Data Parallel Programming in Futhark

Troels Henriksen (athas@sigkill.dk)

DIKU
University of Copenhagen

19th of April, 2018

$\lambda x.x$

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

Agenda

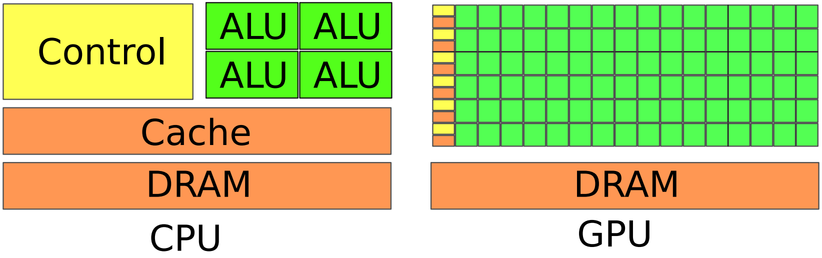
- GPUs—why and how
- Basic Futhark programming
- Compiler transformation—fusion and moderate flattening
- Real world Futhark programming
    - 1D smoothing and benchmarking
    - Talking to the outside world
    - Maybe some hints for the lab assignment

GPUs—why and how

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).

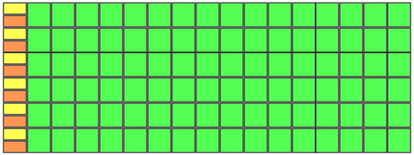
GPUs vs CPUs



- 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*.

Massively parallel processing is currently a special case, but will be the common case in the future.

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

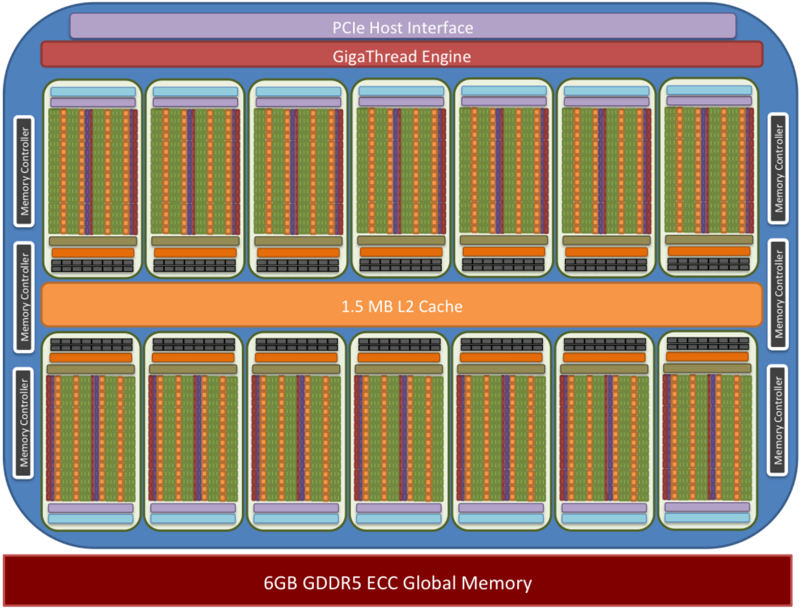
Both branches are executed in all threads, but in those threads where the condition is false, a mask bit is set to treat the instructions inside the branch as no-ops.

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

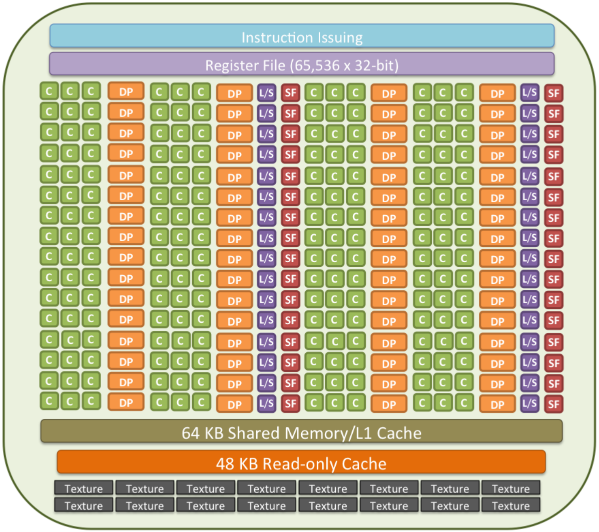
Execution Model

- 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.

GPU layout


PCIe Host Interface
GigaThread Engine
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
Memory Controller
1.5 MB L2 Cache
6GB GDDR5 ECC Global Memory

SM layout



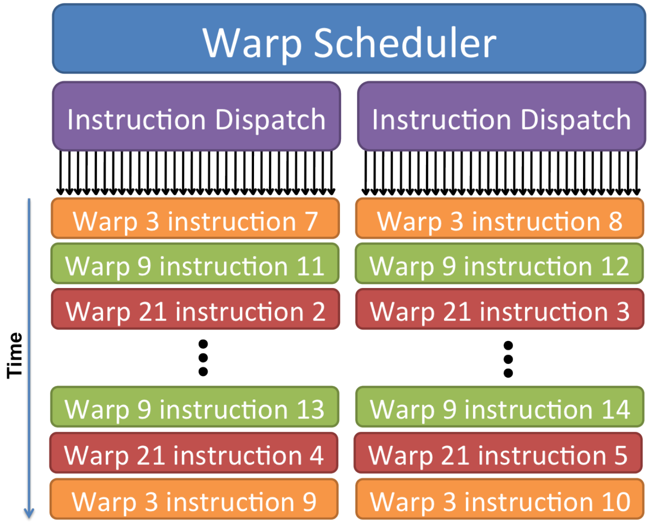
C single precision/integer CUDA core

DP double precision FP unit

L/S memory load/store unit

SF special function unit

Warp scheduling



Do GPUs exist in theory as well?

GPU programming is a close fit to the *bulk synchronous parallel* paradigm:

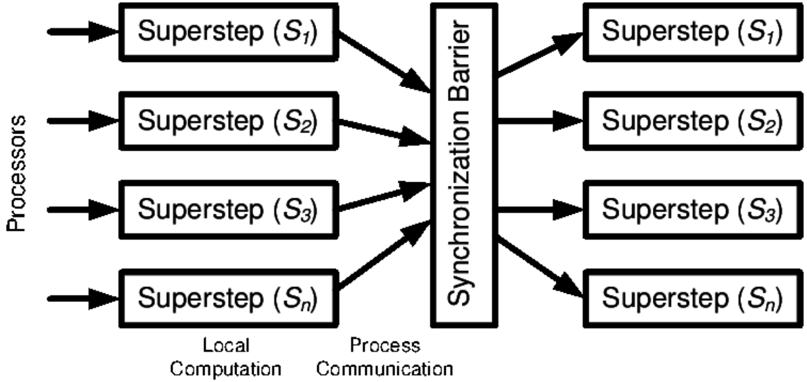


Illustration by Aftab A. Chandio; observation by Holger Fröning.

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.

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]);
}
```

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;          ∼   let ys = map f xs
     i < n;
     i++) {
  ys[i] = f(xs[i]);
}
```

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

Existing functional languages are a poor fit

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

- GPUs are too restricted (no stack, no allocations inside kernels, 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 still a good fit for libraries (REPA) or as a 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 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 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 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 access pattern optimisations.

The problem with full flattening

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

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

Flattens to:

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

Problem: Intermediate arrays of size $n \times n \times m$.
We will return to this.

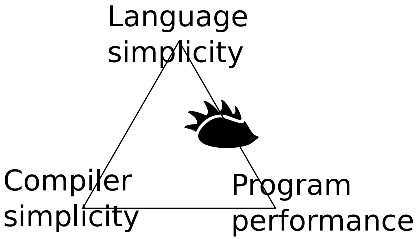
Clearly NESL is still too flexible in some respects. Let's restrict it further to make the compiler *even more feasible*: **Futhark!**

The philosophy of Futhark

The philosophy of Futhark

The philosophy of Futhark

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



- **Futhark is not a GPU language!** It is a hardware-agnostic language, but *our best compiler* generates GPU code.

Futhark at a Glance

Small eagerly evaluated pure functional language with data-parallel constructs. Syntax is a combination of C, SML, and Haskell.

- **Data-parallel loops**

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

- **Array construction**

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

– Only regular arrays: `[[1,2], [3]]` is illegal.

- **Sequential loops**

```
loop x = 1 for i < n do
  x * (i + 1)
```

COMPILER OPTIMISATIONS

Oh, look! It changed shape! Did you see that?!
—Miles “Tails” Prower (Sonic Adventure, 1998)

Loop Fusion

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

`sumrows (increment` a`)`

- A naive compiler would first run `increment`, 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.
- It is easy to implement for a data-parallel language as *loop fusion*.

An Example of a Fusion Rule

The expression

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

is *always* equivalent to

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

- 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

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

- We have avoided the temporary matrix, 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 operation.
- Important to remember: a **`map`** going into a **`reduce`** is an efficient pattern.

Handling Nested Parallelism

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

Handling Nested Parallelism

The 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 segmented reduction), each of which can become a GPU kernel.

Handling Nested Parallelism

The 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 segmented reduction), each of which can become a GPU kernel.

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

$$\Downarrow$$

```
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]i32) ->
         let ass = map (\(p: i32): [m]i32 ->
                            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: i32 ->
                             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]i32`.

(b) Distribution.

```
let asss: [m][m][m]i32 =
  map (\(ps: [m]i32) ->
         let ass = map (\(p: i32): [m]i32 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  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]i32 =
  map (\(ps: [m]i32) ->
         let ass = map (\(p: i32): [m]i32 ->
                            let cs = scan (+) 0 (iota p)
                            let r = reduce (+) 0 cs
                            in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  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]i32 =
  map (\(ps: [m]i32) ->
        let ass = map (\(p: i32): [m]i32 ->
                        let cs = scan (+) 0 (iota p)
                        let r = reduce (+) 0 cs
                        in map (+r) ps) ps
        in ass) pss
let bss: [m][m]i32 =
  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]i32 =
  map (\(ps: [m]i32) ->
        let ass = map (\(p: i32): [m]i32 ->
                        let cs = scan (+) 0 (iota p)
                        let r = reduce (+) 0 cs
                        in map (+r) ps) ps
        in ass) pss
let bss: [m][m]i32 =
  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]i32 =
  map (\(ps: [m]i32) ->
         let ass = map (\(p: i32): [m]i32 ->
                         let cs = scan (+) 0 (iota p)
                         let r = reduce (+) 0 cs
                         in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  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]i32 =
  map (\(ps: [m]i32) ->
         let ass = map (\(p: i32): [m]i32 ->
                                 let cs = scan (+) 0 (iota p)
                                 let r = reduce (+) 0 cs
                                 in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  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]i32 =
  map (\(ps: [m]i32) ->
         let ass = map (\(p: i32): [m]i32 ->
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in map (+r) ps) ps
         in ass) pss
let bss: [m][m]i32 =
  loop wss=pss for i < n do
    let dss: [m][m]i32 =
      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 =
  map (\(ps: [m]i32) ->
       let ass = map (\(p: i32): [m]i32 ->
                      let cs = scan (+) 0 (iota p)
                      let r = reduce (+) 0 cs
                      in map (+r) ps) ps
       in ass) pss
let bss: [m][m]i32 = ...
```

(f) Distributing inner map.

```
let rss: [m][m]i32 =
  map (\(ps: [m]i32) ->
         let rss = map (\(p: i32): i32 ->
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in r) ps
         in rss) pss
let asss: [m][m][m]i32 =
  map (\(ps: [m]i32) (rs: [m]i32) ->
         map (\(r: i32): [m]i32 ->
                map (+r) ps) rs
      ) pss rss
let bss: [m][m]i32 = ...
```

(g) Cannot distribute as it would create irregular array.

```
let rss: [m][m]i32 =
  map (\(ps: [m]i32) ->
         let rss = map (\(p: i32): i32 ->
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in r) ps
         in rss) pss
let asss: [m][m][m]i32 = ...
let bss: [m][m]i32 = ...
```

Array `cs` has type `[p]i32`, and p is variant to the innermost map nest.

(h) These statements are sequentialised

```
let rss: [m][m]i32 =
  map (\(ps: [m]i32) ->
         let rss = map (\(p: i32): i32 ->
                          let cs = scan (+) 0 (iota p)
                          let r = reduce (+) 0 cs
                          in r) ps
         in rss) pss
let asss: [m][m][m]i32 = ...
let bss: [m][m]i32 = ...
```

Array `cs` has type `[p]i32`, and p is variant to the innermost map nest.

Result

```
let rss: [m][m]i32 = map (\ps -> map (...) ps) pss
let asss: [m][m][m]i32 =
  map (\ps rs -> map (\r -> map (...) ps) rs) pss rss
let bss: [m][m]i32 =
  loop wss=pss for i < n do
    let dss: [m][m]i32 = map (\ass -> map (reduce ...) ass)
                                     asss
    in map (\ws ds -> map (...) ws ds ) asss dss
```

From a single kernel with parallelism m to four kernels of parallelism m^2, m^3, m^3, and m^2.
The last two kernels are executed n times each.

Real world Futhark programming

Aw, yeah! This is happenin'!
—Sonic the Hedgehog (Sonic Adventure, 1998)

Simple 1D Stencil

Simple 1D Stencil

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

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

Making Futhark useful

$$\begin{tikzcd}\text{Sequential CPU program} \arrow[r, shift left=2] \arrow[r, leftarrow] \arrow[r, shift right=2] & \text{Parallel GPU program}\end{tikzcd}$$

The controlling CPU program does not have to be *fast*. It can be generated in a language that is *convenient*.

Compiling Futhark to Python+PyOpenCL

```
entry sum_nats (n: i32): i32 =
  reduce (+) 0 (1...n)
```

```
$ futhark-pyopencl --library sum.fut
```

This creates a Python module `sum.py` which we can use as follows:

```
$ python
>>> from sum import sum
>>> c = sum()
>>> c.sum_nats(10)
55
>>> c.sum_nats(1000000)
1784293664
```

Good choice for all your integer summation needs!

Compiling Futhark to Python+PyOpenCL

```
entry sum_nats (n: i32): i32 =
  reduce (+) 0 (1...n)
```

```
$ futhark-pyopencl --library sum.fut
```

This creates a Python module `sum.py` which we can use as follows:

```
$ python
>>> from sum import sum
>>> c = sum()
>>> c.sum_nats(10)
55
>>> c.sum_nats(1000000)
1784293664
```

Good choice for all your integer summation needs!

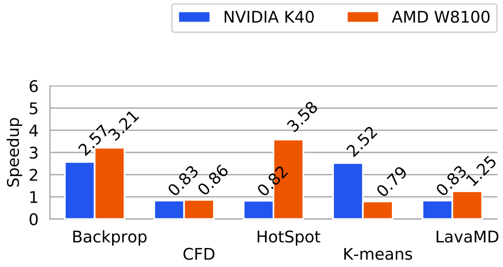
Or, we could have our Futhark program return an array containing pixel colour values, and use Pygame to blit it to the screen...

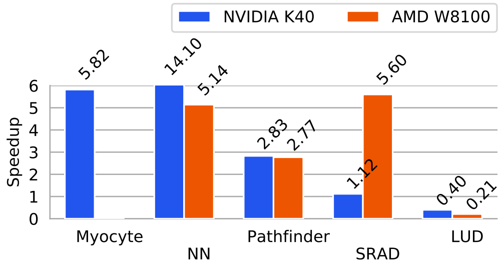
So is it fast?

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.

Rodinia





- CUDA and OpenCL implementations of widely varying quality.
- This makes them “realistic”, in a sense.

On the Lab Exercise

Do I need a reason to want to help out a friend?
—Sonic the Werehog (Sonic Unleashed, 2008)

Largest element and its index

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

Example of `scatter`

```
let filter [n] 'a (p: a -> bool) (as: [n]a): []a =
  let flags = map p as
  let offsets = scan (+) 0 (map i32.bool flags)
  let put_in i f = if f then i-1 else -1
  let is = map2 put_in offsets flags
  in take (offsets[n-1]) (scatter (copy as) is as)
```

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

```
flags   = [false, true, false, false, true]
offsets = [    0,    1,     1,     1,    2]
is      = [   -1,    0,    -1,    -1,    1]
```

Visulisation 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`

Basis library documentation

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

Of particular interest:

- `/futlib/soac`
- `/futlib/functional`
- `/futlib/array`
- `/futlib/random`
- `/futlib/sobol`