Purely Functional GPU Programming with Futhark

Troels Henriksen (athas@sigkill.dk)

DIKU
University of Copenhagen

11th of July — GPU Day 2019

- 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.

Two Helpful 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 Helpful 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)

The problems we evolved to solve

The problems we are now trying to solve

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?

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;        ~  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]);
}
```

So this is solved?

Problem: Turns out purely functional languages are really slow when compiled naively, and GPUs only support certain restricted forms of parallelism anyway.

So this is solved?

Problem: Turns out purely functional languages are really slow when compiled naively, and GPUs only support certain restricted forms of parallelism anyway.

Solution: Spend many years years co-designing a simple language and a non-simple optimising compiler capable of compiling it to efficient GPU code: **Futhark!**

Futhark is a high-level language!

Sequential semantics, parallel operation
: Futhark is *not* a "GPU language"—it is a hardware-agnostic parallel language.

Co-design of language and compiler
: No language features that we do not know how to compile efficiently. (No recursion! (Yet.))

Futhark is a high-level language!

Sequential semantics, parallel operation
: Futhark is *not* a “GPU language”—it is a hardware-agnostic parallel language.

Co-design of language and compiler
: No language features that we do not know how to compile efficiently. (No recursion! (Yet.))

This presentation is a tour of the language design and compilation techniques for generating good GPU code.

Futhark at a Glance

- **Array construction**
  ```
  iota 5              =         [0,1,2,3,4]
  replicate 3 1337 = [1337, 1337, 1337]
  ```
- **Only regular arrays:** `[[1,2], [3]]` is illegal.
- **Second-Order Array Combinators (*SOACs*)**

$$
\begin{array}{rcl}
\textbf{map}\ f\ [x_1,\ldots,x_n] & \rightarrow & [f\ x_1,\ \ldots,\ f\ x_n] \\
\textbf{map2}\ g\ [x_1,\ldots,x_n]\ [y_1,\ldots,y_n] & \rightarrow & [g\ x_1\ y_1,\ \ldots,\ g\ x_n\ y_n] \\
\texttt{reduce}\ \odot\ 0_\odot\ [x_1,\ldots,x_n] & \rightarrow & x_1 \odot \ldots \odot x_n \\
\texttt{scan}\ \odot\ 0_\odot\ [x_1,\ldots,x_n] & \rightarrow & [\texttt{reduce}\ \odot\ 0_\odot\ [x_1], \\
 & & \ \texttt{reduce}\ \odot\ 0_\odot\ [x_1,x_2], \\
 & & \ \ldots, \\
 & & \ \texttt{reduce}\ \odot\ 0_\odot\ [x_1,\ldots,x_n]]
\end{array}
$$

Operator restrictions

Functions/operators used for **`reduce`** and **`scan`** must be *associative* and have a *neutral element*.

Associativity

$$(x \odot y) \odot z = x \odot (y \odot z)$$

Neutral element

$$x \odot 0_{\odot} = 0_{\odot} \odot x = x$$

Example: $*$ is associative and has 1 as neutral element.

Automatically checking this is *undecidable*, so we trust the programmer.

Futhark at a Glance, continued

- **Data-parallel loops**

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

- **Sequential loops**

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

- **Everything else**

  if expressions, higher-order functions, tuples, records, module system, type inference, etc. Most of what you expect in a functional language.

Compiling Futhark to Python+PyOpenCL

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

Compiling Futhark to Python+PyOpenCL

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

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

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...

FLATTENING NESTED DATA PARALLELISM

The Problem

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

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.

The Problem

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

Solution: Have the compiler rewrite program to perfectly nested maps 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 map2 (\xs y -> map (+y) xs) xss ys
```

Moderate flattening is a heuristic

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

$$\Downarrow$$

```
let ys = map (\xs -> reduce (+) 0 xs) xss
in map2 (\xs y -> map (+y) xs) xss ys
```

Maybe *the fastest thing* is to launch one thread per element of `xss`, even if that is less parallel?

Consider Matrix Multiplication

```
for i < n:
  for j < m:
    acc = 0
    for l < p:
      acc += xss[i,l] * yss[l,j]
    res[i,j] = acc
```

Turning it Functional

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

Using `redomap` notation

```
map (\xs ->
     map (\ys ->
          redomap2 (+) (*) 0 xs ys)
         (transpose yss))
    xss
```

Using `redomap` notation

```
map (\xs ->
     map (\ys ->
          redomap2 (+) (*) 0 xs ys)
         (transpose yss))
    xss
```

$$\mathbf{redomap2}\ \odot\ f\ 0_\odot\ x\ y \ \equiv\ \mathbf{reduce}\ \odot\ 0_\odot\ (\mathbf{map2}\ f\ x\ y)$$

Emphasizes that a **map-reduce** composition can be turned into a fused tight sequential loop, or into a parallel reduction.

So how should we parallelise this on GPU?

So how should we parallelise this on GPU?

Full flattening

```
map (\xs ->
  map (\ys ->
    redomap2 (+) (*) 0
             xs ys)
   (transpose yss))
 xss
```

- **All parallelism exploited**
- Some communication overhead.
- *Best if the outer* **maps** *do not saturate the GPU.*

So how should we parallelise this on GPU?

Full flattening

```
map (\xs ->
  map (\ys ->
    redomap2 (+) (*) 0
                 xs ys)
   (transpose yss))
 xss
```

- **All parallelism exploited**
- Some communication overhead.
- *Best if the outer* **maps** *do not saturate the GPU.*

Moderate flattening

```
map (\xs ->
  map (\ys ->
    redomap2 (+) (*) 0
                 xs ys)
   (transpose yss))
 xss
```

- **Only cheap outer parallelism**
- The **redomap2** can then be block tiled.
- *Best if the outer* **maps** *saturate the GPU.*

There is no *one size fits all*—and both situations may be encountered during the program runtime.

Simple Incremental Flattening

At every level of `map`-nesting we have two options:

1. Continue flattening inside the `map`, exploiting the parallelism there.
2. Sequentialise the `map` body; exploiting only the parallelism on top.

- **Moderate flattening**—Futhark's old technique—uses a heuristic to pick between these options. E.g, nested **`redomap`**s are always sequentialised.
- **Incremental flattening** generates *both* versions and uses a predicate to pick at runtime.

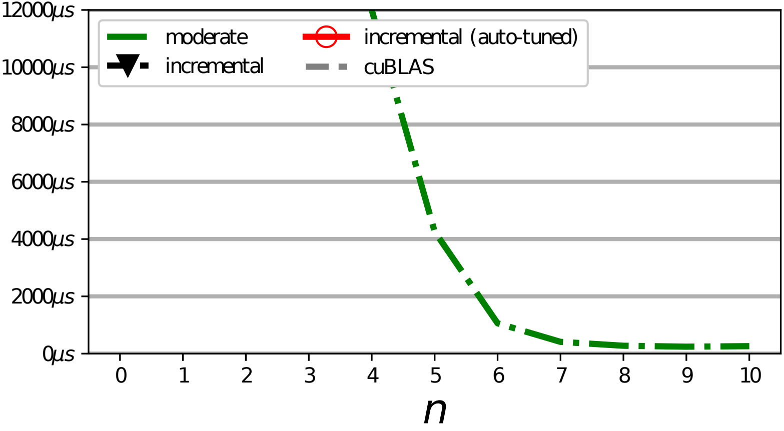
Multi-versioned matrix multiplication

```
xss :  [n][p]i32
yss :  [p][m]i32.
```

```
if n * m > t0 then
  map (\xs ->
         map (\ys ->
                redomap2 (+) (*) xs ys)
             (transpose yss))
      xss
else
  map (\xs ->
         map (\ys ->
                redomap2 (+) (*) xs ys)
             (transpose yss))
      xss
```

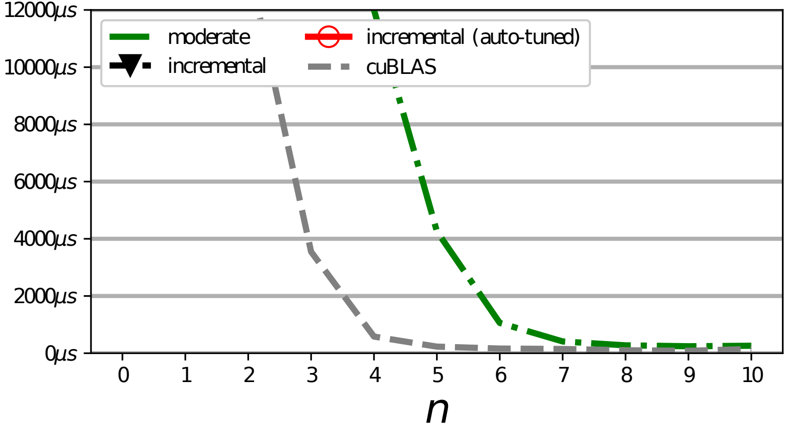
The t_0 *threshold parameter* is used to select between the two versions—and should be auto-tuned on the concrete hardware.

Matrix multiplication on NVIDIA K40



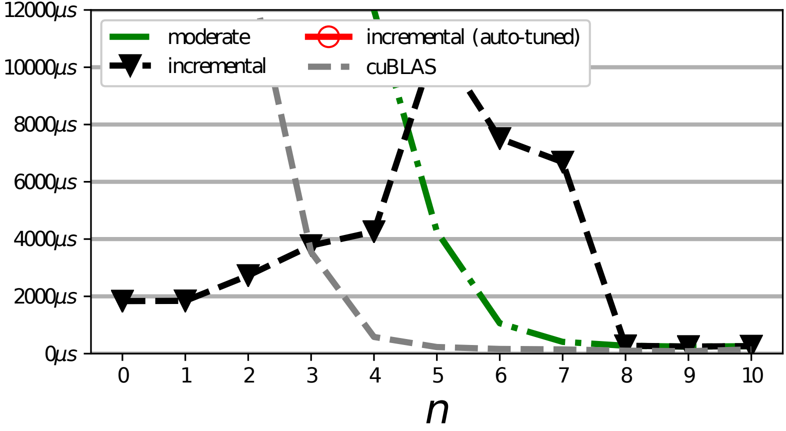
Multiplying matrices of size $2^n \times 2^m$ and $2^m \times 2^n$, where $m = 25 - 2n$, meaning that work is constant as we vary n.

Matrix multiplication on NVIDIA K40



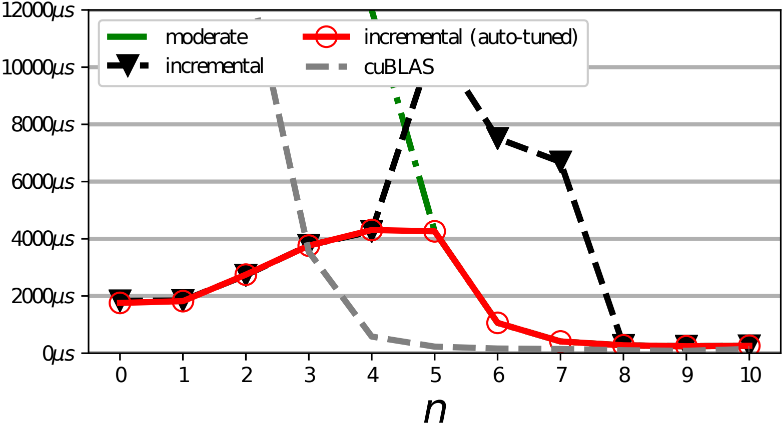
Multiplying matrices of size $2^n \times 2^m$ and $2^m \times 2^n$, where $m = 25 - 2n$, meaning that work is constant as we vary n.

Matrix multiplication on NVIDIA K40



Multiplying matrices of size $2^n \times 2^m$ and $2^m \times 2^n$, where $m = 25 - 2n$, meaning that work is constant as we vary n.

Matrix multiplication on NVIDIA K40



Multiplying matrices of size $2^n \times 2^m$ and $2^m \times 2^n$, where $m = 25 - 2n$, meaning that work is constant as we vary n.

INTRA-GROUP PARALLELISM

More complex nested parallelism

The following is the essential core of the LocVolCalib benchmark from the FinPar suite.

```
map (\xss ->
      map (\xs ->
            let bs = scan ⊕ d⊕ xs
            let cs = scan ⊗ d⊗ bs
            in  scan ⊙ d⊙ cs)
          xss)
    xsss
```

How can we map the application parallelism to hardware parallelism?

Option I: sequentialise the inner scans

```
map (\xss ->
       map (\xs ->
              let bs = scan ⊕ d⊕ xs
              let cs = scan ⊗ d⊗ bs
              in  scan ⊙ d⊙ cs)
           xss)
    xsss
```

scan is relatively expensive in parallel, so this is a good option if the outer dimensions provide enough parallelism.

Option II: flatten and parallelise inner scans

Moderate and incremental flattening uses *loop distribution* (or *fission*) to create **map** nests:

```
map (\xss ->
      map (\xs ->
            let bs = scan ⊕ d⊕ xs
            let cs = scan ⊗ d⊗ bs
            in  scan ⊙ d⊙ cs)
          xss)
    xsss
```

Option II: flatten and parallelise inner scans

Moderate and incremental flattening uses *loop distribution* (or *fission*) to create **map** nests:

```
let csss =
  map (\xss ->
        map (\xs ->
              let bs = scan ⊕ d⊕ xs
              let cs = scan ⊗ d⊗ bs
              in  cs)
            xss)
      xsss
in
  map (\css -> map (\cs -> scan ⊙ d⊙ cs)
                   css)
      csss
```

Option II: flatten and parallelise inner scans

Moderate and incremental flattening uses *loop distribution* (or *fission*) to create **map** nests:

```
let bsss =
  map (\xss -> map (\xs -> scan ⊕ d⊕ xs) xss)
      xsss
let csss =
  map (\bss -> map (\bs -> scan ⊗ d⊗ bs) bss)
      bsss
in
  map (\css -> map (\cs -> scan ⊙ d⊙ cs) css)
      csss
```

- Each **map** nests correspond to a segmented scan operation, which is straightforward to execute on the GPU.
- Moderate flattening does this.

Option III: Mapping innermost parallelism to the workgroup level

```
map (\xss ->
       map (\xs ->
              let bs = scan ⊕ d⊕ xs
              let cs = scan ⊗ d⊗ bs
              in  scan ⊙ d⊙ cs)
            xss)
    xsss
```

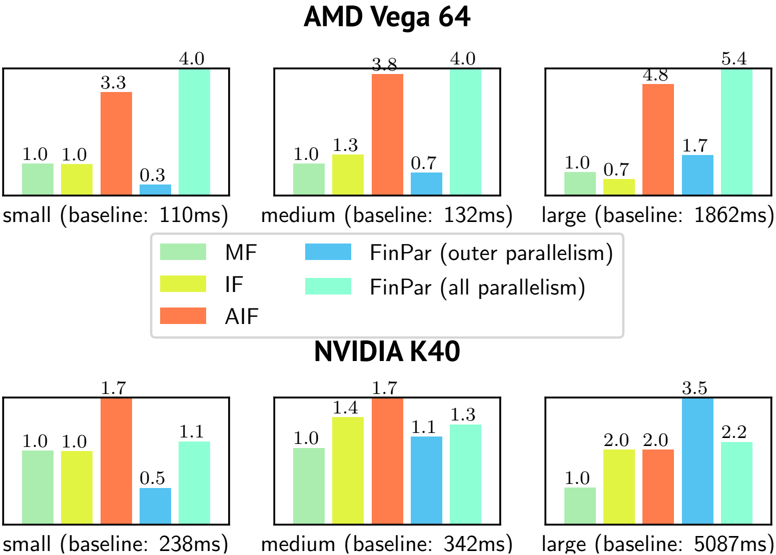
- Each iteration of the outer **map**s is assigned to one GPU workgroup[1], and each **scan** is executed intra-workgroup and in local memory[2].
- Only works if the innermost parallelism fits in a workgroup.

---

[1] *Thread block* in CUDA

[2] *Shared memory* in CUDA

LocVolCalib performance



Speedup versus moderate flattening. Higher is better.

Other Optimisations Performed by the Futhark Compiler

- Aggressive fusion:

$$\textbf{map}\ f\ (\textbf{map}\ g\ xs) \Rightarrow \textbf{map}\ (f \circ g)\ xs$$

Other Optimisations Performed by the Futhark Compiler

- Aggressive fusion:

$$\textbf{map}\ f\ (\textbf{map}\ g\ xs) \Rightarrow \textbf{map}\ (f \circ g)\ xs$$

- Pervasive *struct-of-arrays* representation, i.e. representing

$$[(1, 2), (3, 4), (5, 6)]$$

as

$$([1, 3, 5], [2, 4, 6])$$

Other Optimisations Performed by the Futhark Compiler

- Aggressive fusion:

$$\textbf{map}\ f\ (\textbf{map}\ g\ xs) \Rightarrow \textbf{map}\ (f \circ g)\ xs$$

- Pervasive *struct-of-arrays* representation, i.e. representing

$$[(1,2),(3,4),(5,6)]$$

  as

$$([1,3,5],[2,4,6])$$

- Automatically rearrange representation of arrays to ensure coalesced memory access, e.g. picking column- or row-major (or both!) as necessary.

Other Optimisations Performed by the Futhark Compiler

- Aggressive fusion:

$$\mathbf{map}\ f\ (\mathbf{map}\ g\ xs) \Rightarrow \mathbf{map}\ (f \circ g)\ xs$$

- Pervasive *struct-of-arrays* representation, i.e. representing

  $$[(1,2),(3,4),(5,6)]$$

  as

  $$([1,3,5],[2,4,6])$$

- Automatically rearrange representation of arrays to ensure coalesced memory access, e.g. picking column- or row-major (or both!) as necessary.
- Local memory block tiling when threads access same data.

Other Optimisations Performed by the Futhark Compiler

- Aggressive fusion:

$$\textbf{map}\ f\ (\textbf{map}\ g\ xs) \Rightarrow \textbf{map}\ (f \circ g)\ xs$$

- Pervasive *struct-of-arrays* representation, i.e. representing

$$[(1,2),(3,4),(5,6)]$$

  as

$$([1,3,5],[2,4,6])$$

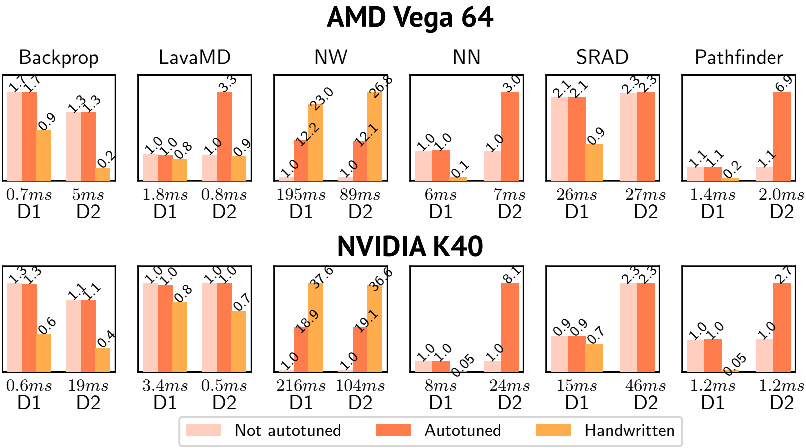
- Automatically rearrange representation of arrays to ensure coalesced memory access, e.g. picking column- or row-major (or both!) as necessary.
- Local memory block tiling when threads access same data.
- Standard compiler optimisations: inlining, CSE, constant folding, constant propagation, etc...

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.

Futhark versus hand-written OpenCL



- Higher is better.
- Handwritten OpenCL of widely varying quality.
- This makes them “realistic”, in a sense.

Conclusions

- Futhark is a data-parallel array language with an optimising compiler that generates CUDA and OpenCL.
- **Futhark will not outcompete hand-tuned primitives,** but application performance is often competitive.
- Everything is under a free software license.

Try out Futhark for yourself! `futhark-lang.org`

Conclusions

- Futhark is a data-parallel ar … n optimising compiler that generates CUD…
- **Futhark will not outcompete har … -tune … primitives,** but application performance is ofte … compe … tive.
- Everything is under a free s … re li … e.

Try out Futhark for yourself!

`futhark-lang.org`

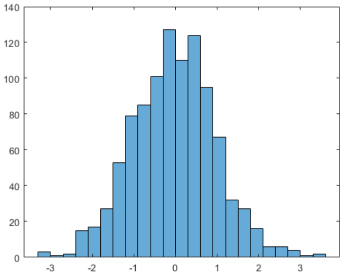
APPENDICES

Computing Histograms

We are given an integer constant *k* and an array
`is : [n]i32`

and we must produce an array
`hist : [k]i32`

where `hist[i]` is the number of occurences of `i` in `is`.



Imperative Implementation

```
int hist[k] = {0, ..., 0}
for (int i = 0; i < n; i++) {
  var j = is[i];
  hist[j]++;
}
```

- $O(k+n)$ work.
- (May have cache issues for large k, but we'll ignore that.)
- **Neither parallel nor functional.**

Data-parallel Implementation

```
let histogram [n] (k: i32) (is: [n]i32): [k]i32 =
  reduce (map2 (+))
         (replicate k 0)
         (map (\i -> replicate k 0 with [i] = 1)
              is)
```

- $O(k \cdot n)$ work—**Bad**.
- $O(\log(n))$ span—**Good**.

Alternative data-parallel Implementation

```
let histogram [n] (k: i32) (is: [n]i32): [k]i32 =
  map (\j -> reduce (+) 0
                    (map (\i -> if i == j
                                then 1
                                else 0)
                     is))
      (iota k)
```

- $O(k \cdot n)$ work—**Bad**.
- $O(\log(n))$ span—**Good**.

Theoretically efficient implementation

```
let histogram [n] (k: i32) (is: [n]i32) =
  let is' =
    radix_sort i32.num_bits i32.get_bit is
  let flags =
    map2 (!=) is' (rotate (-1) is')
  in segmented_reduce (+) 0
                      flags
                      (replicate n 1)
```

- $O(k+n)$ work—**Good**.
- $O(\log(n))$ span—**Good**.
- Assumes bins are non-empty (can be fixed).
- **That radix sort is really slow in practice.**

How can we do better?

Atomic operations in OpenCL

```
int atomic_add(volatile __global int *p,
               int val)

int atomic_cmpxchg(volatile __global int *p,
                   int cmp,
                   int val)
```

- CPUs and GPUs support certain atomic operations with hardware-level synchronisation.
- Can support very efficient histograms.
- Side-effecting, so cannot expose directly in a functional language.

Generalised Histograms

```
val reduce_by_index [k] [n] 't :
     [k]a
  -> (a -> a -> a) -> a
  -> [n]i32 -> [n]a
  -> [k]a
```

Semantically, an application

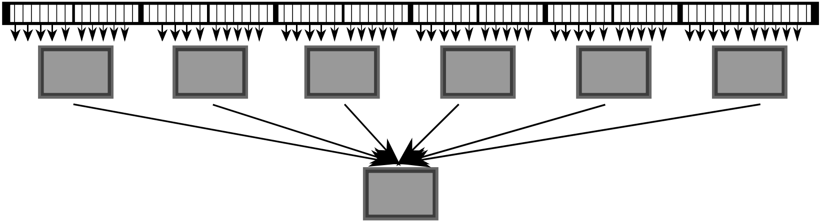
```
reduce_by_index hist f y is xs
```

returns the array `dest`, but modified according to the following imperative pseudocode:

```
for (int j = 0; j < n; j < n) {
  int i = is[j];
  if (i >= 0 && i < k) {
    hist[i] = f(hist[i], xs[j]);
  }
}
```

Generalised Histograms on the GPU

To avoid bin conflicts, threads are grouped, with each group producing a *subhistogram*, which is then combined to a single result.



- Atomics are used to compute the subhistograms, and a segmented reduction for the final result.
- Use specialised atomic if possible; fall back to spinlock with compare-and-exchange for complex operators.
- Subhistograms in local memory[3] if small enough.

[3] *Shared memory* in CUDA terms.

Histogram performance on Vega 64 GPU

$n = 10^6$