FCL: A low-level functional GPU language

Martin Dybdal
dybber@dybber.dk

DIKU
University of Copenhagen

22 September 2016

Joint work w. Mary Sheeran, Joel Svensson and Martin Elsman

Agenda

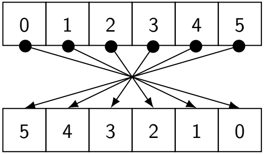
- Why a new low-level language for GPU computing?
- FCL by example
- Performance results
- Formalising FCL (just the highlights)
- Future work

Why a new low-level language for GPU computing?

- Composability and agility
- Built in fusion, user-controlled
- Compare programs/algorithms
- Predictability, no black box wrt. performance
- Ability to optimize
- A quest for a GPU language for algorithms researchers
- Intermediate language for optimizing compilers

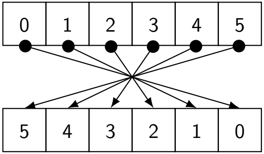
Reverse: A simple example

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



Reverse: A simple example

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



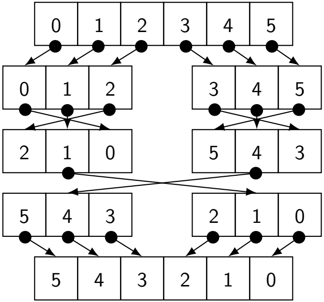
How is this executed on a GPU?

- Sequential in a single thread?
- Performed in a single block?
- Performed among threads in a grid?

Reverse: Distributed

```
sig revBlock : [a] -> [a]<block>
fun revBlock arr = push <block> (reverse arr)

sig revDistribute : int -> [a] -> [a]<grid>
fun revDistribute chunkSize arr =
  splitUp chunkSize arr
   |> map reverseBlock
   |> reverse
   |> concat chunkSize
```



Reverse: Generated OpenCL

For block size 256, FCL generates:

```
__kernel void revDistribute(__global int* arrInput_0, int lenInput_1,
                            __global int* arrOutput_3) {
    int n_2 = lenInput_1 / 256;
    int blocksQ_5 = n_2 / get_num_groups(0);
    for (int i_6 = 0; i_6 < blocksQ_5; i_6++) {
        int j_8 = (get_group_id(0) * blocksQ_5) + i_6;
        arrOutput_3[((j_8 * 256) + get_local_id(0))] =
            arrInput_0 [((((n_2 - j_8) - 1) * 256) + ((256 - get_local_id(0)) - 1))];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    if (get_group_id(0) < (n_2 % get_num_groups(0))) {
        int j_16 = (get_num_groups(0) * blocksQ_5) + get_group_id(0);
        arrOutput_3[((j_16 * 256) + get_local_id(0))] =
            arrInput_0 [((((n_2 - j_16) - 1) * 256) + ((256 - get_local_id(0)) - 1))];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Two array types
(from Obsidian)

- Pull arrays: for organizing computation, indexable no concatenation

```
[int], [bool], [[int]]
```

- Push arrays: for writing to memory, non indexable, supports concatenation

```
[int]<thread>, [int]<block>, [int]<grid>
```

- Both supporting fusion

Array types in reverse

```
sig revDistribute : int -> [a] -> [a]<grid>
fun revDistribute chunkSize arr =
  splitUp chunkSize arr    -- [[a]]
   |> map reverseBlock     -- [[a]<block>]
   |> reverse              -- [[a]<block>]
   |> concat chunkSize     -- [a]<grid>

sig push : <lvl> -> [a] -> [a]<lvl>
sig splitUp : int -> [a] -> [[a]]
sig concat : int -> [[a]<lvl>] -> [a]<1+lvl>
```

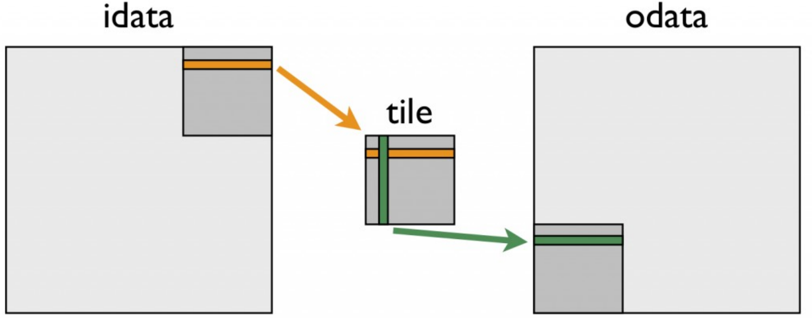
Matrix transposition

A basic approach

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

Matrix transposition

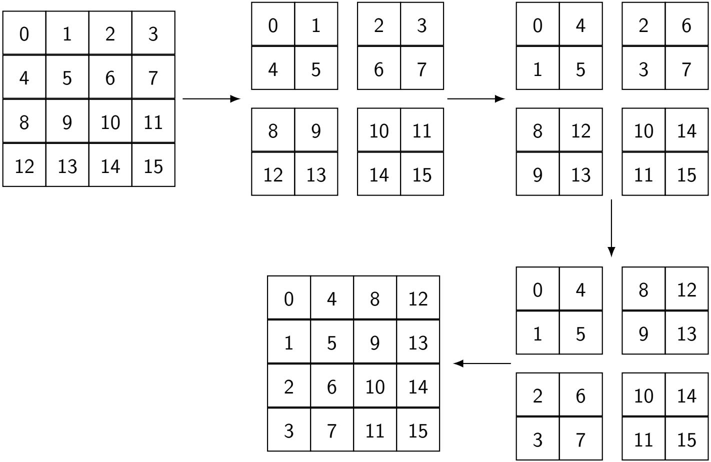
Tiled



(Figure by NVIDIA)

Matrix transposition

Tiled



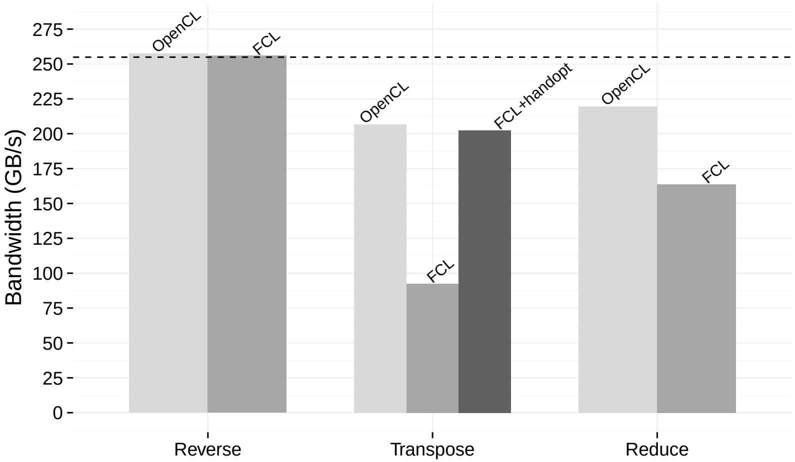
Matrix transposition

Tiled, using shared-memory

```
sig transposeTiled : int -> int -> int -> [a] -> [a]<grid>
fun transposeTiled tileDim cols rows mat =
  let n = cols / tileDim
      m = rows / tileDim
  in split2DGrid tileDim cols n m mat
       |> map (force . push <block>)
       |> map (transpose tileDim tileDim)
       |> transpose n m
       |> map (push <block>)
       |> concat2DGrid tileDim n rows

sig force : [a]<lvl> -> [a]
```

Performance



NVIDIA GeForce GTX 780 Ti (2880 cores, 875 Mhz, 3GB GDDR5)
Peak bandwidth 254.90 GB/s (measured)

Highlights from formalism

- Polymorphic type system, restricting e.g. the valid nesting of arrays
- Dynamic semantics, explaining mapping to threads, blocks, warps and grids
- Formalism informed our implementation
- Abstracts away from block/warp-virtualization
- Classic type safety properties

Built-in operators

$$
\begin{aligned}
\texttt{lengthPull} &: [\alpha] \to \texttt{int} \\
\texttt{lengthPush} &: [\alpha]\langle \mathsf{lvl} \rangle \to \texttt{int} \\
\texttt{mapPull} &: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\texttt{mapPush} &: (\alpha \to \beta) \to [\alpha]\langle \mathsf{lvl} \rangle \to [\beta]\langle \mathsf{lvl} \rangle
\end{aligned}
$$

$$
\begin{aligned}
\texttt{generate} &: \texttt{int} \to (\texttt{int} \to \alpha) \to [\alpha] \\
\texttt{index} &: [\alpha] \to \texttt{int} \to \alpha
\end{aligned}
$$

$$
\begin{aligned}
\texttt{push} &: \langle \mathsf{lvl} \rangle \to [\alpha] \to [\alpha]\langle \mathsf{lvl} \rangle \\
\texttt{force} &: [\alpha]\langle \mathsf{lvl} \rangle \to [\alpha] \\
\texttt{concat} &: \texttt{int} \to [[\alpha]\langle \mathsf{lvl} \rangle] \to [\alpha]\langle 1 + \mathsf{lvl} \rangle
\end{aligned}
$$

$$
\texttt{while} : ([\alpha] \to \texttt{bool}) \to ([\alpha] \to [\alpha]\langle \mathsf{lvl} \rangle) \to [\alpha]\langle \mathsf{lvl} \rangle \to [\alpha]
$$

Future work on FCL

- Generalize hierarchy and mapping
  (e.g. ability to add layers, like multiple GPUs)
- Tracking communication costs in semantics
- Use as back-end for our APL-compiler (TAIL)
- Multi-dimensional arrays

Summary

- I argue that we need to focus on performance reasoning and cost-models
- Nested arrays, but only non-nested arrays can be materialized.
- Level hierarchy controls mapping to sequential/parallel loops

References

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

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

Thank you

Martin Dybdal
Ph.D. student
DIKU, University of Copenhagen
dybber@dybber.dk

FCL is available at: http://github.com/dybber/fcl

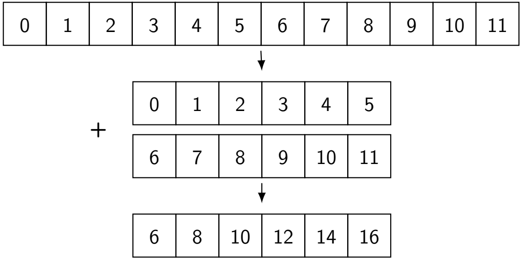
More future work on FCL

- Manual memory-management
- Sequential loops
- Larger examples
- Host-code generation

Reduction

```
sig halve : [a] -> ([a], [a])
sig zipWith : (a -> b -> c) -> [a] -> [b] -> [c]

sig step : <lvl> -> (a -> a -> a) -> [a] -> [a]<lvl>
fun step <lvl> f arr =
  let x = halve arr
  in push <lvl> (zipWith f (fst x) (snd x))
```



Reduction

```
sig red : <lvl> -> (a -> a -> a) -> [a] -> [a]<lvl>
fun red <lvl> f arr =
  while (fn arr => 1 != lengthPull arr)
        (step <lvl> f)
        (step <lvl> f arr)
  |> push <lvl>
```

Reduction

OpenCL

```
__kernel void reduceAdd(__local volatile uchar* sbase,
                        __global int* arrInput_0, int lenInput_1,
                        __global int* arrOutput_2) {
    int ub_3 = lenInput_1 >> 8;
    int blocksQ_4 = ub_3 / get_num_groups(0);
    for (int i_5 = 0; i_5 < blocksQ_4; i_5++) {
        int j_7 = (get_group_id(0) * blocksQ_4) + i_5;
        __local volatile int* arr_12 = (__local volatile int*) (sbase + 0);
        arr_12[get_local_id(0)] = arrInput_0 [((j_7 * 256) + get_local_id(0))]
                                + arrInput_0 [((j_7 * 256) + (get_local_id(0) + 128))];
        barrier(CLK_LOCAL_MEM_FENCE);
        if (get_local_id(0) < 64) {
            arr_12[get_local_id(0)] = arr_12 [get_local_id(0)] + arr_12 [(get_local_id(0) + 64)];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        if (get_local_id(0) < 32) {
            arr_12[get_local_id(0)] = arr_12 [get_local_id(0)] + arr_12 [(get_local_id(0) + 32)];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        if (get_local_id(0) < 16) {
            arr_12[get_local_id(0)] = arr_12 [get_local_id(0)] + arr_12 [(get_local_id(0) + 16)];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        if (get_local_id(0) < 8) {
            arr_12[get_local_id(0)] = arr_12 [get_local_id(0)] + arr_12 [(get_local_id(0) + 8)];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        if (get_local_id(0) < 4) {
            arr_12[get_local_id(0)] = arr_12 [get_local_id(0)] + arr_12 [(get_local_id(0) + 4)];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        if (get_local_id(0) < 2) {
            arr_12[get_local_id(0)] = arr_12 [get_local_id(0)] + arr_12 [(get_local_id(0) + 2)];
```