Compiling a Subset of APL into Performance Efficient GPU Programs

Martin Elsman, DIKU, University of Copenhagen

Joined work with Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, and Cosmin Oancea

@ Dyalog’16
Motivation

**Goal:**
High-performance at the fingertips of domain experts.

**Why APL:**
APL provides a **powerful and concise** notation for array operations.

APL programs are inherently parallel - not just parallel, but **data-parallel**.

There is lots of APL code around - some of which is looking to run faster!

**Challenge:**
APL is dynamically typed. To generate efficient code, we need **type inference**:

- Functions are **rank-polymorphic**.
- Built-in operations are overloaded.
- Types are **value-sensitive** (e.g., any integer 0, 1 is considered boolean).

Type inference algorithm compiles APL into a **typed array intermediate language** called TAIL (ARRAY'14).

APL 🔄 TAIL 🔄 Futhark
APL Supported Features

Dfns-syntax for functions and operators (incl. trains).

Dyalog APL compatible built-in operators and functions (limitations apply).

Scalar extensions, identity item resolution, overloading resolution.

Limitations:
- Static scoping and static rank inference
- Limited support for nested arrays
- Whole-program compilation
- No execute!
- Type system **expressive enough** for many APL primitives.
- Simplify certain primitives into other constructs...
- Multiple backends...
**APL:**

mean ← +/÷×
var ← mean({ω*2} ← mean)
stddev ← {ω*0.5} var
all ← mean, var, stddev
□ ← all 54 44 47 53 51 48 52 53 52 49 48

Type check: Ok
Evaluation:
[3](50.0909,8.8099,2.9681)

**TAIL:**

let v2:[int]1 = [54,44,47,53,51,48,52,53,52,49,48,52] in
let v1:[int]0 = 11 in
let v15:[double]1 = each(fn v14:[double]0 =>
subd(v14,divd(i2d(reduce(addi,0,v2)),i2d(v1))),each(i2d,v2)) in
let v17:[double]1 = each(fn v16:[double]0 => powd(v16,2.0),v15) in
let v21:[double]1 = divd(reduce(add,0.0,v17),i2d(v1)) in
let v31:[double]1 = each(fn v30:[double]0 =>
subd(v30,divd(i2d(reduce(addi,0,v2)),i2d(v1))),each(i2d,v2)) in
let v33:[double]1 = each(fn v32:[double]0 => powd(v32,2.0),v31) in
let v41:[double]1 =
prArrD(cons(divd(i2d(reduce(addi,0,v2)),i2d(v1)),[divd(reduce(add
d,0.0,v33),i2d(v1)),powd(v21,0.5)])) in 0

Simple interpreter
Compiling Primitives

APL:

let v1:[int]2 = reshape([3,2],iotaV(5)) in
let v2:[int]2 = transp(v1) in
let v9:[int]3 = transp2([2,1,3],reshape([3,3,2],v1)) in
let v15:[int]3 = transp2([1,3,2],reshape([3,2,3],v2)) in
let v20:[int]2 = reduce(addi,0,zipWith(muli,v9,v15)) in
let v25:[int]0 = reduce(muli,1,reduce(addi,0,v20)) in
i2d(v25)

Evaluating
Result is []((65780.0)

Notice: Quite a few simplifications happen at TAIL level..

Guibas and Wyatt, POPL’78
Futhark

Pure eager functional language with second-order parallel array constructs.

Support for “imperative-like” language constructs for iterative computations (i.e., graph shortest path).

A sequentialising compiler...

Close to performance obtained with hand-written OpenCL GPU code.

```haskell
fun [int] addTwo ([int] a) = map(+2, a)
fun int sum ([int] a) = reduce(+, 0, a)
fun [int] sumrows([[int]] as) = map(sum, a)
fun int main(int n) =
    loop (x=1) = for i<n do x*(i+1)
    in x
```

Performs general optimisations
- **Constant folding.** E.g., remove branch inside code for `take(n,a)` if `n ≤ ⊃ρa`.
- **Loop fusion.** E.g., fuse the many small “vectorised” loops in idiomatic APL code.

Attempts at flattening nested parallelism
- E.g., reduction (/) inside each (¨).

Allows for indexing and sequential loops
- Needed for indirect indexing and ⍣.

Performs low-level GPU optimisations
- E.g., optimise for coalesced memory accesses.
An Example

**APL:**

\[ f \leftarrow \{ 2 \div \omega + 2 \} \]  
\[ X \leftarrow 1000000 \]  
\[ \text{domain} \leftarrow 10 \times (\iota X) \div X \]  
\[ \text{integral} \leftarrow +/ \ (f \circ \text{domain}) \div X \]

**TAIL:**

```
let domain:<double>1000000 = 
  eachV(fn v4:[double]0 => muld(10.0,v4),
  eachV(fn v3:[double]0 => divd(v3,1000000.0),
  eachV(i2d,iotaV(1000000))) in
let integral:[double]0 = 
  reduce(addd,0.0,
  eachV(fn v9:[double]0 => divd(v9,1000000.0),
  eachV(fn v7:[double]0 => divd(2.0,addd(v7,2.0)),
    domain))) in
integral
```

**Futhark - before optimisation:**

```
let domain = 
  map (fn (t_v4: f64): f64 => 10.0f64*t_v4)
  (map (fn (t_v3: f64): f64 => t_v3/1000000.0f64)
  (map i2d (map (fn (x: int): int => x+1)
    (iota 1000000))) in
let integral = 
  reduce (+) 0.0f64
  (map (fn (t_v9: f64): f64 => t_v9/1000000.0f64)
  (map (fn (t_v7: f64): f64 => 2.0f64/(t_v7+2.0f64))
    domain)) in
integral
```

Notice: TAIL2Futhark compiler is quite straightforward...
Performance Compute-bound Examples

Integral benchmark:

\[ f \leftarrow \{ 2 \div \omega + 2 \} \]  
\[ X \leftarrow 10000000 \]  
\[ \text{domain} \leftarrow 10 \times (\omega X) \div X \]  
\[ \text{integral} \leftarrow \left( f \uparrow \text{domain} \right) \div X \]

- Function \( \backslash x. 2 / (x+2) \)
- Valuation points per unit
- Integrate from 0 to 10
- Compute integral

OpenCL runtimes from an NVIDIA GTX 780
CPU runtimes from a Xeon E5-2650 @ 2.6GHz
Performance Stencils

Life benchmark:

\[
\text{life} \leftarrow \{
\begin{align*}
\text{rs} & \leftarrow \{ (-1\phi \omega) + \omega + 1\phi \omega \} \\
n & \leftarrow (\text{rs} - 1\omega) + (\text{rs} \cdot \omega) + \text{rs} \cdot 1\omega \\
(n=3) & \lor (n=4) \land \omega
\end{align*}
\}
\]

\[
\text{res} \leftarrow +/+ (\text{life} \times 100) \text{ board}
\]
Different Mandelbrot Implementations

**Parallel inner loop:**

*mandelbrot1.apl*

seq for $i < \text{depth}$:
par for $j < n$:
  points[$j$] = $f$(points[$j$])

**Parallel outer loop:**

*mandelbrot2.apl*

par for $j < n$:
p = points[$j$]
seq for $i < \text{depth}$:
p = $f$(p)
points[$j$] = p

---

**Graph 1:**

- Speedup vs. Width and height
- Memory bound

**Graph 2:**

- Speedup vs. Width and height
- Compute bound
Performance Mandelbrot

The chart shows the speedup of different methods for Mandelbrot set computations. The methods compared are Baseline C, TAIL Futhark OpenCL, and Futhark OpenCL. The chart indicates that Futhark OpenCL has the highest speedup, followed by TAIL Futhark OpenCL and Baseline C, respectively. The data points provided are for two datasets: mandelbrot1 and mandelbrot2.
Interoperability Demos

Mandelbrot, Life, AplCam

With Futhark, we can generate reusable modules in various languages (e.g., Python) that internally execute on the GPU using OpenCL.

```futhark
onChannels ← {
  m ← 3 1 2 ∘ w
  m ← (aα h w p m) ; (aα h w p1 m) ; aα h w p2 m
  2 3 1 ∘ 3 h w p m
}
diff ← {
  n ← [degree
    255 \ n×w−1 \ w
  }
  image ← diff onChannels image
```
Related Work

APL Compilers
- Co-dfn compiler by Aaron Hsu. Papers in ARRAY’14 and ARRAY’16.

Type Systems for APL like Languages

Futhark work
- Papers on language and optimisations available from [hiperfit.dk](http://hiperfit.dk).
- Futhark available from [futhark-lang.org](http://futhark-lang.org).

Other functional languages for GPUs
- Accelerate. Haskell library/embedded DSL.
- Obsidian. Haskell embedded DSL.
- FCL. Low-level functional GPU programming. FHPC’16.

Libraries for GPU Execution
- Thrust, cuBLAS, cuSPARSE, ...
Conclusions

- We have managed to get a (small) subset of APL to run efficiently on GPUs.
  - [https://github.com/henrikurms/tail2futhark](https://github.com/henrikurms/tail2futhark).
  - [https://github.com/melsman/apltail](https://github.com/melsman/apltail).

Future Work

- More real-world benchmarks.
- Support a wider subset of APL.
- Improve interoperability...
- Add support for APL “type annotations” for specifying programmer intentions...
mandelbrot1.apl and mandelbrot2.apl

A grid-size in left argument (e.g., (1024 768))
A X-range, Y-range in right argument

mandelbrot1 ← {
    X ← ⍳α ⋄ Y ← ⍳1α
    xRng ← 2×ω ⋄ yRng ← 2×ω
    dx ← ((xRng[2])−xRng[1]) ÷ X
    dy ← ((yRng[2])−yRng[1]) ÷ Y
    cx ← Y X p (xRng[1]) + dx ⋄ iX ⋄ Y
    cy ← ⋄ Y X p (yRng[1]) + dy ⋄ iY
    mandel1 ← {
        a one iteration
        count ← Y X pω[3]
        zzx ← cx + (zx ⋄ zx) − zy ⋄ zy
        zzy ← cy + (zx ⋄ zy) + zx ⋄ zy
        conv ← 4 > (zxzx + zzzx) + zzy ⋄ zzy
        count2 ← count + 1 − conv
        (zxzx zzy count2)
    } pl ← Y X p 0
    N ← 255
    res ← (mandel1 × N) (pl pl pl)
    res[3] ÷ N
}

mandelbrot2 ← {
    X ← ⍳α ⋄ Y ← ⍳1α
    xRng ← 2×ω ⋄ yRng ← 2×ω
    dx ← ((xRng[2])−xRng[1]) ÷ X
    dy ← ((yRng[2])−yRng[1]) ÷ Y
    cxA ← Y X p (xRng[1]) + dx ⋄ iX ⋄ Y
    cyA ← ⋄ Y X p (yRng[1]) + dy ⋄ iY
    N ← 255
    mandel1 ← {
        cx ← α ⋄ cy ← ω
        f ← {
            arg ← ω
            count ← arg[3]
            dummy ← arg[4]
            zx ← cx+(xxx)−(yxy)
            zy ← cy+(xxx)+(xyy)
            conv ← 4 > (zx ⋄ zx) + zy ⋄ zy
            count2 ← count + 1 − conv
            (zx zy count2 dummy)
        }
        res ← (f × N) (0 0 0 'dummy') A N iterations
        res[3]
    } res ← cxA mandel1− cyA
    res ÷ N
}