APL on GPUs – A Progress Report with a Touch of Machine Learning

Martin Elsman, DIKU, University of Copenhagen
Joined work with Troels Henriksen and Cosmin Oancea

@ Dyalog’17, Elsinore
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.
- Some **subtyping** is required (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 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!

```apl
else ← {(αα×α)(ωω×(~α))ω}
mean ← +/÷≠
```
- Type system **expressive enough** for many APL primitives.
- Simplify certain primitives into other constructs...
- Multiple backends...

**Value typing**

\[ \Gamma \vdash e : \tau \]

1. \[ \Gamma \vdash e : \rho \]
2. \[ \Gamma \vdash \text{int} \to \text{int} \]
3. \[ \Gamma \vdash [\text{int}]^\rho \]
4. \[ \Gamma \vdash [\text{double}]^\rho \]

**Expression typing**

\[ \Gamma \vdash e : \tau \]

1. \[ \Gamma \vdash \text{add} \to \text{int} \to \text{int} \]
2. \[ \Gamma \vdash \text{iota} \to [\text{int}]^\rho \]
3. \[ \Gamma \vdash \text{each} \to \text{int} \to \text{int} \]
4. \[ \Gamma \vdash \text{shape} \to \text{int} \to \text{int} \]
5. \[ \Gamma \vdash \text{reshape} \to \text{int} \to \text{int} \]
6. \[ \Gamma \vdash \text{reverse} \to \text{int} \to \text{int} \]
7. \[ \Gamma \vdash \text{rotate} \to \text{int} \to \text{int} \]
8. \[ \Gamma \vdash \text{transp} \to \text{int} \to \text{int} \]
9. \[ \Gamma \vdash \text{take} \to \text{int} \to \text{int} \]
10. \[ \Gamma \vdash \text{drop} \to \text{int} \to \text{int} \]
11. \[ \Gamma \vdash \text{zipwith} \to \text{int} \to \text{int} \]
12. \[ \Gamma \vdash \text{cat} \to \text{int} \to \text{int} \]
13. \[ \Gamma \vdash \text{cons} \to \text{int} \to \text{int} \]
14. \[ \Gamma \vdash \text{snoc} \to \text{int} \to \text{int} \]

**Small Step Reductions**

\[ e \leftarrow e' \]

\[ \frac{e \leftarrow E \neq \mathbb{E}}{E[e] \Rightarrow E'[e']} \]

\[ \frac{e \leftarrow E \neq \mathbb{E}}{E[e] \Rightarrow \mathbb{E}} \]

\[ \text{let } x = v \text{ in } e 
\Rightarrow e[v/x] \]

\[ \Gamma \vdash [\text{int}]^\rho \]

\[ \Gamma \vdash [\text{double}]^\rho \]

\[ \Gamma \vdash \text{add}(i_1, i_2) \leftarrow i \]

\[ \Gamma \vdash \text{add}(d_1, d_2) \leftarrow d \]

\[ \Gamma \vdash \text{iota}(n) \leftarrow [1, \ldots, n]^\rho \]

\[ \Gamma \vdash \text{reshape}(\delta, e) \]

\[ \Gamma \vdash e \to \tau \]

\[ \Gamma \vdash \text{reduce}(\delta, e) \]

\[ \Gamma \vdash e \to e' \]

\[ \Gamma \vdash e : \tau \]

\[ \Gamma \vdash \text{add} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{iota} : [\text{int}]^\rho \]

\[ \Gamma \vdash \text{each} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{shape} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{reshape} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{reverse} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{rotate} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{transp} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{take} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{drop} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{zipwith} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{cat} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{cons} : \text{int} \to \text{int} \to \text{int} \]

\[ \Gamma \vdash \text{snoc} : \text{int} \to \text{int} \to \text{int} \]
**TAIL Example**

**APL:**

\[
\text{mean} \leftarrow +/\div\# \\
\text{var} \leftarrow \text{mean} \{\omega*2\} \leftarrow \text{mean} \\
\text{stddev} \leftarrow \{\omega*0.5\} \text{ var} \\
\text{all} \leftarrow \text{mean, var, stddev} \\
\]

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

**TAIL:**

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

Guibas and Wyatt, POPL’78

TAIL:

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)

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

```
let addTwo (a:[]i32) : []i32 = map (+2) a
let sum (a:[]i32) : i32 = reduce (+) 0 a
let sumrows(a:[][]i32) : []i32 = map sum a
let main(n:i32) : i32 =
  loop x=1 for i < n do x * (i+1)
```

**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:**

\[ f \leftarrow \{ 2 \div \omega + 2 \} \quad \text{a Function } \lambda x. \ 2 / (x+2) \]
\[ X \leftarrow 1000000 \quad \text{a Valuation points per unit} \]
\[ \text{domain} \leftarrow 10 \times (iX) \div X \quad \text{a Integrate from 0 to 10} \]
\[ \text{integral} \leftarrow +/ \ (f"\text{domain})\div X \quad \text{a Compute integral} \]

**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,addv(v7,2.0)),
    domain))) in
integral

---

**Futhark - before optimisation:**

let domain =
    map (\ (t_v4: f64): f64 -> 10.0f64*t_v4)
        (map (\ (t_v3: f64): f64 -> t_v3/1000000.0f64)
            (map i2d (map (\ (x: int): int -> x+1)
                (iota 1000000))))
let integral =
    reduce (+) 0.0f64
        (map (\ (t_v9: f64): f64 -> t_v9/1000000.0f64)
            (map (\ (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:

\[
\begin{align*}
f & \leftarrow \{ 2 + \omega + 2 \} & \text{a Function } \backslash x. \ 2 / (x+2) \\
X & \leftarrow 10000000 & \text{a Valuation points per unit} \\
\text{domain} & \leftarrow 10 \times (\backslash X) \div X & \text{a Integrate from } 0 \text{ to } 10 \\
\text{integral} & \leftarrow +/ (f"\text{domain})\div X & \text{a Compute integral}
\end{align*}
\]

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

Life benchmark:

```plaintext
life ← { 
  rs ← { (¬1φω) + ω + 1φω }
  n ← (rs ¬1ωω) + (rs ω) + rs 1ωω
  (n=3) v (n=4) ^ ω
}
res ← +/+/ (life * 100) board
```
Performance Mandelbrot

![Performance Mandelbrot Diagram]

- **Baseline C**
- **TAIL Futhark OpenCL**
- **Futhark OpenCL**

### Chart Details:
- **x-axis**: mandelbrot1, mandelbrot2
- **y-axis**: Speedup
- **Values**:
  - mandelbrot1:
    - Baseline C: 1.0
    - TAIL Futhark OpenCL: 11.0
    - Futhark OpenCL: 10.5
  - mandelbrot2:
    - Baseline C: 1.0
    - TAIL Futhark OpenCL: 672.7
    - Futhark OpenCL: 747.5
New Features Since Dyalog’16

Complex number support:

- Mandelbrot one-liner:
- Compared to Dyalog APL, additional parentheses are needed around bindings (←) and around the power operator (×).

akin ← ' #'[1+9>|(ₘ⁻₃×.75⁻₉ₐ⁺₀j₁ₓₐ←(₁+ₙ+₁)/(ₙ⁻₂₈))]

Efficient parallel segmented reductions (Troels’ + Rasmus’ FHPC’17 paper).
- A special segmented reduction form is possible in APL:
  +/20000 10⍴200000    +/100 2000⍴200000

Futhark components (library routines).
- Linear algebra routines, sobol sequences, sorting, random numbers, ...

Many Futhark internal optimisations.
Neural Network for Digit Recognition

Task: Train a 3-layer neural network using back-propagation.

MNIST data set:
- *Training* set size: 50,000 classified images (28x28 pixel intensities; floats)
- *Test* set size: 10,000 classified images

Network:

<table>
<thead>
<tr>
<th>Input layer</th>
<th>Layer 2 (Hidden)</th>
<th>Layer 3 (Output)</th>
</tr>
</thead>
<tbody>
<tr>
<td>784 (28x28)</td>
<td>30 sigmoid neurons</td>
<td>10 sigmoid neurons</td>
</tr>
<tr>
<td><strong>weights:</strong> 30x784 matrix</td>
<td><strong>weights:</strong> 10x30 matrix</td>
<td><strong>biases:</strong> 30 vector</td>
</tr>
</tbody>
</table>
Some APL NN Snippets

- The sigmoid function

\[ \sigma(z) \equiv \frac{1}{1 + e^{-z}} \]

- Turn a digit into a 10d unit vector

\[ \text{from_digit} \leftarrow \{ \omega = ^{-1} + i 10 \} \]

- Predict a digit based on the output

\[ \text{predict_digit} \leftarrow \{ ^{-1} + + (i \neq \omega) \times \omega = / \omega \} \]

- Apply a 3-layer network to an input vector

\[
\text{feedforward3} \leftarrow \\
\{ \\
\text{feedforward} \leftarrow \\
\{ \\
b \leftarrow \alpha[1] \circ \omega \leftarrow \alpha[2] \\
\text{sigmoid} b + w + .\times \omega \\
\} \\
\alpha[2] \text{ feedforward} (\alpha[1] \text{ feedforward} \omega) \\
\} \\
\]

<table>
<thead>
<tr>
<th>Digit</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.23</td>
</tr>
<tr>
<td>1</td>
<td>0.32</td>
</tr>
<tr>
<td>2</td>
<td>0.03</td>
</tr>
<tr>
<td>3</td>
<td>0.45</td>
</tr>
<tr>
<td>9</td>
<td>1.25</td>
</tr>
</tbody>
</table>

sigmoid function
NN Implementation in Futhark

Original in Python - neuralnetworksanddeeplearning.com

Back-propagation algorithm based on stochastic gradient descent:

Pseudo code:
epochs ← 20
N ← (train_data ← random_permute train_data
  batches ← split train_data
  nablas ← ω backprop batches
  ω ← +/nablas
  )*epochs) init_N (28×28) 30 10

Futhark supports arrays of ‘pairs of arrays’, which can be processed in parallel using the generic Futhark map function.

The argument function to map may itself return structured values.
NN Implementation in APL

20x slowdown with respect to native Futhark.

More investigations are needed to identify the performance issues.

400 lines of APL code.

How does Dyalog APL perform on this benchmark?

How should it be written in Dyalog APL for it to hit peak performance?

See https://github.com/melsman/neural-networks-and-deep-learning
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 ⊙ ω
  m ← (αα h w ρ m) ; (αα h w ρ1+m) ; αα h w ρ2+m
  2 3 1 ⊙ 3 h w ρ m
}

diff ← {
  n ← [degree
    255\lfloor n×+ω−1\rfloorω
  }
  image ← diff onChannels image
```
Related Work

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

Type Systems for APL like Languages
- K. Trojahner and C. Grelck. Dependently typed array programs don’t go wrong. NWPT’07.

Futhark work
- Papers on language and optimisations available from hiperfit.dk.
- Futhark available from 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/HIPERFIT/futhark-fhpc16](https://github.com/HIPERFIT/futhark-fhpc16)
- [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...
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 $j < n$:
$p = \text{points}[$j$]$
seq for $i < \text{depth}$:
$p = f(p)$
points[$j$] = $p$

- Memory bound
- Compute bound

![Chart 1: Speedup vs. Width and Height](chart1)
![Chart 2: Speedup vs. Width and Height](chart2)
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 ← →1↓α
xRng ← 2↓ω • yRng ← 2↓ω
dx ← ((xRng[2])−xRng[1]) ÷ X
dy ← ((yRng[2])−yRng[1]) ÷ Y
cx ← Y X ρ (xRng[1]) + dx × iX a real plane
cy ← Y X ρ (yRng[1]) + dy × iY a img plane
mandel1 ← {
  a one iteration
  zx ← Y X ρω[1] • zy ← Y X ρω[2]
  count ← Y X ρω[3] a count plane
  zzx ← cx + (zx × zx) − zy × zy
  zzy ← cy + (zx × zy) + zx × zx
  conv ← 4 > (zzx × zzx) + zzy × zzy
  count2 ← count + 1 − conv
  (zzx zzy count2)
}
pl ← Y X ρ 0 a zero-plane
N ← 255 a iterations
res ← (mandel1 × N) (pl pl pl)
res ← (res[3] ÷ N) a count plane
}

mandelbrot2 ← {
  X ← →α • Y ← →1↓α
  xRng ← 2↓ω • yRng ← 2↓ω
dx ← ((xRng[2])−xRng[1]) ÷ X
  dy ← ((yRng[2])−yRng[1]) ÷ Y
cxA ← Y X ρ (xRng[1]) + dx×iX a real plane
  cyA ← Y X ρ (yRng[1]) + dy×iY a img plane
N ← 255 a iterations
  mandel1 ← {
    cx ← α • cy ← ω
    f ← {
      arg ← ω
      x ← arg[1] • y ← arg[2]
      count ← arg[3]
      dummy ← arg[4]
      zx ← cx+(xxy)-(yxy)
      zy ← cy+(xxy)+(yxy)
      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
}