Futhark:
Purely Functional GPU-programming with Nested Parallelism and in-place Array Updates

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, Cosmin Oancea

Presented by:
-Zaid Qureshi

Motivation

- GPUs are traditionally programmed using sequential programming languages
    - Requires expertise to exploit the parallelism provided by GPUs
- Functional programming languages provide parallelizable primitives (ie. map, reduce, scan)
    - But when compiled naively, their performance is very bad

Futhark

- Purely Functional Array programming language for GPUs
    - To ease GPU programming
- Expresses computation/parallelism using basic and streaming second-order array combinators (SOACs)
- Type system that allows expression of race-free in-place updates
- Compiler implements partial flattening to allow for more parallelism without destroying memory access patterns

Futhark Syntax

e	$::=$	$k \mid v$	(Constant/Variable)
	$\mid$	$(v_1, \ldots, v_n)$	(Tuple)
	$\mid$	$v_1 \odot v_2$	(Apply binary operator)
	$\mid$	$\textbf{if } v_1 \textbf{ then } e_2 \textbf{ else } e_3$	
	$\mid$	$v\,[v_1, \ldots, v_n]$	(Array indexing)
	$\mid$	$v\ v_1\ \ldots\ v_n$	(Function call)
	$\mid$	$\textbf{let } (p_1, \ldots, p_n) = e_1 \textbf{ in } e_2$	(Let binding)
	$\mid$	$v \textbf{ with } [v_1, \ldots, v_n] \leftarrow v$	(In-place update)
	$\mid$	$\textbf{loop } (pv) \textbf{ for } v < v \textbf{ do } e$	(Loop)
	$\mid$	$\textbf{iota } v$	$([0, \ldots, v-1])$
	$\mid$	$\textbf{replicate } n\ v$	($[v, \ldots, v]$ of size n)
	$\mid$	$\textbf{rearrange } (\bar{k})\ v$	(Rearrange dimensions)
	$\mid$	$\textbf{map } l\ v_1\ \ldots\ v_n$	
	$\mid$	$\textbf{reduce } l\ (v_1, \ldots, v_n)\ v'_1\ \ldots\ v'_n$	
	$\mid$	$\textbf{scan } l\ (v_1, \ldots, v_n)\ v'_1\ \ldots\ v'_n$	
	$\mid$	$\textbf{stream\_seq } l\ (v_1, \ldots, v_n)\ v'_1\ \ldots\ v'_m$	
	$\mid$	$\textbf{stream\_seq } l\ v'_1\ \ldots\ v'_m$	
	$\mid$	$\textbf{stream\_red } l_1\ l_2\ (v_1, \ldots, v_n)\ v'_1\ \ldots\ v'_m$	

Basic SOACs

$$\text{map } f \; [a_1, \ldots, a_n] \; = \; [f \; a_1, \ldots, f \; a_n]$$

$$\text{reduce } \; \oplus \; 0_{\oplus} \; [a_1, \ldots, a_n] \; = \; 0_{\oplus} \oplus a_1 \oplus \ldots \oplus a_n$$

$$\text{scan } \; \oplus \; 0_{\oplus} \; [a_1, \ldots, a_n] \; = \; [a_1, \ldots, a_1 \oplus \ldots \oplus a_n]$$

Example Futhark Code

```
fun main (matrix : [n][m]f32): ([n][m]f32, [n]f32) =
   map (λrow : ([m]f32, f32) →
            let row' = map (λx : f32 → x+1.0) row
            let s = reduce (+) 0 row
            in (row',s))
        matrix
```

Example Futhark Code

INPUT: nxm matrix

```
fun main (matrix : [n][m]f32): ([n][m]f32, [n]f32) =
    map (λrow : ([m]f32, f32) →
            let row' = map (λx : f32 → x+1.0) row
            let s = reduce (+) 0 row
            in (row',s))
        matrix
```

Example Futhark Code

INPUT: nxm matrix

OUTPUT: tuple of nxm matrix, array of size n

```
fun main (matrix : [n][m]f32): ([n][m]f32, [n]f32) =
    map (λrow : ([m]f32, f32) →
            let row' = map (λx : f32 → x+1.0) row
            let s = reduce (+) 0 row
            in (row',s))
        matrix
```

Example Futhark Code

Map over the rows of the matrix

```
fun mai ][m]f32): ([n][m]f32, [n]f32) =
    map (λrow : ([m]f32, f32) →
            let row' = map (λx : f32 → x+1.0) row
            let s = reduce (+) 0 row
            in (row',s))
        matrix
```

Example Futhark Code

```
fun main (matr                ): ([n][m]f32, [n]f32) =
    map (λrow               ) →
            let row' = map (λx : f32 → x+1.0) row
            let s = reduce (+) 0 row
            in (row',s))
        matrix
```

Generate new row

Example Futhark Code

```
fun main (matrix : [n][m]f32): ([n][m]f32, [n]f32) =
  map (λrow          f32) →
        let          ap (λx : f32 → x+1.0) row
        let s = reduce (+) 0 row
        in (row',s))
    matrix
```

Get sum of row

Example Futhark Code

```
fun main (matrix : [n][m]f32): ([n][m]f32, [n]f32) =
    map (λrow : ([m]f32, f32) →
            le                p (λx : f32 → x+1.0) row
            le                 (+) 0 row
            in (row',s))
        matrix
```

Return tuple of new row and sum

Example Futhark Code

```
fun main (matrix : [n][m]f32): ([n][m]f32, [n]f32) =
   map (λrow : ([m]f32, f32) →
          let row' = map (λx : f32 → x+1.0) row
          let s = reduce (+) 0 row
          in (row',s))
      matrix
```

Parallel operator *sFold* and Streaming Operators

$$\text{sFold}\,(\oplus)\; f\; (v_1 \# \ldots \# v_k) = (f\ \epsilon) \oplus (f\ v_1) \oplus \ldots \oplus (f\ v_k)$$

- concat

ϵ - empty partition

Parallel operator *sFold* and Streaming Operators

$$\text{sFold}\,(\oplus)\,f\,(v_1\#\ldots\#v_k) = (f\ \epsilon)\oplus(f\ v_1)\oplus\ldots\oplus(f\ v_k)$$

$$\textbf{stream\_map}\ f \quad = \quad \text{sFold}\,(\#)\,f$$

\# - concat
ϵ - empty partition

Applies *f* to each partition and then concatenates the resulting partitions

Parallel operator *sFold* and Streaming Operators

$$\text{sFold}\,(\oplus)\ f\ (v_1 \# \ldots \# v_k) = (f\ \epsilon) \oplus (f\ v_1) \oplus \ldots \oplus (f\ v_k)$$

- concat
ϵ - empty partition

$$\textbf{stream\_map}\ f \quad = \quad \text{sFold}\,(\#)\ f$$

Applies *f* to each partition and then concatenates the resulting partitions

$$\textbf{stream\_red}\,(\oplus)\ f \quad = \quad \text{sFold}\,((\oplus) * (\#))\ f$$

Extends **stream_map** by allowing each chunk to produce an additional output which is reduced in parallel

Sequential Histogramming in Futhark

```
let counts = loop (counts = replicate k 0) for i < n do
    let cluster = membership[i]
    in counts with [cluster] ← counts[cluster] + 1
```

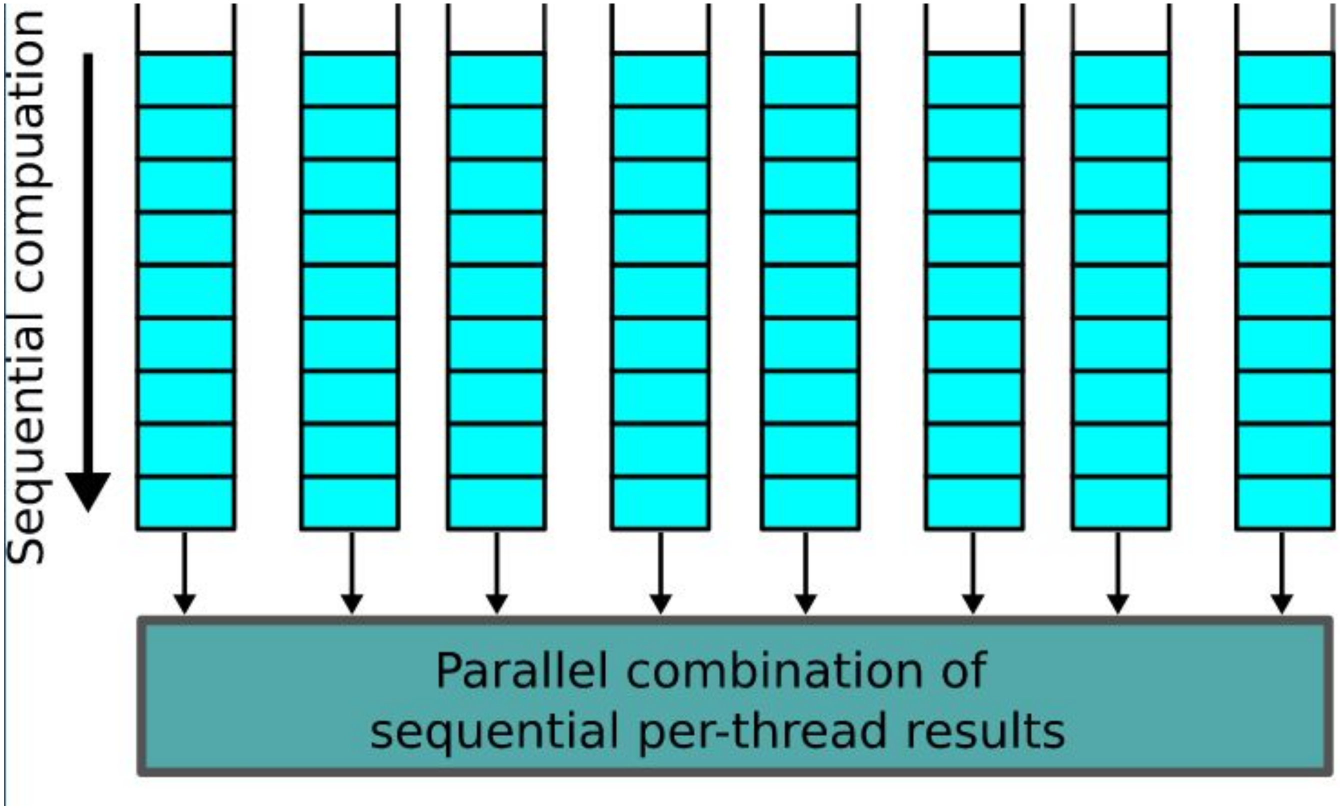
Parallel Histogramming in Futhark

```
let increments =
    map(λ(cluster: int): [k]int →
          let incr = replicate k 0 in let incr[cluster] = 1 in incr)
        membership
let counts =
    reduce (λ(x: [k]int) (y: [k]int): [k]int → map (+) x y)
          (replicate k 0) increments
```

Efficient Parallel Histogramming in Futhark

```
let counts = stream_red (map(+))
    (λ(acc: *[k]int) (chunk: [chunksize]int): [k]int →
        let res = loop (acc) for i < chunksize do
                        let cluster = chunk[i]
                        in acc with [cluster] ← acc[cluster]+1
        in res)
  (replicate k 0) membership
```

Efficient Parallel Histogramming in Futhark



In-Place Updates and Uniqueness Types

```
fun modify (n: int) (a: *[n]int) (i: int) (x: [n]int): *[n]int =
  a with [i] ← (a[i] + x[i])
```

- In purely functional languages array updates require copying array and updating copy (to avoid side effects)
- If it is known that the original array won’t be used after the update, the update can occur in place

In-Place Updates and Uniqueness Types

$$\textbf{fun}\ \text{modify}\ (n\text{: }\textbf{int})\ (a\text{: }*[n]\textbf{int})\ (i\text{: }\textbf{int})\ (x\text{: }[n]\textbf{int})\text{: }*[n]\textbf{int} =$$
$$a\ \textbf{with}\ [i] \leftarrow (a[i] + x[i])$$

- Futhark has *uniqueness types* that allow programmer to specify function arguments that won't be referenced by the caller after the function call
  - The callee gains ownership of that argument
- An array is consumed when it is source of in-place update or when it is passed as a unique parameter.
- After the consumption point, the array or its aliases may not be used.
  - Type system checks this via aliasing rules

Aliasing Rules

- Alias sets for values produced by SOACs are empty (new copies)
- Scalar read from an array does not alias its origin array (alias set not modified)
- Array slicing aliases origin array
- Function application:
  - If the result being returned is unique the alias set is empty
  - Otherwise the result aliases all non-unique parameters
- Other rules can be found in Figure 5 of the paper

In-Place Update Checking

- Each expression *e* has a observed set of variables (*O*) and a consumed set of variables (*C*)
  - the pair <*C*,*O*> forms the occurrence trace for *e*
- Inference rules used to check uniqueness and parameter consumption (Figure 6)

Sequence Judgement

$$\boxed{\langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_2, \mathcal{O}_2 \rangle : \langle \mathcal{C}_3, \mathcal{O}_3 \rangle}$$

Inference Rule

$$\frac{(\mathcal{O}_2 \cup \mathcal{C}_2) \cap \mathcal{C}_1 = \emptyset}{\langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_2, \mathcal{O}_2 \rangle : \langle \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{O}_1 \cup \mathcal{O}_2 \rangle} \quad (\text{OCCURENCE-SEQ})$$

If-then-else uniqueness inference rule

$$\frac{\begin{array}{c} v_1 \;\triangleright \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \quad e_2 \;\triangleright \langle \mathcal{C}_2, \mathcal{O}_2 \rangle \quad e_3 \;\triangleright \langle \mathcal{C}_3, \mathcal{O}_3 \rangle \\ \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_2, \mathcal{O}_2 \rangle : \langle \mathcal{C}'_2, \mathcal{O}'_2 \rangle \\ \langle \mathcal{C}_1, \mathcal{O}_1 \rangle \gg \langle \mathcal{C}_3, \mathcal{O}_3 \rangle : \langle \mathcal{C}'_3, \mathcal{O}'_3 \rangle \end{array}}{\textbf{if } v_1 \textbf{ then } e_2 \textbf{ else } e_3 \;\triangleright \langle \mathcal{C}'_2 \cup \mathcal{C}'_3, \mathcal{O}'_2 \cup \mathcal{O}'_3 \rangle} \quad (\text{SAFE-IF})$$

In-Place Update Checking (Example)

$$\textbf{let } \mathsf{bs} = \textbf{map } (\lambda(\mathsf{a}) \rightarrow \mathsf{a}\ \textbf{with } [0] \leftarrow 2)\ \mathsf{as}$$

$$\textbf{let } \mathsf{d} = \textbf{iota } \mathsf{m}$$

This program passes as the function of the map consumes its parameter as

$$\textbf{let } \mathsf{cs} = \textbf{map } (\lambda(\mathsf{i}) \rightarrow \mathsf{d}\ \textbf{with } [\mathsf{i}] \leftarrow 2)\ (\textbf{iota } \mathsf{n})$$

This program doesn't pass as it implies d, bound outside the function of the map, is consumed for every iteration of the map

Streaming SOAC Fusion

$$\mathcal{F}1:\ \textbf{map}\ \text{f}\ \overline{b} \implies \textbf{stream\_map}\ (\lambda(\overline{b_c}) \to \textbf{map}\ \text{f}\ \overline{b_c})\ \overline{b}$$

$$\mathcal{F}2:\ \textbf{map}\ \text{f}\ \overline{b} \implies \textbf{stream\_seq}\ (\lambda(\text{a},\overline{b_c}) \to (0,\ \textbf{map}\ \text{f}\ \overline{b_c}))\ (0)\ \overline{b}$$

$$\begin{aligned}
&\mathcal{F}3:\ \textbf{reduce}\ \oplus\ \overline{e}\ \overline{b} \implies \\
&\qquad \textbf{stream_red}\ \oplus\ (\lambda(\overline{a},\overline{b_c}) \to \overline{a} \oplus \textbf{reduce}\ \oplus\ \overline{e}\ \overline{b_c})\ (\overline{e})\ \overline{b}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{F}4:\ \textbf{reduce}\ \oplus\ \overline{e}\ \overline{b} \implies \\
&\qquad \textbf{stream_seq}\ (\lambda(\overline{a},\overline{b_c}) \to (\overline{a}) \oplus \textbf{reduce}\ \oplus\ \overline{e}\ \overline{b_c})\ (\overline{e})\ \overline{b}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{F}5:\ \textbf{scan}\ \oplus\ \overline{e}\ \overline{b} \implies \textbf{stream_seq}(\lambda(\overline{a},\overline{b_c}) \to \\
&\qquad\qquad \textbf{let}\ \overline{x_c} = \textbf{scan}\ \oplus\ \overline{e}\ \overline{b_c} \\
&\qquad\qquad \textbf{let}\ \overline{y_c} = \textbf{map}\ (\overline{a}\ \oplus)\ \overline{x_c} \\
&\qquad\qquad \textbf{in}\ (\text{last}(\overline{y_c}),\ \overline{y_c}))\ (\overline{e})\ \overline{b}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{F}6: && \textbf{let}\ (\overline{r_1},\ \overline{r_2},\ \overline{x},\ \overline{y},\ \overline{z}) = \textbf{stream_red} \\
& && \quad (\lambda(\overline{c_1},\ \overline{d_1},\ \overline{c_2},\ \overline{d_2}) \to \\
& && \qquad (\overline{c_1} \oplus \overline{c_2},\quad \overline{d_1} \odot \overline{d_2})) \\
&\textbf{let}\ (\overline{r_1},\overline{x},\overline{y}) = && \quad (\lambda(\overline{e_1},\ \overline{e_2},\ \overline{a_c},\ \overline{b_c}) \to \\
&\quad \textbf{stream_red}\ \oplus\ \text{f}\ (\overline{e_1})\ \overline{a} \implies && \qquad \textbf{let}\ (\overline{r_1},\overline{x_c},\overline{y_c}) = \text{f}\ \overline{e_1}\ \overline{a_c} \\
&\textbf{let}\ (\overline{r_2},\overline{z}) = && \qquad \textbf{let}\ (\overline{r_2},\overline{z_c}) = \text{g}\ \overline{e_2}\ \overline{x_c}\ \overline{b_c} \\
&\quad \textbf{stream_red}\ \odot\ \text{g}\ (\overline{e_2})\ \overline{x}\ \overline{b} && \qquad \textbf{in}\ (\overline{r_1},\ \overline{r_2},\ \overline{x_c},\ \overline{y_c},\ \overline{z_c})) \\
& && \quad (\overline{e_1},\ \overline{e_2})\ \overline{a}\ \overline{b}
\end{aligned}$$

$$\begin{aligned}
& && \textbf{let}\ (\overline{r_1},\ \overline{r_2},\ \overline{x},\ \overline{y},\ \overline{z}) = \textbf{stream_seq} \\
&\mathcal{F}7: && \quad (\lambda(\overline{a_1},\ \overline{a_2},\ \overline{b_c},\ \overline{d_c}) \to \\
&\textbf{let}\ (\overline{r_1},\overline{x},\overline{y}) = && \qquad \textbf{let}\ (\overline{r_1},\overline{x_c},\overline{y_c}) = \text{f}\ \overline{a_1}\ \overline{b_c} \\
&\quad \textbf{stream_seq}\ \text{f}\ (\overline{e_1})\ \overline{b} \implies && \qquad \textbf{let}\ (\overline{r_2},\overline{z_c}) = \text{g}\ \overline{a_2}\ \overline{x_c}\ \overline{d_c} \\
&\textbf{let}\ (\overline{r_2},\overline{z}) = && \qquad \textbf{in}\ (\overline{r_1},\ \overline{r_2},\ \overline{x_c},\ \overline{y_c},\ \overline{z_c})) \\
&\quad \textbf{stream_seq}\ \text{g}\ (\overline{e_2})\ \overline{x}\ \overline{d} && \quad (\overline{e_1},\ \overline{e_2})\ \overline{b}\ \overline{d}
\end{aligned}$$

Streaming SOAC Fusion Example

```
fun main(n: int): int =
  let Y = stream_map (λ(iss: [m]int): [m]int →
                          let a = f^ind (iss[0])
                          let t = map (g a) iss
                          let y = scan ⊙ 0 t
                          in y)
                       (iota n)
  let b = reduce (+) 0 Y
  in b
```

(a) Program before fusion.

```
fun main(n: int): (int,int) =
  stream_red (+) (λ(e1: int) (iss: [m]int): int →
                      let a = f^ind(iss[0])
                      let t = map (g a) iss
                      let y = scan (⊙) 0 t
                      let b = reduce (+) e1 y
                      in b)
                 (0) (iota n)
```

(b) Program after fusion at outer level.

Moderate Flattening

- Flattening algorithm based on map-loop interchange and map distribution
- Attempt to exploit some top-level parallelism
  - Not seeking parallelism inside branches
  - Terminating map distribution when it would introduce irregular arrays

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

Moderate Flattening

- Flattening algorithm based on map-loop interchange and map distribution
- Attempt to exploit partial top-level parallelism
  - Not seeking parallelism inside branches
  - Terminating map distribution when it would introduce irregular arrays

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

```
let bss: [m][m]i32 =
  map (\(ps: [m]i32) (ps: [m]i32)
->
      loop (ws=ps) for i < n do
        map (\w -> w * 2) ws)
    pss
```

```
let bss: [m][m]i32 =
  loop (wss=pss) for i < n do
    map (\ws ->
          map (\w -> w * 2) ws)
        wss
```

Locality of Reference Optimizations

- Naive translation of Flattened and Fused code can lead to bad memory access patterns
- Futhark compiler can optimize memory access patterns by transforming data

Transpose:

```
map (λxs → reduce (+) 0 xs) xss.
```

```
let xss' = as_column_major xss
in map (λxs → reduce (+) 0 xs) xss'
```

Tiling:

```
map (λp → stream_seq (λa (ps': [q]int) →
                          g a ps') ps) ps
```

```
map (λp → stream_seq (λa (ps': [q]int) →
                          let ps'' = local ps'
                          in g a ps'') (0) ps) ps
```

Evaluation Methodology

- Tested with 2 GPUs
  - Nvidia GX 780
  - AMD W8100
- Generated OpenCL code is run on both GPUs
- Baseline implementations taken from benchmark suites

Benchmark	Dataset	Suite
Backprop	Input layer size equal to 2^{20}	Rodinia
CFD	`fvcorr.domn.193K`	Rodinia
HotSpot	1024×1024; 360 iterations	Rodinia
K-means	`kdd_cup`	Rodinia
LavaMD	`boxes1d=10`	Rodinia
Myocyte	`workload=65536`, `xmax=3`	Rodinia
NN	Default Rodinia dataset duplicated 20 times	Rodinia
Pathfinder	Array of size 10^5	Rodinia
SRAD	502×458; 100 iterations	Rodinia
LocVolCalib	large dataset	FinPar
OptionPricing	large dataset	FinPar
MRI-Q	large dataset	Parboil
Crystal	Size 2000, degree 50	Accelerate
Fluid	3000×3000; 20 iterations	Accelerate
Mandelbrot	4000×4000; 255 limit	Accelerate
N-body	$N = 10^5$	Accelerate

Table 2: Benchmark dataset configurations.

Results

	NVIDIA GTX780		AMD W8100	
Benchmark	**Ref.**	**Futhark**	**Ref.**	**Futhark**
Backprop	46.9	20.7	41.5	12.9
CFD	1878.2	2235.9	3610.0	4177.5
HotSpot	35.9	45.3	260.4	72.6
K-means	1597.7	572.2	1216.1	1534.9
LavaMD	5.1	6.7	9.0	7.1
Myocyte	2733.6	555.4	—	2979.8
NN	178.9	11.0	193.2	37.6
Pathfinder	18.4	7.4	18.2	6.5
SRAD	19.9	16.1	195.1	34.8
LocVolCalib	1211.1	1293.2	3117.0	5015.8
OptionPricing	136.0	106.8	429.5	360.8
MRI-Q	20.2	15.5	17.9	14.3
Crystal	41.0	8.4	—	8.4
Fluid	268.7	100.4	—	221.8
Mandelbrot	30.8	8.1	—	14.8
N-body	613.2	89.5	—	269.8

Table 1: Average benchmark runtimes in milliseconds.

Results

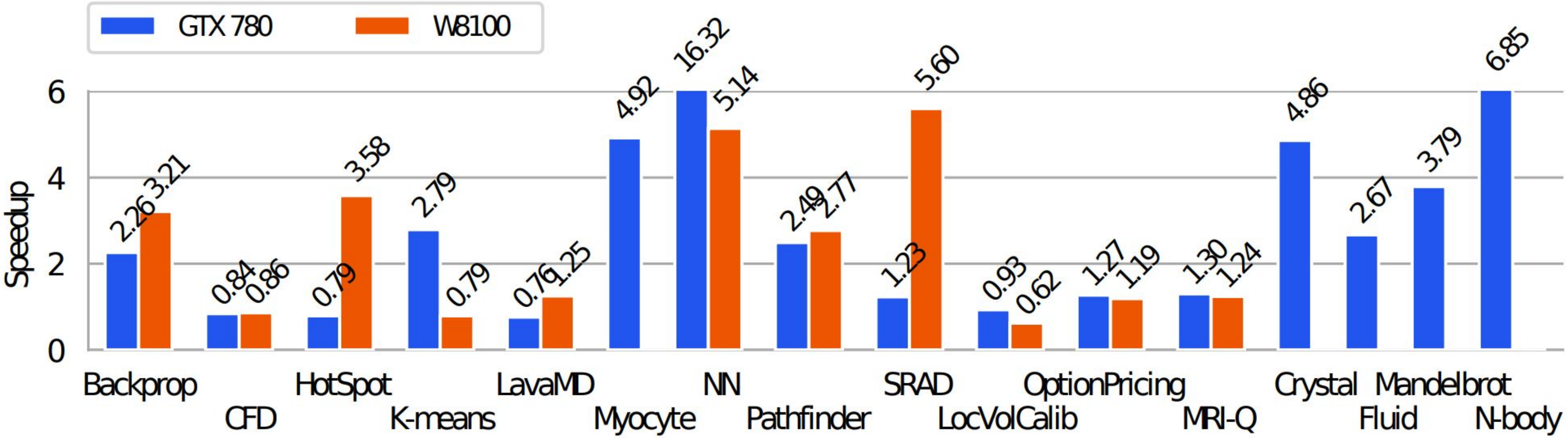


Figure 13: Relative speedup compared to reference implementations (some bars are truncated for space reasons).

Results

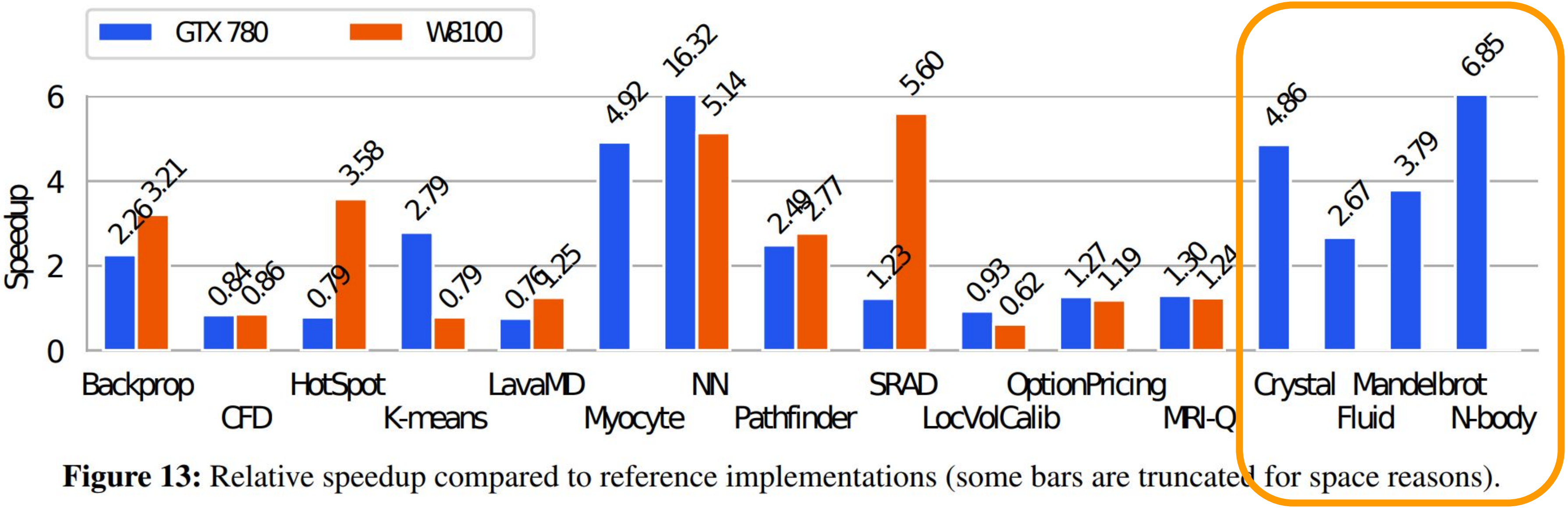


Figure 13: Relative speedup compared to reference implementations (some bars are truncated for space reasons).

Futhark performs better than other functional programming environments for GPU due to higher level optimizations

Results

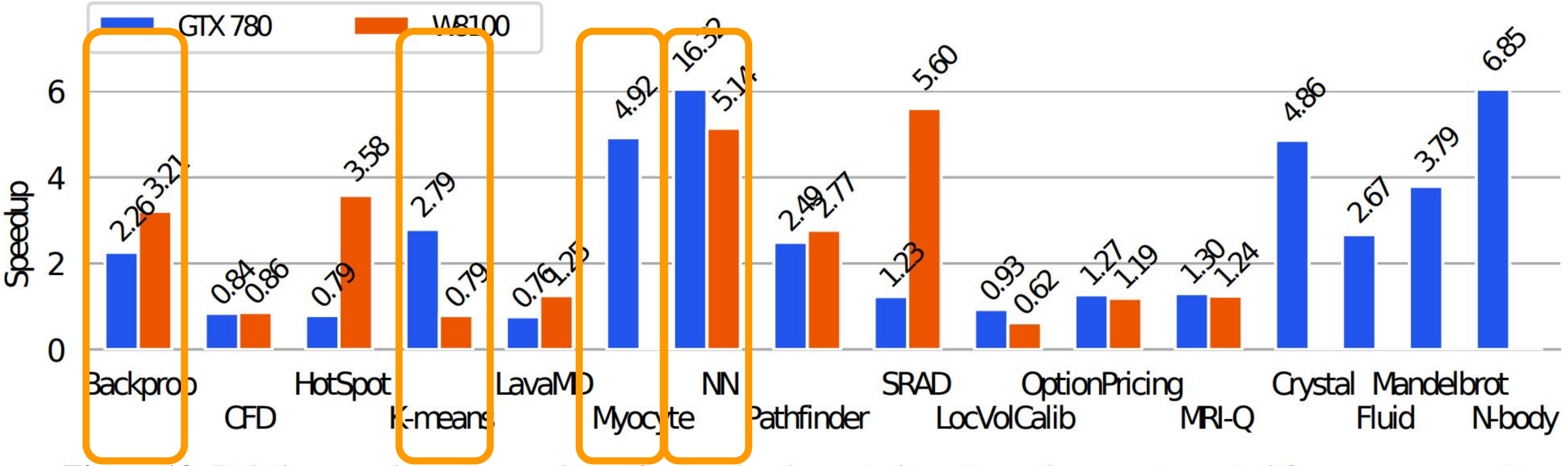


Figure 13: Relative speedup compared to reference implementations (some bars are truncated for space reasons).

Rodinia doesn't implement all optimizations: sequential reductions (Backprop, NN), not parallelizing computation of new cluster centers (k-means), not coalescing all accesses (Myocyte)

Results

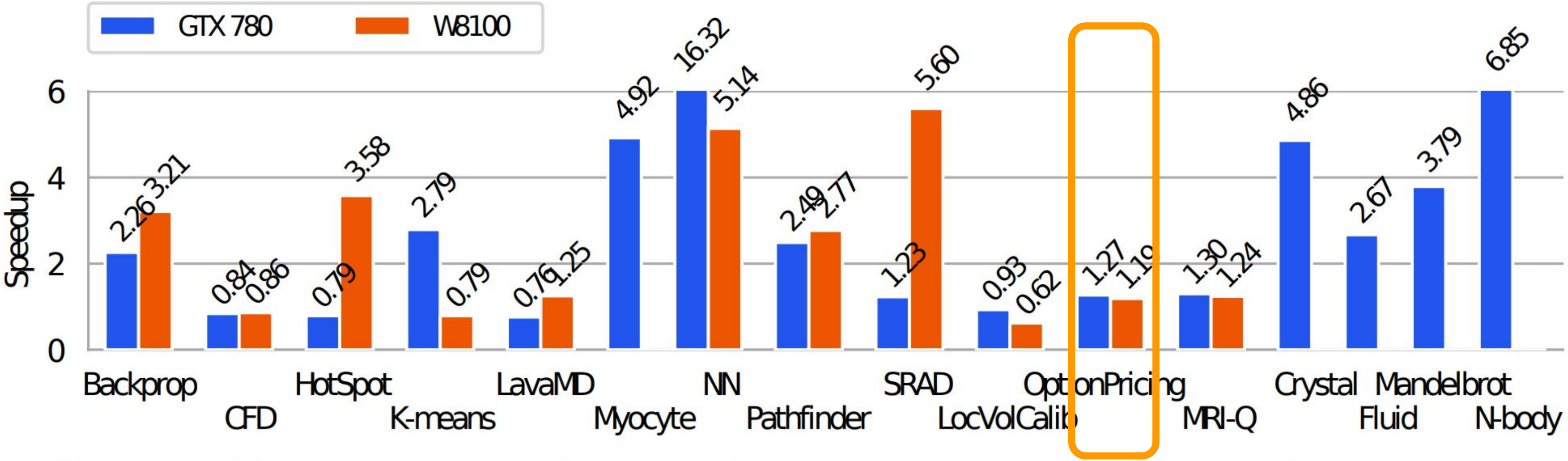


Figure 13: Relative speedup compared to reference implementations (some bars are truncated for space reasons).

For OptionPricing, Futhark sequentializes excessive parallelism.

Results

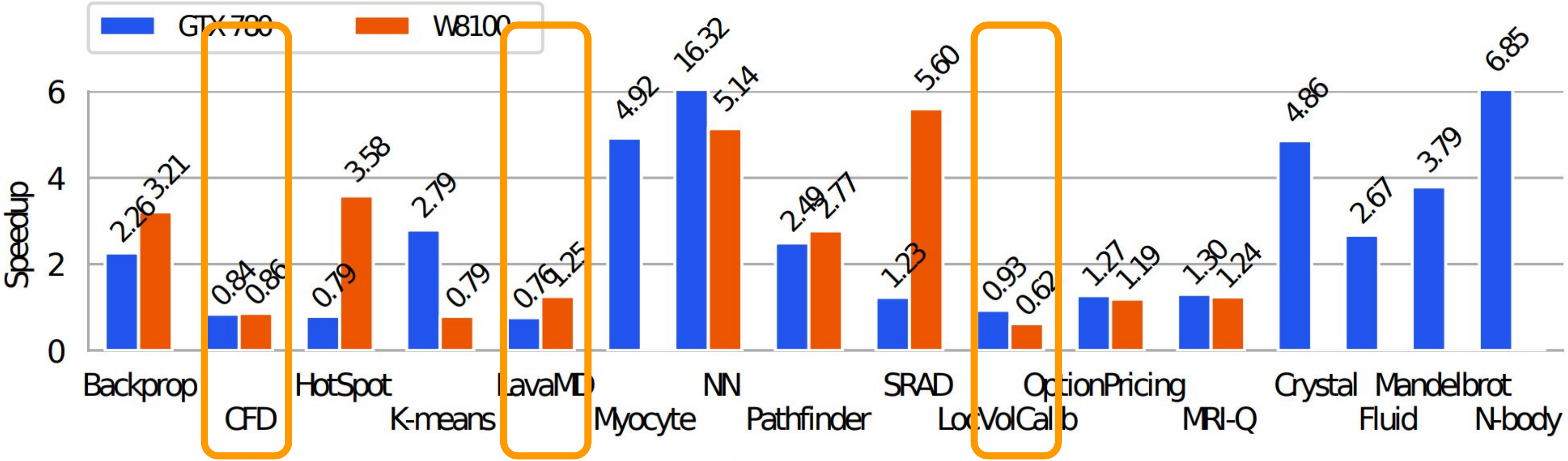


Figure 13: Relative speedup compared to reference implementations (some bars are truncated for space reasons).

Furthark gets around 70-80% of the performance of hand-tuned code.

Impact of Optimizations

- SOAC Fusion
  - K-means (x1.42), LavaMD (x4.55), Myocyte (x1.66), SRAD (x1.21), Crystal (x10.1), LocVolCalid (x9.4)
  - Without fusion OptionPricing, N-body, and MRI-Q have too high memory requirements
- In-place Updates
  - K-means (x8.3), LocVolCalib (x1.7)
  - OptionPricing can't even be implemented without in-place updates
- Coalescing
  - K-means (x9.26), Myocyte (x4.2), OptionPricing (x8.79), LocVolCalib (x8.4)
- Tiling
  - LavaMD (x1.35), MRI-Q (x1.33), N-body (x2.29)

Conclusion

Pros:

- Futhark code is independent of the underlying hardware
- Futhark's type system allows expression of race-free in-place updates
- Optimizations done by compiler using higher level functions/reasoning
- Compiler implements partial flattening to allow for more parallelism without destroying memory access patterns
- Compiler can aggressively fuse and decompose code to best use available parallelism

Cons:

- Requires rewrite of applications
- Although it does optimizations like flattening and fusion, Futhark's compiler can't optimize all the time
    - Ie. it can't convert inefficient histogramming to the efficient one
    - Still leaves a huge design space for the programmer to explore to write good performant code

Thank you!

Other slides

Futhark Syntax

e	$::=$	$k \mid v$	(Constant/Variable)
	$\mid$	$(v_1, \ldots, v_n)$	(Tuple)
	$\mid$	$v_1 \odot v_2$	(Apply binary operator)
	$\mid$	**if** v_1 **then** e_2 **else** e_3	
	$\mid$	$v[v_1, \ldots, v_n]$	(Array indexing)
	$\mid$	$v\ v_1\ \ldots\ v_n$	(Function call)
	$\mid$	**let** $(p_1, \ldots, p_n) = e_1$ **in** e_2	(Let binding)
	$\mid$	v **with** $[v_1, \ldots, v_n] \leftarrow v$	(In-place update)
	$\mid$	**loop** (pv) **for** $v < v$ **do** e	(Loop)
	$\mid$	**iota** v	($[0,\ldots,v-1]$)
	$\mid$	**replicate** $n\ v$	($[v, \ldots, v]$ of size n)
	$\mid$	**rearrange** $(\bar{k})\ v$	(Rearrange dimensions)
	$\mid$	**map** $l\ v_1\ \ldots\ v_n$	
	$\mid$	**reduce** $l\ (v_1, \ldots, v_n)\ v_1'\ \ldots\ v_n'$	
	$\mid$	**scan** $l\ (v_1, \ldots, v_n)\ v_1'\ \ldots\ v_n'$	
	$\mid$	**stream_seq** $l\ (v_1, \ldots, v_n)\ v_1'\ \ldots\ v_m'$	
	$\mid$	**stream_seq** $l\ v_1'\ \ldots\ v_m'$	
	$\mid$	**stream_red** $l_1\ l_2\ (v_1, \ldots, v_n)\ v_1'\ \ldots\ v_m'$	

Results

	NVIDIA GTX780		AMD W8100	
Benchmark	**Ref.**	**Futhark**	**Ref.**	**Futhark**
Backprop	46.9	20.7	41.5	12.9
CFD	1878.2	2235.9	3610.0	4177.5
HotSpot	35.9	45.3	260.4	72.6
K-means	1597.7	572.2	1216.1	1534.9
LavaMD	5.1	6.7	9.0	7.1
Myocyte	2733.6	555.4	—	2979.8
NN	178.9	11.0	193.2	37.6
Pathfinder	18.4	7.4	18.2	6.5
SRAD	19.9	16.1	195.1	34.8
LocVolCalib	1211.1	1293.2	3117.0	5015.8
OptionPricing	136.0	106.8	429.5	360.8
MRI-Q	20.2	15.5	17.9	14.3
Crystal	41.0	8.4	—	8.4
Fluid	268.7	100.4	—	221.8
Mandelbrot	30.8	8.1	—	14.8
N-body	613.2	89.5	—	269.8

Table 1: Average benchmark runtimes in milliseconds.

Basic SOACs

$$\text{map}\ f\ [a_1, \ldots, a_n]\ =\ [f\ a_1, \ldots, f\ a_n]$$

$$\text{reduce}\ \oplus\ 0_{\oplus}\ [a_1, \ldots, a_n]\ =\ 0_{\oplus} \oplus a_1 \oplus \ldots \oplus a_n$$

$$\text{scan}\ \oplus\ 0_{\oplus}\ [a_1, \ldots, a_n]\ =\ [a_1, \ldots, a_1 \oplus \ldots \oplus a_n]$$

Can be implemented with Parallel Operator *fold*

$$\text{fold}\,(\oplus, 0_{\oplus})\,g\,[b_1, \ldots, b_n] = 0_{\oplus} \oplus (g\,b_1) \oplus \ldots \oplus (g\,b_n)$$

$$\mathcal{F}1 : \textbf{map}\ \mathsf{f}\ \overline{b} \implies \textbf{stream\_map}\ (\lambda(\overline{b_c}) \to \textbf{map}\ \mathsf{f}\ \overline{b_c})\ \overline{b}$$

$$\mathcal{F}2 : \textbf{map}\ \mathsf{f}\ \overline{b} \implies \textbf{stream\_seq}\ (\lambda(\mathsf{a},\overline{b_c}) \to (0,\ \textbf{map}\ \mathsf{f}\ \overline{b_c}))\ (0)\ \overline{b}$$

$$\begin{aligned}\mathcal{F}3 : {} & \textbf{reduce} \oplus \overline{e}\ \overline{b} \implies \\ & \quad \textbf{stream\_red} \oplus (\lambda(\overline{a},\overline{b_c}) \to \overline{a} \oplus \textbf{reduce} \oplus \overline{e}\ \overline{b_c})\ (\overline{e})\ \overline{b}\end{aligned}$$

$$\begin{aligned}\mathcal{F}4 : {} & \textbf{reduce} \oplus \overline{e}\ \overline{b} \implies \\ & \quad \textbf{stream\_seq}\ (\lambda(\overline{a},\overline{b_c}) \to (\overline{a}) \oplus \textbf{reduce} \oplus \overline{e}\ \overline{b_c})\ (\overline{e})\ \overline{b}\end{aligned}$$

$$\begin{aligned}\mathcal{F}5 : \textbf{scan} \oplus \overline{e}\ \overline{b} \implies {} & \textbf{stream\_seq}(\lambda(\overline{a},\overline{b_c}) \to \\ & \quad \textbf{let}\ \overline{x_c} = \textbf{scan} \oplus \overline{e}\ \overline{b_c} \\ & \quad \textbf{let}\ \overline{y_c} = \textbf{map}\ (\overline{a} \oplus)\ \overline{x_c} \\ & \quad \textbf{in}\ (\overline{\mathsf{last}(y_c)},\ \overline{y_c}))\ (\overline{e})\ \overline{b}\end{aligned}$$

$$\mathcal{F}6 :$$

$$\begin{array}{lcl} \textbf{let}\ (\overline{r_1}, \overline{x}, \overline{y}) = & & \textbf{let}\ (\overline{r_1},\ \overline{r_2},\ \overline{x},\ \overline{y},\ \overline{z}) = \textbf{stream\_red} \\ \quad \textbf{stream\_red} \oplus \mathsf{f}\ (\overline{e_1})\ \overline{a} & & \quad (\lambda(\overline{c_1},\ \overline{d_1},\ \overline{c_2},\ \overline{d_2}) \to \\ \textbf{let}\ (\overline{r_2}, \overline{z}) = & \implies & \qquad (\overline{c_1} \oplus \overline{c_2},\quad \overline{d_1} \odot \overline{d_2})) \\ \quad \textbf{stream\_red} \odot \mathsf{g}\ (\overline{e_2})\ \overline{x}\ \overline{b} & & \quad (\lambda(\overline{e_1},\ \overline{e_2},\ \overline{a_c},\ \overline{b_c}) \to \\ & & \qquad \textbf{let}\ (\overline{r_1}, \overline{x_c}, \overline{y_c}) = \mathsf{f}\ \overline{e_1}\ \overline{a_c} \\ & & \qquad \textbf{let}\ (\overline{r_2}, \overline{z_c}) = \mathsf{g}\ \overline{e_2}\ \overline{x_c}\ \overline{b_c} \\ & & \qquad \textbf{in}\ (\overline{r_1},\ \overline{r_2},\ \overline{x_c},\ \overline{y_c},\ \overline{z_c})) \\ & & \quad (\overline{e_1},\ \overline{e_2})\ \overline{a}\ \overline{b} \end{array}$$

$$\mathcal{F}7 :$$

$$\begin{array}{lcl} \textbf{let}\ (\overline{r_1}, \overline{x}, \overline{y}) = & & \textbf{let}\ (\overline{r_1},\ \overline{r_2},\ \overline{x},\ \overline{y},\ \overline{z}) = \textbf{stream\_seq} \\ \quad \textbf{stream\_seq}\ \mathsf{f}\ (\overline{e_1})\ \overline{b} & \implies & \quad (\lambda(\overline{a_1},\ \overline{a_2},\ \overline{b_c},\ \overline{d_c}) \to \\ \textbf{let}\ (\overline{r_2}, \overline{z}) = & & \qquad \textbf{let}\ (\overline{r_1}, \overline{x_c}, \overline{y_c}) = \mathsf{f}\ \overline{a_1}\ \overline{b_c} \\ \quad \textbf{stream\_seq}\ \mathsf{g}\ (\overline{e_2})\ \overline{x}\ \overline{d} & & \qquad \textbf{let}\ (\overline{r_2}, \overline{z_c}) = \mathsf{g}\ \overline{a_2}\ \overline{x_c}\ \overline{d_c} \\ & & \qquad \textbf{in}\ (\overline{r_1},\ \overline{r_2},\ \overline{x_c},\ \overline{y_c},\ \overline{z_c})) \\ & & \quad (\overline{e_1},\ \overline{e_2})\ \overline{b}\ \overline{d} \end{array}$$

Futhark Compiler Architecture

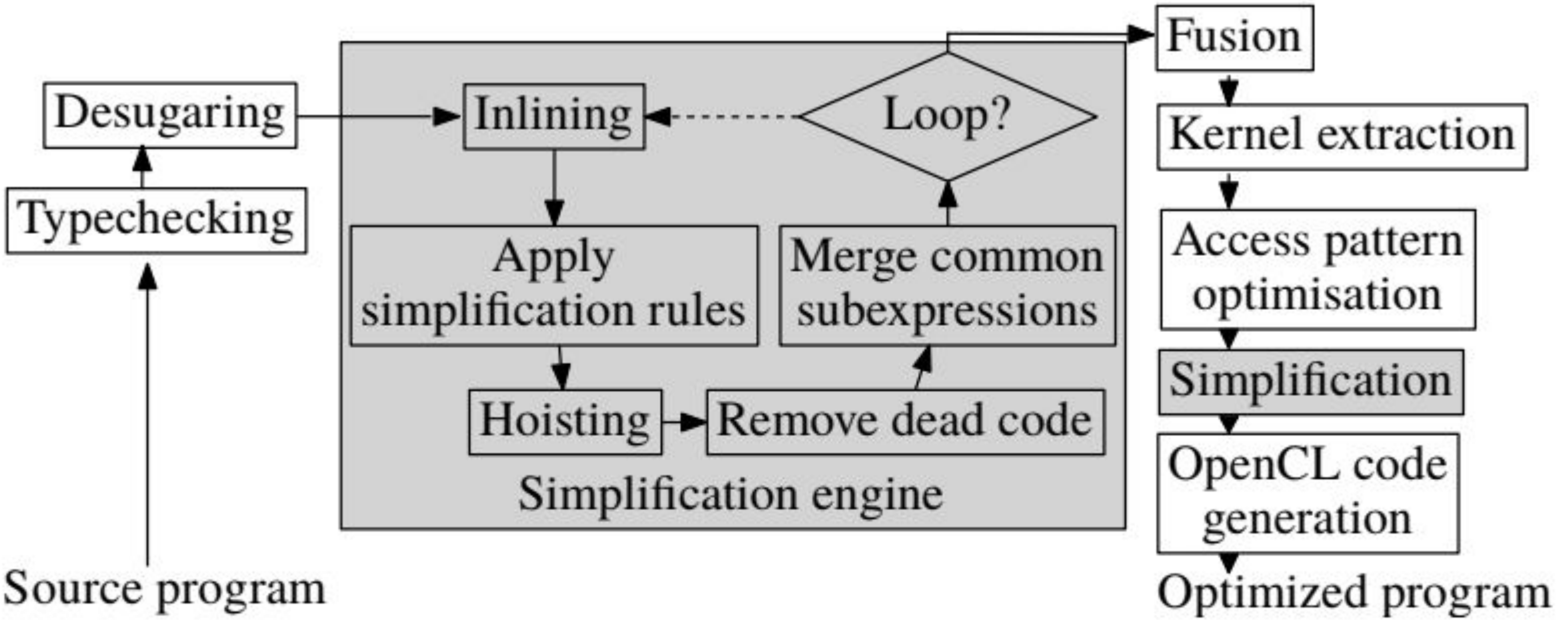


Figure 3: Compiler architecture.