APL on GPUs—A TAIL from the Past, Scribbled in Futhark

Troels Henriksen Martin Dybdal Henrik Urms Anna Sofie Kiehn Daniel Gavin
Hjalte Abelskov Martin Elsman Cosmin Oancea
HIPERFIT, Department of Computer Science, University of Copenhagen (DIKU), Denmark
athas@sigkill.dk, dybber@dybber.dk, a.kiehn89@gmail.com, urhmshenrik@gmail.com, danielgavin4@msn.com,
hjalte.diku@gmail.com, mael@di.ku.dk, cosmin.oancea@di.ku.dk

Abstract

This paper demonstrates various translation schemes by which programs written in a functional subset of APL can be compiled to code that is run efficiently on general purpose graphical processing units (GPGPUs). Furthermore, the generated programs can be straightforwardly interoperated with mainstream programming environments, such as Python, for example for purposes of visualization and user interaction. Finally, a careful empirical evaluation shows that the GPGPU translation achieves speedups in the range of hundred of times faster than the sequential C compiled code.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming; D.3.4 [*Processors*]: Compiler

Keywords GPGPU, APL, auto-parallelization, functional language

1. Introduction

One of the major challenges of modern computing is to allow for programmers to utilize parallel hardware efficiently using high-level compositional programming techniques.

One approach to tackle this challenge is based on the concept of *parallelizing optimizing compilers*, which takes as input sequential programs and outputs programs that utilize the parallel computing units of the underlying machine. While this approach can be shown to be successful in some cases, very often the parallelism present in the algorithm has vanished in the process of the programmer expressing the algorithm as a program. An alternative approach is to have the programmer express the intent of the algorithm using high-level array combinators, which, essentially, are parallel operations, lending themselves to a rich set of algebraic array fusion techniques and other high-level program transformations.

In this paper, we present a toolbox (languages and compilers) for executing high-level functional array computations, efficiently, on parallel hardware. In particular, we demonstrate how programs written in a functional subset of APL can be compiled into programs running efficiently on graphics processing units (GPGPUs).

The toolbox employed includes

1. Futhark [7], a statically typed, data-parallel, purely functional array language designed to be used as a target language for higher-level languages and aimed at efficient GPU execution.
2. APLtail, a compiler that compiles a subset of APL into TAIL [3, 6], a typed array intermediate language.
3. a compiler that compiles TAIL into Futhark programs.

The contributions of the paper are the following:

- We demonstrate that, with no particular knowledge about the target architecture, decent parallel high-performance can be achieved from the finger-tips of APL experts by using native data-parallel APL language constructs.
- Non-trivial extensions to TAIL and its type system, in particular support for tuples in conjunction with repeated computations (power operator). These extensions allow the APL programmer to tune at a high-level the parallel algorithm to a particular target architecture. For example moving a convergence loop inside a bulk-parallel operator may change the GPU behaviour from memory bound to compute bound. [1]
- We demonstrate that Futhark is suitable as a target language for efficient GPU computations by describing how the multi-dimensional array language TAIL can be compiled to another array language with all map nests and reduction-nests made explicit. In particular, for the domain of multi-dimensional array programming, we show promising evidence that automatic techniques for array fusion and transformations for achieving coalesced memory access is a doable approach for utilizing data-parallel hardware.
- We demonstrate that the generated programs interoperate easily with visualization and interaction routines in Python. In particular, we demonstrate how APL programs for Conway's Game of Life and a Mandelbrot set explorer can interoperate with Python for visualization and interaction. This step was enabled by a relatively straightforward code generator from Futhark to Python+PyOpenCL.
- Finally, on nine APL applications, we report GPU speedups with a geomean of 125× relative to C compiled code, and make our benchmarking framework publicly available in order to encourage reproducibility of results.

2. TAIL

The TAIL to Futhark compiler takes TAIL programs as input, for which the grammar is shown in Figure 1. Compared to previous presentations of TAIL [3, 6], the present grammar supports charac-

[1] We believe that such insight into high-level optimization trade-offs requires a gentle learning curve.

$$
\begin{array}{rcll}
v & ::= & i \mid f \mid c \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{inf} & \text{(base values)} \\
e & ::= & v \mid x \mid \texttt{[}\,\bar{e}\,\texttt{]} \mid \texttt{(}\bar{e}\texttt{)} \mid \texttt{prj}(i,e) & \text{(expressions)} \\
 & \mid & x\,\iota\texttt{(}\,\bar{e}\,\texttt{)} \mid \texttt{fn}\ x : \tau\ \texttt{=>}\ e & \\
 & \mid & \texttt{let}\ x : \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 & \\
 & & & \\
\kappa & ::= & \texttt{int} \mid \texttt{float} \mid \texttt{char} \mid \texttt{bool} \mid \alpha & \text{(base types)} \\
\rho & ::= & i \mid \gamma \mid \rho + \rho' & \text{(shape types)} \\
\tau & ::= & [\kappa]^{\rho} \mid \langle\kappa\rangle^{\rho} \mid \mathrm{S}(\kappa,\rho) \mid \mathrm{SV}(\kappa,\rho) & \text{(types)} \\
 & \mid & \tau \to \tau \mid \tau_1 \times \cdots \times \tau_n & \\
\iota & ::= & \texttt{\{}\ \bar{\kappa}\ \texttt{,}\ \bar{\rho}\ \texttt{\}} \mid \epsilon & \text{(instance lists)} \\
\sigma & ::= & \forall\bar{\alpha}\bar{\gamma}.\tau & \text{(type schemes)}
\end{array}
$$

Figure 1: Grammar for the TAIL language.

ter arrays, boolean arrays, and tuples—in particular loop structures over tuples of arrays.

We often write $\bar{z}^{(n)}$ to denote the sequence $z_0, z_1, \ldots, z_{n-1}$ of objects of the same kind. If the exact length of the sequence is irrelevant, we write $\bar{z}$. We assume a denumerable infinite set of program variables (x). A *base value* (v) is either an integer (i), a float (f), 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 tuple expression, a tuple projection, a function call, a `fn`-expression, or a `let`-expression. For presentation purposes, a TAIL program consists of one top-level expression, with a set of built-in primitives bound in the top-level initial environment.

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, written $[\kappa]^{\rho}$, a type for vectors of a specific length ρ, written $\langle\kappa\rangle^{\rho}$, singleton types for integers and booleans, written $\mathrm{S}(\kappa, \rho)$, singleton types for single-element integer and boolean vectors, written $\mathrm{SV}(\kappa, \rho)$, a type for functions, written $\tau \to \tau'$, and the type for n-ary tuples, written $\tau_1 \times \ldots \times \tau_n$. As special notation, we often write κ to denote the scalar array type $[\kappa]^0$. Type schemes σ specify 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 base 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 `zipWith` function is given as:

$$\texttt{zipWith} : \forall\alpha_1\alpha_2\beta\gamma.\,(\alpha_1 \to \alpha_2 \to \beta) \to [\alpha_1]^{\gamma} \to [\alpha_2]^{\gamma} \to [\beta]^{\gamma}$$

The type is parameterized over four type parameters α_1, α_2, β, and γ, where α_1, α_2, and β, denote base types and γ a rank. Notice, as mentioned, that we, for instance, write α_1 to denote the scalar array type $[\alpha_1]^0$. An instantiation list for a call to `zipWith` contains the particular values of the type parameters for that call.[2] The following call to `zipWith` has type `int`:

```
zipWith{[int,int,int],[1]}(addi,[11,12],[1,2])
```

Here is the type scheme for power$n$, TAIL's equivalent of APL's ⍣ operator, which iterates a function over tuples of size n, a given number of times:

$$
\begin{array}{lcl}
\texttt{power}n & : & \forall\alpha_1\cdots\alpha_n\gamma_1\cdots\gamma_n.\,(\tau \to \tau) \to \texttt{int} \to \tau \to \tau \\
\text{where} & & \tau = [\alpha_1]^{\gamma_1} \times \cdots \times [\alpha_n]^{\gamma_n}
\end{array}
$$

Figure 2 lists the type schemes for a selection of other TAIL operations, including the APL mnemonic for the operation.

[2] Formally, instantiation lists are defined in terms of a notion of substitution, which we, for space reasons, will not develop here.

APL	*op*(*s*)		Type scheme
+,..	`addi,subi`	:	$\texttt{int} \to \texttt{int} \to \texttt{int}$
−,..	`negi,absi`	:	$\texttt{int} \to \texttt{int}$
⍳	`iota`	:	$\texttt{int} \to [\texttt{int}]^1$
¨	`each`	:	$\forall\alpha_1\alpha_2\gamma.(\alpha_1 \to \alpha_2) \to [\alpha_1]^{\gamma} \to [\alpha_2]^{\gamma}$
/	`reduce`	:	$\forall\alpha\gamma.(\alpha \to \alpha \to \alpha) \to \alpha$ $\to [\alpha]^{1+\gamma} \to [\alpha]^{\gamma}$
/	`compress`	:	$\forall\alpha\gamma.[\texttt{bool}]^{\gamma} \to [\alpha]^{\gamma} \to [\alpha]^{\gamma}$
\	`scan`	:	$\forall\alpha\gamma.(\alpha \to \alpha \to \alpha) \to [\alpha]^{\gamma} \to [\alpha]^{\gamma}$
⍴	`reshape`	:	$\forall\alpha\gamma\gamma'.\langle\texttt{int}\rangle^{\gamma'} \to \alpha \to [\alpha]^{\gamma} \to [\alpha]^{\gamma'}$
⌽	`rotate`	:	$\forall\alpha\gamma.\texttt{int} \to [\alpha]^{\gamma} \to [\alpha]^{\gamma}$
⍉	`transp`	:	$\forall\alpha\gamma.[\alpha]^{\gamma} \to [\alpha]^{\gamma}$
↑	`take`	:	$\forall\alpha\gamma.\texttt{int} \to \alpha \to [\alpha]^{\gamma} \to [\alpha]^{\gamma}$
↓	`drop`	:	$\forall\alpha\gamma.\texttt{int} \to [\alpha]^{\gamma} \to [\alpha]^{\gamma}$
⊃	`first`	:	$\forall\alpha\gamma.\alpha \to [\alpha]^{\gamma} \to \alpha$
,	`cat`	:	$\forall\alpha\gamma.[\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1}$
,	`cons`	:	$\forall\alpha\gamma.[\alpha]^{\gamma} \to [\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1}$

Figure 2: A selection of TAIL operator type schemes.

As described in [6], 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. We shall not present a dynamic semantics and a formal type system for the TAIL language here, but mention that the semantics of the `reduce` and `scan` operations reduces (or scans) the argument array along its last dimension, following the traditional APL semantics [9, 11].

The implementation of the APL compiler uses a hybrid approach of type inference and local context querying for resolving array ranks, scalar extensions, and identity items (neutral elements) during TAIL program generation. The inference is based on a simple unification algorithm using conditional unification for the implementation of a limited form of subtyping inference.

3. Futhark

Futhark is a monomorphic, statically typed, eagerly evaluated, first-order pure functional language, in which potentially nested parallelism is expressed via a set of data-parallel second-order array combinators (SOACs), such as `map`, `reduce`, and `scan`. The Futhark language itself is high-level and language-agnostic, but compilers for both CPUs and GPUs have been implemented.

As for TAIL, we use x to range over variables and i and f to range over integer constants and floating point constants, respectively. Figure 3 presents the abstract syntax of a subset of the Futhark language. Notice that the syntactical construct denoted by l can only occur in the SOACs `map`, `reduce` and `scan`. By representing an algorithm as a composition of SOACs, the compiler can take advantage of the invariants guaranteed by these constructs to perform aggressive transformations such as loop fusion [7].

Futhark has special syntax for a specific pattern of tail-recursive functions, namely the ones that express simple `for`-loops. The correspondence between a Futhark `loop` expression and a tail recursive function is shown on Figure 4. The motivation for this specialised representation is to enable the compiler to perform loop transformations (such as interchange), without first having to analyse recursion patterns to determine which semantically correspond to simple loops. Many of the restrictions in the language are to simplify such transformations, which in the end enable efficient code generation.

When translating from another language to Futhark, we wish to make use of as many of the built-in language constructs as possible,

k	$::=$	$i \mid f \mid$ `[`$\bar{k}$`]`	(scalar or array value)
	$\mid$	`true` $\mid$ `false`	
g, h	$::=$	id	(function names)
p	$::=$	$\tau\ x$	(typed variable)
t	$::=$	`int` $\mid$ `bool` $\mid$ `f32` $\mid$ `f64`	(basic types)
τ	$::=$	`[`τ`]` $\mid$ `(`$\bar{\tau}$`)` $\mid t$	(types)
$\odot$	$::=$	`+` $\mid$ `-` $\mid$ `abs` $\mid$ `...`	(operators)
l	$::=$	`fn` τ `(`$\bar{p}$`) =>` $e \mid \odot$	(fun param)
e	$::=$	$x \mid k$	(variable or value)
	$\mid$	`(`$\bar{e}^{(n)}$`)`	(n-tuple exp)
	$\mid$	x`[`$\bar{e}$`]`	(array indexing)
	$\mid$	$e_1 \odot e_2$	(binop-call)
	$\mid$	$\odot\ e$	(unop-call)
	$\mid$	g`(`$\bar{e}$`)`	(function-call)
	$\mid$	`if` e `then` e_1 `else` e_2	(if binding)
	$\mid$	`let (`$\bar{x}$`) =` e_1 `in` e_2	(let-binding)
	$\mid$	`iota(`e`)`	($[0,\ldots,e-1]$)
	$\mid$	`rearrange((`$\bar{i}$`),` x`)`	(permute dims of x)
	$\mid$	`rotate(`i, e, e'`)`	(rotate dim i of e' by e)
	$\mid$	`zip(`$\bar{e}^{(n)}$`)`	(zip n arrays)
	$\mid$	`unzip(`e`)`	(unzip to n arrays)
	$\mid$	`map(`l, e`)`	(map)
	$\mid$	`reduce(`l, e, e'`)`	(reduce with assoc op l)
	$\mid$	`scan(`l, e, e'`)`	(scan with assoc op l)
	$\mid$	`loop (`$\bar{x} = \bar{e}$`) =`	(sequential do-loop)
		`for` $s < e'$ `do` e''	(next iter $\bar{p} \leftarrow e''$)
P	$::=$	`fun` $\tau\ g$`(`p`) =` $e; P \mid \epsilon$	(named function defs)

Figure 3: Simplified Futhark syntax.

```
loop ((x1,...,xn) =                let fun t f(int i, int n, t1 x1, ..., tn xn) =
      (a1,...,an)) =                   if i >= n then x
  for i < n do              ⇒          else f(i+1, n, g(i,x1,...,xn))
    g(i, x1,...,xn)                in f(0, n, a1, ..., an)
```

Figure 4: Loop to recursive function.

in order to maximise the ability of the Futhark compiler to generate efficient code.

4. From TAIL to Futhark

When e is some TAIL expression, and e' is some Futhark expression we specify the translation as conversion rules of the form $[\![e]\!] = e'$. The rules are syntax-directed in the sense that they follow the structure of e, recursively.

Figure 6 presents the main part of the compilation scheme and Figure 7 shows the conversion rules for SOAC function parameters. For brevity we do not give the complete translation table for binary and unary operators, but rather assume a translation function $[\![op]\!]_{\text{binop}}$, which translates a TAIL binary operator into a Futhark infix operator (e.g., `addi` is translated into +) and a translation function $[\![op]\!]_{\text{unop}}$, which translates a TAIL unary operator into a Futhark unary operator (e.g., `absi` translates into `abs`). Where possible, TAIL primitives have been mapped directly to their corresponding versions in the Futhark language. Care is taken to map the 1-indexed arrays of TAIL to the 0-indexed arrays of Futhark. Where direct translation is not possible, the approach has been to use existing operations as much as possible and generate code to bridge the gap.

```
fun [t] take1_t(int l, [t] x) =
  if 0 <= l then -- not undertake?
    if l <= size(0, x) -- padding not needed?
    then let (v1, _) = split((l),x) in v1
    else concat(x, replicate(abs(l) - size(0, x), 0))
  else
    if 0 <= l + size(0, x) -- padding not needed?
    then let (_, v2) = split(l + size(0, x), x)
         in v2
    else concat(replicate(abs(l) - size(0, x), 0), x)
```

Figure 5: Skeleton for $\texttt{take1}_{[\![t]\!]}$, type-parametric over t.

Some TAIL operations, typically those that exhibit significant dynamic behaviour, do not correspond easily to any single Futhark construct. For example, $\texttt{take}_{[t]}(d, a)$ can be passed a negative number for the amount of elements d to take from the array a, which is called *undertaking*. In this case, the semantics is to take the last $-d$ elements of a (recall that d is negative), and if necessary prepend with zeroes to ensure that the final array is of size $-d$. In our translation scheme, this maps to a "polymorphic" Futhark function $\texttt{take1}_{[\![t]\!]}$, which is indexed by the (statically known) type $[\![t]\!]$. Futhark itself does not support polymorphic functions, so in this case our compiler generates a specialised version of `take1` from a generic skeleton. This skeleton can be seen on Figure 5.

Another example is the `cons` TAIL operator, which appends one element at the beginning of each innermost row of a multi-dimensional array. While various translations are possible, the one that is best supported by the current version of Futhark compiler is a `map` nest of depth equal to the array's rank, in which the innermost map performs the append operation.

4.1 Handling IO

One particular nuisance is that while Futhark is a pure functional language, TAIL is not. At any point in execution, TAIL can read and write to arbitrary files. However, very often side effects are only used in TAIL to read initial program inputs and to write the final result. We support this common pattern by translating top-level I/O reads and writes to respectively parameters and return values of the Futhark `main` function. The ordering of parameters and return values corresponds to the order in which they appear in the TAIL source code. This is done by a preprocessing step that filters out the top-level I/O-operations before passing it to the main translation function. When such I/O operations are found in the TAIL program, the ordinary return expression of the program (the last expression) is ignored. An example is shown in Figure 8.

5. Bridging APL and Python

Futhark's primary purpose is to provide a functional IR that is well suited for expressing data-parallel computational kernels, together with a compiler infrastructure that can effectively optimize for GPU execution. Sections 4 and 6 validate this design by showing that a subset of a high-level language, APL, can be directly translated to TAIL and then to Futhark, and from there, efficiently executed on GPUs.

A collateral advantage of aggresively migrating the application's code to GPU is that, from a performance point of view, it becomes irrelevant what programming language is used for the CPU-glue code that orchestrates the GPU execution. This enables language interoperability: for example, computational kernels expressed in APL (and translated to Futhark) can be executed in a Python environment that is well suited for ease of scripting and visualization. This remaining of this section demonstrates this view

$$\begin{array}{lcllcllcl}
\llbracket x \rrbracket & = & x & \llbracket i \rrbracket & = & i & \llbracket f \rrbracket & = & f \\
\llbracket c \rrbracket & = & \mathit{ascii}(c) & \llbracket \texttt{tt} \rrbracket & = & \texttt{true} & \llbracket \texttt{ff} \rrbracket & = & \texttt{false}
\end{array}$$

$$\begin{array}{lcl}
\llbracket \texttt{negi}\,(e)\,\rrbracket & = & \texttt{-}\llbracket e \rrbracket \\
\llbracket \texttt{let}\ x : t = e_1\ \texttt{in}\ e_2 \rrbracket & = & \texttt{let}\ x = \llbracket e_1 \rrbracket\ \texttt{in}\ \llbracket e_2 \rrbracket \\
\llbracket\,\texttt{[}e_1, \ldots, e_n\texttt{]}\,\rrbracket & = & \texttt{[}\llbracket e_1 \rrbracket, \ldots, \llbracket e_n \rrbracket\texttt{]} \\
\llbracket \mathit{op}\,(e_1, e_2)\,\rrbracket & = & \llbracket e_1 \rrbracket \odot \llbracket e_2 \rrbracket, \quad \text{if } \llbracket \mathit{op} \rrbracket_{\text{binop}} = \odot \\
\llbracket \mathit{op}\,(e)\,\rrbracket & = & \odot\,\llbracket e \rrbracket, \quad \text{if } \llbracket \mathit{op} \rrbracket_{\text{unop}} = \odot \\
\llbracket \texttt{each}_{[t_1,t_2,r]}\,(f, a)\,\rrbracket & = & \begin{cases} \texttt{map}\,(\llbracket f \rrbracket_{\text{fn}}^{\llbracket t_2 \rrbracket}, \llbracket a \rrbracket) & r = 1 \\ \texttt{map}\,(\texttt{fn}\ \llbracket [t_2]^{r-1} \rrbracket\ (\llbracket [t_1]^{r-1} \rrbracket\ x)\ \texttt{=>}\ \llbracket \texttt{each}_{[t_1,t_2,r-1]}\,(f, x)\,\rrbracket, \llbracket a \rrbracket) & r > 1 \end{cases} \\
\llbracket \texttt{reduce}_{[t,r]}\,(f, n, a)\,\rrbracket & = & \begin{cases} \texttt{reduce}\,(\llbracket f \rrbracket_{\text{fn}}^{\llbracket t \rrbracket}, \llbracket n \rrbracket, \llbracket a \rrbracket) & r = 1 \\ \texttt{map}\,(\texttt{fn}\,(\llbracket [t]^{r-2} \rrbracket\,(\llbracket [t]^{r-1} \rrbracket\ x))\,\texttt{=>}\,(\llbracket \texttt{reduce}_{[t,r-1]}\,(f, n, x)\,\rrbracket, \llbracket a \rrbracket)) & r > 1 \end{cases} \\
\llbracket \texttt{zipWith}_{[t_1,t_2,t_3,r]}\,(f, a_1, a_2)\,\rrbracket & = & \begin{cases} \texttt{map}\,(\llbracket f \rrbracket_{\text{fn}}^{\llbracket t_3 \rrbracket}, \texttt{zip}\,(\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket)) & r = 1 \\ \texttt{map}\,(\texttt{fn}\ \llbracket [t_3]^{r-1} \rrbracket\ (\llbracket [t_1]^{r-1} \rrbracket\ x, \llbracket [t_2]^{r-1} \rrbracket\ y)\ \texttt{=>}\ \llbracket \texttt{zipWith}_{[t_1,t_2,t_3,r-1]}\,(f, x, y)\,\rrbracket & \\ \qquad \texttt{zip}\,(\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket)) & r > 1 \end{cases} \\
\llbracket \texttt{vrotate}_{[t,r]}\,(i, a)\,\rrbracket & = & \texttt{rotate}\,(0, \llbracket i \rrbracket, \llbracket a \rrbracket) \\
\llbracket \texttt{rotate}_{[t,r]}\,(i, a)\,\rrbracket & = & \texttt{rearrange}\,((r-1, \ldots, 0), \llbracket \texttt{vrotate}_{[t,r]}\,(i, \texttt{transp}_{[t,r]}\,(a))\,\rrbracket) \\
\llbracket \texttt{vreverse}_{[t,r]}\,(a)\,\rrbracket & = & \texttt{let}\ y = \llbracket a \rrbracket\ \texttt{in}\ \texttt{map}\,(\texttt{fn}\ x\ \texttt{=>}\ y\texttt{[}\texttt{size}\,(0, y) - x - 1\texttt{]}, \texttt{iota}\,(\texttt{size}\,(0, y))) \\
\llbracket \texttt{reverse}_{[t,r]}(a) \rrbracket & = & \texttt{rearrange}\,((r-1, \ldots, 0), \llbracket \texttt{vreverse}_{[t,r]}\,(\texttt{transp}_{[t,r]}\,(a))\,\rrbracket) \\
\llbracket \texttt{reshape}_{[t,r_1,r_2]}\,(a_1, a_2)\,\rrbracket & = & \texttt{let}\ (x, y)\ \texttt{=}\ (\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket)\ \texttt{in} \\
 & & \quad \texttt{reshape}\,(x, \texttt{reshape1}_{\llbracket t \rrbracket}\,(\mathit{osize}, \texttt{reshape}\,(\mathit{isize}, y))) \\
 & & \qquad \text{where } \mathit{osize} = \texttt{size}\,(0, x) * \ldots * \texttt{size}\,(r_1, x) \\
 & & \qquad\qquad \mathit{isize} = \texttt{size}\,(0, y) * \ldots * \texttt{size}\,(r_2, y) \\
\llbracket \texttt{cat}_{[t,r]}\,(a_1, a_2)\,\rrbracket & = & \begin{cases} \texttt{concat}\,(\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket) & r = 1 \\ \texttt{map}\,(\texttt{fn}\ \llbracket [t]^{r-1} \rrbracket\ (\llbracket [t]^{r-1} \rrbracket\ x, \llbracket [t]^{r-1} \rrbracket\ y)\ \texttt{=>}\ \llbracket \texttt{cat}_{[t,r-1]}\,(x, y)\,\rrbracket,\ \texttt{zip}\,(\llbracket a_1 \rrbracket,\ \llbracket a_2 \rrbracket)) & r > 1 \end{cases} \\
\llbracket \texttt{first}_{[t,r]}\,(a)\,\rrbracket & = & \texttt{let}\ x = \llbracket a \rrbracket\ \texttt{in}\ x\texttt{[}\bar{0}^{(r)}\texttt{]} \\
\llbracket \texttt{take}_{[t,r]}\,(i, a)\,\rrbracket & = & \texttt{let}\ (x, y)\ \texttt{=}\ (\llbracket i \rrbracket, \llbracket a \rrbracket)\ \texttt{in} \\
 & & \quad \texttt{reshape}\,(\mathit{oshape}, \texttt{take1}_{\llbracket t \rrbracket}\,(\mathit{osize}, \texttt{reshape}\,(\mathit{isize}, y))) \\
 & & \qquad \text{where } \mathit{oshape} = (\texttt{abs}\,(x), \texttt{size}\,(1, y), \ldots, \texttt{size}\,(r, y)) \\
 & & \qquad\qquad \mathit{osize} = (x * \texttt{size}\,(1, y) * \ldots * \texttt{size}\,(r, y)) \\
 & & \qquad\qquad \mathit{isize} = \texttt{size}\,(0, y) * \ldots * \texttt{size}\,(r, y) \\
\llbracket \texttt{drop}_{[t,r]}\,(i, a)\,\rrbracket & = & \texttt{let}\ (x, y)\ \texttt{=}\ (\llbracket i \rrbracket, \llbracket a \rrbracket)\ \texttt{in} \\
 & & \quad \texttt{reshape}\,(\mathit{oshape}, \texttt{drop1}_{\llbracket t \rrbracket}\,(\mathit{osize}, \texttt{reshape}\,(\mathit{isize}, y))) \\
 & & \qquad \text{where } \mathit{oshape} = (\texttt{max}\,(0, \texttt{size}\,(0, y) - \texttt{abs}\,(x)), \texttt{size}\,(1, y), \ldots, \texttt{size}\,(r, y)) \\
 & & \qquad\qquad \mathit{osize} = x * \texttt{size}\,(1, y) * \ldots * \texttt{size}\,(r, y) \\
 & & \qquad\qquad \mathit{isize} = \texttt{size}\,(0, y) * \ldots * \texttt{size}\,(r, y) \\
\llbracket \texttt{transp}_{[t,r]}\,(a)\,\rrbracket & = & \texttt{rearrange}\,((r-1, \cdots, 0), \llbracket a \rrbracket) \\
\llbracket \texttt{cons}_{[t,r]}\,(e, a)\,\rrbracket & = & \begin{cases} \texttt{let}\ x = \llbracket a \rrbracket\ \texttt{in} & \\ \quad \texttt{map}\,(\texttt{fn}\ \llbracket t \rrbracket\ (\texttt{int}\ y)\ \texttt{=>}\ \texttt{if}\ y > 0\ \texttt{then}\ x\texttt{[}y - 1\texttt{]}\ \texttt{else}\ \llbracket e \rrbracket & \\ \qquad \texttt{, iota}\,(\texttt{size}\,(0, x) + 1)) & r = 1 \\ \texttt{map}\,(\texttt{fn}\ \llbracket [t]^{r-1} \rrbracket\,(\llbracket [t]^{r-2} \rrbracket\ x,\ \llbracket [t]^{r-1} \rrbracket\ y)\ \texttt{=>} & \\ \quad \llbracket \texttt{cons}_{[t,r-1]}\,(x, y)\,\rrbracket, \texttt{zip}\,(\llbracket e \rrbracket, \llbracket a \rrbracket)) & r > 1 \end{cases} \\
\llbracket \texttt{iota}\,(a)\,\rrbracket & = & \texttt{map}\,(\texttt{fn int}\ (\texttt{int}\ x)\ \texttt{=>}\ x + 1, \texttt{iota}\,(\llbracket a \rrbracket)) \\
\llbracket \texttt{shape}_{[t,r]}\,(a)\,\rrbracket & = & \texttt{let}\ x = \llbracket a \rrbracket\ \texttt{in}\ \texttt{[}\texttt{size}\,(0, x), \ldots, \texttt{size}\,(r-1, x)\texttt{]}
\end{array}$$

Figure 6: Conversion rules for expressions. For each rule, x and y are considered fresh.

$$\begin{array}{lcllcl}
\llbracket \texttt{int} \rrbracket & = & \texttt{i32} & \llbracket \texttt{float} \rrbracket & = & \texttt{f32}\ \ (\text{or } \texttt{f64}) \\
\llbracket \texttt{bool} \rrbracket & = & \texttt{bool} & \llbracket \texttt{char} \rrbracket & = & \texttt{i32} \\
\llbracket \tau_1 \times \ldots \times \tau_n \rrbracket & = & (\llbracket \tau_1 \rrbracket, \ldots, \llbracket \tau_n \rrbracket) & \llbracket [\tau]^0 \rrbracket & = & \llbracket \tau \rrbracket \\
\llbracket [\tau]^{i+1} \rrbracket & = & \texttt{[}\llbracket [\tau]^i \rrbracket\texttt{]} & \llbracket \langle \tau \rangle^i \rrbracket & = & \texttt{[}\llbracket \tau \rrbracket\texttt{]} \\
\llbracket \mathrm{S}(\tau, i) \rrbracket & = & \llbracket \tau \rrbracket & \llbracket \mathrm{SV}(\tau, i) \rrbracket & = & \texttt{[}\llbracket \tau \rrbracket\texttt{]}
\end{array}$$

$$\begin{array}{lcll}
\llbracket \texttt{fn}\ x : t\ \texttt{=>}\ e \rrbracket_{\text{fn}}^{\tau} & = & \texttt{fn}\ \tau\ (\llbracket t \rrbracket\ x)\ \texttt{=>}\ \llbracket e \rrbracket & \\
\llbracket \texttt{fn}\ x : t_1\ \texttt{=>}\ \texttt{fn}\ y : t_2\ \texttt{=>}\ e \rrbracket_{\text{fn}}^{\tau} & = & \texttt{fn}\ \tau\ (\llbracket t_1 \rrbracket\ x, \llbracket t_2 \rrbracket\ y)\ \texttt{=>}\ \llbracket e \rrbracket & \\
\llbracket \mathit{op} \rrbracket_{\text{fn}}^{\tau} & = & \odot & \text{if } \llbracket \mathit{op} \rrbracket_{\text{binop}} = \odot \lor \llbracket \mathit{op} \rrbracket_{\text{unop}} = \odot
\end{array}$$

Figure 7: Conversion rules for types and SOAC function parameters.

```
let v1:[int]1 = readIntVecFile("first") in
let v2:[int]1 = readIntVecFile("second") in
let v5:[int]1 = prArrI(zipWith{[int,int,int],[1]}
                       (subi,v1,v2)) in
let v8:[int]1 = prArrI(zipWith{[int,int,int],[1]}
                       (subi,v2,v1)) in
0.0
```

(a) TAIL program with IO.

```
fun [int] main([int] v1, [int] v2) =
  let v5 = map(-, zip(v1, v2))
  let v8 = map(-, zip(v2, v1))
  in (v5, v8)
```

(b) Corresponding Futhark program—the IO operations have been turned into parameters and return values.

Figure 8: Handling IO operations.

by reporting a straightforward code generator from Futhark to Python+PyOpenCL [10]:[3]

- A Futhark program is translated to a Python module. While in principle, the Futhark code generator support multiple exports (entry points) per program, the APL translation is currently restricted to export only one entry point, (the `main` function) due to the implementation technique explained in Section 4.1.
- Futhark arrays are mapped to PyOpenCL buffers and basic-type values are represented with the use of Numpy scalars in order to maintain the Futhark semantics (e.g., single versus double-precision floats). The latter indirection leads to inefficient Python code, but this is not a concern because the runtime is dominated by the GPU execution.
- Each entry point is translated to a Python function than converts its parameters to the types expected by the translation. For example a Numpy array argument is transfomed to a PyOpenCL (device) buffer, a device-buffer argument is left unchanged, and an array result is delivered as a PyOpenCL buffer. The latter prevents unnecessary device-to-host copying in the case when multiple point entries are used in a consumer-producer fashion in a Python program.

Figure 9 demonstrates interoperation between APL and Python: the computational kernels for Game of Life and Mandelbrot were implemented in APL, translated to Python+PyOpenCL code, and are used from a Python environment that supports real-time visualization and user interaction by the use of Pygame library. For example, the user can interactively insert additional glider guns to Game of Life, and can zoom into Mandelbrot figure and adjust accuracy by increasing/decreasing the convergence loop count. Note that smooth visualization requires a per-frame computation time of about 16 ms; this is achieved by the GPU code, but *not* by the sequential C code, which takes hundreds of miliseconds (see Table 1).

6. Performance

In this section, we evaluate the performance of the Futhark backend on a small number of benchmark programs. For each program, we compile to sequential C using TAILs built-in code generator, as well as compiling to Futhark using `tail2futhark`. The resulting Futhark program is then compiled to both sequential C as well as parallel OpenCL code. To measure the quality of the Tail-to-Futhark translation, we have also implemented each benchmark directly in Futhark.

[3] In essence a similar code generator can be used to interoperate Futhark with other mainstream languages, such as Java, as long as they are interfaced with OpenCL (or CUDA).

6.1 Experimental Methodology and Hardware

Our experimental setup is publicly available at the URL:

`https://github.com/HIPERFIT/futhark-fhpc16`

We strongly believe in the value of reproducible results, and have attempted to automate the reproduction of our experiments. We have also documented common technical problems with running the experiments.

All benchmarks were executed on a system with an Intel Xeon CPU E5-2650 and an NVIDIA GeForce GTX 780 Ti GPU. Each program was executed 30 times and we report averages of wall-clock timings. Time spent on file I/O to read input datasets, or write results, has not been included in the timing. Likewise, for GPU executions, the time taken to initialise a GPU context, to compile the GPU kernel code, and the host-to-device transfer of the program input and result were also excluded from measurements. The resulting runtimes are shown in table 1, and speedups compared to a sequential baseline shown on Figure 10.

6.2 Results and Analysis

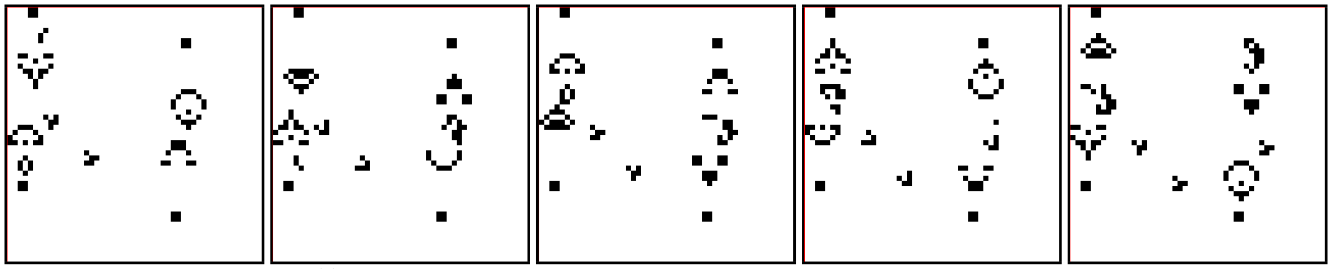
The *Integral* benchmark is a numerical integration program via sampling. *Signal* is a signal processing program derived from the APEX benchmark suite [1]. We represent the input signal as a materialized array. *Easter* is a (modified) program from Dyalog Ltd. that calculates the date of Easter for all years between year 1 and 10^6. *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.

The *Game of Life* benchmark simulates 100 steps of a 1200×1200 board. This involves a sequential loop surrounding a rank-1 stencil computation, and therefore requires efficient interplay between the sequential and embarassingly parallel parts of the code.

The *Sobol-π* benchmark calculates π based on Monte Carlo simulation using Sobol sequences. The Sobol generation algorithm used is a naive inductive algorithm where each Sobol number is computed independently. While this is not the most efficient way to compute Sobol numbers, it is embarassingly parallel, which means the only moderate speedup on the GPU is somewhat disappointing. The reason for this sub-par performance is a deficiency in the Futhark compiler that leads it to generate code with non-coalesced memory accesses.

The *Mandelbrot* benchmark computes the Mandelbrot set for 1000×1000 points in the area $(-2 - 0.75i, 0.75 + 0.75i)$, with an iteration limit of 255. This benchmark is particularly interesting, because there are two different ways of representing the 255-iteration sequential per-point loop: either outside the parallel loops (the "vectorised" programming style), or inside. These are implemented as *mandelbrot1* and *mandelbrot2*, respectively. In this case, the latter style is dramatically more efficient on the GPU, as it makes the program as a whole *compute-bound* instead of *memory-bound*. On the CPU, the vectorised style is slightly more efficient, although the difference is much less than on the GPU. We believe this is due to the C compiler used for final machine code generation, which can more easily make use of vectorised SIMD instructions for this style.

Our final benchmark is *HotSpot*, which is adapted from the Rodinia [4] benchmark suite. This program iteratively solves a series of differential equations for estimating processor temperature distribution. This is done through an outer sequential time-series loop surrounding an inner rank-1 stencil. It is thus similar to the Game of Life benchmark, but with more complicated edge conditions. Our implementation is ported from an ELI implementation by WM Ching et al [5]. Performance on generated Futhark code is poor due


(a) Visualisation of two Gosper glider guns, each image spaced five time steps apart.

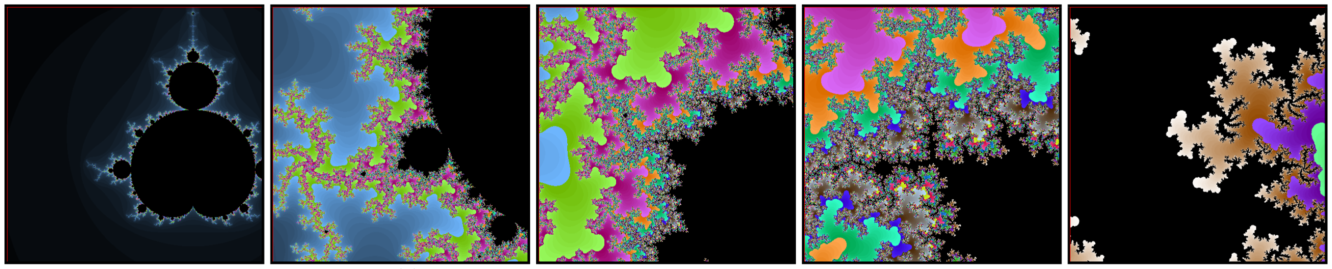

(b) Different zoom levels of a visualisation of the Mandelbrot set.

Figure 9: Game of Life and Mandelbrot benchmarks, compiled to PyOpenCL and rendered in real-time.

to the edge conditions being implemented through the use of several array concatenations. The Futhark compiler is not yet able to fuse neighboring small concatenations into one larger concatenation.

7. Related Work

This paper is a successor to previous work on compiling APL to parallel GPU code [3]. Our benchmark suite is an extension of the one found in this previous work. We are now able to compile every program in the suite to parallel GPU code, and achieve better speedups than previously reported. The TAIL-to-Futhark translation is based on a bachelor's thesis by two of the authors [12].

There have been many attempts at compiling APL, including Timothy Budd's APL compiler [2], ELI [5], and APEX [1]. More recent work includes the Co-dfns compiler [8], which is itself written in a data-parallel style in APL.

8. Conclusions and Future Work

We have presented a compiler that compiles a subset of APL to Futhark through a typed array intermediate language. The generated Futhark programs can be further compiled to either CPU or GPU code, and interoperated by means of host-code generators with mainstream programming environments, such as Python.

We rely on the Futhark compiler to perform optimisation and code generation. Our experimental evaluation shows that we obtain performance close to hand-written Futhark in several cases, and significant speedups over sequential code in almost all cases. However, in some cases the APL programming style gives rise to code that the Futhark compiler is not presently able to compile efficiently. We are investigating enhancing both the optimisation passes in the Futhark compiler itself, as well as generating Futhark code that does not exhibit the problematic patterns.

Acknowledgments

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' (`http://hiperfit.dk`) under contract number 10-092299.

References

[1] R. Bernecky. APEX: The APL parallel executor, 1997.

[2] T. Budd. *An APL compiler*. Springer Science & Business Media, 2012.

[3] M. Budde, M. Dybdal, and M. Elsman. Compiling APL to Accelerate through a Typed Array Intermediate Language. In *Procs. Int. Workshop on Lib. Lang. and Compilers for Array Prog. (ARRAY)*. ACM, 2015.

[4] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009. doi: 10.1109/IISWC.2009.5306797.

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

[6] M. Elsman and M. Dybdal. Compiling a Subset of APL Into a Typed Intermediate Language. In *Procs. Int. Workshop on Lib. Lang. and Compilers for Array Prog. (ARRAY)*. ACM, 2014.

[7] T. Henriksen and C. E. Oancea. A T2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 47–58. ACM, 2013.

[8] A. W. Hsu. The key to a data parallel compiler. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 32–40, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4384-8. doi: 10.1145/2935323.2935331. URL `http://doi.acm.org/10.1145/2935323.2935331`.

[9] K. E. Iverson. *A Programming Language*. John Wiley and Sons, Inc, May 1962.

[10] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3):157–174, 2012. ISSN 0167-8191. doi: 10.1016/j.parco.2011.09.001.

[11] B. Legrand. *Mastering Dyalog APL*. Dyalog Limited, November 2009.

[12] H. Urms and A. S. Kiehn. Compiling TAIL to Futhark (bachelor's thesis), 2015.

			TAIL Futhark		Hand-written Futhark	
Benchmark	**Problem size**	**TAIL C**	**Sequential**	**Parallel**	**Sequential**	**Parallel**
Integral	$N = 10,000,000$	241.90	171.51	0.13	172.93	0.13
Signal	$N = 50,000,000$	1312.73	696.52	2.01	326.79	0.82
Game of Life	$1200 \times 1200, N = 100$	1375.13	2061.52	17.11	1671.05	12.99
Easter	$N = 10,000,000$	315.77	361.64	1.77	180.30	0.80
Black-Scholes	$N = 10,000,000$	6016.47	6944.40	11.14	5285.38	8.81
Sobol MC-π	$N = 10,000,000$	465.13	314.99	36.39	153.03	1.69
HotSpot	$512 \times 512, N = 360$	1338.13	1372.00	297.62	729.36	17.50
Mandelbrot1	$1000 \times 1000, N = 255$	-	940.80	55.01	1472.54	61.80
Mandelbrot2	$1000 \times 1000, N = 255$	-	1109.49	1.42	1124.14	1.39

Table 1: Benchmark timings in miliseconds. The timings are averages over 30 executions. TAIL C is the APL-compiler using a sequential C-code backend; TAIL Futhark is the APL-compiler using Futhark, which is then compiled to either sequential code or parallel GPU code.

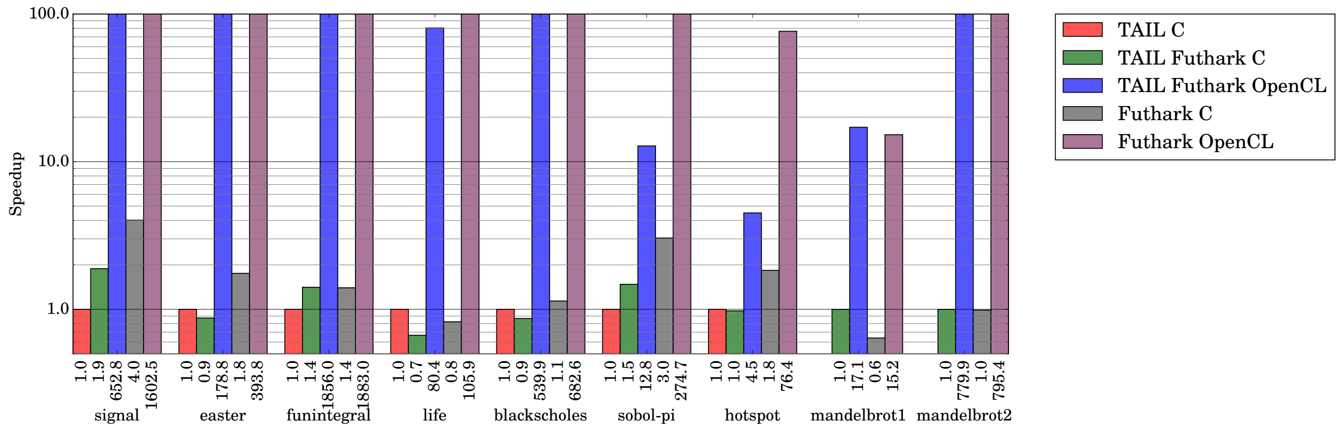


Figure 10: Normalized speedup with respect to sequential C-compiled code.