Advanced Topics in Programming Languages (ATPL)

Synthesizing Futhark Quantum Circuit Simulation

Martin Elsman

Department of Computer Science
University of Copenhagen
DIKU

December 3, 2024

Context: Data-Parallel Programming

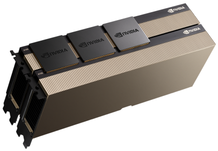
- High-performance demands have lead to SIMD architectures, which are well-suited for data-parallel programming.
- Parallel solutions to many problems benefit from support for *nested parallelism* (e.g., maps within maps) and even for *irregular nested parallelism* (i.e., performing uneven amounts of parallel work in parallel).

Gap: A large performance penalty (open research problem)

- General approaches to supporting irregular parallelism exist (i.e., Blelloch's flattening transformation, as implemented in NESL) but it has a **significant overhead in practice**, due to the administration of segment descriptors (i.e., flag vectors).

Futhark — 101

- Futhark is a strict data-parallel pure functional programming language. It is also a compiler aimed at running Futhark programs efficiently on massively-parallel hardware (e.g., GPUs).



- Futhark shows impressive performance numbers! In some cases it beats even hand-optimized OpenCL and CUDA programs.

- Futhark incrementally turns nested regular data-parallelism into flat data-parallelism and creates multiple versions of code that utilises different levels of parallelism.

- Futhark provides a series of *zero-cost abstraction* features including **higher-order modules**, **type polymorphism**, a restricted form of **higher-order functions**, and **size polymorphism**.

Futhark – the Language

- Syntactically and conceptually similar to Haskell and ML.
- Simple functional language with conditionals and support for looping.
- Datatypes are booleans, basic numeric types, such as `i32` and `f64`, and the types of multi-dimensional arrays ($[]\tau$, where τ is any type.)
- Futhark is augmented with a set of **parallel** SOACs for operating on arrays.

```
-- Simple looping
let fact(n: i32): i32 =
  loop res = 1
    for i < n do (i+1) * res
```

```
-- Looping
let fib(n: i32): i32 =
  fst(loop (x,y) = (1,1)
        for i<n do (y,x+y))
```

```
-- Initialize and fill
let fib (n: i32): []i32 =
  loop arr = replicate n 1
    for i < n-2 do
      let arr[i+2] =
         arr[i] + arr[i+1]
      in arr
```

Futhark Second-Order Array Combinators (SOACs)

`map` $f\ a$	Probably the simplest parallel SOAC. Maps f over the outer dimension of a.
`map2` $f\ a_1\ a_2$	Applies the binary function f pair-wise to the outer elements of a_1 and a_2.
`reduce` $\oplus\ ne\ a$	Similar to Haskell's `foldl` and `foldr`, but assumes $A \oplus (B \oplus C) = (A \oplus B) \oplus C$ and $ne \oplus A = A \oplus ne = A$.
`scan` $\oplus\ ne\ a$	Parallel inclusive prefix scan. Returns an array of the same shape as a but with each element being identical to the result of reducing the prefix of the array a. The `scan` SOAC is critical for implementing parallel algorithms.[1]

Other built-in data-parallel operations include `filter` and `scatter`.

[1] Blelloch, Guy E. 1989. "Scans as Primitive Parallel Operations." IEEE Transactions on Computers 38(11), pp. 1526–1538.

Size Polymorphism and Statically-Enforced Size-Constraints

Programmers may specify that functions are parametric in array sizes:

```
def dotprod [n] (xs: [n]f64) (ys: [n]f64) : f64 =
  f64.sum (map2 (*) xs ys)
```

Here is the type of map2:

```
val map2 'a 'b 'c [n] : (a→b→c) → [n]a → [n]b → [n]c
```

We could also have written:

```
def dotp (n:i64) (x: [n]f64) (y: [n]f64): f64 =
  f64.sum (map2 (*) x y)
```

Notice: Size polymorphism makes the size argument implicit; instances are inserted by the compiler.

Dependent Sizes

```
val iota : (n:i64) : [n]i64
```

```
iota (x+y) : [x+y]i64
```

Like in dependantly typed-languages:

```
val concat 'a [n][m] : [n]a → [m]a → [n+m]a
```

```
concat (iota (x+y)) (iota z) : [x+y+z]i64
```

Existential Size Types

```
val filter 'a [n] : (a→bool) → [n]a → ?[m].[m]a
```

Existential size-types are "opened implicitly" in `let`-bindings making it possible for type inference to track sizes and report size errors statically.

Size Expressions are Syntactic

```
val flatten   'a [n][m] : [n][m]a → [n*m]a
```

More interestingly:

```
val unflatten 'a [n][m] : [n*m]a → [n][m]a
val split     'a [n][m] : [n+m]a → ([n]a, [m]a)
```

Size Constraints

- A size constraint x :> [n*2]f64 has type [n*2]f64.
- It is checked statically that x is an f64-array and dynamically that the array has size n*2.

Static Properties and Restrictions

- It is a static error if a size-argument cannot be determined by the context:

```
def iiota [n] () = iota n        -- invalid!
```

- For an array of arrays, all elements must be of the same size.

- It is enforced that coherent sizes are given to functions like matmul:

```
def matmul [n][p][k] (A: [n][p]f64) (B: [p][k]f64)
  : [n][k]f64 =
  map (\Arow → map (\Bcol → dotprod Arow Bcol)
                   (transpose B))
      A
```

Dynamic Checks

- Array indexing is checked dynamically, but may be discharged statically.

- iota n may fail dynamically (if n<0).

Generating Futhark Programs from Standard ML

- Futhark is **not** meant to be a general programming language.
- For instance, it does not directly support recursive data types.

Use a Host Language (e.g., Standard ML) for Generating Futhark Code!

- It is straightforward to embed a simple Futhark expression language in a functional language using algebraic data types.
- Use a monad to keep track of local `let`-bindings.
- A simple pretty-printer outputs well-formed Futhark code.

Futhark Matrix Utility Library

```
module type MAT = {
  type t
  val tensor [m][n][p][q] : [m][n]t → [p][q]t → [m*p][n*q]t
  val control          [m] : [m][m]t → [2*m][2*m]t
  val matmul     [m][n][k] : [m][n]t → [n][k]t → [m][k]t
  ...
}
```

Implementation

```
module type NUM = { type t
                    val i64 : i64 → t
                    val *   : t → t → t
                    val +   : t → t → t
                  }

module mk_mat (T : NUM) : MAT with t = T.t = {
  type t = T.t
  def scale [m][n] (s:t) (a:[m][n]t) : [m][n]t =
    map (map (T.((s*)))) a
  def tensor [m][n][p][q] (a:[m][n]t) (b:[p][q]t) : [m*p][n*q]t =
    map (map (\s → scale s b)) a
    |> map transpose |> flatten |> map flatten
  ...
}
```

Futhark Expressions and Bindings in Standard ML

```
structure Futhark :> sig
  type var = string
  type ty = string
  datatype exp = VAR of var
               | BINOP of string * exp * exp
               | CONST of string
               | APP of string * exp list
               | ARR of exp list
               | TYPED of exp * ty
               | SEL of int * exp
  type α M
  val ret       : α → α M
  val >>=       : α M * (α → β M) → β M
  val LetNamed : var → exp → var M
  val FunNamed : var → (exp → exp M) → ty → ty → var M
  val Let       : exp → var M
  val Fun       : (exp → exp M) → ty → ty → var M
  val run       : exp M → string
  val runBinds : unit M → string
end
```

Compiling Circuits To Futhark Matrix Expression

```
fun comp (t:Circuit.t) : Futhark.exp Futhark.M =
    let open Circuit
        open Futhark infix >>=
    in case t of
           I ⇒ ret (APP("I",[]))
         | X ⇒ ret (APP("X",[]))
         | Y ⇒ ret (APP("Y",[]))
         | Z ⇒ ret (APP("Z",[]))
         | H ⇒ ret (APP("H",[]))
         | SW ⇒ ret (APP("SW",[]))
         | Seq(t1,t2) ⇒
           comp t1 >>= (fn e1 ⇒
           comp t2 >>= (fn e2 ⇒
           let val n = Int.toString (pow2 (dim t))
               val ty = "[" ^ n ^ "][" ^ n ^ "]C.complex"
           in ret (TYPED(APP("matmul", [e2,e1]),ty))
           end))
         | Tensor(t1,t2) ⇒
           comp t1 >>= (fn e1 ⇒
           comp t2 >>= (fn e2 ⇒
           let val n = Int.toString (pow2 (dim t))
               val ty = "[" ^ n ^ "][" ^ n ^ "]C.complex"
           in ret (TYPED(APP("tensor", [e1,e2]),ty))
           end))
         | C t' ⇒ comp t' >>= (fn e ⇒
           let val n = Int.toString (pow2 (dim t))
               val ty = "[" ^ n ^ "][" ^ n ^ "]C.complex"
           in ret (TYPED(APP("control",[e]),ty))
           end)
    end
```

Prelude with Futhark Quantum Gates

```
import "lib/github.com/diku-dk/complex/complex"
import "futlib"
module C = mk_complex(f64)
module mat = mk_mat(C)
open mat

def I () : [2][2]C.complex =
  [[C.i64 1,C.i64 0], [C.i64 0,C.i64 1]]
def H () : [2][2]C.complex =
  [[C.mk_re (1.0 / f64.sqrt(2.0)),C.mk_re (1.0 / f64.sqrt(2.0))],
   [C.mk_re (1.0 / f64.sqrt(2.0)),C.mk_re ((-1.0) / f64.sqrt(2.0))]]
def X () : [2][2]C.complex =
  [[C.i64 0,C.i64 1], [C.i64 1,C.i64 0]]
def Y () : [2][2]C.complex =
  [[C.i64 0,C.mk_im (-1)], [C.mk_im 1,C.i64 0]]
def Z () : [2][2]C.complex =
  [[C.i64 1,C.i64 0], [C.i64 0,C.i64 (-1)]]
def SW () : [4][4]C.complex =
  [[C.i64 1,C.i64 0,C.i64 0,C.i64 0], [C.i64 0,C.i64 0,C.i64 1,C.i64 0],
   [C.i64 0,C.i64 1,C.i64 0,C.i64 0], [C.i64 0,C.i64 0,C.i64 0,C.i64 1]]
```

```
$ futhark repl prelude.fut
[0]> tensor (matmul (X()) (H())) (Y())
[[(0.0, 0.0), (0.0, -0.7071067811865475), (-0.0, 0.0), (0.0, 0.7071067811865475)],
 [(0.0, 0.7071067811865475), (0.0, 0.0), (-0.0, -0.7071067811865475), (-0.0, 0.0)],
 [(0.0, 0.0), (0.0, -0.7071067811865475), (0.0, 0.0), (0.0, -0.7071067811865475)],
 [(0.0, 0.7071067811865475), (0.0, 0.0), (0.0, 0.7071067811865475), (0.0, 0.0)]]
[1]>
```

Example Generated Futhark Matrix Expression

```
Circuit: (I ** H oo C X oo Z ** Z oo C X oo I ** H) ** I oo I ** SW oo C X ** Y
```

```
              .---.
----------*---| Z |---*-----------------*----
          |   '---'   |                 |
          |           |                 |
  .---. .-+-. .---. .-+-. .---.       .-+-.
--| H |-| X |-| Z |-| X |-| H |-.   .-| X |--
  '---' '---' '---' '---' '---'  \ /  '---'
                                  /
                                 / \  .---.
--------------------------------'   '-| Y |--
                                      '---'
```

$$
\begin{bmatrix}
-i & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & i & 0 & 0 & 0 & 0 & 0 \\
0 & -i & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & i & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -i & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & i \\
0 & 0 & 0 & 0 & -i & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & i & 0
\end{bmatrix}
$$

Generated Futhark Matrix Expression:

```
def m = (matmul ((tensor ((control (X())):>[4][4]C.complex)
  (Y())):>[8][8]C.complex) ((matmul ((tensor (I()) (SW())):>[8][8]C.complex)
  ((tensor ((matmul ((tensor (I()) (H())):>[4][4]C.complex) ((matmul ((control
  (X())):>[4][4]C.complex) ((matmul ((tensor (Z()) (Z())):>[4][4]C.complex)
  ((matmul ((control (X())):>[4][4]C.complex) ((tensor (I())
  (H())):>[4][4]C.complex)):>[4][4]C.complex)):>[4][4]C.complex))
  :>[4][4]C.complex)):>[4][4]C.complex)
  (I())):>[8][8]C.complex)):>[8][8]C.complex)):>[8][8]C.complex
```

Can We Simulate Quantum Circuits Without Tensor-Explosions?

Instead of first generating a matrix from a circuit and then applying the matrix to a state vector, perhaps "apply the circuit to the state vector":

```
type vec = C.complex vector
val interp : Circuit.t → vec → vec
```

Use the "Vec Trick"[2] and the Definition of Matrix Multiply:

- If $A : [m][m]\mathbb{C}$ and $B : [n][n]\mathbb{C}$, then $(A \otimes B) : [mn][mn]\mathbb{C}$.
- $(A \otimes B)\ v \ = \ \texttt{vec}(BVA^T) \ = \ \texttt{vec}((A(BV)^T)^T),\ \text{where } V = \texttt{unvec}\ v$
- $\texttt{vec}\ V : [nm]\mathbb{C}$ is the result of "stacking" all columns in $V : [n][m]\mathbb{C}$.
- $\texttt{unvec}\ v$ is the matrix $V : [n][m]\mathbb{C}$ such that $\texttt{vec}\ V = v$.
- $[\![C_A]\!]V = (\texttt{map}\ (\texttt{interp}\ C_A)\ V^T)^T,\ \text{where } [\![C_A]\!] = A$

[2] See https://en.wikipedia.org/wiki/Kronecker_product

Utility Functions for Kronecker-Free Quantum Simulation

```
fun flatten (a:mat) : vec =
    let val (rows,cols) = M.dimensions a
    in Vector.tabulate(rows * cols,
                       fn i => M.sub(a,i div cols,i mod cols))
    end

fun unflatten (r,c) (v:vec) : mat =
    if Vector.length v <> r * c then raise Fail "unflatten"
    else M.tabulate(r,c,fn (i,j) => Vector.sub(v,i*c+j))

fun unvec (r:int,c:int) (v:vec) : mat =
    unflatten (r,c) v |> M.transpose

fun vec (a:mat) : vec =
    M.transpose a |> flatten

fun vecSplit (v:vec) : vec * vec =
    let val n = Vector.length v div 2
    in (VectorSlice.vector(VectorSlice.slice(v,0,SOME n)),
        VectorSlice.vector(VectorSlice.slice(v,n,NONE)))
    end

fun mapRows (f:vec->vec) (a:mat) : mat =
    List.tabulate(M.nRows a,
                  fn i => f(M.row i a))
                 |> M.fromVectorList
```

The Kronecker-Free Quantum Simulation Function

```
fun interp (t:Circuit.t) (v:vec) : vec =
    case t of
        Circuit.Seq(t1,t2) ⇒ interp t2 (interp t1 v)
      | Circuit.Tensor(A,B) ⇒
        let val V = unvec (pow2(Circuit.dim A),
                           pow2(Circuit.dim B)) v
            val W = mapRows (interp B) (M.transpose V)
            val Y = mapRows (interp A) (M.transpose W)
        in vec Y
        end
      | Circuit.C A ⇒
        let val (v1,v2) = vecSplit v
        in Vector.concat[v1,interp A v2]
        end
      | Circuit.I ⇒ v
      | _ ⇒ eval t v
```

Derivation for the Tensor Case

$$\begin{array}{rcll}
\texttt{interp}\,(A \otimes B)\,v & = & \texttt{vec}((A(BV)^T)^T) & BV = (\texttt{map}\,(\texttt{interp}\,B)\,(V^T))^T \\
& = & \texttt{vec}((A((\texttt{map}\,(\texttt{interp}\,B)\,(V^T))^T)^T)^T) & \\
& = & \texttt{vec}((A(\texttt{map}\,(\texttt{interp}\,B)\,(V^T)))^T) & \\
& = & \texttt{let}\,W = \texttt{map}\,(\texttt{interp}\,B)\,(V^T) & \\
& & \texttt{in}\,\texttt{vec}((AW)^T) & AW = (\texttt{map}\,(\texttt{interp}\,A)\,(W^T))^T \\
& = & \texttt{let}\,W = \texttt{map}\,(\texttt{interp}\,B)\,(V^T) & \\
& & \texttt{in}\,\texttt{vec}(((\texttt{map}\,(\texttt{interp}\,A)\,(W^T))^T)^T) & \\
& = & \texttt{let}\,W = \texttt{map}\,(\texttt{interp}\,B)\,(V^T) & \\
& & \texttt{in}\,\texttt{vec}(\texttt{map}\,(\texttt{interp}\,A)\,(W^T)) & \\
& = & \texttt{let}\,W = \texttt{map}\,(\texttt{interp}\,B)\,(V^T) & \\
& & \texttt{let}\,Y = \texttt{map}\,(\texttt{interp}\,A)\,(W^T) & \\
& & \texttt{in}\,\texttt{vec}(Y) &
\end{array}$$

qed.

Utilities for Futhark Quantum Simulation Synthesis

```
fun allI (t:Circuit.t) : bool =
    let open Circuit
    in case t of I ⇒ true
               | Tensor(a,b) ⇒ allI a andalso allI b
               | Seq(a,b) ⇒ allI a andalso allI b
               | _ ⇒ false
    end
fun vecTyFromDim d =
    "[" ^ Int.toString(pow2 d) ^ "]C.complex"
fun FunC A (f:F.exp → F.exp F.M) : F.var option F.M =
    if allI A then ret NONE
    else let val ty = vecTyFromDim (Circuit.dim A)
         in Fun f ty ty >>= (ret o SOME)
         end
fun splitF d v =
    let val ty = "[" ^ Int.toString (pow2 d) ^ "+" ^
                 Int.toString (pow2 d) ^ "]C.complex"
    in APP("split",[TYPED(v,ty)])
    end
fun concatF d a b = TYPED(APP("concat",[a,b]),vecTyFromDim d)
fun unvecF (e,ty) = APP("unvec", [TYPED(e,ty)])
fun vecF e = APP("vec",[e])
fun mapF f e = APP("map", [VAR f,e])
fun matvecmulF m v = APP("matvecmul", [m,v])
fun transposeF m = APP("transpose", [m])
fun mapF' NONE e = e
  | mapF' (SOME f) e = mapF f e
```

Futhark Quantum Simulation Synthesis

```
fun icomp (t:Circuit.t) (v:exp) : exp M =
    case t of
        Circuit.I ⇒ ret v
      | Circuit.Seq(t1,t2) ⇒ icomp t1 v >>= (icomp t2)
      | Circuit.C t' ⇒
        Let (splitF (Circuit.dim t') v) >>= (fn p ⇒
        icomp t' (SEL(1,VAR p)) >>= (fn v1 ⇒
        ret (concatF (Circuit.dim t) (SEL(0,VAR p)) v1)))
      | Circuit.Tensor(A,B) ⇒
        FunC A (icomp A) >>= (fn Af ⇒
        FunC B (icomp B) >>= (fn Bf ⇒
        let val dA = pow2(Circuit.dim A)
            val dB = pow2(Circuit.dim B)
            val ty = "[" ^ Int.toString dA ^ "*" ^
                     Int.toString dB ^ "]C.complex"
        in Let (unvecF(v,ty)) >>= (fn V ⇒
           Let (mapF' Bf (transposeF (VAR V))) >>= (fn W ⇒
           Let (mapF' Af (transposeF (VAR W))) >>= (fn Y ⇒
           ret (TYPED(vecF (VAR Y),vecTyFromDim (Circuit.dim t))))))
        end))
      | Circuit.H ⇒ ret (matvecmulF (APP("H",[])) v)
      | Circuit.X ⇒ ret (matvecmulF (APP("X",[])) v)
      | Circuit.Y ⇒ ret (matvecmulF (APP("Y",[])) v)
      | Circuit.Z ⇒ ret (matvecmulF (APP("Z",[])) v)
      | Circuit.SW ⇒ ret (matvecmulF (APP("SW",[])) v)
```

Example Simulator

```
def f (v6:[8]C.complex) : [8]C.complex =
  let f0 (v7:[4]C.complex) : [4]C.complex =
    let f1 (v8:[2]C.complex) : [2]C.complex = matvecmul (H()) v8
    let v9 = unvec (v7:>[2*2]C.complex)
    let v10 = map f1 (transpose v9)
    let v11 = transpose v10
    let v12 = split (((vec v11):>[4]C.complex):>[2+2]C.complex)
    let f2 (v13:[2]C.complex) : [2]C.complex = matvecmul (Z()) v13
    let f3 (v14:[2]C.complex) : [2]C.complex = matvecmul (Z()) v14
    let v15 = unvec(((concat(v12.0)(matvecmul (X())(v12.1))):>[4]C.complex):>[2*2]C.complex)
    let v16 = map f3 (transpose v15)
    let v17 = map f2 (transpose v16)
    let v18 = split (((vec v17):>[4]C.complex):>[2+2]C.complex)
    let f4 (v19:[2]C.complex) : [2]C.complex = matvecmul (H()) v19
    let v20 = unvec(((concat (v18.0)(matvecmul(X())(v18.1))):>[4]C.complex):>[2*2]C.complex)
    let v21 = map f4 (transpose v20)
    let v22 = transpose v21
    in (vec v22):>[4]C.complex
  let v23 = unvec (v6:>[4*2]C.complex)
  let v24 = transpose v23
  let v25 = map f0 (transpose v24)
  let f5 (v26:[4]C.complex) : [4]C.complex = matvecmul (SW()) v26
  let v27 = unvec (((vec v25):>[8]C.complex):>[2*4]C.complex)
  let v28 = map f5 (transpose v27)
  let v29 = transpose v28
  let f6 (v30:[4]C.complex) : [4]C.complex =
    let v31 = split (v30:>[2+2]C.complex)
    in (concat (v31.0) (matvecmul (X()) (v31.1))):>[4]C.complex
  let f7 (v32:[2]C.complex) : [2]C.complex = matvecmul (Y()) v32
  let v33 = unvec (((vec v29):>[8]C.complex):>[4*2]C.complex)
  let v34 = map f7 (transpose v33)
  let v35 = map f6 (transpose v34)
  in (vec v35):>[8]C.complex
```

Exercises

See `http://github.com/diku-dk/atpl-sml-quantum`

- Clone the repository and read the `README.md` file.

Project Ideas

- The `README.md` file also mentions a series of project ideas.