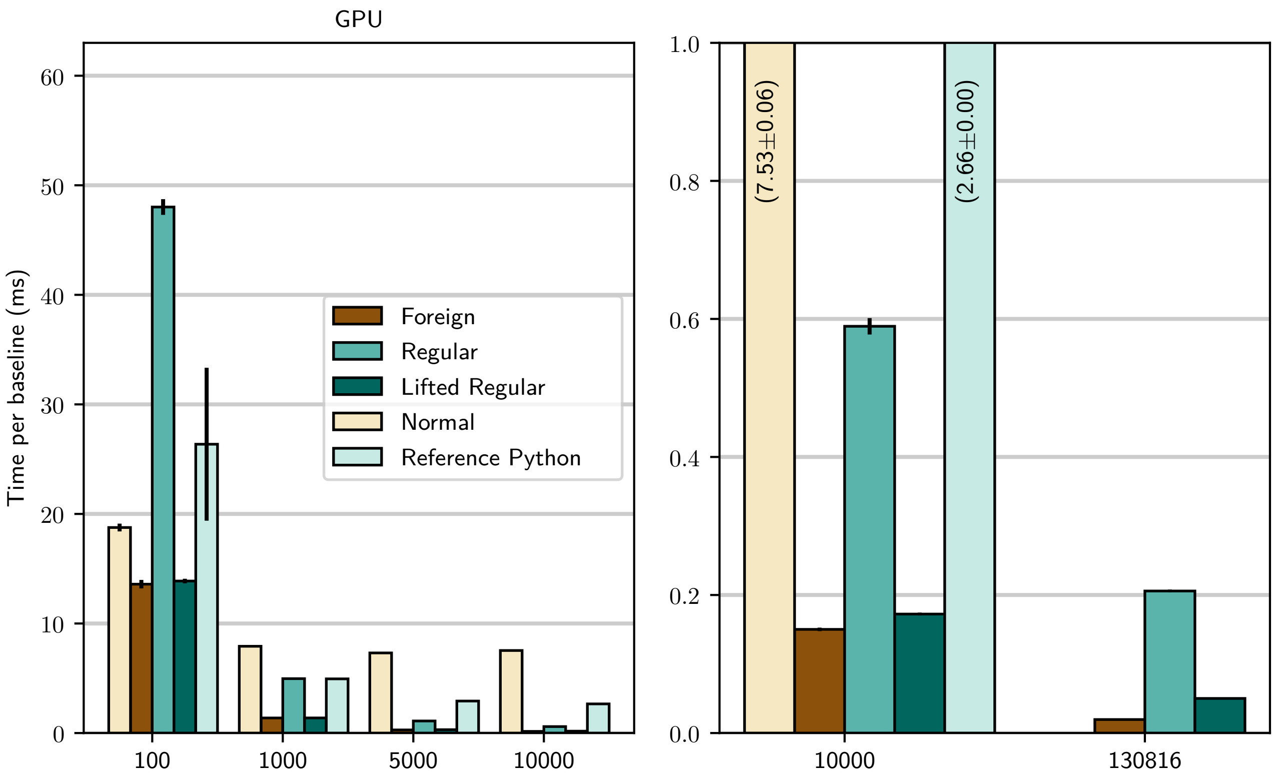


Figure 15: The GPU benchmarks of different versions of the gridding algorithm running, where we vary the number of baselines that are processed. We have the normal Accelerate version (Normal) that uses the cuFFT and processes the baselines sequentially, the sequences version that uses the cuFFT (Foreign), the sequences version that uses the pure FFT (Regular), and a sequences version where we manually lifted the pure function (Lifted Regular) and the reference Python implementation (Reference Python).

7.0.3 CPU results

The results of the benchmarks using the CPU can be found in Figures 13and 14, and in Table 3. We also give the compilation times for the algorithms implemented in Accelerate in Table 5. We make a few key observations:

- On the CPU version the Accelerate sequences version is actually the slowest. It is also using the pure Accelerate implementation, whilst most of the others use a quicker library for this. But if we look our manual lifted regular version, we see that we are about three times slower than the optimum speed of what we can expect of this lifted version of Accelerate. The normal version is even faster than this implementation, although it uses the foreign libraries. But this results differs from the FFT results, where we were at least faster than the normal implementation. Apparently we are less efficient, if the nested parallel part is more complex.
- If we use the foreign libraries in the nested parallelism version, we are in fact the fastest on the CPU. Thus, the flattening of nested parallelism, seems the best approach on the CPU. We are also faster than the Python version. However, that implementation wasn't optimized of course.

7.0.4 GPU results

The results of the benchmarks for the GPU can be found in Figures 15 and 14, and in Table 4. We also give the compilation times for the algorithms implemented in Accelerate in Table 5. We make a few key observations:

Table 4: The total running times for the different versions of the gridding algorithm on the GPU. Times are in seconds (s).

Version	**N = 100**	**N = 1000**	**N = 5000**	**N = 10000**	**N = 130816**
Foreign	1.36(4)	1.377(24)	1.425(16)	1.502(25)	2.56(8)
Regular	4.80(7)	4.96(4)	5.50(7)	5.89(12)	26.92(26)
Lifted Regular	1.388(21)	1.380(26)	1.535(23)	1.724(21)	6.559(35)
Normal	1.88(4)	7.92(7)	36.56(26)	75.3(6)	
Reference Python	2.6(7)	4.946(19)	14.606(18)	26.60(4)	

Table 5: The compilation times for the algorithms implemented in Accelerate. Times are in seconds (s).

Platform	**Foreign**	**Regular**	**Lifted Regular**	**Normal**
CPU	1.461(33)	57.1(5)	3.41(6)	0.932(27)
GPU	4.6(10)	45(4)	11.14(9)	2.44(4)

- On the GPU, the regular version is faster, than the normal implementation, if we have more than 1000 elements. Thus the flattening of the nested parallelism pays off more on the GPU. Which we would expect, because the GPU needs more available parallelism, to work efficient. And processing a single array of size 32×32 at the time, isn't sufficient.
- The own lifted version is again a lot faster, the scaling is about 4 times better. Thus also for the GPU the lifting is still far from the optimum.
- The foreign sequence implementation is again the fastest, and also by a large margin. It scales again 4 times better than the lifted regular version, and 240 times better than the Python version. Although the Python version is purely on the CPU, and the GPU has more raw computation power then the CPU.

To summarize, making use of all the parallelism in the gridding algorithm benefits the performance. Especially on the GPU the additional parallelism is needed. However, specialized lifted versions of the FFT perform significantly better than the automatic lifted one. So we suspect that improvements can be made in the results of the automatic lifting. But we would need to analyse the resulting code to see where exactly.

Part III

Closing Remarks

8 Related Work

In this section, we review the most important other work that researchers have done in the field of data parallel languages and nested data parallelism. First, we discuss the most used approach to handle nested data parallelism. Then, we discuss other more recent approaches to data parallelism in general. Lastly, we talk about how our work compares to others.

8.1 Nested Data Parallelism in General

We must first mention the work of Blelloch and Sabot [3], which is one of the first papers addressing the problem of compiling nested parallelism languages. The flattening idea that they proposed is what we are still using in the Accelerate compiler today. Based on this research, they created the language NESL [4], which inspired most work on nested data parallelism.

8.1.1 Data Parallel Haskell

The Data Parallel Haskell [22] is based on the key insights of the work of Blelloch and Sabot [3] and implements these ideas in Haskell targeting the CPU. However, where NESL is a first-order language with only a few data types, Haskell is a higher-order language with a rich type system. The concepts they implemented are almost the same, but they made two significant innovations: the non-parametric representations of arrays and the treatment of first-class functional values. Compiling the Data Parallel Haskell programs consists of the following steps

1. The compiler desugars the code is to the GHC core language
2. It applies vectorisation to transform the nested data parallelism into flat data parallelism
3. It applies fusion for optimisations
4. The compiler divides the parallel operations into chunks, so they can be assigned to threads. For example, in a CPU, each core gets a thread.

We focus on the vectorisation since that is most relevant for our research.

They denote the parallel arrays by `[:a:]` and these are the collection on which the parallel functions work on. Internally they change them to a type `PA a` and how they are stored depend on the type. This is unlike standard arrays in Haskell, which the computer stores as a list of pointers. For example `[:Int:]` is stored as a contiguous memory block of 32-bit Integer values. Therefore, the implementation is dependent on the type of the elements (so non-parametric). This implementation is then captured by the type class `PAElem`, which defines the underlying type of `PA a`, the length of an array, and some more operations. In general base types like Integers and Floats are just stored as a contiguous array, but a tuple `[:(a, b):]` is split into a pair of arrays. Therefore, it changed from an array of structures to a structure of arrays. For nested arrays, they take a similar approach is as the work of Blelloch and Sabot [3]. They use a flat value array for the values (`PA a`), and add a segment descriptor with type `PA (Int, Int)`, which stores the starting position and the length of each subarray. Thus the main difference is that the starting position is also stored. For recursive data types, the internal data representation is also recursive. For example for the type **data** `Tree = Node Int [:Tree:]` is represented as **data** `PA Tree = PANode (PA Int) (PA (PA Tree))`, where the latter is a nested tree, where we can use the ideas from nested. In practice, this means that each recursion level has its own data (the `PA Int`) and the computer can process each recursion level in parallel.

Another big difference in Haskell compared with NESL, is that Haskell is higher-order. This means that functions can be passed as arguments and the compiler treats them like normal values. The basic transformation is lifting a function, such that it can work on a parallel array. They lift functions in a similar matter as int the work of Blelloch and Sabot [3]; the compiler replaces calls to regular functions to their lifted counterpart. This means that if we pass a function as an argument, we always need to pass the regular and lifted version. A second challenge that they encounter is a parallel array of functions. In data parallel languages, we work with the SIMD model; this means that we have similar instructions. They represent an array of functions as the same instructions (one pointer to a function), that is initialised by a certain environment. It contains all the free variables occurring in the function. So for example the function `f x = x + a` has the free variable `a` which is the environment in the function. Moreover, when lifting, the environment is made into a parallel array, since it can have different values for different execution paths. It would still be possible to use different functions, like an array of `Float → Float` functions such as [`cos`, `sin`], but then conditionals should be used. Also, for conditionals, there is an overhead in executing, which means there is no efficient implementation possible.

With these adjustments, they make similar transformations as Blelloch and Sabot [3] to vectorise the whole program. We can find the full list of vectorisation transformation in the work of Peyton Jones et al. [22, see fig. 6 chap. 6]. For example, we can consider transforming the following simple function:

```
inc :: Float → Float
inc = λx → x + 1
```

The full vectorised function will be

```
inc_V :: Float : − > Float
inc_V = Clo () inc_S inc_L
inc_S :: () → Float → Float
inc_S = λe x → case e of () → (+_L) $: x $: 1
inc_L :: PA () → PA Float → PA Float
inc_L = λe x → case e of ATup_0 n → (+_L) $:_L x $:_L (replicatePA n 1)
```

Here `Clo` is constructor that is they use for the new vectorised function, it contains the environment (in this case (), because there were no free variables), the serial function inc$_S$ and the lifted function inc$_L$ and it has type `a : − > b` (in this case Float to Float). All the function calls (+) are replaced by their vectorised version ($+_L$). `$:` extracts the serial version from the `Clo` constructor with environment applied. $\$:_L$ does a similar thing for the lifted version, but for the `AClo` constructor which is a constructor used for functions in parallel arrays. ATup$_0$ `n` is the constructor for the empty tuple in a parallel array of length `n`. So the constant 1 that was in the function is replicated `n` times so that it lifted functions can use it.

Eventually the `mapP :: (a → b) → [:a:] → [:b:]` is one of the core functions, that is going to deal with the nested parallelism. Internally it is vectorised as follows:

```
mapP_V :: (a : − > b) : − > PA a : − > PA b
mapP_V     = Clo () mapP_1 mapP_2
mapP_1 :: () → (a : − > b) → PA a : − > PA b
mapP_1 _ f = Clo f mapP_S mapP_L
mapP_2 :: PA () → PA (a : − > b) → PA (PA a : − > PA b)
mapP_2 _ f = Clo f mapP_S mapP_L
mapP_S :: (a : − > b) → PA a → PA b
mapP_S (Clo Γ _ fl) xss
           = fl (replicatePA (lengthPA xss) Γ) xss
mapP_L :: PA (a : − > b) → PA (PA a) → PA (PA b)
mapP_L (AClo Γ _ fl) xss
           = unconcatPA xss (fl (expandPA xss Γ) (concatPA xss))
```

The function mapP$_L$ does the important work for the nesting. First off the environment for each subarray is originally only one element, but when it is flattened, we need this environment for each element of the subarray. Thus, we need to replicate the environmental values via the `expandPA` function, which uses the segment descriptor of `xss`. Then the array is flattened by `concatPA`, so that lifted function `fl` can process the data in parallel. Finally, the result is lifted again by the `unconcatPA` using the segment descriptor of `xss`.

Above we have summarised the approach of flattening data parallelism in this research. We focussed mainly on the actual transformations but did not mention the performance. This was also the main problem for Data

Parallel Haskell; the scope was extensive and could deal with all nested data parallelism. However, it proved too difficult to implement this fully in the existing compiler of Haskell (GHC), while getting the performance that is competitive with low-level handwritten code for SIMD architectures. The choice was made to take some critical insights from this research and build this into an embedded DSL: Accelerate. Accelerate chose to implement a subset of the functionality and focus on optimising that. From that basis, new features can be added, while keeping the performance in mind. In the next subsection, we discuss some other approaches to data parallel programming.

8.2 Other Data Parallel Languages

Since the work discussed in the last section, there have been different approaches to data parallel programming. In this subsection, we briefly address some more modern languages that people created. Most have taken a domain-specific approach. This limits the usability to one domain but makes it is easier to make assumptions to optimise the code.

The work of Bergstrom and Reppy [1] was the first that tried to compile a nested data parallel language to the GPU. They took the NESL language [4], which compiles which to the VCODE language, which is intended to run on a vector machine. The C Vector Library (CVL) implements the VCODE vector operations in C, and in their work, they ported CVL to CUDA. A simple port would lead to performance issues since it would produce all the instructions as separate kernels that share their results via global memory in the GPU. This leads to inefficient code. Thus they optimise the generated VCODE by fusing it. They compare their work with the CPU-based NESL, Data Parallel Haskell, Copperhead[28], and hand-coded CUDA code. For the benchmarked algorithms, they are faster than the other languages except for the hand-coded CUDA code. This work was a first effort to compile a simple data parallel language to the GPU, but there are still many opportunities to improve on the expressiveness of the language and performance. The original work has been improved upon [24], by more fusion optimisations, and better memory management on the GPU. The work of Bergstrom et al. [2] has taken similar steps, but here MIMD architectures were targeted. The main difference is that they do flatten all the nested data, but do not fully vectorise it. This makes the most sense for conditionals since we need to take all branches for all the data in SIMD architectures, but this is not necessary for MIMD architectures. They build this in Manticore, a compiler for Parallel ML [11].

[23] created Halide for the imaging and tensor calculation domains. It is a DSL embedded in C++ that has a functional style of declaring algorithms while being able to define a schedule independent of the algorithm to look for optimisations. The main strength of Halide is being able to try different executing schedules relatively easy, looking for the best performance. It can compile both to the CPU and GPU.

Futhark [14] is a new functional language that focusses on the domain of array computations. It takes a similar approach to Accelerate, but does this as a stand-alone language. It comes with the possibility of in-place modifications of arrays to reduce memory usage and some form of nested parallelism. It focusses on second-order array combinators (like map or reduce), to express parallel patterns that can be used to make parallel programs. The in-place modification is made possible with the ability to specify a uniqueness type. Furthermore, it does higher-order reasoning on the program to perform rewrite rules. This makes it possible to have fusion, to expose the parallelism for some form of nested parallelism and some locality optimisations. A simple approach of making nested parallelism work is to perform the work sequentially, but potentially much parallelisation is lost this way. In Futhark they make sure no parallelisation is lost for nested parallelism if the nested operations work on regularly shaped data and there are no conditional branches. The locality optimisation is used to utilise the fast memory on modern GPUs. More recently they added more support for a certain kind of irregular nested parallelism [10, 15].

Single Assignment C (SaC) [12] is a pure functional language, that targets computations on multidimensional arrays. The language is functional but aims to have a syntax that resembles imperative languages like C/C++/C# or Java. We can then compile the language to different architectures, like multi-socket, multi-core and graphics accelerators (GPGPU) via CUDA. SaC aims at outperforming sequential imperative codes on parallel hardware. SaC does not support higher-order functions, currying or partial applications, unlike most functional languages. It also is a strict language. The language supports a subset of C, without pointers, side effects and some others, and adds stateless arrays, an IO/system and some more. These arrays, have a certain rank, the number of dimensions, and a shape vector, which stores the extent of each dimension. The base functions on arrays are very basic, getting dimension and shape information, modifying the shape, selecting and modifying a value at a specific index and

[28]Copperhead is an extension of the Python language which supports data parallel operations.

a `with`-loop that goes over the array. The with loops support general generating, mapping and folding arrays. From these basics, they can define most other functions, and they did this in a separate library. SaC has no general support for nested data parallelism for their arrays, but do support a form of rank-polymorphism in their `with`-loop.

Lift [26] is an intermediate language, which data parallel languages can target to generate efficient parallel code. Because of the complexity of writing code for the GPU, language designers find a part of the solution in Domain-Specific Languages (DSLs). They make it easier to write code for GPUs that is performing fast. However, each DSL has to hard code its compiler for each back-end, which is a lot of work. Lift is a language that these DSL can target. Lift uses provable rewrite rules to optimise the code and compiles to OpenCL.

This section is a brief overview of nested parallelism. Next we will talk about the analyses we explained in our work and how they compare to other research.

8.3 Compared to Other Research

There are a few pieces of research that link closely to the work present here on nested data parallelism. The independent analysis is very similar to vectorisation avoidance. Keller et al. [16] addresses this topic, and Clifton-Everest et al. [8], Madsen et al. [18] added this to Accelerate. Because the vectoriser can only avoid terms if they are independent. Part of this was already present in Accelerate, and probably the work can be fused with the work we have done. However, we can make some improvements to our work that could distinguish it. For example, we could add the possibility to track element independence separately, and we can mark more cases as independent. Since a shape can later be transformed to be independent again. Although admittedly, we do not know for good use case of this now.

The shape analysis is most similar to shape inferencing that compilers in array languages perform. Many people have already researched shape inferencing [17]. Although in the language of Accelerate, this was less interesting for flat data parallelism. Normally we give the sizes of arrays in Accelerate, so the shape is automatically known. However, with nested data parallelism, this information has to be computed. This has been done in Accelerate by Madsen et al. [18], but also in Futhark by Henriksen et al. [13]. Especially this last work is similar to ours, since it also tries to do some static analysis, to better determine the shape dynamically. Although in the end, these analyses are interested in getting the exact shape. However, in our work, we do not care about the exact shape, only how two shapes compare with each other. Which means we have fewer restrictions and can do some other reasoning. Also, we stay completely static. In the Futhark work, they add new code to pass shape information, which the computer executes dynamically.

Moreover, as far as we know, this analysis is not done before. Also, not for vectorising functions. We think this analysis can be useful for most functional languages that want to flatten nested data parallelism.

9 Conclusion

To summarise this work, we showed that in the case of doing multiple FFTs, expressing this with one level deep nested parallelism is beneficial. It can expose all the available parallelism, which leads to an implementation that executes faster on both the CPU and GPU. If FFTs can use a regular implementation, instead of an irregular one, this also leads to faster running code.

We have presented two ideas that help preserve regularity in conditional and loop control statements. These ideas are the basis for the independent and shape analysis, which we both implemented in the Accelerate compiler. The independent analysis allows the Split-Radix FFT algorithm to stay regular. When considering other algorithms for FFTs, we cannot prove regularity yet. To accommodate these use cases, we implemented a language feature in Accelerate, that allows the programmer to provide his own lifted version of a function.

The implementation of the gridding algorithm is quicker than the simple reference version which others wrote in Python. In the CPU case, this is only true if we make use of foreign libraries. In the GPU case, we are quicker when we have enough parallelisation available. We notice an execution time overhead in the pure Accelerate version which the compiler vectorises automatically, but it is unclear yet what the cause is. Also, the compile-time of the pure Accelerate version is significant, and even more than the actual running time. Therefore, we conclude that there is still work to improve on when dealing with nested parallelism in Accelerate.

10 Future Work

We want to address several issues and ideas we encountered in this work, which we think deserves some more attention in future work.

10.1 Analyses

Concerning the analyses, we can use the independent analysis only on the Split-Radix FFT algorithm. We would want to find out if the analysis can be improved, to prove regularity for the other FFT algorithms as well. Alternatively, if this deems too difficult, it may be useful to find different ways to express these algorithms differently, which makes proving regularity easier. A new language pattern might achieve this.

The shape analysis is now only used for proving regularity in control statements. We believe it can also be useful in other stages of the compiler. For instance, when we zip two arrays together, we can omit bound checks if they have equal shapes. Also, when vectorising irregular nested parallelism, we would not need to alter the segments descriptors of the results in some instances (while and conditional statements and maybe others). Which should lead to better performance, but this is not done in the compiler yet. However, if the compiler is using this analysis more often, it is interesting to look at the running time of the analysis something we ignored up to now.

We made the two analyses with respect to a flat data parallel world. The independent analysis keeps some track of variables that might live in the nested world. The shape analysis is not considering it all. It is interesting to see what adjustment one must make, to have it work in the arbitrarily nested data parallel case.

For the implementation of the analysis, we use the type-safeness that Haskell provides as much as possible. Which means we adhere to the types of the AST of Accelerate, but we ran into a problem with a double application of two non-injective type families. E.g. `TakeOnlyCircles` (`MakePurple` `a`) $\equiv$ `MakePurple` (`TakeOnly Circles` `a`), where we describe two type families that work on coloured drawings-types. It would be nice to investigate if this would already be possible in a Haskell compiler, or what adjustments in the type system are needed to make this possible.

We tried to decide everything statically with the analysis. However, it could be worthwhile to perform dynamic checks as well, to decide if we can stay regular. If we do this, we will induce some compile and execution overhead since we would need to compile more code paths and spend time doing these checks. It could also influence fusion since we would not know which code path we take yet at that stage. An idea is to only have these dynamic checks for particular cases, where we suspect there is a high chance that we can stay regular. We could also use the information of the shape analysis during the dynamic checks.

10.2 Nested Data Parallelism in Accelerate

When we used to sequence work, with support for one level deep nested parallelism, we encountered some problems. The scheduling of the chunks could be improved since it is merely doubling the work each iteration until it slows down. There is also overhead in both execution and compile time. A big chunk of the execution seems constant, unrelated to the number of elements that were processed. We suspect that the compiler adds much code during vectorisation that is unnecessary. For example, when we try to vectorise the most general FFT algorithm in Accelerate, the compiler produced an AST that is over 1 million lines (it had to assume irregular data).

Nevertheless, when we lift the FFT ourselves, we do not see this overhead. Thus, it is worthwhile to see if we can reduce this. In the work Clifton-Everest et al. [8], they did not report this overhead. So we suspect this has to do with the lifting of control statements, which we think is the main difference and was also partly implemented by ourselves.

Although we just mentioned that we think one can improve the lifting in Accelerate, it might also be simply not possible to that for specific problems. Thus having the option to have the programmer lift itself might be necessary from a more practical point of view. Although having more languages designs to express nested data parallelism, is also an excellent way to look. The work of Elsman et al. [10] is a fine example of this.

10.3 Nested Data Parallelism in General

In general, the problem of finding a data parallel language where we can freely express nested data parallelism while being fast enough is not solved yet. There are several directions to take, where we think that adding more language constructs, that safely add more forms of nested data parallelism is the best one. This forces the programmer to express their problem in a certain way, while the compiler can use this, to make better assumptions. Therefore it can produce faster code.

10.4 Gridding Algorithm

The last thing we want to address is the gridding algorithm. We only compared it with a simple reference implementation, which the maintainers did not optimise for speed. Although Accelerate does have the same complexity as this algorithm, computational wise. There are other gridding algorithms available, that should be quicker. For example, the work of [28] is an optimised algorithm targetted at the GPU. It would be interesting to see how our implementation compares and if one can express this new algorithm in Accelerate.

11 Acknowledgements

I want to thank Dr. Trevor L. McDonell,for all the hours we spend discussing this work and supervising me during my master thesis. I want to thank Prof. dr. Gabriele K. Keller for her valuable remarks during the process. I want to thank Michael van den Hoogenband, Lukas Arts, Ivo Gabe de Wolff and Cas van der Rest for proofreading (versions) of this work, and their invaluable remarks and criticism.

References

[1] Lars Bergstrom and John Reppy. Nested data-parallelism on the gpu. *SIGPLAN Not.*, 47(9):247–258, September 2012. ISSN 0362-1340. doi: 10.1145/2398856.2364563. URL `http://doi.acm.org/10.1145/2398856.2364563`.

[2] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. Data-only flattening for nested data parallelism. *SIGPLAN Not.*, 48(8):81–92, February 2013. ISSN 0362-1340. doi: 10.1145/2517327.2442525. URL `http://doi.acm.org/10.1145/2517327.2442525`.

[3] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8(2):119–134, February 1990. ISSN 0743-7315. doi: 10.1016/0743-7315(90)90087-6. URL `http://dx.doi.org/10.1016/0743-7315(90)90087-6`.

[4] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *SIGPLAN Not.*, 28(7):102–111, July 1993. ISSN 0362-1340. doi: 10.1145/173284.155343. URL `http://doi.acm.org/10.1145/173284.155343`.

[5] L. Bluestein. A linear filtering approach to the computation of discrete fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, December 1970. ISSN 0018-9278. doi: 10.1109/TAU.1970.1162132.

[6] 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.

[7] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 666–675, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. ISBN 0-89791-412-0. URL `http://dl.acm.org/citation.cfm?id=110382.110597`.

[8] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. Streaming Irregular Arrays. In *Haskell '17: The 10th ACM SIGPLAN Symposium on Haskell*, pages 174–185. ACM, September 2017.

[9] Pierre Duhamel and Henk Hollmann. Split radix'fft algorithm. *Electronics letters*, 20(1):14–16, 1984.

[10] Martin Elsman, Troels Henriksen, and Niels Gustav Westphal Serup. Data-parallel flattening by expansion. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, pages 14–24, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6717-2. doi: 10.1145/3315454.3329955. URL `http://doi.acm.org/10.1145/3315454.3329955`.

[11] Matthew Fluet, Nic Ford, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Status report: the manticore project. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 15–24. ACM, 2007.

[12] Clemens Grelck. Single assignment c (sac) high productivity meets high performance: High productivity meets high performance. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, pages 207–278, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-32095-8. doi: 10.1007/978-3-642-32096-5_5. URL `http://dx.doi.org/10.1007/978-3-642-32096-5_5`.

[13] Troels Henriksen, Martin Elsman, and Cosmin E. Oancea. Size slicing: A hybrid approach to size inference in futhark. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '14, pages 31–42, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3040-4. doi: 10.1145/2636228.2636238. URL `http://doi.acm.org/10.1145/2636228.2636238`.

[14] 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. *SIGPLAN Not.*, 52 (6):556–571, June 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062354. URL `http://doi.acm.org/10.1145/3140587.3062354`.

[15] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 53–67, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6225-2. doi: 10.1145/3293883.3295707. URL `http://doi.acm.org/10.1145/3293883.3295707`.

[16] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. Vectorisation avoidance. *SIGPLAN Not.*, 47(12):37–48, September 2012. ISSN 0362-1340. doi: 10.1145/2430532.2364512. URL `http://doi.acm.org/10.1145/2430532.2364512`.

[17] Dietmar Kreye. A compilation scheme for a hierarchy of array types. In Thomas Arts and Markus Mohnen, editors, *Implementation of Functional Languages*, pages 18–35, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-46028-2.

[18] Frederik M. Madsen, Robert Clifton-Everest, Manuel M T Chakravarty, and Gabriele Keller. Functional Array Streams. In *FHPC '15: The 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*, pages 23–34. ACM, September 2015.

[19] 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.

[20] 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.

[21] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[22] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel MT Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462176. URL `http://doi.acm.org/10.1145/2499370.2462176`.

[24] John Reppy and Nora Sandler. Nessie: A NESL to CUDA compiler. Presented at the *Compilers for Parallel Computing Workshop (CPC '15)*, January 2015. Imperial College, London, UK.

[25] Steven W Smith et al. The scientist and engineer's guide to digital signal processing.

[26] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 74–85, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4931-8. URL `http://dl.acm.org/citation.cfm?id=3049832.3049841`.

[27] A. Richard Thompson, James Moran, and George W. Swenson Jr. *Interferometry and Synthesis in Radio Astronomy*. Springer International Publishing, 2017. ISBN 9783319444314. doi: 10.1007/978-3-319-44431-4. URL `https://www.springer.com/gp/book/9783319444291`.

[28] B. Veenboer, M. Petschow, and J. W. Romein. Image-domain gridding on graphics processors. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 545–554, May 2017. doi: 10.1109/IPDPS.2017.68.

[29] R. Yavne. An economical method for calculating the discrete fourier transform. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 115–125, New York, NY, USA, 1968. ACM. doi: 10.1145/1476589.1476610. URL `http://doi.acm.org/10.1145/1476589.1476610`.