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

<u>Goal</u>:
High-performance at the fingertips of domain experts.

<u>Why APL</u>:
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!

<u>Challenge</u>:
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 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!

```
else ← {(αα⍣α)(ωω⍣(~α))ω}

mean ← +/÷≢
```

TAIL - as an IL

- Type system ***expressive enough*** for many APL primitives.
- Simplify certain primitives into other constructs…
- Multiple backends...

APL	op(s)		TySc(op)
	`addi,...`	:	$\mathtt{int} \to \mathtt{int} \to \mathtt{int}$
	`addd,...`	:	$\mathtt{double} \to \mathtt{double} \to \mathtt{double}$
⍳	`iota`	:	$\mathtt{int} \to [\mathtt{int}]^1$
¨	`each`	:	$\forall\alpha\beta\gamma.(\alpha \to \beta) \to [\alpha]^\gamma \to [\beta]^\gamma$
/	`reduce`	:	$\forall\alpha\gamma.(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha]^{1+\gamma} \to [\alpha]^\gamma$
⍴	`shape`	:	$\forall\alpha\gamma.[\alpha]^\gamma \to \mathsf{Sh}\ \gamma$
⍴	`reshape0`	:	$\forall\alpha\gamma\gamma'.\mathsf{Sh}\ \gamma' \to [\alpha]^\gamma \to [\alpha]^{\gamma'}$
⍴	`reshape`	:	$\forall\alpha\gamma\gamma'.\mathsf{Sh}\ \gamma' \to \alpha \to [\alpha]^\gamma \to [\alpha]^{\gamma'}$
⌽	`reverse`	:	$\forall\alpha\gamma.[\alpha]^\gamma \to [\alpha]^\gamma$
⌽	`rotate`	:	$\forall\alpha\gamma.\mathtt{int} \to [\alpha]^\gamma \to [\alpha]^\gamma$
⍉	`transp`	:	$\forall\alpha\gamma.[\alpha]^\gamma \to [\alpha]^\gamma$
⍉	`transp2`	:	$\forall\alpha\gamma.\mathsf{Sh}\ \gamma \to [\alpha]^\gamma \to [\alpha]^\gamma$
↑	`take`	:	$\forall\alpha\gamma.\mathtt{int} \to \alpha \to [\alpha]^\gamma \to [\alpha]^\gamma$
↓	`drop`	:	$\forall\alpha\gamma.\mathtt{int} \to [\alpha]^\gamma \to [\alpha]^\gamma$
⊃	`first`	:	$\forall\alpha\gamma.\alpha \to [\alpha]^\gamma \to \alpha$
	`zipWith`	:	$\forall\alpha_1\alpha_2\beta\gamma.(\alpha_1 \to \alpha_2 \to \beta) \to [\alpha_1]^\gamma \to [\alpha_2]^\gamma \to [\beta]^\gamma$
,	`cat`	:	$\forall\alpha\gamma.[\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1}$
,	`cons`	:	$\forall\alpha\gamma.[\alpha]^{\gamma} \to [\alpha]^{\gamma+1} \to [\alpha]^{\gamma+1}$
,	`snoc`	:	$\forall\alpha\gamma.[\alpha]^{\gamma+1} \to [\alpha]^{\gamma} \to [\alpha]^{\gamma+1}$

Value typing $\boxed{\Gamma \vdash v : \tau}$

$$\frac{\vdash \delta : \rho}{\Gamma \vdash [\vec{i}]^\delta : [\mathtt{int}]^\rho}\ (7) \qquad \frac{\vdash \delta : \rho}{\Gamma \vdash [\vec{r}]^\delta : [\mathtt{double}]^\rho}\ (8)$$

$$\frac{}{\Gamma \vdash [\vec{i}]^{\langle n \rangle} : \mathsf{Sh}\ n}\ (9) \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\ (10)$$

$$\frac{}{\Gamma \vdash [n]^{\langle 1 \rangle} : \mathsf{VI}\ n}\ (11) \qquad \frac{}{\Gamma \vdash n : \mathsf{I}\ n}\ (12)$$

Expression typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \mathsf{I}\ n}{\Gamma \vdash [e] : \mathsf{VI}\ n}\ (13) \qquad \frac{\Gamma(x) \geq \tau}{\Gamma \vdash x : \tau}\ (14) \qquad \frac{\tau \subseteq \tau' \quad \Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'}\ (15)$$

$$\frac{\Gamma \vdash e_i : \kappa \quad i = [0;n[}{\Gamma \vdash [\vec{e}^{(n)}] : [\kappa]^1}\ (16) \qquad \frac{\Gamma \vdash e_i : \mathtt{int} \quad i = [0;n[}{\Gamma \vdash [\vec{e}^{(n)}] : \mathsf{Sh}\ n}\ (17)$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\ (18) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \mathrm{fv}(\vec{\alpha}\vec{\gamma}) \cap \mathrm{fv}(\Gamma, \tau') = \emptyset \quad \Gamma, x : \forall\vec{\alpha}\vec{\gamma}.\tau \vdash e_2 : \tau'}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau'}\ (19)$$

$$\frac{\mathrm{TySc}(op) \geq \tau_0 \to \ldots \to \tau_{(n-1)} \to \tau \quad \mathrm{arity}(op) = n \quad \Gamma \vdash e_i : \tau_i \quad i = [0;n[}{\Gamma \vdash op(\vec{e}^{(n)}) : \tau}\ (20)$$

Small Step Reductions $\boxed{e \hookrightarrow e'/\mathbf{err}}$

$$\frac{e \hookrightarrow e' \quad E \neq [\cdot]}{E[e] \hookrightarrow E[e']}\ (21) \qquad \frac{e \hookrightarrow \mathbf{err} \quad E \neq [\cdot]}{E[e] \hookrightarrow \mathbf{err}}\ (22)$$

$$\frac{}{\mathtt{let}\ x = v\ \mathtt{in}\ e \hookrightarrow e[v/x]}\ (23) \qquad \frac{}{(\lambda x.e)\ v \hookrightarrow e[v/x]}\ (24)$$

$$\frac{}{[\vec{a}^{(n)}] \hookrightarrow [\vec{a}^{(n)}]^{\langle n \rangle}}\ (25)$$

$$\frac{i = i_1 + i_2}{\mathtt{addi}(i_1, i_2) \hookrightarrow i}\ (26) \qquad \frac{d = d_1 + d_2}{\mathtt{addd}(d_1, d_2) \hookrightarrow d}\ (27)$$

$$\frac{n \geq 0}{\mathtt{iota}(n) \hookrightarrow [1, \cdots, n]^{\langle n \rangle}}\ (28) \qquad \frac{n < 0}{\mathtt{iota}(n) \hookrightarrow \mathbf{err}}\ (29)$$

$$\frac{e = [v_f\ a_0, \cdots, v_f\ a_{(n-1)}]}{\mathtt{each}(v_f, [\vec{a}^{(n)}]^\delta) \hookrightarrow \mathtt{reshape0}(\delta, e)}\ (30)$$

$$\frac{\delta = \langle \vec{n}, m \rangle \quad k = \mathrm{product}(\vec{n}) \quad i = [0;m[\quad e_i = v_f\ a_{(i*k)}\ (\cdots(v_f\ a_{(i*k+m-1)}\ v)\cdots)}{\mathtt{reduce}(v_f, v, [\vec{a}^{(n)}]^\delta) \hookrightarrow \mathtt{reshape0}(\langle \vec{n} \rangle, [\vec{e}^{(k)}])}\ (31)$$

$$\frac{m = \mathrm{product}(\delta') \quad f(i) = i \bmod n \quad n > 0}{\mathtt{reshape}(\delta', a, [\vec{a}^{(n)}]^\delta) \hookrightarrow [a_{f(0)}, \cdots, a_{f(m-1)}]^{\delta'}}\ (32)$$

$$\frac{m = \mathrm{product}(\delta') \quad a_i = a \quad i = [0;m[}{\mathtt{reshape}(\delta', a, []^\delta) \hookrightarrow [a_0, \cdots, a_{(m-1)}]^{\delta'}}\ (33)$$

$$\frac{\delta' = \mathrm{rev}(\delta) \quad f = \mathrm{fromSh}_{\delta'}\ o\ \mathrm{rev}\ o\ \mathrm{toSh}_\delta}{\mathtt{transp}([\vec{a}^{(n)}]^\delta) \hookrightarrow [a_{f(0)}, \cdots, a_{f(n-1)}]^{\delta'}}\ (34)$$

TAIL Example

APL → TAIL → Futhark

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)

Simple interpreter

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]0 = divd(reduce(addd,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
```

Compiling Primitives

APL → **TAIL** → Futhark

APL:

Guibas and Wyatt, POPL'78

```
dot ← {
  WA ← (1↓ρω),ρα
  KA ← (⊃ρρα)-1
  VA ← ι ⊃ ρWA
  ZA ← (KA⌽¯1↓VA),¯1↑VA
  TA ← ZA⍉WAρα
  WB ← (¯1↓ρα),ρω
  KB ← ⊃ ρρα
  VB ← ι ⊃ ρWB
  ZB0 ← (-KB) ↓ KB ⌽ ι(⊃ρVB)
  ZB ← (¯1↓(ι KB)),ZB0,KB
  TB ← ZB⍉WBρω
  αα / TA ωω TB
}

A ← 3 2 ρ ι 5
B ← ⍉ A
R ← A + dot × B
R2 ← ×/ +/ R
```

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)
```

Evaluating
Result is [](65780.0)

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.

```
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 → TAIL → Futhark

APL:

```
f ← { 2 ÷ ω + 2 }                ⍝ Function \x. 2 / (x+2)
X ← 1000000                      ⍝ Valuation points per unit
domain ← 10 × (⍳X) ÷ X           ⍝ Integrate from 0 to 10
integral ← +/ (f¨domain)÷X       ⍝ Compute integral
```

TAIL:

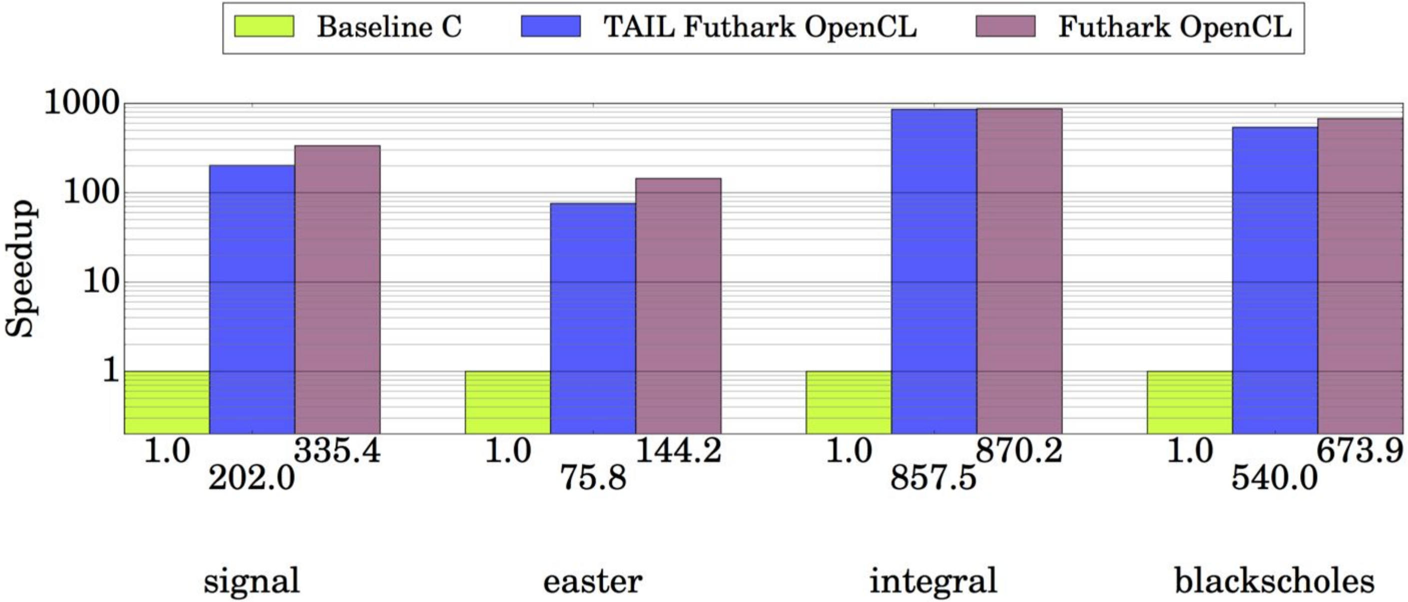
```
let domain:<double>1000000 =
  eachV(fn v4:[double]0 => muld(10.0,v4),
   eachV(fn v3:[double]0 => divd(v3,1000000.0),
    eachV(i2d,iotaV(1000000)))) in
let integral:[double]0 =
  reduce(addd,0.0,
   eachV(fn v9:[double]0 => divd(v9,1000000.0),
    eachV(fn v7:[double]0 => divd(2.0,addd(v7,2.0)),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

Notice: TAIL2Futhark compiler is quite straightforward...

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
```

Performance Compute-bound Examples

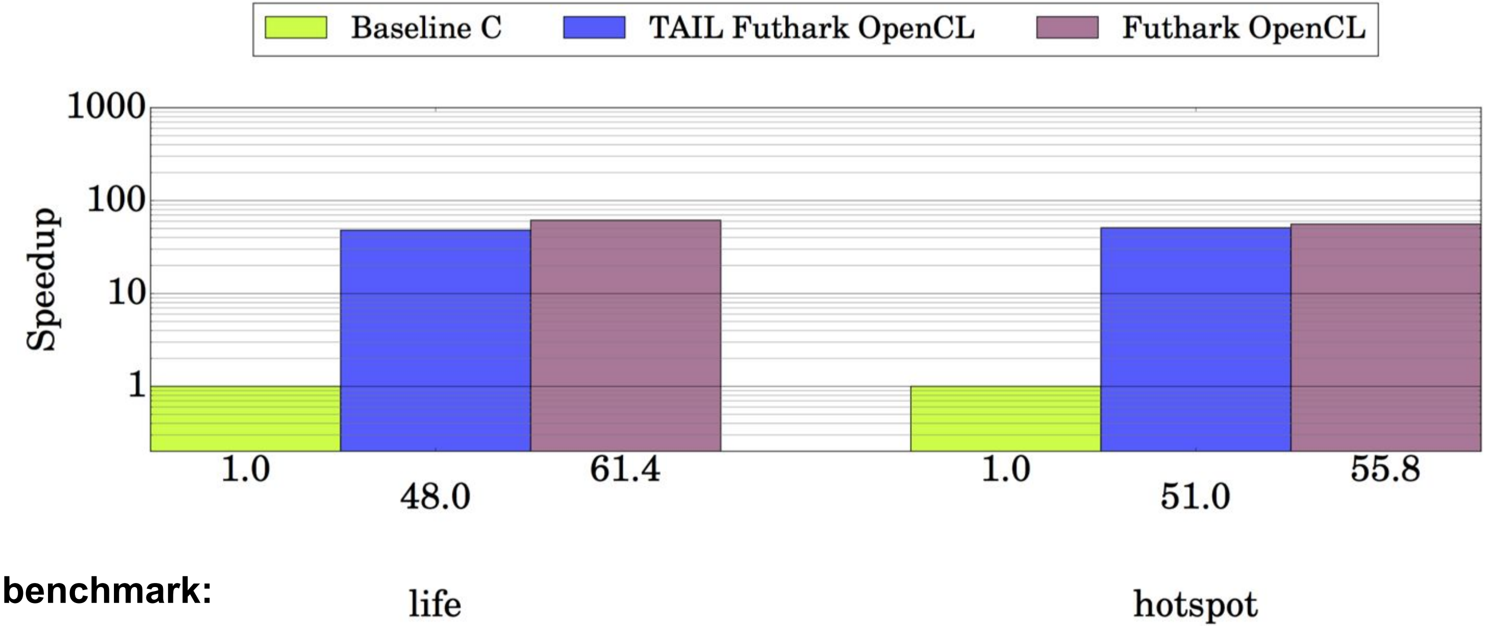


Integral benchmark:

```
f ← { 2 ÷ ⍵ + 2 }              ⍝ Function \x. 2 / (x+2)
X ← 10000000                   ⍝ Valuation points per unit
domain ← 10 × (⍳X) ÷ X         ⍝ Integrate from 0 to 10
integral ← +/ (f¨domain)÷X     ⍝ Compute integral
```

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

Performance Stencils



Life benchmark:

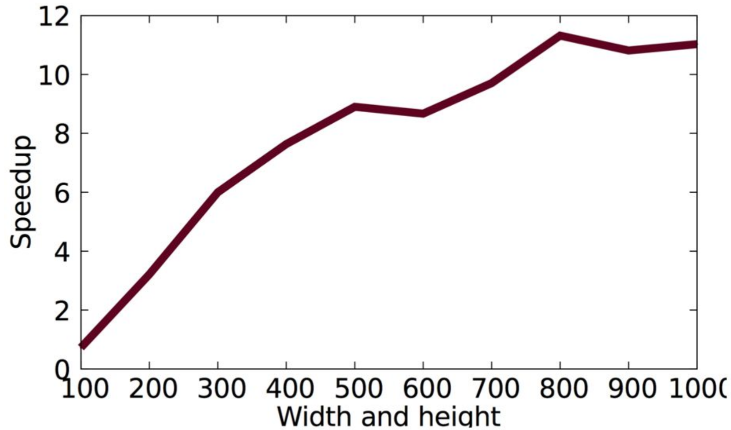
```
life ← {
  rs ← { (¯1⌽ω) + ω + 1⌽ω }
  n ← (rs ¯1⊖ω) + (rs ω) + rs 1⊖ω
  (n=3) ∨ (n=4) ∧ ω
}
res ← +/+/ (life ⍣ 100) board
```

Different Mandelbrot Implementations

Parallel inner loop:
mandelbrot1.apl

```
seq for i < depth:
  par for j < n:
    points[j] = f(points[j])
```

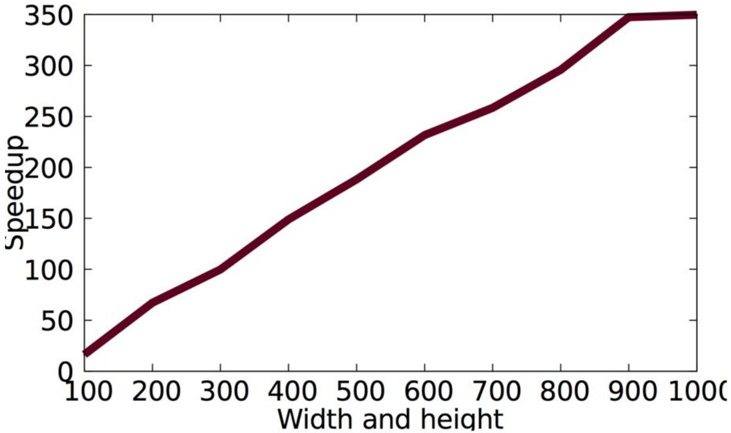
Memory bound



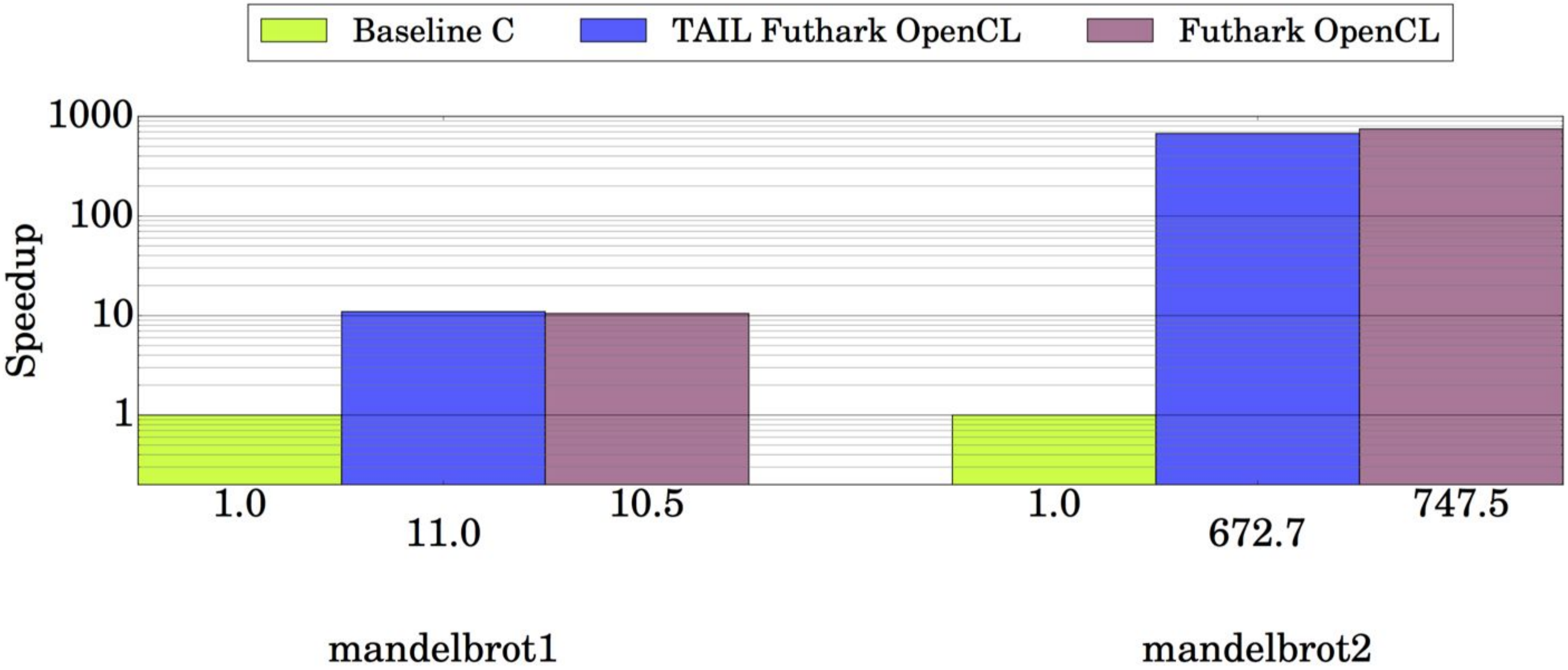
Parallel outer loop:
mandelbrot2.apl

```
par for j < n:
  p = points[j]
  seq for i < depth:
    p = f(p)
  points[j] = p
```

Compute bound



Performance Mandelbrot

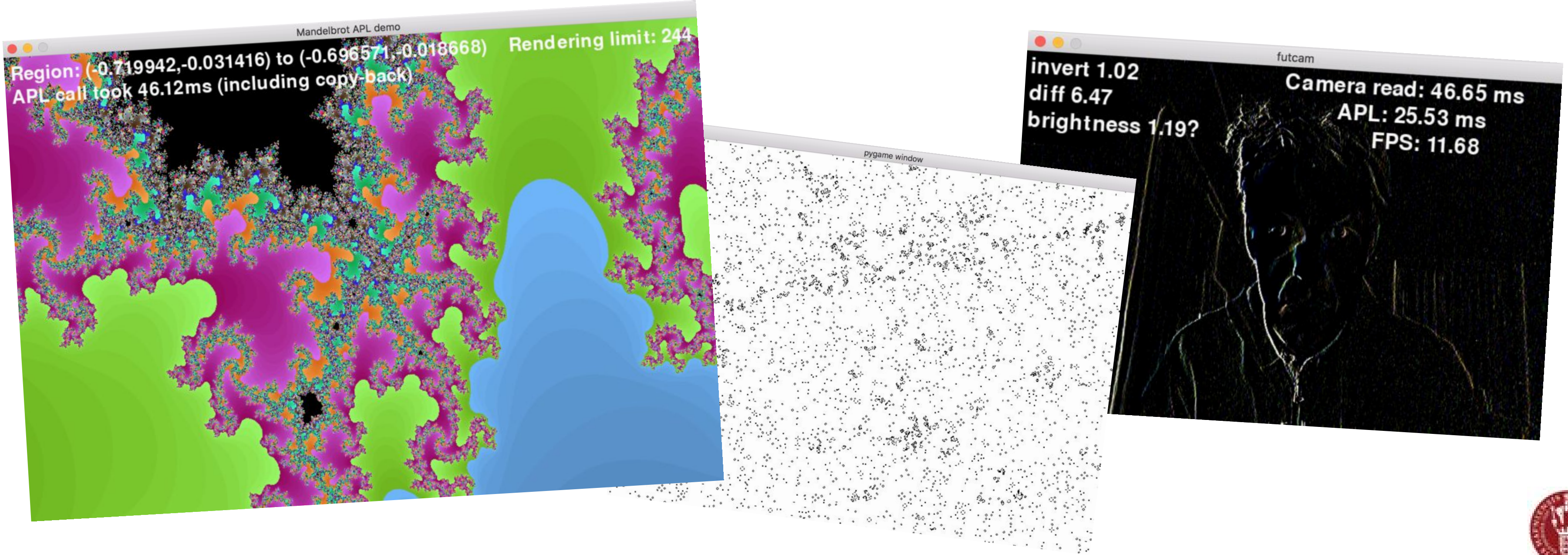


Interoperability Demos

Mandelbrot, Life, AplCam

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

```
onChannels ← {
  m ← 3 1 2 ⍉ ⍵
  m ← (⍺⍺ h w ⍴m) ⍪ (⍺⍺ h w ⍴1↓m) ⍪ ⍺⍺ h w ⍴2↓m
  2 3 1 ⍉ 3 h w ⍴ m
}
diff ← {
  n ← ⌈degree
  255⌊n×+⍵-1⌽⍵
}
image ← diff onChannels image
```



Related Work

APL Compilers

- Co-dfns compiler by Aaron Hsu. Papers in ARRAY'14 and ARRAY'16.
- C. Grelck and S.B. Scholz. *Accelerating APL programs with SAC*. APL'99.
- R. Bernecky. *APEX: The APL parallel executor*. MSc Thesis. University of Toronto. 1997.
- L.J. Guibas and D.K. Wyatt. *Compilation and delayed evaluation in APL*. POPL'78.

Type Systems for APL like Languages

- K. Trojahner and C. Grelck. *Dependently typed array programs don't go wrong*. NWPT'07.
- J. Slepak, O. Shivers, and P. Manolios. *An array-oriented language with static rank polymorphism*. ESOP'14.

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/henrikurms/tail2futhark.
    - 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...

HIPERFIT

mandelbrot1.apl and mandelbrot2.apl

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

mandelbrot1 ← {
  X ← ⊃⍺ ⋄ Y ← ⊃1↓⍺
  xRng ← 2↑⍵ ⋄ yRng ← 2↓⍵
  dx ← ((xRng[2])-xRng[1]) ÷ X
  dy ← ((yRng[2])-yRng[1]) ÷ Y
  cx ← Y X ⍴ (xRng[1]) + dx × ⍳X    ⍝ real plane
  cy ← ⍉ X Y ⍴ (yRng[1]) + dy × ⍳Y  ⍝ img plane
  mandel1 ← {                       ⍝ one iteration
    zx ← Y X ⍴⍵[1] ⋄ zy ← Y X ⍴⍵[2]
    count ← Y X ⍴ ⍵[3]              ⍝ count plane
    zzx ← cx + (zx × zx) - zy × zy
    zzy ← cy + (zx × zy) + zx × zy
    conv ← 4 > (zzx × zzx) + zzy × zzy
    count2 ← count + 1 - conv
    (zzx zzy count2)
  }
  pl ← Y X ⍴ 0                      ⍝ zero-plane
  N ← 255                           ⍝ iterations
  res ← (mandel1 ⍣ N) (pl pl pl)
  res[3] ÷ N                        ⍝ count plane
}
```

```
mandelbrot2 ← {
  X ← ⊃⍺ ⋄ Y ← ⊃1↓⍺
  xRng ← 2↑⍵ ⋄ yRng ← 2↓⍵
  dx ← ((xRng[2])-xRng[1]) ÷ X
  dy ← ((yRng[2])-yRng[1]) ÷ Y
  cxA ← Y X ⍴ (xRng[1])+dx×⍳X    ⍝ real plane
  cyA ← ⍉ X Y ⍴ (yRng[1])+dy×⍳Y  ⍝ img plane
  N ← 255                        ⍝ iterations
  mandel1 ← {
    cx ← ⍺ ⋄ cy ← ⍵
    f ← {
      arg ← ⍵
      x ← arg[1] ⋄ y ← arg[2]
      count ← arg[3]
      dummy ← arg[4]
      zx ← cx+(x×x)-(y×y)
      zy ← cy+(x×y)+(x×y)
      conv ← 4 > (zx × zx) + zy × zy
      count2 ← count + 1 - conv
      (zx zy count2 dummy)
    }
    res ← (f⍣N) (0 0 0 'dummy')  ⍝ N iterations
    res[3]
  }
  res ← cxA mandel1¨ cyA
  res ÷ N
}
```