UTRECHT UNIVERSITY

FACULTY OF SCIENCE

DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES

ARRAY-ORIENTED LANGUAGES AND POLYHEDRAL COMPILATION

Author
Justin Paston-Cooper

Supervisor
Prof. Dr. G.K. (Gabriele) Keller

Secondary Examiner
Dr. T.L. (Trevor) McDonell

ICA-3730786

June 16, 2020

Abstract

Collection-oriented languages are characterised by their provision of a set of primitives which work on data in aggregate, and are intrinsically suited to parallelisation. Polyhedral compilation systems aim at optimising aggregate computations using an abstract, mathematical representation. However, they are typically run as optimisation passes not over high-level programmes representing aggregate computations, but over the low-level intermediate representation they compile down to. This means that information can be lost if code is not written in the correct form. Accelerate[1] is a collection-oriented language embedded in Haskell. Programmes are compiled into LLVM-IR, that code being passed to the LLVM compiler. Polly[2], a polyhedral optimisation pass of LLVM, aims specifically at optimising programmes typical in Accelerate, namely computations over regular, multi-dimensional arrays. In this project we investigate the strengths of each system in order to see how they can together produce faster code.

Contents

1 Introduction **3**
1.1 Research Questions 4
1.2 Document Structure 4

2 Background **4**
2.1 Polyhedral Compilation 5
2.1.1 Polly 6
2.1.2 Other Polyhedral Compilation Tools 7
2.2 Collection-Oriented Languages 8
2.2.1 Accelerate 8

3 The Potential for Polyhedral Compilation in Accelerate **12**
3.1 Benchmarking Polly 13
3.2 Benchmarking Accelerate 15

4 Polyhedral Compilation of Accelerate Programmes **17**
4.1 Optimising the `2mm` kernel 17
4.2 Optimising Other Kernels 20
4.3 Parallelism, Polly and Accelerate 21
4.4 Larger Examples 23
4.4.1 Salt Marsh Simulation 23
4.4.2 K-Means Clustering 24
4.4.3 Integral Image 25

5 Conclusion, Discussion & Future Work **28**

1 Introduction

The defining characteristic of a collection-oriented language is that it provides a set of primitives which work over collections as a whole. Such languages are thus intrinsically suited to parallelisation by the fact that the unit of work is the whole collection. One major sub-class of collection-oriented languages is array-oriented languages. Perhaps the most commonly known, and one of the oldest, array-oriented languages is APL. APL is known for its vast collection of array primitives with special characters as names. Somewhat more recent are Futhark[3] and Accelerate[1]. These languages provide a set of primitives for working with arrays commonly known from functional programming, such as `map`, `fold` and `scanl`. Using these primitives, an an intuitive representation of array computations is allowed. This is in contrast to other array-based languages which require the user to deal with explicit loops and indexing over arrays, such as SaC[4]. The selling point of languages like Futhark and Accelerate is that while allowing such an intuitive representation of array computations, they compile to highly efficient code involving iteration over regular, multi-dimensional arrays. They also handle parallelisation automatically without user intervention. Such performance is typically only seen in programmes written with high-performance computing frameworks such as OpenCL and OpenMP. However, in such frameworks, the user must handle memory allocation and parallelism explicitly.

Polyhedral compilation is a technique which aims at optimising exactly the kind of computational behaviour which these array-oriented languages exhibit. That is, iterations over regular, multi-dimensional arrays. A polyhedral compiler will take a programme, extract a mathematical, polyhedral representation of it, apply a number of transformations this representation with a view to maximising memory locality, and then map it back into a transformed programme. One algorithm used to optimise memory locality is the Pluto[5] algorithm. Polly[2] is an optional polyhedral optimisation pass of the LLVM[6] compiler. Being part of the LLVM compiler framework, it generally deals with programmes written in C or C++, but is on paper independent of source language. Polly uses the Pluto algorithm internally to optimise the polyhedral representations of programmes which it extracts, while also applying a number loop optimisation heuristics such as tiling. The Tiramisu[7] project is a high-performance computation framework for CPUs and GPUs. Instead of applying polyhedral optimisations automatically, it provides a number of polyhedral transformation primitives to be utilised by the user explicitly. PolyMage[8] is an image-processing language with a polyhedral optimiser directly suited to its domain.

Given the domain of array-oriented languages, it might seem that they are directly suited to the application of polyhedral compilation, where regular array iteration is the focus. Polly, being an optimisation pass of LLVM, works with LLVM-IR, which is the LLVM compiler's intermediate representation format. The Accelerate language also generates LLVM-IR from its programmes, using LLVM to compile the LLVM-IR to a final executable. This establishes the possibility for leveraging polyhedral compilation techniques not from a low-level C-like language, but instead from a high-level array-oriented language, a language whose domain is exactly what Polyhedral compilation techniques aim to optimise.

1.1 Research Questions

In this project we investigate the following:

1. There are limits to the type and form of programme Polly will accept and optimise. We investigate exactly what types of programmes Polly accepts, and the extent of the benefit it provides. We also look into classes of programmes which become slower with the application of Polly.
2. Polly can be sensitive to the form of code it will optimise. Even if two pieces of code are equivalent and very similar, it might optimise only one. We investigate how Accelerate's code generation can be adapted such that it is in a form for which Polly can work with.
3. Following our research on what applications are best suited to polyhedral compilation, we implement in Accelerate a range of applications, and measure the benefit of our changes to Accelerate's code generation.
4. Accelerate applies its own domain-specific optimisations to the code it generates. We investigate the interaction of these optimisations with Polly.

1.2 Document Structure

Following this introduction, we provide a background of polyhedral compilation and collection-oriented languages, looking at Polly and Accelerate in detail. Afterwards, we establish the potential for polyhedral compilation in Accelerate. We first explore the extent of programmes optimised by Polly by running a range of benchmarks. We then use our insight from this, in applying Polly for the first time to a programme in Accelerate. Having shown that Polly can indeed optimise a programme written with Accelerate, we implement a number of differing examples in Accelerate to understand the extent of the benefit of Accelerate and Polly when applied together. We then conclude with a discussion of our results and what they mean for polyhedral compilation applied to collection-oriented languages. We also discuss possible future work.

2 Background

Consider the two code snippets in Figure 1. Both implementations of the dot-product produce the same result, but from different approaches. Accelerate, a collection-oriented language, allows the user to express the dot-product using a number of high-level collection-oriented primitives, applying crucial optimisation techniques such as array fusion to reduce the cost of this expressivity. In C, pattern and dependency recovery is left to the optimisation passes of the compiler. For instance, Polly, an optional optimisation pass of LLVM contains special provisions for matrix multiplication, but they trigger only for code of a certain form. This is dealt with in more detail in Section 2.1.1. This can make it difficult for the user to write optimal code without knowing the internals of the compiler.

In this section we look into both approaches, later seeing how they interact. We first give a description of polyhedral compilation. We later give an overview of collection-oriented languages, and Accelerate in particular. After this in Section 3, we round up both approaches, discussing how

```
dp = fold (+) (constant 0) $ zipWith (*) a b
```

(a) Haskell, Accelerate

```
int dp = 0;

for (i = 0; i < N; ++i) {
  dp += a[i] * b[i];
}
```

(b) C

Figure 1: Two different implementations of the dot-product

collection-oriented languages like Accelerate and polyhedral compilation techniques might work together.

2.1 Polyhedral Compilation

Polyhedral compilation has for decades[9] existed as a method of optimising the memory locality of loop nests whose boundaries and array offsets in statements are affine functions of surrounding loop variables and constant parameters. Affine means linear combination plus an optional constant. Such loop nests are common in array computations, and this includes a lot of the code generated by array-oriented languages such as Accelerate and Futhark. For instance in Accelerate, the collective operations `map` and `zipWith` both compile to affine loop nests over arrays. In the literature, maximimal parts of programmes satisfying the above conditions are called static control parts, or (SCoPs)[10][11].

In loop nests satisfying the above conditions, each statement invocation in a loop can be viewed as a point in a multi-dimensional polyhedron, the coordinates of the point being the corresponding loop variable assignments. A polyhedral compiler will detect the suitable maximal loop nests (SCoPs) in a programme and produce a polyhedral representation of them. It will then apply a number of heuristics to this representation with a view to maximising memory locality while preserving data dependencies. This results in a transformation on the initial polyhedral representation of the programme, which is mapped back into the original programming language, providing a new, hopefully optimised schedule of the original programme statements. See Figure 2 for an example of a SCoP and a visual polyhedral representation of its statement invocations.

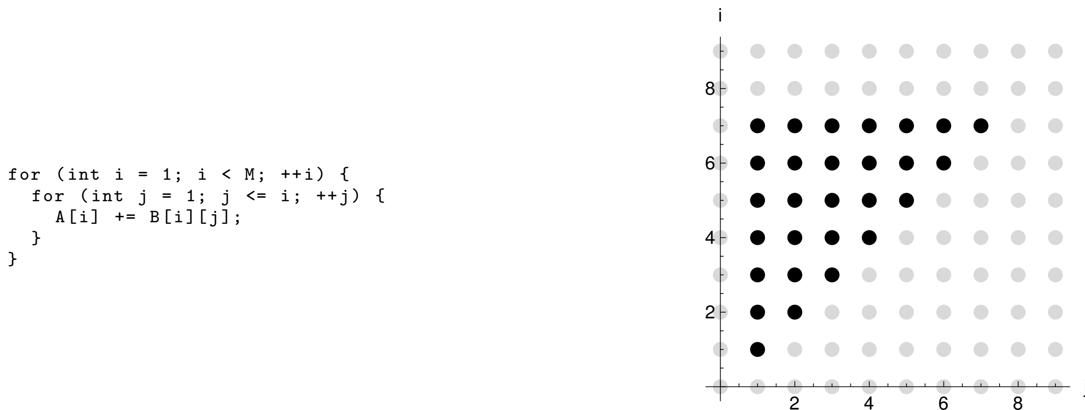


Figure 2: A SCoP and a visual polyhedral representation of its statement invocations.

The set of statement invocations of a SCoP, as illustrated in Figure 2, is known as its iteration domain. Each combination of `i` and `j` for which the single statement in the SCoP executes is denoted by a black dot. The iteration domain may be formally defined by what is called a basic integer set[10]. The basic integer set for a statement encapsulates the affine constraints under which a statement is invoked, and the constant parameters on which it depends. All polyhedrons can be viewed as hyperplanes constrained by a set of affine constraints. As an example, call the single statement in the code in Figure 2 `Statement`. This statement depends on the parameters `i` and `j`. The basic integer set S for `Statement` would be:

$$S = [M] \rightarrow \{\texttt{Statement}[i, j] | 1 \leq i \wedge i < M \wedge 1 \leq j \wedge j \leq i\}$$

In this basic integer set, we see the plane defined by all instances of (i, j) constrained by three affine constraints: $1 \leq i < M$, $1 \leq j$ and $j \leq i$. These three affine constraints applied to the plane form a triangle.

2.1.1 Polly

Polly is a polyhedral optimisation tool which is included as an optional component of the LLVM compiler framework. Polly is implemented as an optimisation pass of LLVM, working on LLVM-IR, the intermediate representation language of LLVM. The fact that Polly works as an optimisation pass on LLVM-IR allows it to work with any front-end language which LLVM supports.

Polly runs in a multi-step process. It takes the input LLVM-IR code and first detects the SCoPs in it. It then produces a formal polyhedral representation of the detected SCoPs, which it passes to the Integer Set Library (isl)[12]. isl deals natively with integer sets as previously defined, and the transformations upon them. It is the backend of Polly, and it is in isl that programme schedule optimisation occurs. One of the optimisation methods which isl can apply to a programme schedule is the Pluto[5] algorithm. The Pluto algorithm translates the polyhedral representation of a SCoP in isl into a linear programme, whose objective function aims to minimise the time between writes and reads of the same data. The solution of the linear programme is the optimised schedule.

Outside of the optimisations carried out by isl, Polly can also apply a number of heuristics itself, a notable one of which is tiling, and is detailed below. These heuristics all work upon the polyhedral representation of programmes in isl. Following optimisation, the final step is extracting the schedule from isl and producing the corresponding, transformed LLVM-IR.

We now look at an example of loop tiling, one of the transformations which Polly commonly applies to code. Loop tiling is also known in the literature as loop blocking[10]. Loop tiling aims at increasing memory locality by executing a loop tile-wise. See Figure 3 the tiling of code for the addition of two `N` by `N` matrices `B` and `C`.

The iteration behaviour of the two versions of matrix multiplication can be visualised as in Figure 4. Each dot corresponds to a single execution of the addition statement. The set of all dots corresponds to the iteration domain, which in this case is square. In the untiled case, we start at the bottom, executing the statements in each row left to right. In the tiled example we see four separate blocks, each denoted by a square surrounding a number of dots. To execute the statements in the tiled manner, we start at block 1, executing each statement contained in it from left to right, bottom to top. Once

```
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    A[i][j] = B[i][j] + C[i][j];
  }
}
```

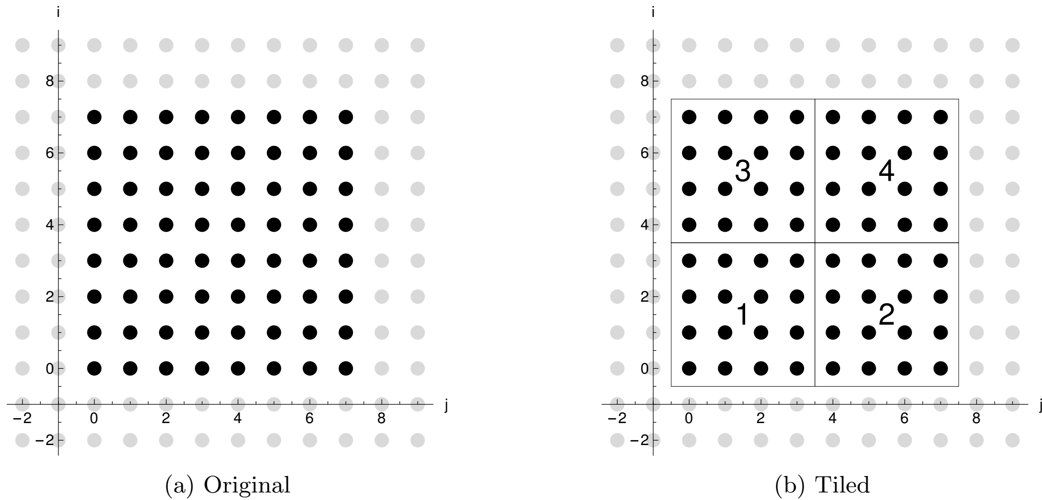
(a) Original

```
for (int ii = 0; i < N; ii += 4) {
  for (int jj = 0; jj < N; jj += 4) {
    for (int i = ii; i < min(N, ii + 4); ++i) {
      for (int j = jj; j < min(N, jj + 4); ++j) {
        A[i][j] = B[i][j] + C[i][j];
      }
    }
  }
}
```

(b) Tiled

Figure 3: Applying tiling to code for matrix addition

finished, we move onto block 2 and execute the statements in the same order, then move onto the remaining two blocks.



(a) Original

(b) Tiled

Figure 4: Tiling matrix addition visually

Polly also has provisions for detecting and optimising matrix multiplication, as laid out in [13]. The matrix multiplication optimisation again uses the polyhedral representation of a programme in isl to both detect matrix multiplication and optimise it. Even though this optimisation works on an abstract polyhedral representation, we have found that it is sensitive to code form. In Figure 5 we see code that it correctly detects as matrix multiplication, and code which it doesn't.

2.1.2 Other Polyhedral Compilation Tools

This is not the first project to consider applying polyhedral compilation techniques to array-oriented languages. Tiramisu[7] is a functional, array-oriented language which instead of running an automated polyhedral optimiser on programmes, provides a set of primitives to apply polyhedral transformations manually, such as loop tiling, splitting and shifting. In the cited paper, it is claimed that automated polyhedral optimisers such as Polly are unable to match the performance of hand-tuned code for examples such as generalised matrix multiplication ($C = \alpha AB + \beta C$). For a high-level, implicitly parallelising language like Accelerate, this is not a concern. We are interested in speed-ups which

```
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    A[i][j] = 0;

    for (int k = 0; k < N; ++k) {
      A[i][j] += B[i][k] * C[k][j];
    }
  }
}
```

(a) Passes matrix multiplication detection

```
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    int sum = 0;

    for (int k = 0; k < N; ++k) {
      sum += B[i][k] * C[k][j];
    }

    A[i][j] = sum;
  }
}
```

(b) Fails matrix multiplication detection

Figure 5: Detecting matrix multiplication

Accelerate can realise without user intervention. This is one of the motivations behind choosing Polly.

PolyMage[8] is an image-processing domain-specific language with a polyhedral optimiser directly suited to its domain. The framework boasts impressive results compared to hand-tuned Halide code over a number of image-processing examples.

2.2 Collection-Oriented Languages

Collection-oriented languages define a set of primitives which work over collections of data as a whole, avoiding explicit looping. Programmes in such languages are intrinsically suited to parallelisation. One class of collection-oriented languages is array-oriented languages, the most notable example of which is probably APL, known for its vast collection of array primitives with special characters as names. Descended from APL is J, with more advanced mechanisms for writing point-free code, and ASCII-based primitive names. Programmes written in these systems are not compiled and run, but are interpreted. Moving away from the APL lineage, we have the Accelerate framework, providing a functional language which also falls into the array-oriented class. In Accelerate, programmes are formed from a number of primitives commonly known from functional programming, such as `map`, `fold` and `scanl`. Programmes in Accelerate are not interpreted directly, but are compiled to LLVM-IR, leaving the rest of the work to the LLVM compiler to create a runnable binary. Futhark[3] is a similar framework, but programmes are compiled into either CUDA or OpenCL code.

2.2.1 Accelerate

In Accelerate, the fundamental data-type is the dense, regular, multi-dimensional array. Such arrays are efficiently represented in a flat way in memory. Accelerate defines a set of primitives which work over these multi-dimensional arrays as a whole. A subset of these primitives is listed in Listing 1.

```haskell
-- Generate an array of a given shape given a function over indices
generate :: (Shape sh, Elt a)
         => Exp sh
         -> (Exp sh -> Exp a)
         -> Acc (Array sh a)

-- Apply a function to each element of an array
map :: (Shape sh, Elt a, Elt b)
    => (Exp a -> Exp b)
    -> Acc (Array sh a)
    -> Acc (Array sh b)

-- Combine two arrays with a function
zipWith :: (Shape sh, Elt a, Elt b, Elt c)
        => (Exp a -> Exp b -> Exp c)
        -> Acc (Array sh a)
        -> Acc (Array sh b)
        -> Acc (Array sh c)

-- Reduce an array with a function over its innermost dimension
fold :: (Shape sh, Elt a)
     => (Exp a -> Exp a -> Exp a)
     -> Exp a
     -> Acc (Array (sh:.Int) a)
     -> Acc (Array sh a)

-- Map over each value and its neighbourhood
stencil :: (Stencil sh a stencil, Elt b)
        => (stencil -> Exp b)
        -> Boundary (Array sh a)
        -> Acc (Array sh a)
        -> Acc (Array sh b)

-- Left-to-right scan along the innermost dimension of an arbitrary rank array
scanl :: (Shape sh, Elt a)
      => (Exp a -> Exp a -> Exp a)
      -> Exp a
      -> Acc (Array (sh:.Int) a)
      -> Acc (Array (sh:.Int) a)

-- Matrix transpose on a two-dimensional array
transpose :: Elt e
          => Acc (Array DIM2 e)
          -> Acc (Array DIM2 e)
```

Listing 1: A subset of the primitives defined in Accelerate

Arrays in Accelerate are represented by the `Array` type in Haskell. This type is parameterised by two types: dimensionality and type of data contained. Accelerate represents at the type level whether an array has two, three or more dimensions, but it does not represent the actual array size in the type. The class of possible dimensionalities is represented by the type-class `Shape` in Haskell. Examples of this class' use can be seen in Listing 1. The inhabitants of each dimensionality are known as shapes. See below for an example of a two-dimensional shape.

```haskell
type DIM2 = (Z :. Int) :. Int -- DIM2, an instance of the Shape type-class

((Z :. 10) :. 10) :: DIM2 -- A two-dimensional example of a shape.
```

Listing 2: A two-dimensional shape

Shapes are not only used to define an array's size, but also as an index into an array. A multi-dimensional array must have a multi-dimensional index with matching dimensionality. This is guaranteed by the type system at compile time.

We now explain the behaviour of the primitives in Listing 1.

`generate` Generate an array of a chosen shape, given a function which takes an index into the array and constructs a value. The result is constructed by calling the given function on every index into the result.

`map` Apply a given function to every element in an array, constructing a new array.

`zipWith` Given a combination function and two arrays of equal dimensionality, this function constructs a new array by applying the combination function to corresponding pairs of values from the two input arrays. The input arrays are generally expected to be of the same shape.

`fold` Given a reduction function and a zero element, this function reduces an array along its inner-most dimension. This produces a result with dimension one less than the input. For a two-dimensional matrix, `fold` will reduce each row into a single value, resulting in a one-dimensional vector.

`stencil` This function is similar to the `map` function, but instead of mapping over each value of the input, each value plus a neighbourhood of a chosen size is mapped over. The `Boundary` argument determines what values are given to the function when the neighbourhood extends over the boundaries of the input array.

`scanl` Computes the cumulative sum of an array along the inner dimension given an associative operation and a starting value.

`transpose` Computes the matrix transpose over a two-dimensional array, swapping its two dimensions.

Accelerate separates aggregate operations and single-value operations into two classes at the type level. Types surrounded in `Acc` denote the results of aggregate operations, and types surrounded in `Exp` denote the results of single-value operations. This prevents irregular nested data-parallelism. For instance, one cannot call a `map` or `fold` inside the argument function to another `map`, as the argument function's result must be an `Exp`, but the result of a `map` or `fold` is an `Acc`. Accelerate is thus limited to flat data-parallelism over regular, multi-dimensional arrays.

In Figure 1, we saw a dot-product implementation both in Accelerate and in C. The operations in collection-oriented languages work over data as a whole. A naive way to compile the Accelerate version would be to have code which carries out the `zipWith`, and code which carries out the `fold` separately. The equivalent of this in C would be as in Figure 6b.

This is a sub-optimal solution when compared to the C implementation in Figure 1. Not only do we need more memory for the intermediate array `p`, we now have two array traversals instead of one. In order for a collection-oriented language like Accelerate to compete with C for memory usage and run-times, it must implement a type of optimisation known as array fusion. This involves, when possible, combining the computations from multiple successive array operations into a single array traversal. With fusion, the two separate loops become one, just like in the original C implementation. To apply fusion, Accelerate separates primitives into two classes:

```
dp = fold (+) (constant 0) $ zipWith (*) a b
```

(a) Dot-product in Accelerate

```
int dp = 0;
int p[N];

for (i = 0; i < N; ++i) {
  p[i] = a[i] * b[i];
}

for (i = 0; i < N; ++i) {
  dp += p[i];
}
```

(b) Accelerate dot-product naively compiled to C

Figure 6: Implementations of dot-product

1. *Producer*: Each output element depends on at most one element of each input array.

2. *Consumer*: Each element depends on multiple elements of each input array.

Out of the primitives listed in Listing 1, `generate`, `map` and `zipWith` are producers. The primitives `fold` and `stencil` are consumers.

Fusion is applied to a programme's AST in two successive phases:

1. *Producer/producer*: Sequences of adjacent producers are fused into a single producer.

2. *Producer/consumer*: Annotates AST as to which producers should be computed separately, and which should be fused into their consumers. This phase is ultimately completed during code generation, where the consumer skeleton is specialised by embedding the code of the fused producer.

We now look at how fusion applies to the dot-product example. In the example, the only producer is `zipWith`, so there is no producer-producer fusion. The producer `zipWith` is fused with the consumer `fold`. When this fused unit is compiled, it results in a fused dot-product computation, analogous to the C implementation in 1, but with provisions for running in parallel which we will elaborate on below. This fusion process greatly reduces computation time and total memory needed for array computations, while maintaining the ease of defining computations.

Accelerate can produce code to run both on CPUs and on GPUs. In this project, we focus only on running programmes on CPUs in parallel on multiple cores. In either case, each separate fused unit of an Accelerate programme is compiled into one or more functions in LLVM's intermediate representation language called LLVM-IR. LLVM optimises the code further and compiles it into object code for the respective computation unit, either the CPU or GPU. In this context, a compiled object is called a kernel.

In Accelerate, each primitive is associated with a number of skeletons which define the code to carry out its operation on an array. Fused units, which are formed of a chain or producers with a consumer at the end, are compiled by taking the skeleton of the consumer, and including code to compute the producer inside that skeleton. Our dot-product example is compiled according to the skeleton for the `fold` operation, the `fold` being the outer consumer.

The operation `fold` in Accelerate can compile in different ways based on the dimensionality of the input. In the dot-product case, the whole of the input to `fold` is reduced to a single element.

This contrasts with taking an array of dimensionality above 1 and reducing along its inner dimension, leaving potentially many values in the result.

The `fold` operation in this example is carried out by two separate skeletons, which each carry out a separate pass in order to compute the result. The first skeleton carries out the bulk of the computation. The skeletons are first instantiated with the code for the producer, namely `zipWith`, and are then compiled into two separate kernels. Equivalent C code to what the first skeleton would compile to is shown in Listing 3. We now come to scheduling, dividing and executing the work. The input is split into as many portions as there are available worker threads. A temporary array is allocated of size equal to the number of portions. Each worker then runs the first kernel on its allocated portion in parallel with the other workers. This first kernel carries out the dot-product operation on its allocated portion, and writes the result to the temporary array at the position corresponding to the portion assigned to it. A single worker runs the kernel for the second pass. This kernel iterates over the temporary array, and reduces the partial results into a single final result.

```
void fold_pass_1 (int ix_start0, int ix_end0, // Begin and end index of portion to be worked on
                  int portion,                // Id of portion to be worked on
                  int * tmp,                  // Temporary array where result will be stored
                  int * a, int a_sh0,         // First input array, passed in with size
                  int * b, int b_sh0) {       // Second input array, passed in with size
  int result = 0;                             // Beginning of the reduction

  for (int i = ix_start0; i < ix_end0; ++i) {
    result += a[i] * b[i];                    // The code for the zipWith has been inserted here
  }

  tmp[portion] = result;                      // Write the partial result for this portion
}
```

Listing 3: Equivalent C code for the kernel of the first pass needed to compute the dot-product

3 The Potential for Polyhedral Compilation in Accelerate

Polyhedral compilation techniques aim to optimise programmes that contain loops whose bounds and array access offsets are affine combinations of surrounding loop variables and constant parameters. In other words, programmes formed of regular iterations over regular, multi-dimensional arrays. On the face of it, array-oriented languages such as APL, Accelerate and Futhark should seem directly suited to polyehdral compilation. In practice, we have found that it can be difficult to get collection-oriented languages and polyhedral compilation to work together.

In this project we look at integrating Polly, a polyhedral optimisation pass of LLVM, with Accelerate, which compiles programmes into LLVM-IR. Polly recovers the polyhedral representation of a programme from LLVM-IR, this LLVM-IR generally being generated from a language like C. We have found from experimentation that it is difficult to know exactly what type of code Polly expects in order to perform its optimisations. This was true starting both from C, and later from LLVM-IR. However, once the right form of code is known, Polly can provide impressive speed-ups for certain applications, for instance with its custom code for matrix multiplication. This shows the potential of collection-oriented languages like Accelerate to use polyhedral compilation. Programming array operations in a

low-level language like C, the user must know the ins and outs of Polly in order to use Polly to its full potential. In contrast, this knowledge can be programmed into a collection-oriented language like Accelerate, and can be made available to the user implicitly, just like Accelerate's implicit parallelism.

In the rest of this section we justify our claims towards the potential for integrating Polly and Accelerate with benchmarks. We first benchmark Polly over a wide range of applications in order to understand what conditions Polly provides the most benefit under. We then implement a number of these applications in Accelerate and investigate how the benefit of Polly carries over to the code generated by Accelerate.

All benchmarks were run on a computer with the specifications in Table 1.

CPU	MD ThreadRipper2950X (16cores @3.5GHz), zenv1 architecture
Cache	L1d:512KiB, L1i:1MiB, L2:8MiB, L3:32MiB
GPU	NVidia RTX2080, Turing architecture
RAM	64GB DDR4,2833MHz
Storage	Two 8TB HDDs and two 1TB NVME drives
OS	GNU/Linux, distribution Ubuntu19.10Eoan

Table 1: Specifications of computer benchmarks were run on

3.1 Benchmarking Polly

In order to understand the potential benefit Polly can provide to Accelerate programme run-times, Polly must be tested over a wide range of programmes which might be implemented in Accelerate. To this end, we have benchmarked Polly with the Polybench[14] suite of algorithms. Polybench defines a number of algorithms over regular, multi-dimensional arrays, ranging over multiple domains and memory access patterns. While Polly supports the generation of parallel code, all benchmarks were run single-threaded. This is in order to find a baseline for the speed-up Polly can provide to work which Accelerate assigns to separate threads.

We have compiled each algorithm of Polybench with `clang`, optimisation level `-O3`, comparing the speed-up Polly provides when enabled to when it is disabled. When compiling with Polly, we also set `-polly-position=early` as an argument, causing Polly to run earlier in the optimisation chain. This has the effect of increasing compilation times, but was found to lead to the fastest code across all Polybench examples. Without this, Polly's affine array subscript detection can fail, especially when C99 variable-length arrays are used. The `-polly-position=early` setting does cause Polly to run before inlining, but this is not an issue, as we wish to measure the speed of the kernels alone.

Polybench allows the user to specify that each kernel takes as arguments arrays either with hard-coded sizes, or with C99 variable lengths. We split our benchmarks into two groups according to this choice. Both choices are illustrated in the benchmarks in Figure 7, the left yellow bar of each pair corresponding to C99 variable-length arrays, the right blue bar corresponding to hard-coded array sizes.

We found Polly to provide a speed-up to a number of kernels in the Polybench suite. The most notable speed-ups were for `2mm` and `3mm`, which involve matrix multiplication. As previously mentioned, Polly contains specialised code for recognising and optimising programmes containing matrix multiplication, which was invoked in this case. We discuss later the difficulty of generating the right

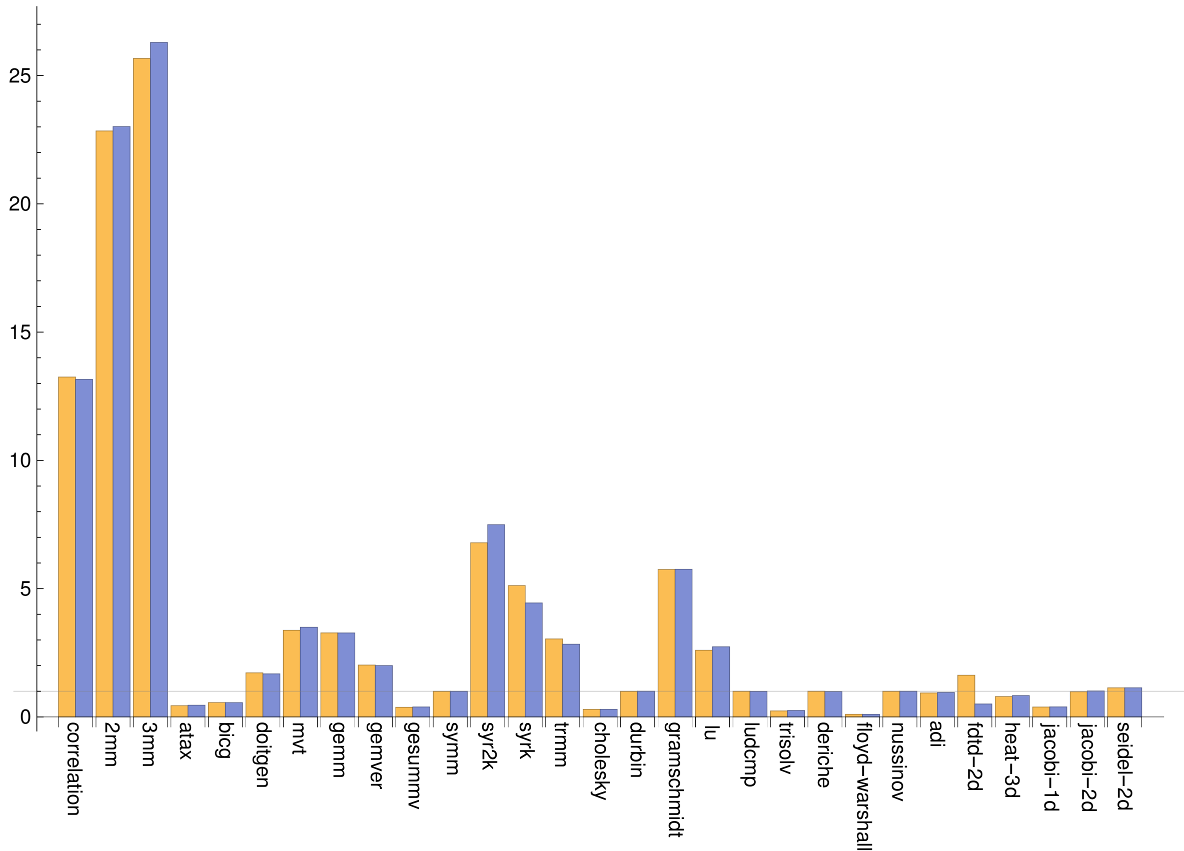


Figure 7: Relative speed-up with Polly enabled to `-O3` optimisation alone. At horizontal grey bar means no speed-up, above means faster. The yellow bar on the left of each pair is speed-up with C99 variable-length arrays, the blue bar on the right is with hard-coded array sizes.

code for this to be invoked. For the remaining kernels where a speed-up was observed, the most notable transformation Polly was found to have applied was tiling. No significant difference of run-times was found between the C99 variable-length versions and hard-coded versions, apart from in the `fdtd-2d` example. The C99 version showed a slight speed-up, but the hard-coded array sizes version showed a slow-down. Polly generated completely different code for each case. It is not immediately clear why.

Looking closely at Polly's benchmark results over the Polybench examples in Figure 7, we see a large class for which either a near-zero speed-up is provided, or in the case of `heat-3d` and `jacobi-1d`, a considerable slow-down. All of these examples fall under the `stencils` directory in Polybench. Stencils are an important class of computation in scientific computing and machine learning.

Stencil computations are characterised by the fact that, for multi-dimensional input and output arrays, an element in the output is computed from an element from the input and a small number of elements surrounding it. This characteristic makes stencil computations amenable to parallelisation and locality optimisations. One such locality optimisation is tiling, which as detailed in Section 2.1.1. Polly applies tiling to most examples, including the stencil examples, and Polly allows the user set the tile size at each loop nesting level. Determining the optimal tiling size for a programme is a computationally intensive task, and there is no single optimal tile size for all examples[15]. Polly does

```
for (t = 1; t <= TSTEPS; t++) {
    for (i = 1; i < N - 1; i++) {
        for (j = 1; j < N - 1; j++) {
            for (k = 1; k < N - 1; k++) {
                B[i][j][k] =   0.125 * (A[i+1][j][k] - 2.0 * A[i][j][k] + A[i-1][j][k])
                             + 0.125 * (A[i][j+1][k] - 2.0 * A[i][j][k] + A[i][j-1][k])
                             + 0.125 * (A[i][j][k+1] - 2.0 * A[i][j][k] + A[i][j][k-1])
                             + A[i][j][k];
            }
        }
    }
    for (i = 1; i < N - 1; i++) {
       for (j = 1; j < N - 1; j++) {
           for (k = 1; k < N - 1; k++) {
               A[i][j][k] =   0.125 * (B[i+1][j][k] - 2.0 * B[i][j][k] + B[i-1][j][k])
                            + 0.125 * (B[i][j+1][k] - 2.0 * B[i][j][k] + B[i][j-1][k])
                            + 0.125 * (B[i][j][k+1] - 2.0 * B[i][j][k] + B[i][j][k-1])
                            + B[i][j][k];
           }
       }
   }
}
```

Listing 4: `heat-3d` implementation

not do anything towards determining the optimal tile size to apply to a given example. If it does apply tiling, which it does in most examples, it uses a default tile size of 32. That is, each iteration variable is iterated in blocks of 32. This turned out to be the wrong choice for the stencil examples, as seen in the benchmark results in Figure 7.

The `heat-3d` example in Polybench simulates heat propagation in a three-dimensional environment. See Listing 4 for its implementation. The `heat-3d` example actually involves the iterated application of two separate stencil computations.

Polly by default fuses these the two stencil loops and applies a tile size of 32 on each loop. As seen in the benchmark results, this causes a slow-down. The only way we found to get a speed-up with Polly was to prevent fusion by inserting an `asm volatile("")` between the two stencil loops, and to set the tile sizes of each loop manually with `-polly-tile-sizes=24,4,1024`. At this point not much is left for Polly to do but apply the chosen tile sizes. The story is the same for `jacobi-1d` and `jacobi-2d`. Fusion plus a default tile size of 32 cause either a slow-down or no change in speed. One case where this strategy seems to work is `fdtd-2d` in the `C99` case. The hard-coded array size case differs considerably. In that case, only loop tiling is applied.

The above results serve towards an initial investigation into the extent of the benefits Polly can provide. In Section we use these results in deciding where to start in investigating which type of programme, when implemented in Accelerate, can be optimised by Polly.

3.2 Benchmarking Accelerate

In Section 3.1 we conducted an initial investigation into the type of code which Polly can provide a benefit to by benchmarking it against a number of examples written in C. We now move the focus to Accelerate, investigating the extent to which Polly can optimise code written in Accelerate. We also look at what changes can be made when Polly cannot optimise code. Out of the Polybench kernels benchmarked, we have chosen to first look at the `2mm` kernel, as this was one of the kernels with the

```
// First loop
for (i = 0; i < NI; i++)
  for (j = 0; j < NJ; j++)
    {
      tmp[i][j] = SCALAR_VAL(0.0);
      for (k = 0; k < NK; ++k)
        tmp[i][j] += alpha * A[i][k] * B[k][j];
    }
// Second loop
for (i = 0; i < NI; i++)
  for (j = 0; j < NL; j++)
    {
      D[i][j] *= beta;
      for (k = 0; k < NJ; ++k)
        D[i][j] += tmp[i][k] * C[k][j];
    }
```

Listing 5: Polybench C implementation of `2mm`

```
tmp a b =
  fold (+) (constant 0) $
    generate (Z_ ::. ni ::. nj ::. nk) $
      λ(_ ::. i ::. j ::. k) →
        let partA = a ! (Z_ ::. i ::. k)
            partB = b ! (Z_ ::. k ::. j)

        in constant alpha * partA * partB

  where (Z_ ::. ni ::. nk) = shape a
        (Z_ ::. _  ::. nj) = shape b
```

Listing 6: Accelerate implementation of first loop of `2mm`

highest speed-up when Polly was enabled. The Polybench C implementation of the `2mm` kernel is shown in Listing 5.

The kernel takes four two-dimensional matrices $\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}$ as input, and sets `D` to $\texttt{alpha}\cdot\texttt{A}\cdot\texttt{B}\cdot\texttt{C}+\texttt{beta}\cdot\texttt{D}$, where `alpha` and `beta` are constants. In Polybench this is implemented with two separate nested loops, where the first loop sets a temporary buffer `tmp` to $\texttt{alpha} \cdot \texttt{A} \cdot \texttt{B}$, and the second loop sets `D` to $\texttt{beta} \cdot \texttt{D} + \texttt{tmp} \cdot \texttt{C}$.

The mean run-time of the Polybench implementation of the `2mm` kernel in C over the `EXTRALARGE` problem size was 43 seconds with Polly disabled, and 1.83 seconds with Polly enabled. We did not observe any difference in run-times between the C99 variable-length array version and the hard-coded array sizes version.

For the Accelerate implementation, we have followed this looping style as closely as possible in order that the code generated by Accelerate is close to the code generated for the C implementation. This was done using the `generate` combinator. The Accelerate implementation of the first loop is shown in Listing 6. A more idiomatic implementation of the `2mm` example follows later on after we first examine the more C-like implementation.

We now explain the semantics of the Accelerate implementation of this loop. If matrix `a` is of size $\texttt{ni} \times \texttt{nk}$ and matrix `b` is of size $\texttt{nk} \times \texttt{nj}$, the `generate` statement represents a three-dimensional matrix `g` of size $\texttt{ni} \times \texttt{nj} \times \texttt{nk}$, where each element $\texttt{g}_{i,j,k} = \texttt{alpha} \cdot \texttt{a}_{i,k} \cdot \texttt{b}_{k,j}$. The `fold` statement then takes this `g` and sums it along the inner-most dimension, indexed by k. This results in the matrix `tmp` defined by:

$$\mathtt{tmp}_{i,j} = \sum_{k=1}^{\mathtt{nk}} \mathtt{g}_{i,j,k}$$

Just like in the Polybench benchmarks, we ran the Accelerate implementation of `2mm` with a single thread. The kernel took a mean average of 79.73 seconds to run on the same `EXTRALARGE` problem size. Comparing with the run-times for the C implementation, this indicates that there is a large potential for optimisation by Polly.

4 Polyhedral Compilation of Accelerate Programmes

Now that we have shown the potential for the speed-up that might be provided by applying Polly to Accelerate programmes, we now investigate in detail how this might be done. Accelerate compiles programmes into LLVM-IR, which is then compiled and optimised further by the LLVM compiler. The resulting objects are then loaded into memory and executed within the Accelerate runtime. Accelerate programmes therefore have the potential to be optimised by Polly, which itself is implemented as an optimisation pass of LLVM. Polly, however, can provide benefit only to code of a certain form, so we must make sure that Accelerate generates suitable code. A more detailed explanation of the compilation process in Accelerate can be found in Section 2.2.1.

4.1 Optimising the `2mm` kernel

To investigate the ability of Polly to optimise code generated by Accelerate, we initially focus on the first loop of the `2mm` kernel alone. The code for this is listed in Section 3.2. This implementation takes a mean average of 34.26 seconds to run with a single thread for the `EXTRALARGE` problem size. Given that the C implementation of the whole of `2mm` runs in under 2 seconds with Polly enabled, we see that the potential for optimisation remains.

The `generate` statement, when compiled alone, results in three regular nested loops, the loop variables being used directly to index into the input and output arrays. Following the fusion process described in Section 2.2, when compiled as part of the larger programme, the `generate` statement is fused with the `fold` statement. With this, no intermediate matrix G is produced.

Following our tests, we found that Polly is not able to optimise the resulting code. This is due to the implementation of `fold`, which is not implemented with a nested loop for each of the output dimensions, but with one flat loop for the outer-most dimensions, and a loop for the inner-most dimension which is folded over. The `generate` statement is defined over multiple dimensions, so the indices into these dimensions must be recovered from the flat loop variable. The pseudocode for the code generated by Accelerate for the first loop is shown in Listing 7.

We now discuss why it is not possible for Polly to optimise this code. For Polly to optimise code, all array indices must be affine combinations of loop variables and constant parameters. In this code, there are three variables used as array indices: `a_index`, `b_index` and `out_index`. In `a_offset`, we have `out_sh1_index`, which is the result of a division operation over a loop variable. Because of this, `a_offset` is not an affine index. The situation is the same in `b_offset`: `out_sh0_index` is the result

```
void kernel (int out_sh1, int out_sh0, double * out,
             int a_sh1  , int a_sh0  , double * a,
             int b_bh1  , int b_sh0  , double * b) {
  int out_size = out_sh1 * out_sh0;

  for (int out_index = 0; out_index < out_size; ++out_index) {
    int out_sh1_index = out_index / out_sh0;
    int out_sh0_index = out_index % out_sh0;

    int out_element = 0;

    for (int k = 0; k < a_sh0; ++k) {
      int a_index = (out_sh1_index * a_sh0) + k;
      int b_index = (k * b_sh0) + out_sh0_index;

      int generate_result = a[a_index] * b[b_index];

      out_element += generate_result;
    }

    out[out_index] = out_element;
  }
}
```

Listing 7: Pseudocode for generated code of the first loop of 2mm

of a remainder operation over a loop variable. It is therefore not affine. The array index `out_index` is affine, as it is itself a loop variable, however this does not change the situation.

We have solved this issue by replacing Accelerate's flat loop implementation of `fold` with an implementation generating a nested loop for each dimension of the output, thus producing code suitable for optimisation by Polly. After this change, even with Polly turned off, LLVM was able to optimise the generated code for the first loop of the `2mm` kernel much better, running in 20.93 seconds. With Polly turned on, we obtained a run-time of 3.54 seconds, down from 34.26. The pseudo-code for this is shown in Listing 8.

Note that the array indices `a_index` and `b_index` are now affine, and that `out_index` also has a new definition. In the definition of `a_index`, the loop variable `out_sh1_index` is multiplied by `a_sh0`, which itself is a constant parameter. The array index is therefore affine. The remaining array indices are defined similarly.

We now focus on the whole of `2mm`. We have already looked at the implementation of the first loop. This leaves the second loop, which we separate into two stages. Firstly, the result of the first loop is multiplied with matrix `C`. This matrix multiplication is implemented in the same way as the first loop, and is now easily optimised by Polly. The second stage involves adding the result of the matrix multiplication to `beta` $\cdot D$, also optimised by Polly as is. We obtain a mean run-time of 9.12 seconds, down from 79.73.

Having shown in Section 3.2 the ability of Polly to optimise the C-like explicit indexing implementation of `2mm` after making the change to the code generation of `fold`, we were interested in seeing whether the benefits carry over to a more idiomatic implementation. This implementation is shown in Listing 9.

Our idiomatic implementation of `2mm` ran in 133 seconds, up from the 79.73 for the C-like version. With our change to the code generation of `fold` alone, the run-time went down to 43 seconds. With Polly enabled, the run-time went down even further to 8.2 seconds, in fact beating the run-time of the

```
void kernel (int out_sh1, int out_sh0, double * out,
             int a_sh1  , int a_sh0  , double * a,
             int b_bh1  , int b_sh0  , double * b) {
  for (int out_sh1_index = 0; out_sh1_index < out_sh1; ++out_sh1_index) {
    for (int out_sh0_index = 0; out_sh0_index < out_sh0; ++out_sh0_index) {
      int out_element = 0;

      for (int k = 0; k < a_sh0; ++k) {
        int a_index = (out_sh1_index * a_sh0) + k;
        int b_index = (k * b_sh0) + out_sh0_index;

        int generate_result = a[a_index] * b[b_index];

        out_element += generate_result;
      }

      int out_index = (out_sh1_index * out_sh0) + out_sh0_index;

      out[out_index] = out_element;
    }
  }
}
```

Listing 8: Pseudocode for the first loop of 2mm after change to `fold` loop behaviour

```
matrixMultiply a b =
  A.fold (+) (constant 0) $ A.zipWith (*) aRep bRep

  where (_ ::. i ::. j) = shape a
        (_ ::. _ ::. k) = shape b

        cAll = constant All

        aRep = A.replicate (Z_ ::. cAll ::. k    ::. cAll) a

        bRep = A.replicate (Z_ ::. i    ::. cAll ::. cAll) $ A.transpose b

kernel2mm a b c d =
  A.zipWith (+) betaD matrixMultiplyPart

  where betaD = A.map (* constant beta) d

        tmp = A.map (* constant alpha) $ matrixMultiply a b

        matrixMultiplyPart = matrixMultiply tmp c
```

Listing 9: Idiomatic implementation of `2mm` in Accelerate

first implementation.

Neither our initial nor the idiomatic implementation of `2mm` in Accelerate managed to match the speed of the Polybench C implementation, which ran at 1.83 seconds with Polly. Polly has special provisions for matrix multiplication which are triggered by the Polybench implementation, but is not triggered by the code generated by Accelerate. We compared the code generated by Accelerate with the code generated by the Polybench implementation. The code generated by Accelerate was not far off.

Accelerate divides work by assigning to each thread a portion of the output array to compute. A portion is indicated by a start and end offset for each dimension of the ouput array. The start and end offsets into the output array for each instance of a kernel are set by passing them in as arguments. This is true even when running a single thread. Polly's matrix multiplication detection code expects the loops to begin at zero, not at some start offset passed in as an argument. This causes the detection to fail.

Even though Accelerate does not generate suitable code for the matrix multiplication detector, it is not the easiest to write suitable code in C either. The code only works if we accumulate the dot product into the result matrix itself. It does not work if we accumulate into a separate variable and set that as the result at the end of the loop.

It would be possible to change Accelerate so that it generated loops starting from zero by adding the start offsets of arrays into array addresses appropriately. However, this would provide a benefit only to matrix multiplication, and it is unclear how it would affect the run-times of other programmes.

4.2 Optimising Other Kernels

To show that the benefits of Polly combined with our change to `fold` were not restricted to a single example, we have run tests on a further three kernels from the Polybench suite where large speed-ups were observed: `syr2k`, `correlation` and `gemm`. Not all kernels in the Polybench suite are suited to idiomatic implementation in Accelerate. For instance, the `lu` kernel, which performs an LU decomposition, require complex loop bounds to separately compute the lower and upper triangles of the result, which cannot be expressed by the usual Accelerate primitives which work over arrays as a whole. This is not to say that the `lu` kernel cannot be implemented in Accelerate. We are simply interested in idiomatic implementations in order to find out how well polyhedral optimisation can be applied to the code generated by Accelerate's primitives. Also note that all implementations required the usage of the `fold` primitive.

We ran benchmarks over all combinations of the two following variables:

1. Polly enabled or disabled
2. Our change applied to `fold` or not

We first consider the cases where our change code generation of `fold` was not applied. For all kernels, the run-time was the same whether Polly was enabled or disabled. This was expected, as Polly cannot optimise calls to the unchanged version of `fold`, which form the bulk of the work in these kernels.

We now consider the contribution of our change to `fold` to the run-times of the kernels in question. Even with Polly disabled, the run-times of all kernels showed a significant speed-up with the change to `fold`. With Polly enabled, we observed a even higher speed-up for all kernels again. The relative speed-ups are illustrated in Figure 8.

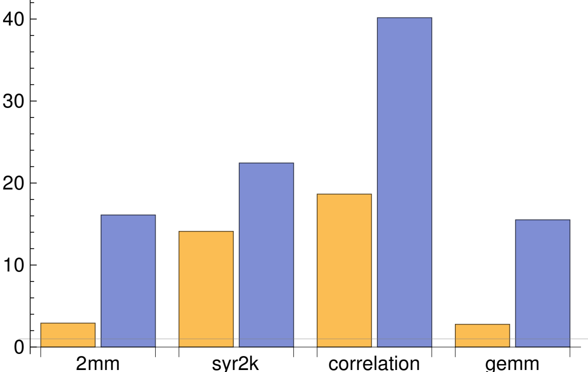


Figure 8: Relative speed-ups of four polybench kernels with change to `fold` relative to unchanged Accelerate code. At horizontal grey bar means no speed-up, above means faster. Yellow bar on left of each pair is with Polly disabled. Blue bar on right of each pair is with Polly enabled.

4.3 Parallelism, Polly and Accelerate

Accelerate is at its core an implicitly parallel language. Having shown the contribution of Polly and the change to `fold` to the run-times of a number of kernels in the single-threaded context, we have verified that this contribution carries over to the multi-threaded context as well. We ran our tests with the same kernels and variables as in the previous section, but this time with thirty-two threads. The results are shown in Figure 9.

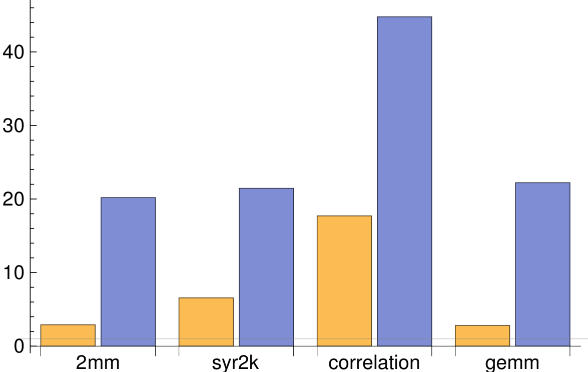


Figure 9: Relative speed-ups of four polybench kernels with change to `fold` relative to unchanged Accelerate code. At horizontal grey bar means no speed-up, above means faster. Yellow bar on left of each pair is with Polly disabled. Blue bar on right of each pair is with Polly enabled.

Polly also provides a way of automatically parallelising code, which can be enabled with the `-polly-parallel` argument, generating code to work with OpenMP[16], the parallel programming

API. In our tests we have used LLVM's own OpenMP back-end. We have compared the relative speed-up provided by Polly both in the single-threaded case, and in the multi-threaded case with `-polly-parallel`. We ran the multi-threaded case with thirty-two threads, just like in our Accelerate tests above. The results are listed in Figure 10.

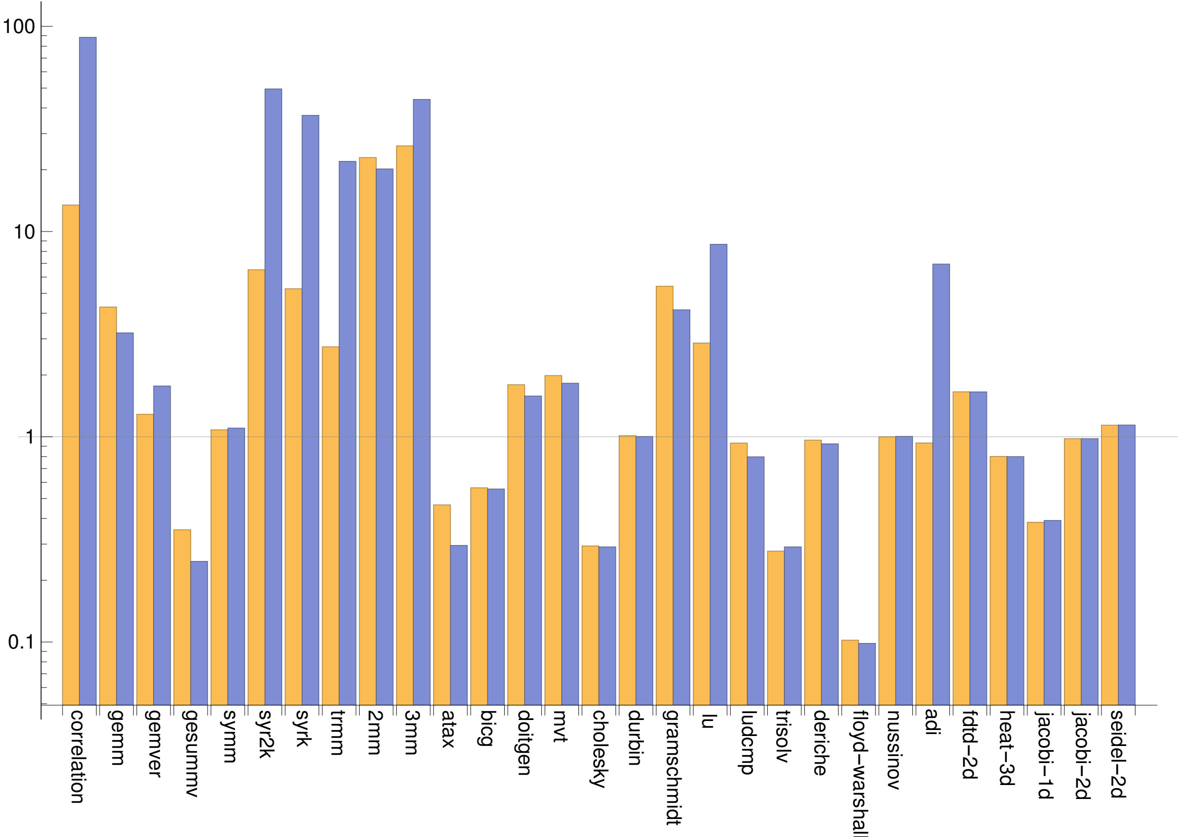


Figure 10: Speed-up with Polly relative to `-O3` alone, either single-threaded or multi-threaded ($\log_2$ scale). At horizontal grey bar means no speed-up, above means faster. The yellow bar on the left of each pair is speed-up with single-threaded Polly, the blue bar on the right is multi-threaded with `-polly-parallel`.

Many kernels show either no change in speed or even a slow-down with `-polly-parallel` enabled. For the cases where there is no change, this means that either Polly couldn't optimise the code in the first place, that Polly couldn't parallelise the code, or that it chose not to parallelise the code. For instance with the `heat-3d` kernel, no change in speed was observed, as Polly chose not to parallelise it. Polly fuses the two loops in such a way that it no longer profitable to parallelise them.

Out of the four examples which we earlier also implemented in Accelerate, `correlation` and `syr2k` showed a significant speed-up with `-polly-parallel`, and `2mm` and `gemm` showed a slow-down. The bulk of the work in the two kernels with a slow-down involves matrix multiplication. Polly's custom matrix multiplication detection transforms this into code which seems not parallelise well.

In Table 2 we compare the Polybench C implementations with Accelerate implementations of the four kernels above, varying whether Polly is enabled and the number of threads the kernels are run

with. For the Polybench C implementation, 32 threads with polly implies `-polly-parallel`.

Running the kernels single-threaded without Polly, C is the clear winner. The C implementations run from 12 to 3 times better. Running without Polly the Accelerate implementations with 32 threads, none of them beats the single-threaded run-times of the C implementations with Polly enabled. However, with `-polly-parallel`, the C implementations of `2mm` and `gemm` show a slow-down, as previously mentioned. With Polly enabled, the Accelerate versions of these two kernels already run faster than the C implementations, with only 4 threads. The C implementations of `correlation` and `syr2k` do show a significant speed-up with `-polly-parallel` at 32 threads. However Accelerate with Polly enabled already beats `correlation` at just 4 threads, and `syr2k` at 8 threads. These results show that in the multi-threaded setting, Accelerate and Polly can make a useful combination.

Kernel	Polybench C			Accelerate							
	No Polly	With Polly		No Polly		With Polly					
	1	1	32	1	32	1	2	4	8	16	32
`2mm`	41.75	1.82	2.06	131.35	8.85	8.16	3.8	1.92	0.99	0.60	0.43
`correlation`	57.32	4.25	0.64	200.12	13.55	4.98	2.04	1.05	0.52	0.39	0.30
`gemm`	9.41	2.19	2.93	119.56	7.90	7.70	3.12	1.57	0.80	0.49	0.35
`syr2k`	43.73	6.71	0.88	133.73	9.17	5.96	2.74	1.40	0.70	0.40	0.42

Table 2: Run-times in seconds of kernels by number of threads. Polybench C, 32 threads with Polly implies `-polly-parallel`.

4.4 Larger Examples

In this section, we move away from the small examples of Polybench and look at how Accelerate and Polly work together on larger examples. We first look at a marsh simulation, then the k-means clustering algorithm, and finally the integral image algorithm.

4.4.1 Salt Marsh Simulation

The salt marsh simulation[17] simulates the sediment and water flow dynamics of a marsh. It is largely a stencil computation, where in addition, at each step, the whole result is iterated over to fix values at the boundary. It is implemented with a `stencil2` operation followed by a `generate` to fix the boundary values. The `stencil2` operation runs over two input arrays. It is like a `zipWith` version of `stencil`. In place of `zipWith` applying a function to corresponding elements of two arrays, `stencil2` applies a function to corresponding neighbourhoods.

The salt marsh simulation involves no `fold` operations, so whether or not our `fold` change was applied is irrelevant. Without Polly enabled, the example ran in 1931 seconds, and with Polly enabled, it ran in 1888 seconds. We wanted to see where exactly in the programme speed-up arose, so we removed the `generate` call that fixes the boundary values, and ran our tests again. Without Polly enabled, the simulation ran in 578 seconds, and with Polly enabled, it ran in 584 seconds, showing a slow-down. This shows that the speed-up was limited to the boundary value fixing operation.

This is not the first time we have observed Polly causing a slow-down of a stencil operation. Most of the Polybench stencil kernels showed either no change, or a slow-down as in the `heat-3d` and

`jacobi-1d` example. This does not mean that Polly cannot provide speed-ups to stencil computations with the right parameters. However, as explained in Section 3.1, it may not be computationally easy to deduce the right parameters such as tile size. In this case, there is nothing that Accelerate can automatically do.

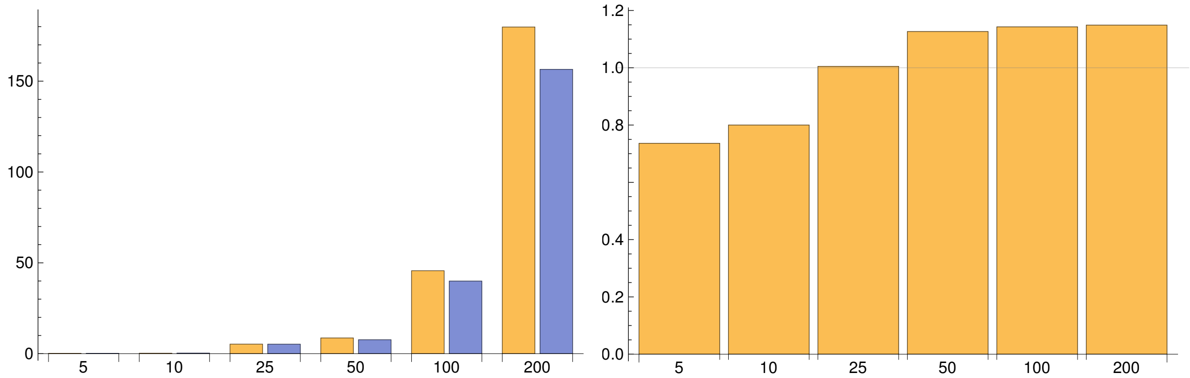
Accelerate unrolls the loops in the code it generates for stencil operations, and we wondered whether this was having an effect on the ability of Polly to optimise the stencil code. We disabled the stencil loop unrolling and ran the tests again. This time with Polly disabled, we observed a slower run-time of 705 seconds, up from 578. This at least shows the benefit of loop unrolling when Polly is disabled. When we enabled Polly, we observed a run-time of 694 seconds, a slight speed-up on the run-time without Polly, but still well above the best run-time we observed when loop unrolling was enabled and Polly was disabled.

From this example and the previous stencil examples from the Polybench suite, we can conclude that Polly is best left disabled for stencil computations. This is not necessarily a negative result, we have merely found one of the limits of Polly. Due to the fact that stencil computations in Accelerate occur through the `stencil` and `stencil2` primitives, it would be simple to implement functionality which decides against enabling Polly in this case.

4.4.2 K-Means Clustering

The k-means clustering algorithm[18] takes a set of vectors and partitions them into k clusters such that each vector belongs to the cluster with the nearest mean. This is implementated in Accelerate at `https://github.com/tmcdonell/accelerate-examples`. The implementation in question defines a benchmark which takes three parameters c, p_{min} and p_{max}. The first parameter defines the number of clusters. The second and third parameters define the minimum and maximum number of points per cluster, the number of points and their positions being chosen randomly between the minimum and maximum.

We ran the benchmark both with Polly and without over a dataset comprising 382500 points. We ran the benchmark multiple times, each time varying the number of clusters while keeping the point set the same. Our change to `fold` has no bearing on the code. In Figure 11 we have two graphs.



(a) Total run-times in seconds of benchmark per number of clusters. Yellow is without Polly, blue is with Polly.

(b) Relative speed-ups per number of clusters with Polly compared to without Polly. At bar means no speed-up, below means slower with Polly, above means faster with Polly.

Figure 11: Run-times ranging from 5 to 200 clusters

From these results, we can see that as the number of clusters increases relative to the number of points, the total run-time of k-means increases. We also see that it is in these examples of more clusters and increased run-times that Polly provides a benefit. Polly does not provide a benefit with lower numbers of clusters, but in that case the run-times are negligible anyway.

4.4.3 Integral Image

The integral image is a data structure that is crucial in the Viola-Jones face detection algorithm[19]. Given an image $I(x, y)$, the integral image $II(x, y)$ at a point (x, y) is equal to the sum of points above and to the left of that point, plus the point itself. It can be defined formally as follows:

$$II(x,y) = \sum_{x' \leq x, y' \leq y} I(x,y)$$

In Accelerate, the integral image over a whole image may be efficiently computed using a combination of calls to `scanl1`, which computes the prefix sum along an array's inner dimension, and `transpose`:

```
integralImage =
  transpose .
  scanl1 (+) .
  transpose .
  scanl1 (+)
```

Listing 10: Integral image implementation in Accelerate

In this algorithm, the `transpose` calls are producers, and the `scanl1` calls are `consumers`. The `scanl1` primitive is the inclusive version of `scanl`. It does not take an argument for initialising each sum. It instead initialises the sum with the first element of the corresponding input row. Following the fusion process, the algorithm results in three fused units:

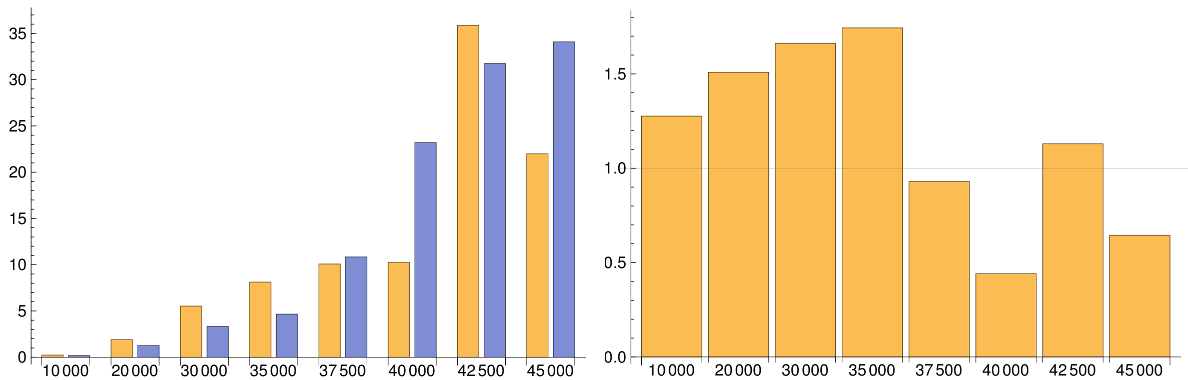
1. `transpose`

2. `scanl1 (+) . transpose`

3. `scanl1 (+)`

After testing, we found that Polly was only able to optimise the `transpose` unit, which compiles to a two-dimensional loop over its two-dimensional input array. With this, it satisfies the requirements for polyhedral optimisation. Given Accelerate's existing implementation, it is not possible for Polly to optimise the `scanl1` units. This is because `scanl1` iterates over its two-dimensional input array using a single loop variable, recovering the two coordinates using integer division and remainder operations, which are not affine. This breaks the affine array indexing requirement for polyhedral compilation. This is exactly the same issue observed in the implementation of `fold` before we changed it.

First leaving the `scanl1` operation unchanged, we tested the integral image algorithm over random images which we generated of varying sizes, running the tests both with and without Polly. We again ran the tests multi-threaded with thirty-two threads. All tests of this algorithm are run multi-threaded unless otherwise mentioned. Remember that the only kernel optimised by Polly here is that corresponding to the outer `transpose` call. Polly simply tiles it. In the results in Figure 12, Image size n means an image of width and length n.



(a) Total run-times of algorithm per image size. Yellow is without Polly, blue is with Polly.

(b) Relative speed-ups per image size with Polly compared to without Polly. At bar means no speed-up, below means slower with Polly, above means faster with Polly.

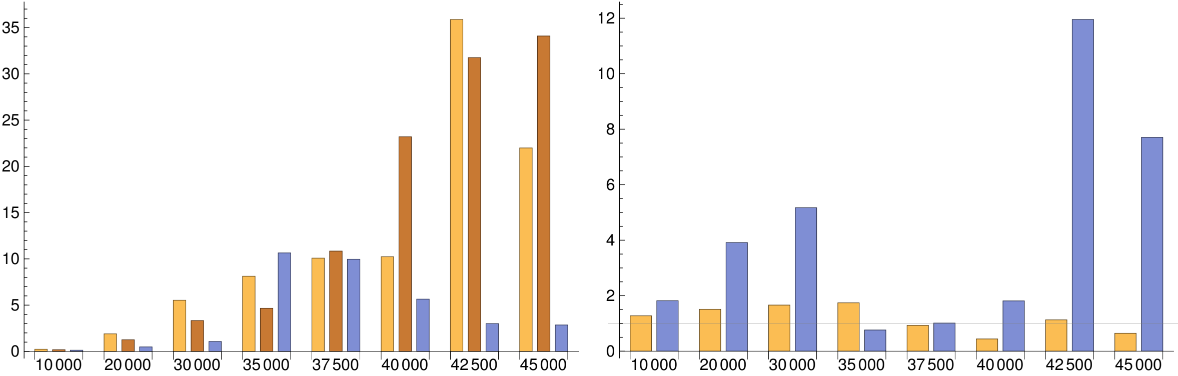
Figure 12: Run-times of integral image algorithm implemented in Accelerate

From the results we can see that until the image size of 37500, the tiling of the final `transpose` by Polly leads to a speed-up, but that it mostly causes a slow-down thereafter. It is not immediately clear why this is, as it would seem that the nature of the transpose operation is directly amenable to the cache friendliness provided by tiling. To narrow the search for the culprit down, we implemented the transpose algorithm in C, and tested it on all image sizes from 0 to 60000 in increments of 5000, thus testing image sizes even larger than we did with Accelerate. We ran the tests with a single thread. Polly tiled the transpose algorithm just as in the Accelerate implementation. We found that Polly provided a significant speed-up to the transpose algorithm over all image sizes.

We also tried running the unchanged integral image algorithm over all image sizes mentioned, but with a single thread. We found in this case as well that Polly provided a speed-up in all cases. The

results of single-threaded test are listed later in Figure 14. It seems therefore that the slow-down on larger image sizes of the algorithm with Polly enabled may be down to how Accelerate divides work and carries out multi-threading, and not down to the implementation of transpose itself. Solving this problem is left as future work.

We changed the iteration behaviour in the code generation of `scanl1` similarly to how we did for `fold`. That is, we ensured that each indexed array dimension corresponded to a unique loop variable. Following this change, Polly was finally able to optimise both `scanl1` units in the integral image algorithm. We ran the algorithm again over the same image sizes and noted significant speed-ups in comparison to the results we observed from the unchanged `scanl1` implementation which we tested with and without Polly. This was apart from the image sizes 35000 and 37500. The total and relative speed-ups are listed in Figure 13. Figure 13a shows the total run-times both with the old `scanl1` and the new. For the old version, run-times both with Polly and without are listed. For the new version, only run-times with Polly are listed. These form the three bars of each group. Figure 13b compares the relative speed-ups Polly provides based on whether the change to `scanl1` is applied. The first bar of each group is with the old version of `scanl1`, the second bar of each group is with the new version.



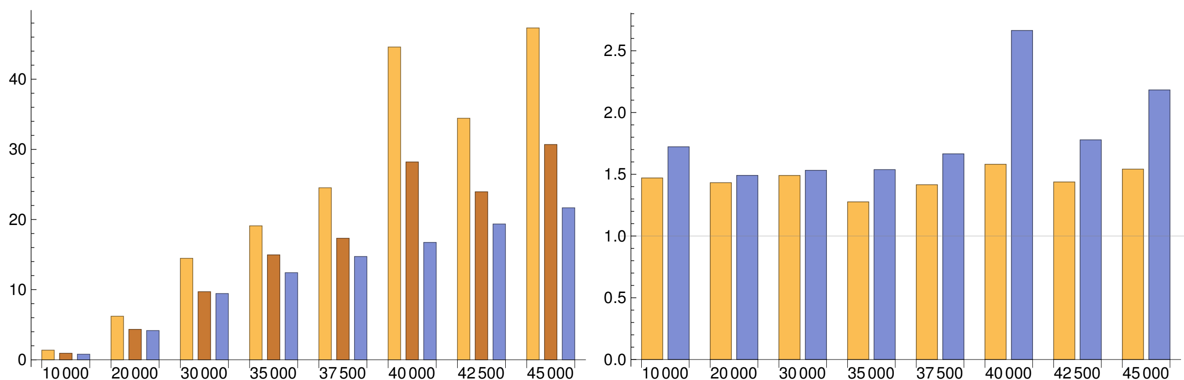
(a) Total run-times of algorithm per image size. Yellow is without change and without Polly, brown is Accelerate without change and with Polly, blue is with change and with Polly.

(b) Relative speed-ups per image size with Polly compared to without Polly. At bar means no speed-up, below means slower with Polly, above means faster with Polly. Yellow is without change, blue is with change.

Figure 13: Run-times of integral image algorithm with or without change to `scanl1`

We ran the algorithm again, this time with the change to `scanl1`, with a single thread. This was to narrow down the possibilities as to origin of the slow-down observed with image sizes 35000 and 37500. In Figure 14 we list the run-times for the unchanged version without and with Polly applied, and the changed version with Polly applied.

The speed-ups for the changed version with Polly applied when run with a single thread were not as dramatic as some of the speed-ups observed in the multi-threaded case. However in comparison, they show a consistent speed-up over all image sizes in comparison to the unchanged version both with and without Polly applied. This leads us to the conclusion that if the anomalies observed when running the algorithm multi-threaded can be solved, the change to `scanl1` applied with Polly can make a useful contribution to the integral image algorithm over all image sizes. Given how our change to `fold` showed a speed-up over a range of applications with Polly applied, it is likely that our change



(a) Total run-times of algorithm per image size. Yellow is without change and without Polly, brown is without change and with Polly, blue is with change and with Polly.

(b) Relative speed-ups per image size with Polly compared to without Polly. At bar means no speed-up, below means slower with Polly, above means faster with Polly. Yellow is without change, blue is with change.

Figure 14: Single-threaded run-times of integral image algorithm with or without change to `scanl1` implementation

to `scanl1` would also.

5 Conclusion, Discussion & Future Work

The aim of this project was to investigate the extent to which array-oriented languages could work together with polyhedral compilation. The number of general-purpose array-oriented frameworks which aim at applying polyhedral compilation without user intervention is limited. For instance PolyMage applies polyhedral optimisation automatically, but is focussed towards image processing. Tiramisu is general purpose, but requires users to apply polyhedral transformations to code explicitly, similarly to Halide.

Based on the literature on polyhedral compilation, and on Polly in particular, we had high hopes as to the ability of Polly to optimise a wide range of array algorithms with minimal user intervention. Our hopes were somewhat broken by the results from the Polybench benchmarks which we ran. We observed slow-downs in a number of cases. In fact, for the stencil class of algorithms, we observed a slow-down in all examples. One of the stencil examples exhibiting a slow-down was `heat-3d`, simulating 3-d heat propagation. After a lot of back and forth on the mailing list for Polly, we finally arrived at a solution for getting Polly to speed `heat-3d` up. It involved preventing loop fusion by editing the code, and setting tiling sizes manually. At that point, not much was left to Polly but to apply the tile sizes.

It is not necessarily a negative insight that the default tile size, and Polly always applies this if it applies tiling, slows stencil operations down. This in fact is exactly the kind of insight we are looking for. Accelerate knows exactly when it is dealing with stencil computations; this is namely when the `stencil` primitive is used. If we had a computationally simple way of deducing tile sizes, then Accelerate could apply it over all programmes involving the `stencil` primitive. However, we previously noted that this is computationally complex[15]. With that, we are left with a weaker rule which could be applied: Accelerate should not apply Polly to any programme involving the stencil

operations.

Tiling size isn't the only optimisation parameter which can be set manually in Polly. Polly has settings for things such as fusion strategy, strip-mining, vectorisation and more. However, the documentation for these parameters is minimal, and can only be found in the argument descriptions for Polly. Given the minimal documentation, it was difficult to know the scope of each parameter, and more importantly for us, whether the value of each parameter could be deduced in a computationally simple manner for automatic application by Accelerate such that it leads to faster code. Such an investigation would have to be left for further work.

Two positive results we obtained in this project were in our changes to the `fold` and `scanl1` operations. We noticed that the code generation for these operations did not fit the requirements of being a SCoP for polyhedral compilation. It was quite simple to change the code generation for `fold` to fit the requirements. The change to `scanl1` took more work for it to be accepted by Polly. We could not find any more operations where this change was needed. As shown in the results, we found that these changes contributed towards significant speed-ups when Polly enabled.

The question of whether Polyhedral compilation can be applied beneficially to a general-purpose array-oriented language like Accelerate without user intervention seems still to be an open question. The promise of such a language in the polyhedral context is that the explicit representation of array-based semantics should allow more efficient communication between language and polyhedral compiler in terms of what a programme represents. With Accelerate and Polly, this communication is realised in two ways. One way is code generation, which we were able to fix for `fold` and `scanl1`, two very important primitives. The other way is through the arguments passed to Polly, such as tiling sizes. We did see that tile sizes are computationally difficult to derive, but Polly provides many more arguments and settings which we did not fully understand, or were not able to investigate due to the lack of documentation. Given greater knowledge of these settings, it might then be possible to optimise a wider range of array-based programme without user intervention. This will have to be left for further work.

References

[1] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating haskell array codes with multicore gpus," in *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, (New York, NY, USA), p. 3–14, Association for Computing Machinery, 2011.

[2] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[3] T. Henriksen, *Design and implementation of the Futhark programming language*. PhD thesis, Department of Computer Science, Faculty of Science, University of Copenhagen, 2017.

[4] S.-B. Scholz, "Single assignment c: efficient support for high-level array operations in a functional setting," *Journal of functional programming*, vol. 13, no. 6, pp. 1005–1059, 2003.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical and fully automatic polyhedral program optimization system," in *ACM SIGPLAN PLDI*, vol. 10, 2008.

[6] "Llvm." `https://llvm.org/`.

[7] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 193–205, IEEE, 2019.

[8] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 429–443, 2015.

[9] C. Lengauer, "Loop parallelization in the polytope model," in *CONCUR'93* (E. Best, ed.), (Berlin, Heidelberg), pp. 398–416, Springer Berlin Heidelberg, 1993.

[10] T. C. Grosser, *Enabling polyhedral optimizations in llvm.* PhD thesis, 2011.

[11] J. Doerfert, "Applicable and sound polyhedral optimization of low-level programs," 2018.

[12] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *International Congress on Mathematical Software*, pp. 299–302, Springer, 2010.

[13] R. Gareev, T. Grosser, and M. Kruse, "High-performance generalized tensor operations: A compiler-oriented approach," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 3, pp. 1–27, 2018.

[14] Louis-Noel Pouchet, Ohio State University, "Polybench." `http://web.cse.ohio-state.edu/~pouchet.2/index.html`, 2015. [Online; accessed 10-February-2020].

[15] D. Adamski, M. Szydłowski, G. Jabłoński, and J. Lasoń, "Dynamic tiling optimization for polly compiler," *International Journal of Microelectronics and Computer Science*, vol. 8, no. 4, 2017.

[16] "Openmp." `https://www.openmp.org/`.

[17] J. van de Koppel, R. Gupta, and C. Vuik, "Scaling-up spatially-explicit ecological models using graphics processors," *Ecological modelling*, vol. 222, no. 17, pp. 3011–3019, 2011.

[18] Wikipedia contributors, "K-means clustering — Wikipedia, the free encyclopedia," 2020. [Online; accessed 20-April-2020].

[19] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, vol. 1, pp. I–I, IEEE, 2001.