# 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, urmshenrik@gmail.com, danielgavin4@msn.com, hjalte.diku@gmail.com, mael@di.ku.dk, cosmin.oancea@di.ku.dk

## Abstract

This paper demonstrates 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, empirical evaluation shows that the GPGPU translation achieves speedups up to hundreds of times faster than 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

A major challenge of modern computing is allowing programmers efficiently to utilize parallel hardware using high-level compositional programming techniques.

One approach to tackling this challenge is *parallelizing optimizing compilers*, which take sequential programs as input, and output corresponding parallel programs. While this approach can be shown to be successful in some cases, very often the parallelism present in the original algorithm has vanished in the process of the programmer expressing it 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 (GPUs).

The toolbox employed includes

1. Futhark [7, 8], a 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.
- 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 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.
- Finally, on nine APL applications, we report GPU speedups with a geometric mean 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 character 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`-

*FHPC'16* September 14, 2016, Nara, Japan
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4433-3/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2975991.2975997

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

**Figure 1:** Grammar for the TAIL language.

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 ($\kappa$), shape types ($\rho$), types ($\tau$), and type schemes ($\sigma$). Shape types ($\rho$) are considered identical upto associativity and commutativity of $+$ and upto evaluation of constant shape-type expressions involving $+$. Types ($\tau$) include a type for multi-dimensional arrays of rank $\rho$, written $[\kappa]^{\rho}$, a type for vectors of a specific length $\rho$, 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 $\kappa$ to denote the scalar array type $[\kappa]^0$. Type schemes $\sigma$ 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 scheme 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 scheme is parameterized over four type parameters $\alpha_1$, $\alpha_2$, $\beta$, and $\gamma$, where $\alpha_1$, $\alpha_2$, and $\beta$, denote base types and $\gamma$ a rank. Notice, as mentioned, that we, for instance, write $\alpha_1$ 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.[1] 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 an instance 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.

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 [10, 12].

[1] 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.

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 barebones 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 machine-agnostic, but optimising compilers for both CPUs and GPUs have been implemented.

As with 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].

When translating from another language to Futhark, we wish to make use of as many of the built-in language constructs as possible, 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 5 presents the main part of the compilation scheme and Figure 6 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.

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

**Figure 3:** Simplified Futhark syntax.

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.

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 4.

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, APL and TAIL are not. At any point in execution, a TAIL program can read and write to arbitrary files. However, usually side effects are only used 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 performing the main translation step. An example is shown in Figure 7. If any other side-effecting operations are found, compilation will fail.

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

## 5. Bridging APL and Python

A collateral advantage of migrating the application's code to GPUs is that it becomes performance-wise irrelevant what 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.[2]

We demonstrate this view by reporting a straightforward code generator from Futhark to Python+PyOpenCL [11]:

- A Futhark program is translated to a Python module. While the Futhark code generator supports multiple 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 that converts its parameters to the types expected by the translation. For example a Numpy array argument is transfomed to a PyOpenCL (device) buffer, and a device-buffer argument is left unchanged. Array return values are delivered as PyOpenCL arrays.
- The Python code issues calls to the PyOpenCL library, which is a thin wrapper around the standard OpenCL library. These calls are used to transfer code (expressed in OpenCL C) and data to the GPU. These implementation details are hidden from the users of the generated Python module.

As a demonstration, computational kernels for Game of Life and Mandelbrot were implemented in APL, compiled to Futhark and from there to Python+PyOpenCL code, and used from a Python program that performs real-time visualization and user interaction by the use of the Pygame library. For example, the user can interactively insert additional glider guns to Game of Life, and can zoom and scroll the Mandelbrot set. 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

[2] This is similar to work in computer-algebra systems [13], where, for example, efficient and expressive languages, suitable for library design were inter-operated with top-level systems based on user-interface priorities.

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

$$
\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 []^{r-1} t_2 \rrbracket\; (\llbracket []^{r-1} t_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 []^{r-2} t \rrbracket\,(\llbracket []^{r-1} t \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 []^{r-1} t_3 \rrbracket\; (\llbracket []^{r-1} t_1 \rrbracket\; x, \llbracket []^{r-1} t_2 \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{rotate}_{[t,r]}\,(i, a)\,\rrbracket & = & \texttt{rotate@}(r-1)\,(\llbracket i \rrbracket, \llbracket a \rrbracket) \\
\llbracket \texttt{cat}_{[t,r]}\,(a_1, a_2)\,\rrbracket & = & \texttt{concat@}(r-1)\,(\llbracket a_1 \rrbracket, \llbracket a_2 \rrbracket) \\
\llbracket \texttt{transp}_{[t,r]}\,(a)\,\rrbracket & = & \texttt{rearrange}\,((r-1, \cdots, 0), \llbracket 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{[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{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{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 []^{r-1} t \rrbracket\,(\llbracket []^{r-2} t \rrbracket\; x,\; \llbracket []^{r-1} t \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}\;\texttt{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{[size@0}\,(x), \ldots, \texttt{size@}(r-1)\,(x)\texttt{]}
\end{array}
$$

**Figure 5:** Conversion rules for expressions. For each rule, $x$ and $y$ are considered fresh.

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

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

**Figure 6:** 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 7:** Handling IO operations.

takes hundreds of miliseconds (see Table 1). These demos are part of our benchmark suite (see Section 6.1). Our benchmark suite also compiles all benchmark programs using our Python+PyOpenCL code generator, in order to quantify the performance loss compared to generating C.

## 6. Performance

We evaluate the performance of the Futhark backend on a small number of benchmark programs. Each is compiled to sequential C using TAIL's built-in code generator, as well as to Futhark. The resulting Futhark program is then compiled to both sequential C, parallel C+OpenCL, and parallel Python+PyOpenCL code. To measure the quality of the TAIL-to-Futhark translation, we have also implemented each benchmark by hand in Futhark, and in sequential C code as a baseline.

### 6.1 Experimental Methodology and Hardware

Our experimental setup is publicly available at the URL:

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

We 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 benchmark 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 on Figure 8.

### 6.2 Results and Analysis

*Integral* performs numerical integration via sampling. *Signal* is a signal processing program derived from the APEX benchmark suite [1]. *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* valuates European options using a closed form solution. These applications perform almost no memory accesses, which is the reason for their extreme speedup.

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

*Sobol-$\pi$* calculates $\pi$ based on Monte Carlo simulation using Sobol sequences, 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. This is due to a deficiency in the Futhark compiler that leads it to generate code with non-coalesced memory accesses.

*Mandelbrot* computes the Mandelbrot set for $1000 \times 1000$ points 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. 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. 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*, originally from the Rodinia [4] benchmark suite, constituting an outer sequential time-series loop surrounding an inner rank-1 stencil. It is similar to the Game of Life benchmark, but with complicated edge conditions implemented using array concatenation. Our implementation is ported from an ELI implementation by WM Ching et al [5]. PyOpenCL performance is disappointing due to a high number of kernel launches. The generated code sets the OpenCL kernel parameters anew before every invocation of the kernel, even if they are the same for every launch. In direct OpenCL, this is a cheap operation, but it is slow in PyOpenCL. This pattern also occurs for Game of Life, but the corresponding kernel takes much fewer parameters, somewhat ameliorating the problem. We are investigating improvements.

## 7. Related Work

This paper is a successor to previous work on compiling APL to parallel GPU code [3]. We are now able to compile every program in the benchmark 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 [15].

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 [9], which is itself written in a data-parallel style in APL. Related to TAIL is Remora [14] which uses dependent types to assigning a static semantics to an APL-like language.

## 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 interoperate 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.

| | | | | TAIL Futhark | | | Hand-written Futhark | | |
|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | **Problem size** | **Baseline C** | **TAIL C** | **Sequential** | **OpenCL** | **PyOpenCL** | **Sequential** | **Parallel** | **PyOpenCL** |
| Integral | $N = 10,000,000$ | 244.86 | 235.10 | 153.40 | 0.27 | 0.44 | 165.45 | 0.27 | 0.46 |
| Signal | $N = 50,000,000$ | 306.93 | 712.20 | 622.56 | 1.49 | 1.59 | 284.37 | 0.84 | 0.93 |
| Game of Life | $1200^2, N = 100$ | 847.74 | 1274.03 | 1689.76 | 16.11 | 19.11 | 1433.93 | 13.65 | 15.52 |
| Easter | $N = 10,000,000$ | 136.92 | 321.80 | 315.83 | 1.80 | 2.13 | 165.39 | 0.94 | 1.06 |
| Black-Scholes | $N = 10,000,000$ | 5689.88 | 5396.10 | 5994.86 | 9.13 | 7.95 | 5418.89 | 8.63 | 6.91 |
| Sobol MC-$\pi$ | $N = 10,000,000$ | 144.95 | 368.67 | 281.61 | 23.68 | 23.87 | 71.85 | 2.04 | 4.70 |
| HotSpot | $512^2, N = 360$ | 874.77 | 1210.57 | 874.73 | 16.84 | 46.88 | 645.44 | 16.10 | 77.35 |
| Mandelbrot1 | $1000^2, N = 255$ | 598.85 | - | 889.63 | 53.85 | 55.38 | 1530.42 | 57.30 | 58.14 |
| Mandelbrot2 | $1000^2, N = 255$ | 1046.13 | - | 1086.21 | 1.52 | 1.63 | 973.58 | 1.50 | 1.57 |

**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, the latter as either C+OpenCL or Python+PyOpenCL.

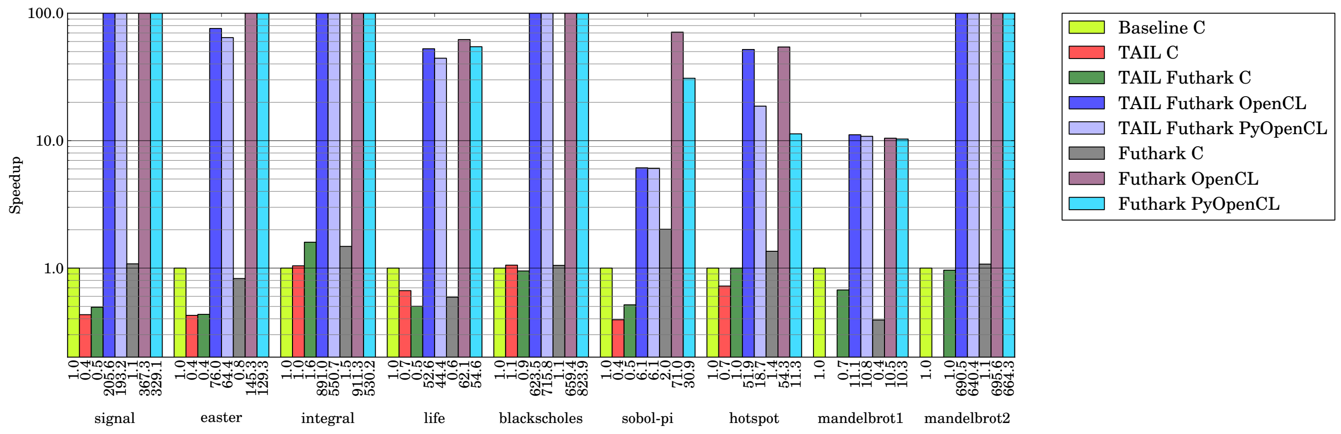

**Figure 8:** Relative speedup compared to sequential hand-written C code.

## 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] T. Henriksen, K. F. Larsen, and C. E. Oancea. Design and gpgpu performance of futhark's redomap construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pages 17–24. ACM, 2016.

[9] 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`.

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

[11] 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.

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

[13] C. E. Oancea and S. M. Watt. Domains and Expressions: An Interface between Two Approaches to Computer Algebra. In *Proc. Int. Symp. on Symbolic and Alg. Comp. (ISSAC)*, pages 261–269. ACM, 2005.

[14] J. Slepak, O. Shivers, and P. Manolios. An array-oriented language with static rank polymorphism. In *European Symposium on Programming Languages and Systems*, pages 27–46. Springer, 2014.

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