Compiling APL to Accelerate through a Typed Array Intermediate Language *

Michael Budde Martin Dybdal Martin Elsman

Department of Computer Science, University of Copenhagen, Denmark
mbudde@gmail.com, dybber@dybber.dk, mael@di.ku.dk

Abstract

We present an approach for compiling a rich subset of APL into data-parallel programs that can be executed on GPUs. The compiler is based on the APLTAIL compiler, which compiles APL programs into a typed array intermediate language, called TAIL [11]. We translate TAIL programs into Haskell source code, employing Accelerate [6], a Haskell-library for general purpose GPU-programming.

We demonstrate the feasibility of the approach by presenting some encouraging results for a number of smaller benchmarks. We also outline some problems that we need to overcome in order for the approach to result in competitive code for larger benchmarks.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classification—Applicative (functional) languages; Concurrent, distributed, and parallel languages

Keywords APL compilation, Data parallelism, GPGPU

1. Introduction

APL is a dynamically typed, second-order programming language developed in the 1960s for manipulating multi-dimensional arrays. The succinct syntax of APL where the large number of built-in operations are written as symbols (e.g., ↑ and ↓ for take and drop, respectively), suggests a data-parallel programming style, which has become increasingly relevant in this multi-core era of programming. Our interest in APL springs from its selection of built-in operations, which, through 50 years of history, have shown to be suitable for a large range of applications. Moreover, a large number of real-world APL programs exists, many of which are written in a data-parallel style and which can be used more or less directly as benchmarks.

Our previous work on APLTAIL [11] compiles a subset of APL into a typed array intermediate language called TAIL. In this paper, we present a compiler from TAIL to Haskell source code, which employs the Accelerate library [6]. Accelerate is an array language embedded in Haskell. It supports multi-dimensional arrays and provides various backends, most prominently a skeleton-based parallel GPU backend using CUDA [22].

TAIL is a statically typed, functional, array programming language, providing a subset of the operations available in APL, with the same semantics as found in APL. For a formal description of the type system and operational semantics of TAIL consult [11].

As a simple example of compilation, consider the following program, which calculates an approximation to $\int_0^{10} 2/(x+2)\ dx$:

```
f ← { 2 ÷ ⍵ + 2 }          ⍝ Function λx. 2 / (x+2)
N ← 10000000               ⍝ No. of valuation points
dx ← 10 ÷ N
domain ← dx × ⍳N           ⍝ Integrate from 0 to 10
integral ← dx × +/f¨domain ⍝ Compute integral
```

This program first declares the function `f` to be integrated (⍵ represents the right argument) and the number of valuation points (`N`). It then computes a discretisation of the integration domain (the ⍳-function, *iota*, generates the array containing values 1 through N) and computes the integral by simple numeric integration, using a sum-reduce (`+/`) over the weighted function values.

The example APL program is compiled into the TAIL program in Figure 1. Notice the presence of vector types with explicit length attributes and the explicit array type `[double]0`, which denotes a scalar double (i.e., a rank zero array.) The TAIL code can be compiled into the Haskell code given in Figure 2, by employing our TAIL to Accelerate compiler, which we refer to as the APLACC compiler in the following [5]. The program references the module `Prim`, which provides Accelerate-implementations for TAIL primitives. Unqualified functions and types come from the Haskell prelude and the Accelerate module.

The contributions of this paper are the following:

1. We give a detailed presentation of a compiler from the intermediate language TAIL to Accelerate. The compiler enables compilation of a subset of APL to programs that, in particular, can use CUDA-supported GPUs to perform calculations in parallel.
2. We demonstrate the feasibility of using Accelerate as the target for an APL compiler by evaluating the performance of the compiled programs, running on a GPU, with the results of using a backend to APLTAIL that compiles the benchmarks into sequential C code.
3. We demonstrate the use of APLTAIL as a generic APL compiler frontend and that larger benchmarks can be expressed in TAIL.

APLACC is open source and available online for download at `github.com/mbudde/aplacc` and is to be used in concert with APLTAIL, available at `github.com/melsman/apltail`.

The remainder of this paper is organised as follows. We give a quick overview of TAIL and Accelerate in Section 2 and Section 3, respectively. In Section 4, which contains the main contribution of

* This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center HIPERFIT: Functional High Performance Computing for Financial Information Technology (`hiperfit.dk`) under contract number 10-092299.

ARRAY'15, June 13, 2015, Portland, OR, USA
Copyright 2015 ACM 978-1-4503-3584-3/15/06...$15.00
http://dx.doi.org/10.1145/2774959.2774966

```
let v5:<double>1000000 =
  eachV{[double,double],[1000000]}(
   fn v4:[double]0 => muld(10.0,v4),
   eachV{[double,double],[1000000]}(
    fn v3:[double]0 => divd(v3,1e6),
    eachV{[int,double],[1000000]}(i2d,
      iotaV(1000000)))) in
let v12:[double]0 =
  reduce{[double],[0]}(addd,0.0,
   eachV{[double,double],[1000000]}(
    fn v9:[double]0 => divd(v9,1e6),
    eachV{[double,double],[1000000]}(
     fn v7:[double]0 =>
     divd(2.0,addd(v7,2.0)),v5))) in
v12
```

Figure 1: The result of compiling the example APL program into TAIL.

$$
\begin{array}{rcll}
v & ::= & i \mid d \mid c \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{inf} & \text{(base values)} \\
e & ::= & v \mid x \mid \texttt{[}\vec{e}\texttt{]} \mid x\,\iota(\,\vec{e}\,) & \text{(expressions)} \\
 & \mid & \texttt{fn}\ x : \tau\ \texttt{=>}\ e & \\
 & \mid & \texttt{let}\ x : \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 & \\
 & & & \\
\kappa & ::= & \texttt{int} \mid \texttt{double} \mid \texttt{char} \mid \texttt{bool} \mid \alpha & \text{(base types)} \\
\rho & ::= & i \mid \gamma \mid \rho + \rho' & \text{(shape types)} \\
\tau & ::= & [\kappa]^\rho \mid \langle\kappa\rangle^\rho \mid \mathrm{S}(\kappa,\rho) \mid \mathrm{SV}(\kappa,\rho) & \text{(types)} \\
 & \mid & \tau \to \tau & \\
\iota & ::= & \texttt{\{}\vec{\kappa}\texttt{,}\ \vec{\rho}\texttt{\}} \mid \epsilon & \text{(instance lists)} \\
\sigma & ::= & \forall\vec{\alpha}\vec{\gamma}.\tau & \text{(type schemes)}
\end{array}
$$

Figure 3: Grammar describing the TAIL language.

the paper, we give a presentation of the compilation from TAIL to Accelerate. In Section 5, we present runtime performance numbers of a handful of compiled example programs and discuss the effectiveness of our approach. The idea of using Accelerate as a target language, and the problems encountered, is discussed in Section 6. We present related work in Section 7 and conclude with Section 8, where we also discuss potential future projects.

2. TAIL

The APLACC compiler takes TAIL programs as input. Figure 3 shows the TAIL syntax supported by the compiler. The syntax deviates somewhat from the grammar described in [11], which has been updated to work with character and boolean arrays.

We assume a denumerable infinite set of program variables (x). For some z, we write $\vec{z}^{(n)}$ to denote the sequence $z_0, z_1, \ldots, z_{n-1}$. If the exact length of the sequence is unknown or irrelevant, we leave it out and write $\vec{z}$.

A *base value* (v) is either an integer (i), a double (d), a character literal (c), a boolean (`tt` or `ff`), or infinity (`inf`). An expression is either a value (v), a variable (x), a vector expression, a function call, a `fn`-expression, or a `let`-expression. For presentation purposes, a TAIL program consists of a single top-level expression. As we shall see, a number of built-in primitives are bound in the top-level initial environment, which, for instance, allow for multi-dimensional arrays, to be constructed during program execution.

Types are segmented into base types (κ), shape types (ρ), types (τ), and type schemes (σ). Shape types (ρ) are considered identical upto associativity and commutativity of $+$ and upto evaluation of constant shape-type expressions involving $+$. Types (τ) include a type for multi-dimensional arrays of rank ρ ($[\kappa]^\rho$), a type for vectors of a specific length ($\langle\kappa\rangle^\rho$), singleton types for integers and booleans ($\mathrm{S}(\kappa,\rho)$), singleton types for single-element integer and boolean vectors ($\mathrm{SV}(\kappa,\rho)$), and a type for functions. As special notation, we often write κ to denote the scalar array type $[\kappa]^0$. Type schemes σ are used for specifying the types for built-in operations. Function calls in TAIL are annotated with *instance lists*, which specify the particular instance of a polymorphic function. The first list contains type instantiations and the second list contains rank instantiations. The numbers of elements in the two lists depend on the function. For example, the type of the `reduce` function is given as:

$$\texttt{reduce} \ : \ \forall\alpha\gamma.\ (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha]^{\gamma+1} \to [\alpha]^\gamma$$

The type is parameterized over two type parameters α and γ, where α denotes a base type and γ a rank. Notice, as just mentioned, that α is used to denote the scalar array type $[\alpha]^0$. An instantiation list for a call to `reduce` contains the particular values of α and γ for that call. Here is an example call to `reduce`, with $\alpha = \texttt{int}$ and $\gamma = 0$:

```
reduce{[int],[0]}(addi, 0, [1,2,3])
```

The type of this expression is `int`. Formally, instantiation lists are defined in terms of a notion of substitution, which we, for space reasons, will not develop here.

We shall only present type schemes for a small number of TAIL primitive operations. We have already seen the type scheme for `reduce`. Here is the type scheme for `eachV`, which takes as arguments a scalar function and a compatible vector of length γ and returns a vector also of length γ:

$$\texttt{eachV} \ : \ \forall\alpha\gamma.\ (\alpha \to \beta) \to \langle\alpha\rangle^\gamma \to \langle\beta\rangle^\gamma$$

As described in [11], the somewhat elaborate type system of TAIL allows for expressing a number of complex operations, such as APL's inner and outer product operators, as derived operations.

3. Accelerate

Accelerate is an array language embedded in Haskell. An array computation represents an abstract syntax tree of the computation. When running an array computation, the tree is optimized in various ways and then passed on to a backend, which takes care of executing the computation and returning the result. There exists a number of different backends to Accelerate but the primary one is the CUDA backend. This backend compiles the Accelerate AST to CUDA code and runs it in parallel on a GPU.

Arrays are multi-dimensional and are represented by the type

$$(\texttt{Shape sh}, \texttt{Elt e}) \Rightarrow \texttt{Array sh e}$$

Shapes have the number of dimensions encoded in the type and are constructed using the *snoc* operator (which is both a type constructor and a type level operator):

```
Z :. 5 :. 8  ::  Z :. Int :. Int
```

Accelerate provides a large number of library functions. To accommodate the APL-style semantics of TAIL, we provide additional operations required by TAIL which are not already provided by Accelerate in a separate module, the aforementioned `Prim` module. For instance, the APL *take* operation adds a default value when overtaking, instead of failing, and *take* also supports a negative argument, in which case it takes elements from the end of the array.

4. From TAIL to Accelerate

In this section, we present the translation from TAIL to Accelerate. The first step is to parse TAIL source code into a TAIL AST. This TAIL AST is then translated into an untyped version of the Accelerate AST, after which it is a straightforward task to output Haskell code with calls to the Accelerate library. In the remainder of this section,

```
program :: IO (Acc (Scalar Double))
program = do let v5 = Prim.eachV (\ v4 -> (constant (10.0 :: Double)) * (v4))
                       (Prim.eachV (\ v3 -> (v3) / (constant (1000000.0 :: Double)))
                        (Prim.eachV Prim.i2d (Prim.iotaV (constant (1000000 :: Int)))))
                         :: Acc (Array DIM1 Double)
             let v12 = the (Prim.reduce (+) (constant (0.0 :: Double))
                            (Prim.eachV (\ v9 -> (v9) / (constant (1000000.0 :: Double)))
                             (Prim.eachV (\ v7 -> (constant (2.0 :: Double)) /
                                                  ((v7) + (constant (2.0 :: Double))))
                              v5)))
                       :: Exp Double
             return (unit v12)
```

Figure 2: The result of compiling the example TAIL program into Accelerate.

we focus on the translation from the TAIL AST to the untyped Accelerate AST.

4.1 Translating Types

Let us first consider the translation of TAIL types to Accelerate types. As we have already seen, in TAIL, we have a number of different type constructors, including type constructors for arrays, shapes, singleton integers, and single-element integer vectors. A scalar value can be typed as either a zero dimensional array or as a singleton ($\mathrm{S}(\kappa, \rho)$) depending on whether the value is known at compile-time. Similarly, single-element vectors can be typed either as a one dimensional array or as a singleton vector ($\mathrm{SV}(\kappa, \rho)$). There are also a number of subtyping relations. Vectors and single-element vectors are subtypes of one dimensional arrays and singleton types are subtypes of zero dimensional arrays (for details, see [11]).

In Accelerate, there are two main data types: `Acc` for array-valued computations and `Exp` for scalar expressions. In addition, there are also plain Haskell values[1], such as Haskell integers. We use p to range over *plain* Haskell types, which include `Int`, `Bool`, `Double`, and `Char`. We first define the following trivial partial translation, $[\![\, \kappa \,]\!]_{\mathrm{b}} = p$, which map concrete base types κ to plain Haskell types:

$$[\![\, \texttt{int} \,]\!]_{\mathrm{b}} = \texttt{Int} \qquad [\![\, \texttt{bool} \,]\!]_{\mathrm{b}} = \texttt{Bool}$$

$$[\![\, \texttt{double} \,]\!]_{\mathrm{b}} = \texttt{Double} \qquad [\![\, \texttt{char} \,]\!]_{\mathrm{b}} = \texttt{Char}$$

In Accelerate, it is possible to convert values between certain types, but subtyping, as in TAIL, is not supported. In particular, we cannot treat a shape vector as an array. Instead, we must arrange that a shape, for instance, is explicitly converted to an array when needed. Accelerate gives us the following three functions to convert between different types:

```
constant : Elt e ⇒ e → Exp e
    unit : Elt e ⇒ Exp e → Acc (Scalar e)
     the : Elt e ⇒ Acc (Scalar e) → Exp e
```

Converting `Int` values to `Double` can be done with the Haskell built-in function `fromIntegral` and converting from `Exp Int` to `Exp Double` can be done with the `fromIntegral` function from Accelerate.

Consider the following TAIL program:

```
let x:[int]0 = 5 in reduce{[int], [0]}(addi, x, [x])
```

When translating this program, we have a choice in which type to give `x`. We can either give it type `Int`, type `Exp Int`, or type `Acc (Scalar Int)`. The choice we make influences which conversions are made in the call to `reduce` since the second argument to `reduce` should be of type `Exp Int` while the vector literal should be of type `Int`. Notice that it is not possible to convert from `Exp a` to `a`. This limitation forces `x` to be of type `Int`; otherwise, we cannot use it in the vector construct.

[1] Not to be confused with the `Plain` associated-type for the `Lift` type class in Accelerate.

$$\begin{array}{lll}
(\epsilon : p) & \rightsquigarrow \texttt{Exp}\ p & = \ \texttt{constant}\ (\epsilon\ \texttt{::}\ p) \\
(\epsilon : p) & \rightsquigarrow \texttt{Acc}\ r\ p & = \ \texttt{unit}\ (\texttt{constant}\ (\epsilon\ \texttt{::}\ p)) \\
(\epsilon : \texttt{Exp}\ p) & \rightsquigarrow \texttt{Acc}\ 0\ p & = \ \texttt{unit}\ \epsilon \\
(\epsilon : \texttt{Acc}\ 0\ p) & \rightsquigarrow \texttt{Exp}\ p & = \ \texttt{the}\ \epsilon \\
(\epsilon : \tau) & \rightsquigarrow \tau & = \ \epsilon \\
(\epsilon : \tau_1) & \rightsquigarrow \tau_2 & = \ \textbf{fail}
\end{array}$$

Figure 4: Type casting rules for converting expressions from one type to another.

The *type translation* from a ground TAIL type τ (a type with no free type variables) to a Haskell/Accelerate type τ', written $[\![\, \tau \,]\!]_{\mathrm{t}} = \tau'$ is defined by the following equations:

$$[\![\, [\kappa]^0 \,]\!]_{\mathrm{t}} = \texttt{Exp}\ [\![\, \kappa \,]\!]_{\mathrm{b}} \qquad [\![\, \mathrm{S}(\kappa, i) \,]\!]_{\mathrm{t}} = \texttt{Exp}\ [\![\, \kappa \,]\!]_{\mathrm{b}}$$

$$[\![\, [\kappa]^i \,]\!]_{\mathrm{t}} = \texttt{Acc}\ i\ [\![\, \kappa \,]\!]_{\mathrm{b}} \qquad [\![\, \langle\kappa\rangle^i \,]\!]_{\mathrm{t}} = \texttt{Acc}\ 1\ [\![\, \kappa \,]\!]_{\mathrm{b}}$$

$$[\![\, \mathrm{SV}(\kappa, i) \,]\!]_{\mathrm{t}} = \texttt{Acc}\ 1\ [\![\, \kappa \,]\!]_{\mathrm{b}}$$

The mapping requires some explanation as it suggests that scalar arrays and singletons are always converted to `Exp`, which is not the whole story. As we will explain later, these types will be converted to plain Haskell types if possible. Second, the mapping suggests that shapes in TAIL (which are represented as integer vectors) are converted to Accelerate vectors, which is also not the whole story. Whenever an Accelerate operation requires an Accelerate shape as argument, the TAIL vector is converted into an Accelerate shape. If the vector is a vector literal containing only integer constants then it is translated into an Accelerate shape using the snoc operator. Otherwise, the vector is converted to an Accelerate shape at run-time using the `shFromVec` primitive.

Because values can be used in different contexts that require different types by Accelerate, we need rules for converting between types. For this reason the APLACC compiler has a set of *type casting* rules. Type casting of a Haskell expression ϵ from type τ_1 to τ_2 takes the form

$$(\epsilon : \tau_1) \rightsquigarrow \tau_2 = \epsilon'$$

where ϵ' is some expression of type τ_2. The explicit type of ϵ is sometimes left out if the type is obvious from the context. Figure 4 shows the type casting rules in APLACC. When casting from a plain type to an `Exp` type, we also add a type signature because Haskell is not always able to deduce the type of ϵ.

$$
\begin{array}{lcl}
[\![\, x \,]\!]\, E\, \tau & = & (x : \tau') \rightsquigarrow \tau \qquad \text{if } E[x] = \tau' \\
[\![\, i \,]\!]\, E\, \tau & = & (i : \texttt{Int}) \rightsquigarrow \tau \\
[\![\, \texttt{tt} \,]\!]\, E\, \tau & = & (\texttt{True} : \texttt{Bool}) \rightsquigarrow \tau \\
[\![\, \texttt{ff} \,]\!]\, E\, \tau & = & (\texttt{False} : \texttt{Bool}) \rightsquigarrow \tau \\
[\![\, d \,]\!]\, E\, \tau & = & (d : \texttt{Double}) \rightsquigarrow \tau \\
[\![\, c \,]\!]\, E\, \tau & = & (c : \texttt{Char}) \rightsquigarrow \tau \\
[\![\, \texttt{inf} \,]\!]\, E\, \tau & = & (\texttt{infinity} : \texttt{Double}) \rightsquigarrow \tau \\
[\![\, \texttt{fn}\ x : \tau^*\ \texttt{=>}\ e \,]\!]\, E\, \tau & = & \texttt{\textbackslash}x\ \texttt{->}\ [\![\, e \,]\!]\ E[x \mapsto [\![\, \tau^* \,]\!]_{\mathrm{t}}]\ \tau \\
[\![\, x\, \iota\ \ (\vec{e}) \,]\!]\, E\, \tau & = & \mathit{convertOp}\ x\ \iota\ \vec{e}\ \tau \\
[\![\, \texttt{let}\ x : \tau^*\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \,]\!]\, E\, \tau & = & \texttt{let}\ x\ \texttt{=}\ \epsilon\ \texttt{::}\ \tau_2\ \texttt{in}\ [\![\, e_2 \,]\!]\ E[x \mapsto \tau_2]\ \tau \\
 & & \quad \text{where } (\epsilon, \tau_2) = \mathit{cancelLift}\ \tau_1\ ([\![\, e_1 \,]\!]\, E\, \tau_1) \\
 & & \quad\quad \tau_1 = [\![\, \tau^* \,]\!]_{\mathrm{t}} \\
[\![\, [\vec{e}^{(n)}] \,]\!]\, E\, (\texttt{Acc 1}\ p) & = & \texttt{use (fromList (Z :. } n\texttt{) [}\vec{\epsilon}\texttt{]) :: Acc (Vector } p\texttt{)} \\
 & & \quad \text{where } \vec{\epsilon} = \epsilon_0, \epsilon_1, \ldots, \epsilon_{n-1} \\
 & & \quad\quad \epsilon_i = [\![\, e_i \,]\!]\, E\, p
\end{array}
$$

Figure 5: Translation of TAIL expressions to Haskell code.

4.2 Translating Expressions

Rules for converting a TAIL expression e to a Haskell expression ϵ take the form

$$[\![\, e \,]\!]\, E\, \tau = \epsilon$$

where E is an environment that maps identifiers to their types and τ is the *type context*. The type context specifies what type the resulting expression is expected to have. For example, the TAIL expression `5` in a plain `Int` context should translate to just the literal `5`, while in an `Exp Int` context, it should translate to `constant (5 :: Int)`, which is what the type casting rules are used for.

Figure 5 shows the rules for translating TAIL expressions to Haskell expressions. In the rules, τ^* denotes a TAIL type while we use τ to range over Accelerate types.

When translating `let`-expressions, we use the function *cancelLift*, which is defined as follows:

$$\begin{array}{l}
\mathit{cancelLift}\ (\texttt{Exp}\ p)\ (\texttt{constant (}\epsilon\ \texttt{::}\ p\texttt{)}) = (\epsilon, p) \\
\mathit{cancelLift}\ \tau\ \epsilon = (\epsilon, \tau)
\end{array}$$

To motivate the need for this function, consider the following TAIL example:

```
let a:[int]0 = 17
in reduce{[int],[0]}(addi, 0, [a, a])
```

At the point the value `17` is bound to the variable `a` of type `[int]0`, we might be tempted to convert it to `Exp Int` using our type conversion rules. This strategy will not work, however, because `a` is used in a vector construct where a plain `Int` is required. Because we cannot cast from `Exp Int` to a plain Haskell `Int`, we instead choose `Int` to be the type of `a`, and postpone the conversion into an `Exp Int`, if needed.

For translating calls to primitive operations, we define a utility function *convertOp*:

$$\epsilon = \mathit{convertOp}\ x\ \iota\ \vec{e}\ \tau$$

The function takes as argument (1) the name of the operation x, (2) an instantiation list ι, (3) a list of arguments $\vec{e}$, and (4) the expected type of the result. The function looks up the function name in the table of all available primitive functions. If the function name is found, the table entry is a function that, given ι and τ, returns information about how to convert each of the arguments and what the return type will be. The argument expressions are then converted according to the specification and the information is combined into the resulting Haskell expression.

Given the translated program, all that is left is to plug the Haskell expression into a Haskell module with the correct imports and `main` function that uses an Accelerate backend to execute the program.

4.3 APL Primitives for Accelerate

We should now have a valid Haskell module that performs the same computations as the TAIL input. The output we get from the APLACC compiler cannot work on its own, however. To compile and run the program, we need Accelerate implementations of the primitive functions.

Some functions have been straightforward to implement while others have been more difficult to implement. Functions that have been straightforward to implement include all scalar functions, such as `addi`, `maxd`, and so on. Also operations such as `each`, `reduce`, `rotate` and `zipWith` are readily available in Accelerate as primitive functions. Examples of functions that are more elaborate to implement include the TAIL operations `take` and `drop`, which need to cover quite a number of cases compared to those that come with Accelerate. The TAIL operation `take` for instance, deals with all the features of the APL take operation (↑), including, for instance, the possibility of taking too many columns from the right of a two-dimensional array [11]. To implement operations such as `transp` and `transp2` (APL's monadic and dyadic transpose, respectively), the Accelerate `backpermute` operation is used [6]. This function is also used for implementing TAIL's `reshape` operation.

Compared to the functionality documented in [11], quite a larger subset of APL is treated, which also means that the number of TAIL operations that needs to be treated by the compilation into Accelerate has increased. In particular, many of the benchmarks that we report on in Section 5 make use of the TAIL `power` operation, which composes a function with itself N times. As recommended in the Accelerate documentation, we first compile the body of the function using `run1` and then execute the compiled kernel repeatedly. Another strategy for compiling `power`, is to use the `awhile` construct, which would require an additional GPU-to-host memory transfer on each iteration. If we instead follow a strategy of repeatedly composing the function with itself, using Accelerate function composition (`>->`), programs are generated that are proportional in size to the number of iterations. Programs generated this way easily reached a size too large for Accelerate to handle.

Some primitives also have TAIL versions that operate on vectors, such as the `eachV` primitive described in Section 2 and the `iotaV` primitive described in Subsection 4.1. Because shapes are converted to arrays, most of these primitives are just aliases of their non-vector counterparts. As an example of a primitive function, here is the implementation of `rotate`:

Table 1: Benchmark timings in milliseconds. The timings are averages over 30 executions. TAIL C is the APL-compiler using a sequential C-code backend, TAIL Acc is the APL-compiler using Accelerate.

Benchmark	Problem size	TAIL C	TAIL Acc
Integral	N = 10,000,000	46.90	3.10
Signal	N = 50,000,000	209.03	16.1
Game-of-Life	40×40, N = 2,000	28.70	2.30
Easter	N = 3,000	33.96	-
Black-Scholes	N = 10,000	54.0	-
Sobol MC π	N = 10,000,000	4881.30	2430.30
HotSpot	1024×1024, N = 360	6072.93	2.03

```
rotate :: (Shape sh, Slice sh, Elt e)
       => Exp Int -> Acc (Array (sh :. Int) e)
       -> Acc (Array (sh :. Int) e)
rotate n arr =
  let sh = Acc.shape arr
      m  = Acc.indexHead sh
      idx sh = Acc.lift $ Acc.indexTail sh :.
                  (Acc.indexHead sh + n) `mod` m
  in Acc.backpermute sh idx arr
```

5. Performance

In this Section, we evaluate the Accelerate backend and compare, for a number of smaller benchmarks, the performance of code generated with the Accelerate-backend to that of code generated with a sequential C-code backend, based on pull-arrays [11]. For one of the benchmarks, we also compare against execution time obtained with a hand-written CUDA implementation. The benchmarks are listed in Table 1.

All benchmarks were executed on an Intel(R) Xeon(R) CPU E5-2650 v2 2.60GHz box with 32 cores and equipped with two NVIDIA GeForce GTX 780 Ti GPUs, each with 3GB ram. All benchmarks were executed 30 times each, and we report averages of wall-clock timings. Time spent on file I/O while reading datasets to memory are not included in the measurements. The benchmark results are shown in Table 1. The reported timings do not include time for compiling the CUDA kernels, which for the Accelerate backend was assured by running the bulk part of a program once, before measuring time for the 30 consecutive executions.

The *Integral* benchmark is the example from Section 1. *Signal* is a signal processing program derived from the APEX benchmark suite [2]. We represent the input signal as a materialized array. *Easter* is a program from Dyalog Ltd. that calculates 300 times the date of Easter for all years between year 1 and 3000. *Black-Scholes* is the well known benchmark (e.g., from the PARSEC benchmark suite) that valuates European options using a closed form solution. In all these micro-benchmarks, the C code generated by TAIL is similar to what you would write by hand. For the Easter and Black-Scholes benchmarks, we see that the APLACC compiler fails to generate executable Haskell code. The problem is that the examples are making use of nested reduce operations, which are not supported by Accelerate. Whereas the generated Haskell code does compile, the Accelerate library issues an error at runtime. We hope to find that newer versions of Accelerate will support these patterns.

We also have a micro benchmark implementing Conway's Game of Life. Here the generated C code is not as you would hope, because of a case of two much fusion and thus code-duplication. We are currently investigating how we can improve on this issue. Our generated Accelerate code though, does not have this problem.

The *Sobol-π* benchmark calculates π based on Monte Carlo simulation using Sobol sequences. We use a naive inductive algorithm, which is why the performance of the C code is not that great, but the algorithm is embarrassingly parallel and thus the Accelerate version should perform much better than the sequential version. We are investigating possibilities for improvement.

Finally, the HotSpot benchmark is a slightly larger program (60 lines of APL), which iteratively solves a series of differential equations for estimating a processor temperature distribution. This benchmark is taken from the Rodinia benchmark suite for heterogeneous computing [7]. We have ported the implementation from code in the APL-like language ELI, originally presented by WM Ching et al. [8]. The Rodinia benchmark suite provides a CUDA implementation and we obtain comparable performance; one millisecond for 360 iterations versus two milliseconds in our case. Similarly, we can report that our generated C code is only slightly slower (around 300 ms) than the C implementation provided with the benchmark suite.

6. Accelerate as a Target Language

We selected Accelerate as a target language, because its selection of built-in operations seemed to fit well with what we needed and our shape types seemed compatible with the shape types of Accelerate. The fact that Accelerate provides segmented reductions and scans, also indicates that it might be possible to perform a NESL-like flattening transformation [3] on TAIL programs and thus allow nested computation. Blelloch's technique has also been applied in the context of Data Parallel Haskell [21] and in the context of compiling NESL [1] to GPU code, but is sometimes incurring a drastic memory overhead. More work in this area is needed for further evaluation.

Some problems did arise in the process of developing the Accelerate backend. The first problem encountered was that the Accelerate AST is not exposed in a version which is easy to target, as one would need to generate Higher-order abstract syntax. Moreover, Accelerate relies essentially on Haskell type inference for inferring shape types. Whereas this dependency is fine when programming directly in Haskell, it requires the code generation strategy to invoke a Haskell compiler after generating Haskell code. There seems to be no way that a compiler for Accelerate, written in Haskell, can avoid invoking a Haskell compiler.

Another problem introduced by the shapes-as-types strategy in Accelerate, is that it is difficult for operations to operate on the outer dimension of an array of arbitrary rank without using inefficient transpositions. For vertical rotation, for instance, we ended up having to write specific Accelerate functions for a variety of ranks, using a fall back transposition strategy (i.e., using `backpermute`) for higher-dimensional arrays.

As mentioned previously, Accelerate documentation recommends using `run1` when the same kernel needs to be executed several times. This suggestion seems odd, as it hinders Accelerate from performing cross-iteration optimizations. A dedicated looping construct is provided (`awhile`), but in our experience, it seemed to generate much slower code. A problem might be that, if let-bound variables outside the loop are only used once, the sharing recovery algorithm of Accelerate will not detect it and thus the computation will be inlined inside the loop, effectively forcing a recomputation on every iteration.

One limiting factor of this work is that Accelerate does not provide for mutable array updates. We have recently extended TAIL (after this project was carried out) with array indexing and updates, which seems crucial for one of our larger benchmarks, an option pricer obtained from a partner company.

Finally, it is often difficult to control when CUDA kernels are generated. In particular, for comparing the efficiency of the generated code, usually, the time used for generating kernels should not be included in the running time. However, with Accelerate, kernels are generated lazily and special attention is needed for forcing a kernel to be generated before execution.

7. Related Work

This paper extends the previous work on compiling APL [11] with a backend for targeting the Haskell Accelerate Library. For compiling a larger set of benchmarks, the previous work has also been extended with support for boolean operations (e.g., compress), new data-parallel operations (e.g., scan), iterative computations (i.e., the power operator $\dot{*}$), and mutable array updates. The enriched treatment of the language also includes a refined type system for TAIL and a more complete coverage of primitives. We are not reporting on these extensions here.

There have been many attempts at compiling APL. For instance, Guibas and Wyatt have demonstrated how a subset of APL can be compiled using a delayed representation of arrays [14], much like how arrays are compiled using the techniques of pull-arrays [10] in Repa [20], Accelerate [6, 22], and Obsidian [9]. Other attempts at compiling APL include Timothy Budd's APL compiler [4] and the ELI-compiler [8]. One of the most serious attempts at compiling APL is the work on APEX [2], which also contains a backend for targeting SAC [12] and thus GPUs [15]. More recent work includes Aaron Hsu's APL co-defns compiler, which also aims at compiling parallel APL constructs [19].

Another pool of related work on array languages is the work on Futhark [16–18], which, as TAIL, and in opposition to SAC [13], holds on to the concept of second order array combinators (named SOACs) in the intermediate representations. One benefit of this approach is its support for fusion even in cases involving filtering and scans. As for Futhark, we seek to allow programmers to express parallel patterns in programs as high-level functional constructs, with the aim of systematically (and automatically) generating efficient (and even data-dependent) parallel code. This design choice has also proven to be useful in the compilation into Accelerate. For comparison, we are investigating the possibility of compiling TAIL programs into Futhark.

8. Conclusion and Future Work

We have presented a compiler that compiles a subset of APL to Accelerate through a typed array intermediate language. The target programs can be executed using a GPU.

We are currently relying on the Accelerate framework to deal with many of the code transformations and analyses necessary for generating efficient low-level CUDA kernel code. These program transformations and analyses include fusion [22]. An interesting line of future work would be to address how an APL programmer could get some controlled influence on how data is processed on the GPU, including what operations are parallelized and whether data is accessed by different GPU threads in a coalesced way. An alternative to using Accelerate as a target for GPU compilation is to use the Obsidian embedded domain specific language [9] as a target. Compared to Accelerate, Obsidian gives the programmer more control over the utilization of a GPU. Thus, using Obsidian as a target for an APL compiler has potential for also giving an APL programmer more control over the GPU.

The present paper also highlights some areas where more work on the APLACC compiler is still needed. Implementation of the missing primitives and better handling of shapes are topics that we are currently working on improving. Finally, it would be interesting to see the results of a more comprehensive benchmark.

References

[1] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. In *17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, pages 247–258, Sept. 2012.

[2] R. Bernecky. APEX: The APL parallel executor. Master's thesis, Department of Computer Science University of Toronto, 1997.

[3] G. Blelloch. Programming Parallel Algorithms. *Communications of the ACM (CACM)*, 39(3):85–97, 1996.

[4] T. Budd. *An APL Compiler*. Springer-Verlag, 1988. ISBN 978-0-387-96643-4.

[5] M. Budde. Compiling APL to Accelerate through a typed IL. MSc project, Department of Computer Science, University of Copenhagen (DIKU), November 2014.

[6] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *6th Workshop on Decl. Aspects of Multicore Programming*, DAMP'11. ACM, 2011.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE Int. Symp. on Workload Characterization*, IISWC'09, 2009.

[8] H. Chen and W.-M. Ching. An ELI-to-C compiler: Production and performance, 2013.

[9] K. Claessen, M. Sheeran, and J. Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *7th Workshop on Decl. Aspects of Multicore Programming*, DAMP'12. ACM, 2012.

[10] C. Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, March 2003.

[11] M. Elsman and M. Dybdal. Compiling a subset of APL into a typed intermediate language. In *1st Int. Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY'14. ACM, 2014.

[12] C. Grelck and S.-B. Scholz. Accelerating APL programs with SAC. In *Conference on APL '99: On Track to the 21st Century*, APL'99, pages 50–57. ACM, 1999.

[13] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *Int. Journal of Parallel Programming*, 34(4):383–427, 2006.

[14] L. J. Guibas and D. K. Wyatt. Compilation and delayed evaluation in APL. In *5th Symposium on Principles of Programming Languages*, POPL'78. ACM, 1978.

[15] J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Decl. Aspects of Multicore Programming*, DAMP'11. ACM, 2011.

[16] T. Henriksen and C. E. Oancea. A T2 graph-reduction approach to fusion. In *2nd Workshop on Functional High-Performance Computing*. ACM, September 2013.

[17] T. Henriksen and C. E. Oancea. Bounds checking: An instance of hybrid analysis. In *1st Int. Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY'14. ACM, 2014.

[18] T. Henriksen, M. Elsman, and C. E. Oancea. Size Slicing - A Hybrid Approach to Size Inference in Futhark. In *3rd Workshop on Functional High-Performance Computing*, FHPC'14. ACM, 2014.

[19] A. Hsu. Co-dfns: Ancient language, modern compiler. In *1st Int. Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY'14. ACM, 2014.

[20] G. Keller, M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *15th Int. Conf. on Functional Programming*, ICFP'10. ACM, 2010.

[21] G. Keller, M. M. Chakravarty, R. Leshchinskiy, B. Lippmeier, and S. Peyton Jones. Vectorisation avoidance. In *Proceedings of the 2012 Haskell Symposium*, Haskell'12. ACM, 2012.

[22] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. In *18th Int. Conference on Functional Programming*, ICFP'13, pages 49–60. ACM, 2013.