# FCL: A low-level functional GPU language

(Very much work in progess)

Martin Dybdal
dybber@dybber.dk

HIPERFIT
DIKU
University of Copenhagen

4 March 2016

# Previously: APL → TAIL

```
pi ← {
 x ← ?ωρ0
 y ← ?ωρ0
 dists ← (x*2) + y*2
 4×(+/1>dists)÷ω
}
pi 1000000
    3.142668
```

# Previously: APL → TAIL

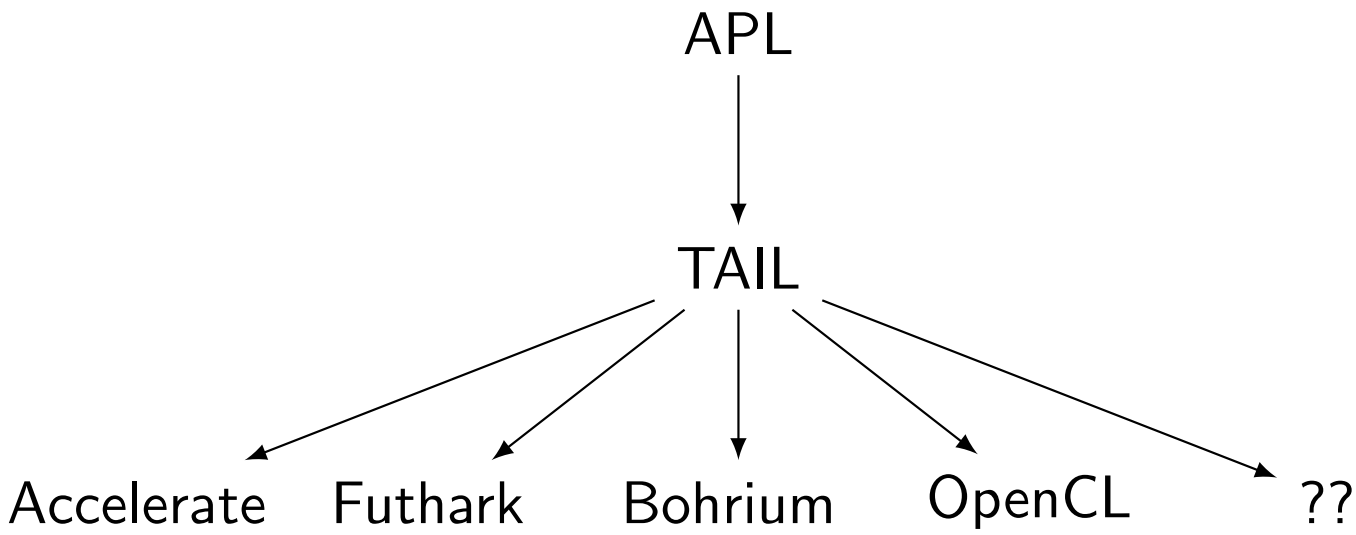
```
pi ← {
 x ← ?ωρ0
 y ← ?ωρ0
 dists ← (x*2) + y*2
 4×(+/1>dists)÷ω
}
pi 1000000
    3.142668
```

⇓

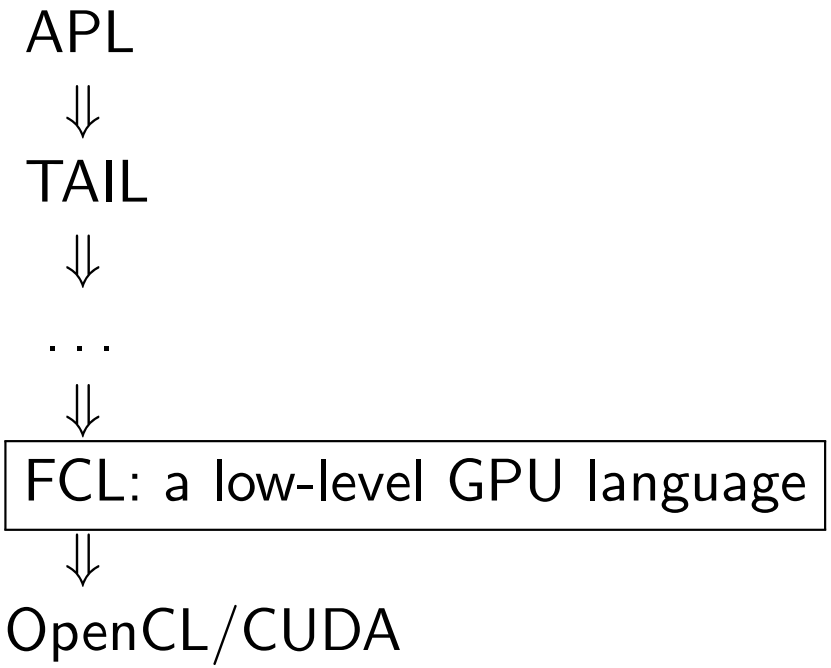
Note: each is just map in APL lingo

```
let v3:[double]1 =
  each(fn v2:[int]0 => roll(v2), reshape([1000000],[0])) in
let v5:[double]1 =
  each(fn v4:[int]0 => roll(v4), reshape([1000000],[0])) in
let v11:[double]1 =
  each(fn v10:[double]0 => powd(v10,divd(1.0,2.0)),
       zipWith(addd, each(fn v7:[double]0 => powd(v7,2.0),v3),
                     each(fn v6:[double]0 => powd(v6,2.0),v5))) in
muld(4.0, divd(i2d(reduce(addi,0,
                     each(b2i,
                       each(fn v12:[double]0 => gtd(1.0,v12),
                            v11)))),
               1000000.0))
```

# APL → TAIL → ?



# Outline



# Obsidian overview

- GPU language embedded in Haskell
- Two-layer design with Haskell as meta-language
- Hierarchy of array types
- Loops are always unrolled
- Some array sizes must be known statically

# Obsidian example: reduction

```
simpleReduce :: Data a
             => (a -> a -> a)
             -> SPull a
             -> Program Block (SPush Block a)
simpleReduce f arr =
  if len arr == 1
  then return (push arr)
  else do let (a1,a2) = halve arr
          arr' <- computePull (zipWith f a1 a2)
          simpleReduce f arr'
```

# Obsidian example: reduction

```
simpleReduce :: Data a
             => (a -> a -> a)
             -> SPull a
             -> Program Block (SPush Block a)
simpleReduce f arr =
  if len arr == 1
  then return (push arr)
  else do let (a1,a2) = halve arr
          arr' <- computePull (zipWith f a1 a2)
          simpleReduce f arr'
```

- Specifies reduction in a single block
- Generates an unrolled loop

# Obsidian example: reduction

```
simpleReduce :: Data a
             => (a -> a -> a)
             -> SPull a
             -> Program Block (SPush Block a)

red :: Data a
    => (a -> a -> a)
    -> DPull (SPull a)
    -> DPush Grid a
red f arr = liftGridMap (execBlock . simpleReduce f) arr

addReduce  :: DPull EInt32 -> DPush Grid EInt32
addReduce = red (+) . splitUp 512
```

# Obsidian example: reduction

```
simpleReduce :: Data a
             => (a -> a -> a)
             -> SPull a
             -> Program Block (SPush Block a)

red :: Data a
    => (a -> a -> a)
    -> DPull (SPull a)
    -> DPush Grid a
red f arr = liftGridMap (execBlock . simpleReduce f) arr

addReduce  :: DPull EInt32 -> DPush Grid EInt32
addReduce = red (+) . splitUp 512
```

- `red` lifts reduction to grid-level
- `splitUp` creates a nested array of arrays
- To do a full reduction, the kernel should be applied repeatedly.

# Obsidian summary

- EDSL
- Generates *individual* CUDA kernels
- Fusion-by-default, but user controlled
- Meta-programming
- Program GPU hierarchy in uniform style

# Goals of FCL

- Close to the metal
- Built in fusion, but user-controlled
- Predictability, no black box
- Ability to optimize
- Expressive enough for all necessary APL primitives
- A GPU language for algorithms researchers?
- Starting point: "Unembedded Obsidian"

# FCL reverse

```
sig reverse : [a] -> [a]
fun reverse arr =
  let n = length arr
  in generate n (fn i => index arr (n - i - 1))
```

# FCL reverse

```
sig reverse : [a] -> [a]
fun reverse arr =
  let n = length arr
  in generate n (fn i => index arr (n - i - 1))

sig distributeReverse : int -> [a] -> [a]
fun distributeReverse splitSize arr =
  splitUp splitSize arr
   |> map (fn subarr => force (reverse subarr))
   |> reverse
   |> concat splitSize
```

# FCL reverse

```
sig reverse : [a] -> [a]
fun reverse arr =
  let n = length arr
  in generate n (fn i => index arr (n - i - 1))

sig distributeReverse : int -> [a] -> [a]
fun distributeReverse splitSize arr =
  splitUp splitSize arr
   |> map (fn subarr => force (reverse subarr))
   |> reverse
   |> concat splitSize

sig reverseKernel : [int] -> [int]
kernel reverseKernel arr = distributeReverse 512 arr
```

# OpenCL

```
__kernel void reverseGridKernel(__local uchar* sbase,__global int* arrInput_0,
                                int lenInput_1, __global int* arrOutput_3) {
  int n_2 = ((lenInput_1 + get_local_size(0)) - 1) / get_local_size(0);
  int ub_5 = n_2;
  int id17 = ub_5 / get_num_groups(0);
  __local int* arr_7 = (__local int*) (sbase + 0);
  for (int id16 = 0; id16 < id17; id16++) {
    int i_4 = (get_group_id(0) * id17) + id16;
    for (int id12 = 0; id12 < 1; id12++) {
      int i_8 = (id12 * get_local_size(0)) + get_local_id(0);
      arr_7[i_8] = arrInput_0 [((((n_2-i_4)-1)*get_local_size(0)) + ((get_local_size(0)-i_8)-1))];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int id14 = 0; id14 < 1; id14++) {
      int i_10 = (id14 * get_local_size(0)) + get_local_id(0);
      arrOutput_3[((i_4 * get_local_size(0)) + i_10)] = arr_7 [i_10];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (get_group_id(0) < (ub_5 % get_num_groups(0))) {
    int i_4 = (get_num_groups(0) * id17) + get_group_id(0);
    for (int id12 = 0; id12 < 1; id12++) {
      int i_8 = (id12 * get_local_size(0)) + get_local_id(0);
      arr_7[i_8] = arrInput_0 [((((n_2-i_4)-1)*get_local_size(0)) + ((get_local_size(0)-i_8)-1))];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    for (int id14 = 0; id14 < 1; id14++) {
      int i_10 = (id14 * get_local_size(0)) + get_local_id(0);
      arrOutput_3[((i_4 * get_local_size(0)) + i_10)] = arr_7 [i_10];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
}
```

# concat and assemble

Concatenation is a derived form:

```
sig concat : int -> [[a]] -> [a]
fun concat n arr =
  assemble n (fn sh => (fst sh * n) + snd sh) arr
```

# concat and assemble

Concatenation is a derived form:

```
sig concat : int -> [[a]] -> [a]
fun concat n arr =
  assemble n (fn sh => (fst sh * n) + snd sh) arr
```

Using a more general construct "assemble":

```
assemble : int -> ((int, int) -> int) -> [[a]] -> [a]
```

# Transpose

```
sig transpose : int -> int -> [a] -> [a]
fun transpose rows cols elems =
  generate (rows * cols)
           (fn n =>
              let i = n / rows in
              let j = n % rows
              in index elems (j * rows + i))
```

# Transpose

```
sig transpose : int -> int -> [a] -> [a]
fun transpose rows cols elems =
  generate (rows * cols)
           (fn n =>
              let i = n / rows in
              let j = n % rows
              in index elems (j * rows + i))

splitGrid : int -> int -> int -> [a] -> [[a]]
concatGrid : int -> int -> [[a]] -> [a]
```

# Transpose

```
sig transpose : int -> int -> [a] -> [a]
fun transpose rows cols elems =
  generate (rows * cols)
           (fn n =>
              let i = n / rows in
              let j = n % rows
              in index elems (j * rows + i))

splitGrid : int -> int -> int -> [a] -> [[a]]
concatGrid : int -> int -> [[a]] -> [a]

sig transposeChunked : int -> int -> int -> [int] -> [int]
kernel transposeChunked splitSize rows cols elems =
  splitGrid splitSize rows cols elems
    |> map (transpose splitSize splitSize)
    |> map (fn arr => force arr)  -- force into shared memory
    |> transpose (rows / splitSize) (cols / splitSize)
    |> concatGrid splitSize (cols / splitSize)
```

# Reduction

```
sig reduceBlock : (a -> a -> a) -> [a] -> [a]
fun reduceBlock f arr =
  let cond = fn arr => 1 <> length arr in
  let step = fn arr => let x = halve arr
                       in zipWith f (fst x) (snd x)
  in while cond step (step arr)
```

# Reduction

```
sig reduceBlock : (a -> a -> a) -> [a] -> [a]
fun reduceBlock f arr =
  let cond = fn arr => 1 <> length arr in
  let step = fn arr => let x = halve arr
                       in zipWith f (fst x) (snd x)
  in while cond step (step arr)

sig reducePart : (a -> a -> a) -> [a] -> [a]
fun reducePart f arr =
  splitUp 512 arr
   |> map (reduceBlock f)
   |> concat 1
```

# Future work on FCL

- GPU-hierarchy in types
- Loop unrolling annotations
- Host-code generation
- Larger examples
- (Shapes and multi-dimensional arrays)

# Future work on TAIL and FCL

- FCL as backend for TAIL
- TAIL annotations for GPU vs. CPU execution
- Multiple devices (another level in the hierarchy?)
- Ability to inline FCL within APL program
- Integration with real APL interpreter (Dyalog)

# References


Compiling a Subset of APL Into a Typed Intermediate Language.
Martin Elsman and Martin Dybdal, 2014
*ARRAY'14*

Obsidian: A domain specific embedded language for parallel programming of graphics processor
Joel Svensson, Mary Sheeran, Koen Claessen, 2011
*IFL'11*

# Questions?