Data Parallel Programming in Futhark

Troels Henriksen (athas@sigkill.dk)

27th of April, 2023

DIKU
University of Copenhagen
Guest lecture in *Parallel Functional Programming* at Chalmers

- Troels Henriksen
- Assistant professor at the Department of Computer Science at the University of Copenhagen (DIKU)
- My research involves working on a high-level purely functional language, Futhark, and its heavily optimising compiler

The why and how of GPUs

The need for a programming model

The Futhark programming language

Compiler transformations

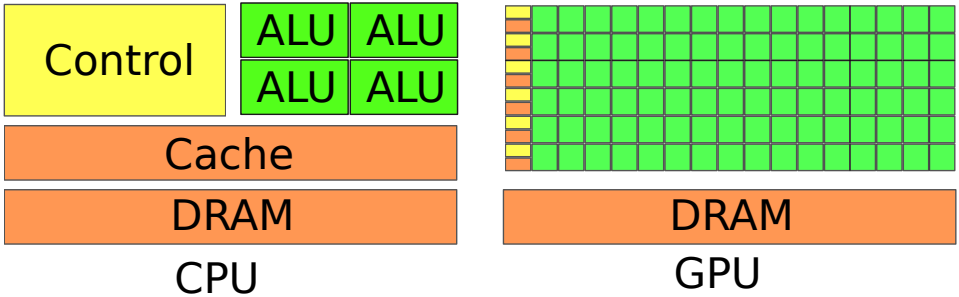
Applied Futhark programming

The why and how of GPUs

The Situation

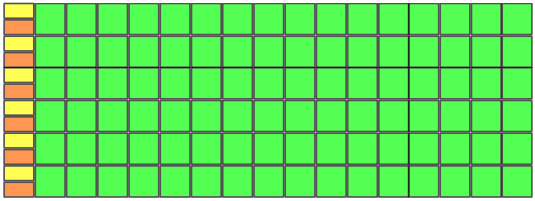
- Transistors continue to shrink, so we can continue to build ever more advanced computers
- CPU clock speed stalled around 3GHz in 2005, and improvements in sequential performance has been slow since then
- Computers still get *faster*, but mostly for parallel code
- General-purpose programming now often done on *massively parallel* processors, like Graphics Processing Units (GPUs)

GPUs vs CPUs



- GPUs have *thousands* of simple cores and taking full advantage of their compute power requires *tens of thousands* of threads
- GPU threads are very *restricted* in what they can do: no stack, no allocation, limited control flow, etc
- Potential *very high performance* and *lower power usage* compared to CPUs, but programming them is *hard*

The SIMT Programming Model



- GPUs are programmed using the SIMT model (*Single Instruction Multiple Thread*)
- Similar to SIMD (*Single Instruction Multiple Data*), but while SIMD has explicit vectors, we provide *sequential scalar per-thread* code in SIMT

Each thread has its own registers, but they all execute the same instructions at the same time (i.e. they share their instruction pointer).

SIMT example

For example, to increment every element in an array `a`, we might use this code:

```
increment(a) {
  tid = get_thread_id();
  x = a[tid];
  a[tid] = x + 1;
}
```

- If `a` has `n` elements, we launch `n` threads, with `get_thread_id()` returning *i* for thread *i*
- This is *data-parallel programming*: applying the same operation to different data

Branching

If all threads share an instruction pointer, what about branches?

```
mapabs(a) {
  tid = get_thread_id();
  x = a[tid];
  if (x < 0) { a[tid] = -x; }
}
```

Masked Execution

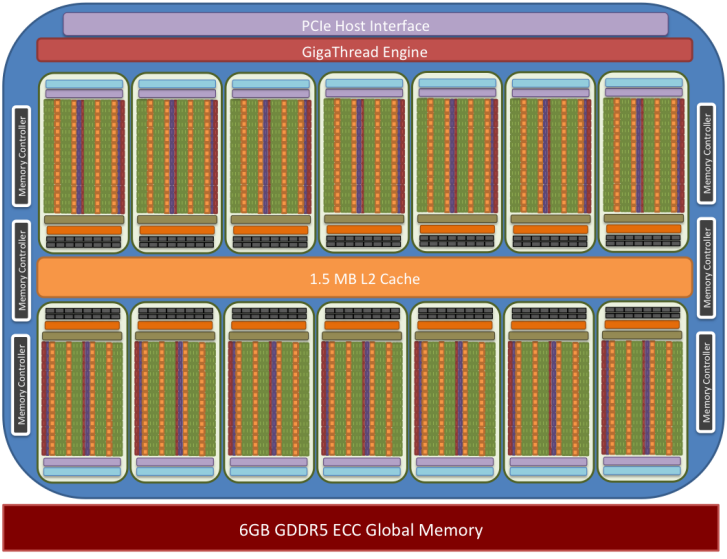
Both branches are executed in all threads, but in those threads where the condition is false, a mask bit is set to treat the instructions inside the branch as no-ops.

When threads differ on which branch to take, this is called *branch divergence*, and can be a performance problem.

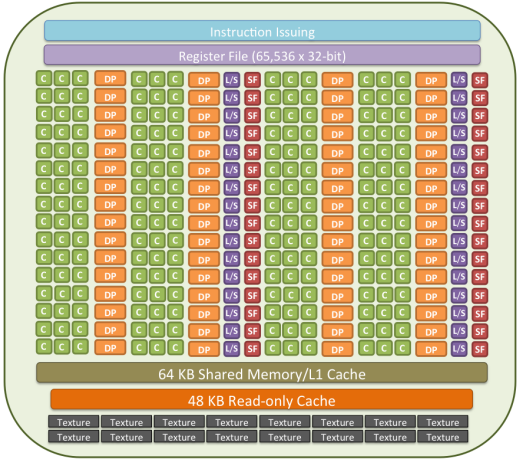
Execution Model

- A GPU program is called a *kernel*
- The GPU bundles threads in groups of 32, called *warps*. These are the unit of scheduling
- Warps are in turn bundled into *workgroups* or *thread blocks*, of a programmer-defined size not greater than 1024
- Using *oversubscription* (many more threads that can run simultaneously) and *zero-overhead hardware scheduling*, the GPU can aggressively *hide latency*
- Following illustrations from `https://www.olcf.ornl.gov/for-users/system-user-guides/titan/nvidia-k20x-gpus/`
  Older K20 chip (2012), but modern architectures are very similar.

K20 GPU layout—an assembly of 14 *streaming multiprocessors*



Streaming multiprocessor layout



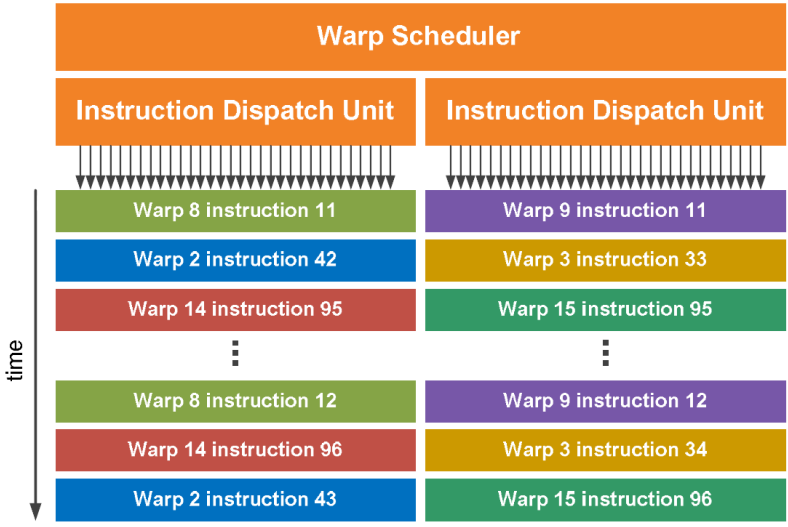
C single precision/integer CUDA core

DP double precision FP unit

L/S memory load/store unit

SF special function unit

Warp scheduling



The need for a programming model

Two guiding quotes

When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.

—Edsger W. Dijkstra (EWD963, 1986)

Two guiding quotes

When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.

—Edsger W. Dijkstra (EWD963, 1986)

The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague.

—Edsger W. Dijkstra (EWD340, 1972)

Human brains simply cannot reason about concurrency on a massive scale

- We need a programming model with *sequential* semantics, but that can be *executed* in parallel.
- It must be *portable*, because hardware continues to change.
- It must support *modular* programming, so we can write reusable components.

Sequential programming for parallel machines

One approach: write imperative code like we've always done, and apply a *parallelising compiler* to try to figure out whether parallel execution is possible:

```
for (int i = 0; i < n; i++) {
  ys[i] = f(xs[i]);
}
```

Is this parallel?

Sequential programming for parallel machines

One approach: write imperative code like we've always done, and apply a *parallelising compiler* to try to figure out whether parallel execution is possible:

```
for (int i = 0; i < n; i++) {
  ys[i] = f(xs[i]);
}
```

Is this parallel? **Yes.** But it requires careful inspection of read/write indices.

Sequential programming for parallel machines

What about this one?

```
for (int i = 0; i < n; i++) {
  ys[i+1] = f(ys[i], xs[i]);
}
```

Sequential programming for parallel machines

What about this one?

```
for (int i = 0; i < n; i++) {
  ys[i+1] = f(ys[i], xs[i]);
}
```

Yes, but hard for a compiler to detect.

- Many algorithms are innately parallel, but phrased sequentially when we encode them in current languages
- A *parallelising compiler* tries to reverse engineer the original parallelism from a sequential formulation
- Possible in theory, is called *heroic effort* for a reason

Why not use a language where we can just say exactly what we mean?

Functional programming for parallel machines

Common purely functional combinators have *sequential semantics*, but permit *parallel execution*.

```
for (int i = 0; i < n; i++) {
  ys[i] = f(xs[i]);
}
```

Functional programming for parallel machines

Common purely functional combinators have *sequential semantics*, but permit *parallel execution*.

```
for (int i = 0; i < n; i++) {    ~   let ys = map f xs
  ys[i] = f(xs[i]);
}
```

```
for (int i = 0; i < n; i++) {
  ys[i+1] = f(ys[i], xs[i]);
}
```

Functional programming for parallel machines

Common purely functional combinators have *sequential semantics*, but permit *parallel execution*.

```
for (int i = 0; i < n; i++) {
  ys[i] = f(xs[i]);
}
```

~

```
let ys = map f xs
```

```
for (int i = 0; i < n; i++) {
  ys[i+1] = f(ys[i], xs[i]);
}
```

~

```
let ys = scan f xs
```

Existing functional languages are a poor fit

Unfortunately, we cannot simply write a Haskell compiler that generates GPU code:

- GPUs are restricted (no stack, no allocation, no function pointers).
- Lazy evaluation makes parallel execution very hard.
- Unstructured/nested parallelism not supported by hardware.
- Common programming style is *not sufficiently parallel!*
  For example:
  - Linked lists are inherently sequential.
  - `foldl` not necessarily parallel.
- Haskell is a good fit for libraries (REPA) or as metalanguage (Accelerate, Obsidian).

We need parallel languages that are restricted enough to make a compiler viable.

The best language is NESL by Guy Blelloch

Good: Sequential semantics; language-based cost model.

Good: Supports irregular arrays-of-arrays such as `[[1], [1,2], [1,2,3]]`.

The best language is NESL by Guy Blelloch

Good: Sequential semantics; language-based cost model.

Good: Supports irregular arrays-of-arrays such as `[[1], [1,2], [1,2,3]]`.

Amazing: The *flattening transformation* can flatten all nested parallelism (and recursion!) to flat parallelism, *while (almost) preserving asymptotic cost*!

The best language is NESL by Guy Blelloch

Good: Sequential semantics; language-based cost model.

Good: Supports irregular arrays-of-arrays such as `[[1], [1,2], [1,2,3]]`.

Amazing: The *flattening transformation* can flatten all nested parallelism (and recursion!) to flat parallelism, *while (almost) preserving asymptotic cost*!

Amazing: Runs on GPUs! *Nested data-parallelism on the GPU* by Lars Berstrom and John Reppy (ICFP 2012).

The best language is NESL by Guy Blelloch

Good: Sequential semantics; language-based cost model.

Good: Supports irregular arrays-of-arrays such as `[[1], [1,2], [1,2,3]]`.

Amazing: The *flattening transformation* can flatten all nested parallelism (and recursion!) to flat parallelism, *while (almost) preserving asymptotic cost*!

Amazing: Runs on GPUs! *Nested data-parallelism on the GPU* by Lars Berstrom and John Reppy (ICFP 2012).

Bad: Flattening preserves *time* asymptotics, but can lead to polynomial *space increases*.

Worse: The constants are horrible because flattening inhibits locality optimisations.

The problem with full flattening

Multiplying $n \times m$ and $m \times n$ matrices:

```
map (\xs -> map (\ys -> let zs = map2 (*) xs ys
                        in reduce (+) 0 zs)
                (transpose yss))
    xss
```

Flattens to:

```
let ysss = replicate n (transpose yss)      -- n x n x m
let xsss = map (replicate n) xss            -- n x n x m
let zsss = map2 (map2 (map2 (*))) xsss ysss -- n x n x m
in map (map (reduce (+) 0)) zsss            -- n x n
```

Problem: Intermediate arrays of size $n \times n \times m$.

Clearly NESL is still too flexible in some respects. Let's restrict it further to make the compiler *even more feasible*: **Futhark!**

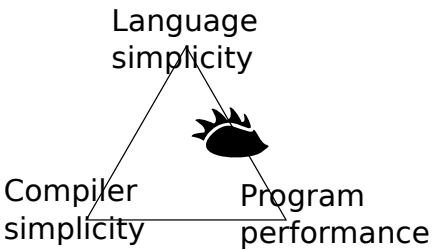
The Futhark programming language

The spirit of Futhark


Gotta go
fast

The philosophy of Futhark

- **Performance is everything!**
- Remove anything we cannot compile efficiently: E.g., recursion(!), irregular arrays.
- Accept a complex compiler—but it should spend its time on *optimisation*, rather than guessing programmer intent.



Goals

- Language co-creator Cosmin Oancea calls it *Functional Fortran*.
- **Futhark is not a GPU language!** It is a hardware-agnostic language, but *our best compiler* generates GPU code.
- Designed by working backwards from the code we wanted to generate.
  1. How would a compiler optimise code to look like that?
  2. What would the input language to that compiler look like?
  3. What does the compiler need to assume about the program?

Futhark syntax

- **Array construction**

```
iota 5            =        [0,1,2,3,4]
replicate 3 1337  = [1337, 1337, 1337]
```

- **Only regular arrays:** `[[1,2], [3]]` is a type error.

- **Second-Order Array Combinators (*SOACs*)**

$$
\begin{aligned}
\textbf{map}\ f\ [x_1, \ldots, x_n] &\rightarrow [f\ x_1,\ \ldots,\ f\ x_n] \\
\textbf{reduce}\ \odot\ 0_\odot\ [x_1, \ldots, x_n] &\rightarrow x_1 \odot \ldots \odot x_n \\
\textbf{scan}\ \odot\ 0_\odot\ [x_1, \ldots, x_n] &\rightarrow [\textbf{reduce}\ \odot\ 0_\odot\ [x_1], \\
&\quad\ \textbf{reduce}\ \odot\ 0_\odot\ [x_1, x_2], \\
&\quad\ \ldots, \\
&\quad\ \textbf{reduce}\ \odot\ 0_\odot\ [x_1, \ldots, x_n]]
\end{aligned}
$$

Futhark at a Glance

- **Size-dependent types**

  An *n* by *m* integer matrix has type `[`*n*`][`*m*`]i32`.

Futhark at a Glance

- **Size-dependent types**

  An *n* by *m* integer matrix has type `[`*n*`][`*m*`]i32`.

- **Nested Parallelism**

```
def add_two [n]    (a:  [n]i32):    [n]i32 = map (+2) a
def     sum [n]    (a:  [n]i32):       i32 = reduce (+) 0 a
def sumrows [n][m] (as: [n][m]i32): [n]i32 = map sum as
```

Futhark at a Glance

- **Size-dependent types**

  An *n* by *m* integer matrix has type `[`*n*`][`*m*`]i32`.

- **Nested Parallelism**

```
def add_two [n]    (a:  [n]i32):    [n]i32 = map (+2) a
def     sum [n]    (a:  [n]i32):       i32 = reduce (+) 0 a
def sumrows [n][m] (as: [n][m]i32): [n]i32 = map sum as
```

- **Parallism must be *regular*; this is not supported:**

```
def f [n] (a: [n]i32) =
  map (\x -> reduce (+) 0 (iota x)) a
```

Uniqueness types

Inspired by Clean; used to permit in-place modification of arrays without violating referential transparency.

```
let y = x with [i] = v
```

- `y` has same elements as `x`, except at position `i` which contains `v`.
- We say that `x` has been *consumed*.
- Compiler verifies that `x` is not used afterwards.

Uniqueness types

Inspired by Clean; used to permit in-place modification of arrays without violating referential transparency.

```
let y = x with [i] = v
```

- `y` has same elements as `x`, except at position `i` which contains `v`.
- We say that `x` has been *consumed*.
- Compiler verifies that `x` is not used afterwards.

Shorthand

When $x \equiv y$, we write:

```
let x[i] = 0
```

This is just syntactical sugar for variable shadowing.

Uniqueness Type Annotations

A function can uniqueness-annotate its parameters and return type:

```
def copy_one (xs: *[]i32) (ys: []i32) (i: i32): *[]i32 =
  let xs[i] = ys[i]
  in xs
```

For a parameter, * means the argument will never be used again by the caller.

For a return value, * means the returned value does not alias any (non-unique) parameter.

An application `let xs' = copy_one xs ys i` is valid if `xs` can be consumed and consumes `xs`. The result `xs'` has no aliases.

Parallel in-place updates

Type

```
val scatter [k] [n] 'a :
     (dest: *[k]a)
  -> (is: [n]i32)
  -> (vs: [n]a)
  -> *[k]a
```

Semantics, operationally

```
for j in 0..k-1:
  i = is[j]
  v = vs[j]
  dest[i] = v
```

Beware! Multiple writes to same index with different values is *undefined*.

Example use of `scatter`

```
def filter [n] 'a (p: a -> bool) (as: [n]a): ?[m].[m]a =
  let flags = map p as
  let flags_int = map i64.bool flags
  let offsets = scan (+) 0 flags_int
  let put_in i f = if f then i-1 else -1
  let is = map2 put_in offsets flags
  let as' = scatter (copy as) is as
  in take (offsets[n-1]) as'
```

Example for `filter (<0) [1,-1,2,3,-2]`:

```
as         = [    1,   -1,     2,     3,   -2]
flags      = [false, true, false, false, true]
flags_int  = [    0,    1,     0,     0,    1]
offsets    = [    0,    1,     1,     1,    2]
is         = [   -1,    0,    -1,    -1,    1]
as'        = [   -1,   -2,     2,     3,   -2]
```

Generalised histograms

Like `scatter`, but uses operator to combine multiple writes to same index.

```
val reduce_by_index [k] [n] 'a :
      (dest: *[k]a)
  -> (f: a -> a -> a) -> (ne: a)
  -> (is: [n]i32) -> (vs: [n]a)
  -> *[k]a
```

Semantics, operationally

```
for j in 0..k-1:
  i = is[j]
  v = vs[j]
  dest[i] = f(dest[i], v)
```

Parallel implementation based on *atomics*: `https://futhark-lang.org/publications/sc20.pdf`

Compiler transformations

Loop fusion

Let's say we wish to first call `add_two`, then `sumrows` (with some matrix *a*):

`sumrows` (`add_two` *a*)

- A naive compiler would first run `add_two`, producing an entire matrix in memory, then pass it to `sumrows`.
- This problem is bandwidth-bound, so unnecessary memory traffic will impact our performance.
- Avoiding unnecessary intermediate structures is known as *deforestation*, and is a well known technique for functional compilers.
- Easy to implement for a data-parallel language as *loop fusion*.

An example of a fusion rule

The expression

$$\mathbf{map}\ f\ (\mathbf{map}\ g\ a)$$

is *always* equivalent to

$$\mathbf{map}\ (f \circ g)\ a$$

- This is an extremely powerful property that is only true in the absence of side effects.
- Fusion is *the* core optimisation that permits the efficient decomposition of a data-parallel program.
- A full fusion engine has much more awkward-looking rules (`zip`/`unzip` causes lots of bookkeeping), but safety is guaranteed.

A fusion example

$$
\begin{aligned}
\texttt{sumrows}\ (\texttt{add\_two}\ a) &= && \text{(Initial expression)} \\
\mathbf{map}\ \texttt{sum}\ (\texttt{add\_two}\ a) &= && \text{(Inline sumrows)} \\
\mathbf{map}\ \texttt{sum}\ (\mathbf{map}\ (\lambda r \rightarrow \mathbf{map}\ (+2)\ r)\ a) &= && \text{(Inline add\_two)} \\
\mathbf{map}\ (\texttt{sum} \circ (\lambda r \rightarrow \mathbf{map}\ (+2)\ r)\ a) &= && \text{(Apply \textbf{map}-\textbf{map} fusion)} \\
\mathbf{map}\ (\lambda r \rightarrow \texttt{sum}\ (\mathbf{map}\ (+2)\ r)\ a) && && \text{(Apply composition)}
\end{aligned}
$$

- Temporary matrix eliminated, but the composition of `sum` and the **`map`** also holds an opportunity for fusion – specifically, **`reduce`**-**`map`** fusion
- Will not cover in detail, but a **`reduce`** can efficiently apply a function to each input element before engaging in the actual reduction.
- **Important**: a **`map`** going into a **`reduce`** is an efficient pattern

Handling nested parallelism

Problem Futhark permits *nested* (regular) parallelism, but GPUs prefer *flat* parallel *kernels*.

Handling nested parallelism

Problem Futhark permits *nested* (regular) parallelism, but GPUs prefer *flat* parallel *kernels*.

Solution Have the compiler rewrite program to *perfectly nested* **map**s containing sequential code (or known parallel patterns such as nested **reduce**), each of which can become a GPU kernel.

```
map (\xs -> let y = reduce (+) 0 xs
            in map (+y) xs)
    xss
```

```
let ys = map (\xs -> reduce (+) 0 xs) xss
in map (\xs y -> map (+y) xs) xss ys
```

Moderate flattening via loop fission

The classic map fusion rule:

$$\text{map } f \circ \text{map } g \Rightarrow \text{map } (f \circ g)$$

Moderate flattening via loop fission

The classic map fusion rule:

$$\text{map } f \circ \text{map } g \Rightarrow \text{map } (f \circ g)$$

We can also apply it backwards to obtain *fission*:

$$\text{map } (f \circ g) \Rightarrow \text{map } f \circ \text{map } g$$

This, along with other higher-order rules (see PLDI paper), are applied by the compiler to extract perfect map nests.

Example: (a) Initial program, we inspect the map-nest

```
let (asss, bss) =
  map (\(ps: [m]i64) ->
        let ass = map (\(p: i64): [m]i64 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in map (+r) ps) ps
        let bs = loop ws=ps for i < n do
                    map (\as w: i64 ->
                           let d = reduce (+) 0 as
                           let e = d + w
                           in 2 * e) ass ws
        in (ass, bs)) pss
```

We assume the type of `pss : [m][m]i64`.

(b) Distribution

```
let (asss: [m][m][m]i64) =
  map (\(ps: [m]i64) ->
        let ass = map (\(p: i64): [m]i64 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in map (+r) ps) ps
        in ass) pss
let (bss: [m][m]i64) =
  map (\ps ass ->
        let bs = loop ws=ps for i < n do
                   map (\as w ->
                          let d = reduce (+) 0 as
                          let e = d + w
                          in 2 * e) ass ws
        in bs) pss asss
```

(c) Interchanging outermost map inwards

```
let (asss: [m][m][m]i64) =
  map (\(ps: [m]i64) ->
        let ass = map (\(p: i64): [m]i64 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in map (+r) ps) ps
        in ass) pss
let (bss: [m][m]i64) =
  map (\ps ass ->
        let bs = loop ws=ps for i < n do
                    map (\as w ->
                           let d = reduce (+) 0 as
                           let e = d + w
                           in 2 * e) ass ws
        in bs) pss asss
```

(c) Interchanging outermost map inwards

```
let (asss: [m][m][m]i64) =
  map (\(ps: [m]i64) ->
        let ass = map (\(p: i64): [m]i64 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in map (+r) ps) ps
        in ass) pss
let (bss: [m][m]i64) =
  loop wss=pss for i < n do
    map (\ass ws ->
          let ws' = map (\as w ->
                           let d = reduce (+) 0 as
                           let e = d + w
                           in 2 * e) ass ws
          in ws') asss wss
```

(d) Skipping scalar computation

```
let (asss: [m][m][m]i64) =
  map (\(ps: [m]i64) ->
        let ass = map (\(p: i64): [m]i64 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in map (+r) ps) ps
        in ass) pss
let (bss: [m][m]i64) =
  loop wss=pss for i < n do
    map (\ass ws ->
          let ws' = map (\as w ->
                           let d = reduce (+) 0 as
                           let e = d + w
                           in 2 * e) ass ws
          in ws') asss wss
```

(d) Skipping scalar computation

```
let (asss: [m][m][m]i64) =
  map (\(ps: [m]i64) ->
         let ass = map (\(p: i64): [m]i64 ->
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let (bss: [m][m]i64) =
  loop wss=pss for i < n do
    map (\ass ws ->
           let ws' = map (\as w ->
                            let d = reduce (+) 0 as
                            let e = d + w
                            in 2 * e) ass ws
           in ws') asss wss
```

(e) Distributing reduction.

```
let (asss: [m][m][m]i64) =
  map (\(ps: [m]i64) ->
         let ass = map (\(p: i64): [m]i64 ->
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let (bss: [m][m]i64) =
  loop wss=pss for i < n do
    map (\ass ws ->
           let ws' = map (\as w ->
                            let d = reduce (+) 0 as
                            let e = d + w
                            in 2 * e) ass ws
           in ws') asss wss
```

(e) Distributing reduction

```
let (asss: [m][m][m]i64) =
  map (\(ps: [m]i64) ->
         let ass = map (\(p: i64): [m]i64 ->
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let (bss: [m][m]i64) =
  loop wss=pss for i < n do
    let dss: [m][m]i64 =
      map (\ass -> map (\as -> reduce (+) 0 as) ass) asss
    in map (\ws ds ->
              let ws' = map (\w d -> let e  = d + w in 2 * e) ws ds
              in ws')
           asss dss
```

(f) Distributing inner map

```
let (asss: [m][m][m]i64) =
  map (\(ps: [m]i64) ->
        let ass = map (\(p: i64): [m]i64 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in map (+r) ps) ps
        in ass) pss
let (bss: [m][m]i64) = ...
```

(f) Distributing inner map

```
let (rss: [m][m]i64) =
  map (\(ps: [m]i64) ->
        let rss = map (\(p: i64): i64 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in r) ps
        in rss) pss
let (asss: [m][m][m]i64) =
  map (\(ps: [m]i64) (rs: [m]i64) ->
        map (\(r: i64): [m]i64 ->
               map (+r) ps) rs
      ) pss rss
let (bss: [m][m]i64) = ...
```

(g) Cannot distribute as it would create irregular array

```
let (rss: [m][m]i64) =
  map (\(ps: [m]i64) ->
        let rss = map (\(p: i64): i64 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in r) ps
        in rss) pss
let (asss: [m][m][m]i64) = ...
let (bss: [m][m]i64) = ...
```

Array `cs` has type `[p]i64`, and `p` is variant to the innermost map nest.

(h) These statements are sequentialised

```
let (rss: [m][m]i64) =
  map (\(ps: [m]i64) ->
        let rss = map (\(p: i364: i64 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in r) ps
        in rss) pss
let (asss: [m][m][m]i64) = ...
let (bss: [m][m]i64) = ...
```

Array `cs` has type `[p]i64`, and `p` is variant to the innermost map nest.

Result

```
let (rss: [m][m]i64) = map (\ps -> map (...) ps) pss
let (asss: [m][m][m]i64) =
  map (\ps rs -> map (\r -> map (...) ps) rs) pss rss
let (bss: [m][m]i64) =
  loop wss=pss for i < n do
    let (dss: [m][m]i64) = map (\ass -> map (reduce ...) ass)
                               asss
    in map (\ws ds -> map (...) ws ds ) asss dss
```

From single kernel with parallelism m to four kernels of parallelism m^2, m^3, m^3, m^2.

The last two kernels are executed n times each.

Applied Futhark programming

Simple 1D Stencil

Simple 1D Stencil

```
def smoothen (centres: []f32) =
  let rights = rotate 1 centres
  let lefts = rotate (-1) centres
  in map3 (\l c r -> (l+c+r)/3) lefts centres rights

def main (xs: []f32) =
  iterate 10 smoothen xs
```

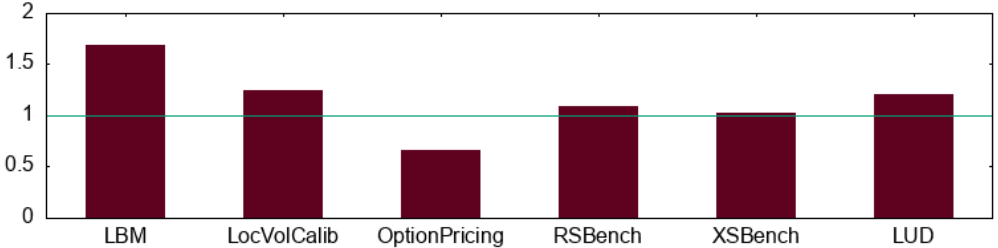
So is it fast?

The Question: Is it possible to construct a purely functional hardware-agnostic programming language that is convenient to use and provides good parallel performance?

Hard to Prove: Only performance is easy to quantify, and even then…

- No good objective criterion for whether a language is “fast”
- Best practice is to take benchmark programs written in other languages, port or re-implement them, and see how they behave
- These benchmarks originally written in low-level CUDA or OpenCL

Speedup of Futhark versus hand-written GPU code—higher is better



- CUDA and OpenCL implementations of widely varying quality.
- This makes them “realistic”, in a sense.
- (We have over 40 benchmarks, but write preparing these slides I was reminded that while it’s easy to re-measure Futhark results, that isn’t the case for the original programs…)

For the lab exercise: largest element and its index

```
def argmax [n] (xs: [n]i32) =
  reduce (\(x, i) (y, j) ->
            if x < y then (y, j) else (x, i))
         (i32.lowest, -1)
         (zip xs (iota n))
```

For the lab exercise: `concat` and sizes

`(++)` produces an unknown size, so we often need to do a size coercion afterwards[1]:

```
def f (xs: [n]i32) (ys: [n]i32) =
  let npn = n + n
  let xs' = xs ++ xs :> [npn]i32
  let ys' = ys ++ ys :> [npn]i32
  in zip xs' ys'
```

See also

```
val concat_to [n] [m] 't :
  (k: i64) -> (xs: [n]t) -> (ys: [m]t) -> [k]t}
```

[1] Unless you are using the unreleased nightly Futhark, which has a more expessive type system.

Visualisation of Ising Model

(If it works...)

Additional Reading

Quickstart guide if you already know functional programming `http://futhark.readthedocs.io/en/latest/versus-other-languages.html`

Glossary

`https://futhark.readthedocs.io/en/latest/glossary.html`

Code examples

`https://futhark-lang.org/examples.html`

Futhark prelude documentation

`https://futhark-lang.org/docs/prelude/`

Relevant packages

- `https://futhark-lang.org/pkgs/github.com/diku-dk/cpprandom/latest/`
- `https://futhark-lang.org/pkgs/github.com/diku-dk/segmented/latest/`