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?

[Diagram showing the theoretical peak (GFLOP/s) vs. release date for various GPU models from Willamette to NVIDIA GPU DP and Intel DP, highlighted with nodes for specific GPU models such as GeForce GTX 7800 GTX, Tesla C2075, Haswell, Ivy Bridge, and Sandy Bridge.]
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**
  
  ```futhark
  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**
  
  ```futhark
  fun main (n : i32): i32 =
    loop (x = 1) = for i < n do
      x * (i + 1)
    in x
  ```

- **Array Construction**

  ```futhark
  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.

```plaintext
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 ≡ y, we write:

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

This is just syntactical sugar for variable shadowing.
Uniqueness checking is entirely intra-procedural. A function can uniqueness-annotate its parameters and return type:

```haskell
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 = 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

```plaintext
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.

```ocaml
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.
Computing Cluster Sizes: the Bad

Use a parallel map to compute “increments”, and then a reduce of these increments.

```haskell
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:

$$\text{fold } \odot \text{ xs } = \text{fold } \odot \left( \text{map } (\text{fold } \odot) (\text{chunk xs}) \right)$$

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*.

```futhark
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.
Broken up as:

```haskell
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

\[
\text{let } \text{cluster\_sizes} = \\
\quad \text{reduce } (\text{map } (\text{map}(+))) (\text{replicate } k \ 0) \\
\quad \text{per\_thread\_results}
\]

and get

\[
\text{let } \text{per\_thread\_results}' : [k][d][\text{num\_threads}]\text{f32} = \\
\quad \text{rearrange } (1,2,0) \ \text{per\_thread\_results} \\
\text{let } \text{cluster\_sizes} = \\
\quad \text{map } (\text{map } (\text{reduce } (+) \ 0)) \ \text{per\_thread\_results}'
\]

- map parallelism of size $k \times d$ - likely not enough.
- Futhark compiler generates a segmented reduction for
  \[
  \text{map } (\text{map } (\text{reduce } (+) \ 0)),
  \]
  which exploits also the innermost reduce parallelism.
Performance of cluster means computation

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

<table>
<thead>
<tr>
<th>Platform</th>
<th>Version</th>
<th>Runtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Chunked (parallel)</td>
<td>17.6ms</td>
<td>$\times7.6$</td>
</tr>
<tr>
<td></td>
<td>Fully parallel</td>
<td>134.1ms</td>
<td></td>
</tr>
<tr>
<td>GPU</td>
<td>Chunked (sequential)</td>
<td>98.3ms</td>
<td>$\times0.92$</td>
</tr>
<tr>
<td></td>
<td>Fully sequential</td>
<td>90.7ms</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td></td>
<td>90.7ms</td>
<td></td>
</tr>
</tbody>
</table>
## Speedup Over Hand-Written Rodinia OpenCL Code on NVIDIA and AMD GPUs

<table>
<thead>
<tr>
<th>Application</th>
<th>GTX 780</th>
<th>W8100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backprop</td>
<td>4.34</td>
<td>2.11</td>
</tr>
<tr>
<td>CFD</td>
<td>0.84</td>
<td>0.85</td>
</tr>
<tr>
<td>HotSpot</td>
<td>0.80</td>
<td>4.41</td>
</tr>
<tr>
<td>K-means</td>
<td>2.76</td>
<td>1.06</td>
</tr>
<tr>
<td>LavaMD</td>
<td>0.80</td>
<td>2.18</td>
</tr>
<tr>
<td>Myocyte</td>
<td>4.12</td>
<td>5.15</td>
</tr>
<tr>
<td>NN</td>
<td>17.91</td>
<td>2.62</td>
</tr>
<tr>
<td>Pathfinder</td>
<td>2.25</td>
<td>1.35</td>
</tr>
<tr>
<td>SRAD</td>
<td>3.26</td>
<td></td>
</tr>
</tbody>
</table>
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