Futhark

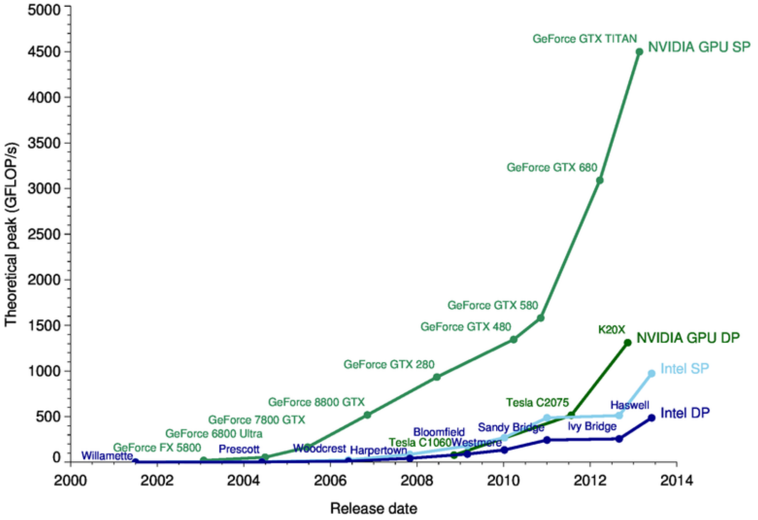
A High-Performance Purely Functional Array Language

Troels Henriksen (athas@sigkill.dk)
Niels G. W. Serup (ngws@metanohi.name)
Martin Elsman (mael@di.ku.dk)
Cosmin Oancea (cosmin.oancea@di.ku.dk)

DIKU
University of Copenhagen

February 9th 2017

Why GPUs?



NVIDIA GPU SP
NVIDIA GPU DP
Intel SP
Intel DP
GeForce GTX TITAN
GeForce GTX 680
GeForce GTX 580
GeForce GTX 480
K20X
GeForce GTX 280
GeForce 8800 GTX
Tesla C2075
Haswell
GeForce 7800 GTX
GeForce 6800 Ultra
Bloomfield
Sandy Bridge
Ivy Bridge
Tesla C1060
Westmere
GeForce FX 5800
Prescott
Woodcrest
Harpertown
Willamette
Theoretical peak (GFLOP/s)
Release date
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
2000
2002
2004
2006
2008
2010
2012
2014

Futhark at a Glance

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

- **Data-parallel loops**

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

- **Sequential loops**

```
fun main (n: i32): i32 =
  loop (x = 1) = for i < n do
    x * (i + 1)
  in x
```

- **Array Construction**

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

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*.
- Type-checker verifies that `x` is not used afterwards, via alias analysis.

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*.
- Type-checker verifies that `x` is not used afterwards, via alias analysis.

Shorthand

When $x \equiv y$, we write:

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

This is just syntactical sugar for variable shadowing.

Uniqueness Type Annotations

Uniqueness checking is entirely intra-procedural. A function can uniqueness-annotate its parameters and return type:

```
fun 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.

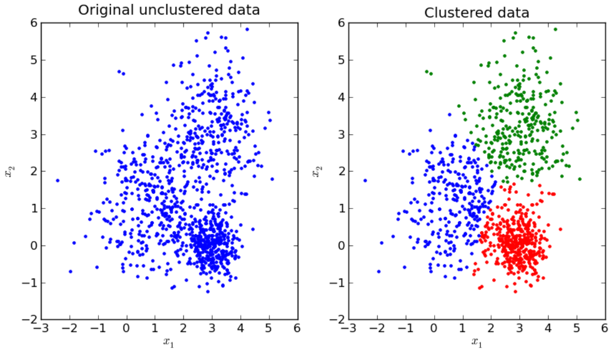
A call `let xs' = copy_one xs ys i` is valid if `xs` can be consumed. The result `xs'` does not alias anything at this point.

Case Study:
k-means Clustering

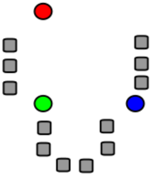
The Problem

We are given n points in some d-dimensional space, which we must partition into k disjoint sets, such that we minimise the inter-cluster sum of squares (the distance from every point in a cluster to the centre of the cluster).

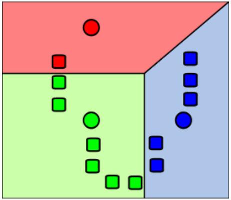
Example with $d = 2, k = 3, n = \textit{more than I can count}$:



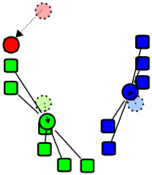
The Solution (from Wikipedia)



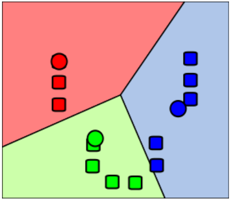
(1) k initial "means" (here $k = 3$) are randomly generated within the data domain.



(2) k clusters are created by associating every observation with the nearest mean.



(3) The centroid of each of the k clusters becomes the new mean.



(4) Steps (2) and (3) are repeated until convergence has been reached.

Computing Cluster Means: the Ugly

```
fun add_centroids(x: [d]f32) (y: [d]f32): [d]f32 =
  map (+) x y

fun cluster_means_seq (cluster_sizes: [k]i32)
                      (points: [n][d]f32)
                      (membership: [n]i32): [k][d]f32 =
  loop (acc = replicate k (replicate d 0.0)) =
    for i < n do
      let p = points[i]
      let c = membership[i]
      let p' = map (/f32(cluster_sizes[c])) p
      let acc[c] = add_centroids acc[c] p'
      in acc
  in acc
```

Computing Cluster Means: the Ugly

```
fun add_centroids(x: [d]f32) (y: [d]f32): [d]f32 =
  map (+) x y

fun cluster_means_seq (cluster_sizes: [k]i32)
                      (points: [n][d]f32)
                      (membership: [n]i32): [k][d]f32 =
  loop (acc = replicate k (replicate d 0.0)) =
    for i < n do
      let p = points[i]
      let c = membership[i]
      let p' = map (/f32(cluster_sizes[c])) p
      let acc[c] = add_centroids acc[c] p'
      in acc
  in acc
```

Problem

$O(n \times d)$ work, but no parallelism.

Computing Cluster Sizes: the Bad

Use a parallel map to compute "increments", and then a `reduce` of these increments.

```
fun cluster_means_par(cluster_sizes: [k]i32)
                     (points: [n][d]f32)
                     (membership: [n]i32): [k][d]f32 =
  let increments : [n][k][d]i32 =
    map (\p c ->
           let a = replicate k (replicate d 0.0)
           let a[c] = map (/(f32(cluster_sizes[c]))) p
           in a)
         points membership
  in reduce (\xss yss ->
               map (\xs ys -> map (+) xs ys) xss yss)
            (replicate k (replicate d 0.0))
            increments
```

Computing Cluster Sizes: the Bad

Use a parallel `map` to compute "increments", and then a `reduce` of these increments.

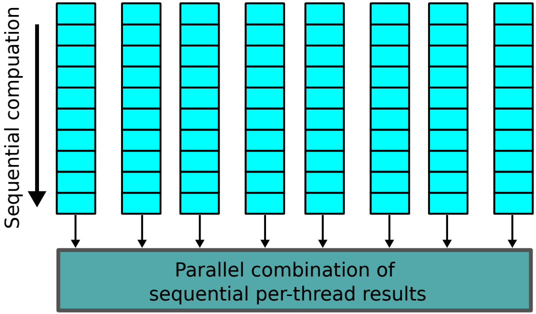
```
fun cluster_means_par(cluster_sizes: [k]i32)
                     (points: [n][d]f32)
                     (membership: [n]i32): [k][d]f32 =
  let increments : [n][k][d]i32 =
    map (\p c ->
           let a = replicate k (replicate d 0.0)
           let a[c] = map (/(f32(cluster_sizes[c]))) p
           in a)
         points membership
  in reduce (\xss yss ->
               map (\xs ys -> map (+) xs ys) xs ys)
            (replicate k (replicate d 0.0))
            increments
```

Problem

Fully parallel, but $O(k \times n \times d)$ work.

One Futhark Design Principle

The hardware is not infinitely parallel - ideally, we use an efficient sequential algorithm for chunks of the input, then use a parallel operation to combine the results of the sequential parts.



The optimal number of threads varies from case to case, so this should be abstracted from the programmer.

Validity of Chunking

Any fold with an associative operator $\odot$ can be rewritten as:

$$\texttt{fold}\ \odot\ xs = \texttt{fold}\ \odot\ (\texttt{map}\,(\texttt{fold}\,\odot)\,(\texttt{chunk}\,xs))$$

The trick is to provide a language construct where the user can provide a specialised implementation of the *inner* fold, which need not be parallel.

Computing cluster sizes: the Good

We use a Futhark language construct called a *reduction stream*.

```
fun cluster_means_stream (cluster_sizes: [k]i32)
                         (points: [n][d]f32)
                         (membership: [n]i32): [k][d]f32 =
  streamRed
  (\(acc: [k][d]f32) (elem: [k][d]f32) ->
     map add_centroids acc elem)
  (\(inp: [chunksize]([d]f32,i32)) ->
     let (points', membership') = unzip inp
     in cluster_means_seq cluster_sizes points' membership')
  (zip points membership)
```

For full parallelism, set chunk size to 1.

For full sequentialisation, set chunk size to n.

GPU Code Generation for `streamRed`

Broken up as:

```
let per_thread_results : [num_threads][k][d]f32 =
  oneChunkPerThread ... points membership
-- combine the per-thread results
let cluster_means =
  reduce (map (map (+))) (replicate k 0) per_thread_results
```

The reduction with `map (map (+))` is not great - the accumulator of a reduction should ideally be a scalar. The compiler will recognise this pattern and perform a transformation called *Interchange Reduce With Inner Map* (IRWIM); moving the reduction inwards at a cost of a transposition.

After IRWIM

We transform

```
let cluster_sizes =
  reduce (map (map(+))) (replicate k 0)
         per_thread_results
```

and get

```
let per_thread_results' : [k][d][num_threads]f32 =
  rearrange (1,2,0) per_thread_results
let cluster_sizes =
  map (map (reduce (+) 0)) per_thread_results'
```

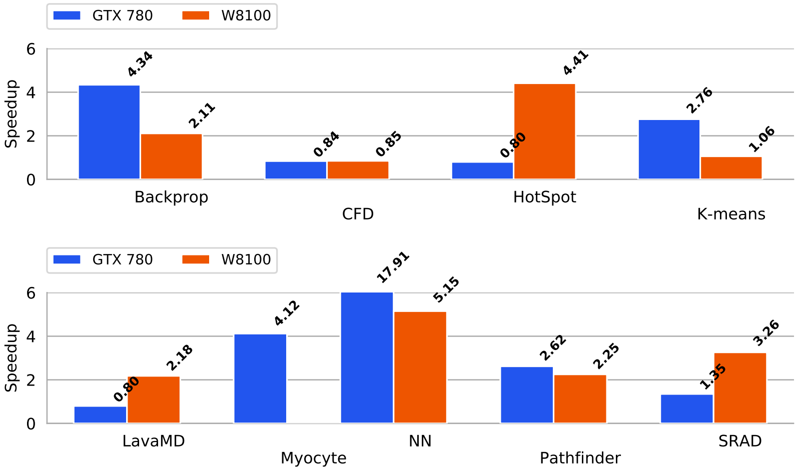
- `map` parallelism of size $\mathtt{k} \times \mathtt{d}$ - likely not enough.
- Futhark compiler generates a segmented reduction for `map (map (reduce (+) 0))`, which exploits also the innermost `reduce` parallelism.

Performance of cluster means computation

Seqential performance on Intel Xeon E6-2750 and GPU performance on NVIDIA Tesla K40. Speedup of `streamRed` over alternative. $k = 5$; $n = 10,000,000$; $d = 3$.

Platform	Version	Runtime	Speedup
GPU	Chunked (parallel)	17.6ms	$\times 7.6$
	Fully parallel	134.1ms	
CPU	Chunked (sequential)	98.3ms	$\times 0.92$
	Fully sequential	90.7ms	

Speedup Over Hand-Written Rodinia OpenCL Code on NVIDIA and AMD GPUs



Conclusions

- Futhark is a small high-level functional data-parallel language with a GPU-targeting optimising compiler.
- Chunking data-parallel operators permit a balance between efficient sequential code and all necessary parallelism.
- Performance is okay.

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

Code `https://github.com/HIPERFIT/futhark`

Benchmarks `https://github.com/HIPERFIT/futhark-benchmarks`