Implementing Map-Scan Fusion in the Futhark Compiler

Brian Spiegelhauer & William Sprent

DIKU
University of Copenhagen

June 21, 2016

Table of Contents

Introduction

- Performance increases on hardware increasingly comes from parallisation.
- GPUs present an opportunity for massive parallelism.
- GPGPU libraries can be low level and cumbersome.
- Increasing need for high performance parallel languages.

Futhark

Futhark is:

- Functional language with GPU execution.
- Uses Second Order Array Combinators for bulk array operations.
- Aggressive optimisation strategy.

SOACs

- `map`, `reduce`, `scan`, `filter` etc.
- Equivalent to higher-order functions found in functional programming.
- Can have strong invariants. Useful with regards to parallelisation, and optimisation.

$$\texttt{map}\ f\ a\ :\ (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\equiv\ [f(a_0), f(a_1), ..., f(a_{n-1})]$$

$$\texttt{scan}\ \odot\ e\ a\ :\ (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]$$
$$\equiv [e \odot a_0, e \odot a_0 \odot a_1, ..., e \odot a_0 \odot ... \odot a_{n-1}]$$

Table of Contents

Loop Fusion

Why loop fusion?

- Combining loops reduces overhead.
- Can optimize memory access patterns.
- Can do away with intermediate arrays.

On GPU – high penalties for global memory accesses.

Example

Listing 1: Producer-Consumer pre-fusion.

```
1  a[n];
2  b[n];
3  for (int i = 0; i < n; i++) {
4    b[i] = f(a[i]);
5  }
6  c[n];
7  for (int i = 0; i < n; i++) {
8    c[i] = g(b[i]);
9  }
```

Example

Listing 2: Producer-Consumer post-fusion.

```
1   a[n];
2   c[n];
3
4   for (int i = 0; i < n; i++) {
5     c[i] = f(g(a[i]))
6   }
```

Fusing SOACs

Why?

- SOACs express easily as loops.
- Compatible SOACs can be fused using simple function composition.
- No difficult loop-dependency analysis required!

How?

- Analyse SOAC inter-compatibility for fusion.
- Express generalised rules for fusing combinations of SOACS.

Previous Example Revisited

$$\begin{array}{l}
\textbf{let}\ \ b = \mathtt{map}\ f\ a\ \ \textbf{in} \\
\textbf{let}\ \ c = \mathtt{map}\ g\ b\ \ \textbf{in} \\
\qquad\quad \Downarrow \\
\textbf{let}\ \ c = \mathtt{map}\ (g \circ f)\ a\ \ \textbf{in}
\end{array}$$

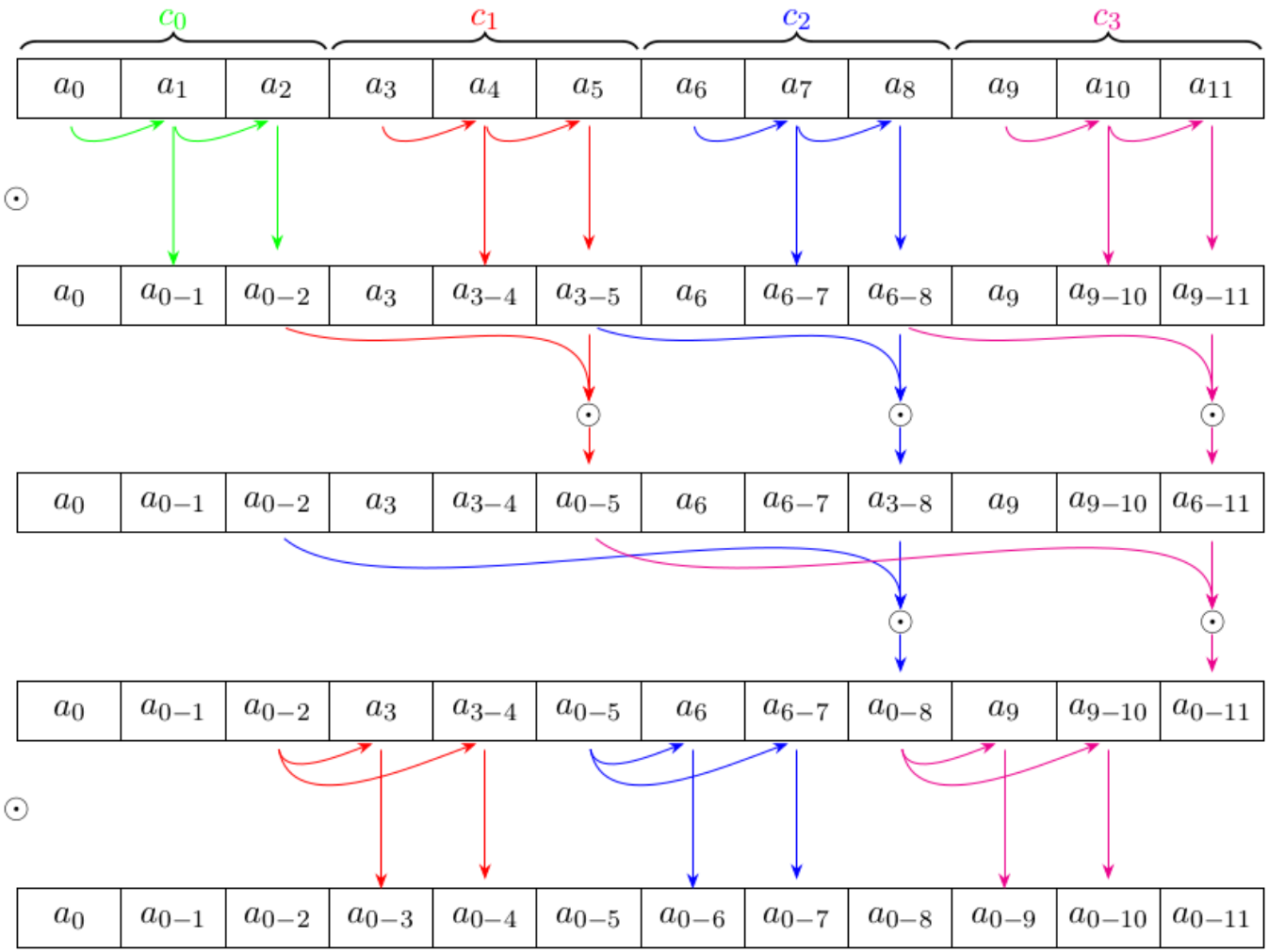
Map-Scan fusion.

Naive approach:

$$
\begin{aligned}
b &= \texttt{map}\ f\ a \\
c &= \texttt{scan}\ \odot\ e\ b \\
&\Downarrow \\
c &= \texttt{scan}\ \odot_f\ e\ a
\end{aligned}
$$

Problem: Scan requires an associative function.

Computing Scan



Scanomap

Solution:

- Look at how chunks are computed sequentially.
- Extend Scan with a sequential folding function.

$$\texttt{scanomap}\ \odot\ \odot_f\ e\ a$$
$$: (\alpha \to \alpha \to \alpha) \to (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to [\alpha]$$
$$\equiv [e \odot_f a_0, (e \odot_f a_0) \odot_f a_1, \ldots, ((e \odot_f a_0) \odot_f \ldots) \odot_f a_{n-1}]$$

Map-Scan Fusion

Using Scanomap to perform Map-Scan fusion:

$$
\begin{aligned}
b &= \texttt{map}\ f\ a \\
c &= \texttt{scan}\ \odot\ e\ b \\
&\Downarrow \\
c &= \texttt{scanomap}\ \odot\ \odot\ e\ b \\
&\Downarrow \\
c &= \texttt{scanomap}\ \odot\ \odot_f\ e\ a.
\end{aligned}
$$

Table of Contents

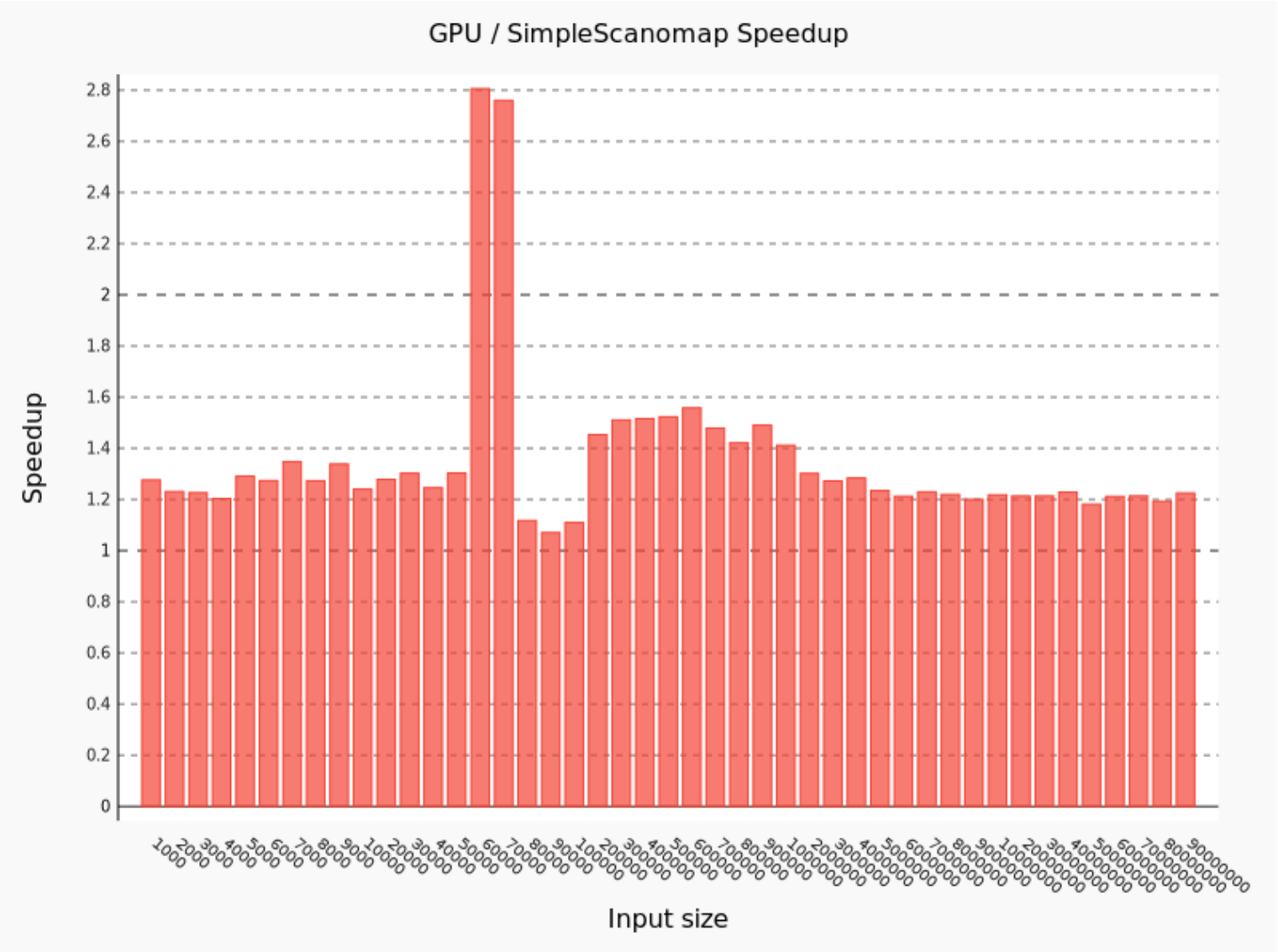
What have we done?

- Extended Futhark's internal representation with Scanomap.
- Added support for Scanomap in the type-checker, interpreter, SOAC module etc.
- Added rules for converting Scanomap into sequential loops.
- Extended the fusion module with fusion logic for Scanomap fusions.

Benchmark: Simple Map-Scan

```
1 fun ([int]) main([int] inp) =
2   let a = map(+10, inp)
3   let b = scan(+, 0, a) in
4   (b)
```

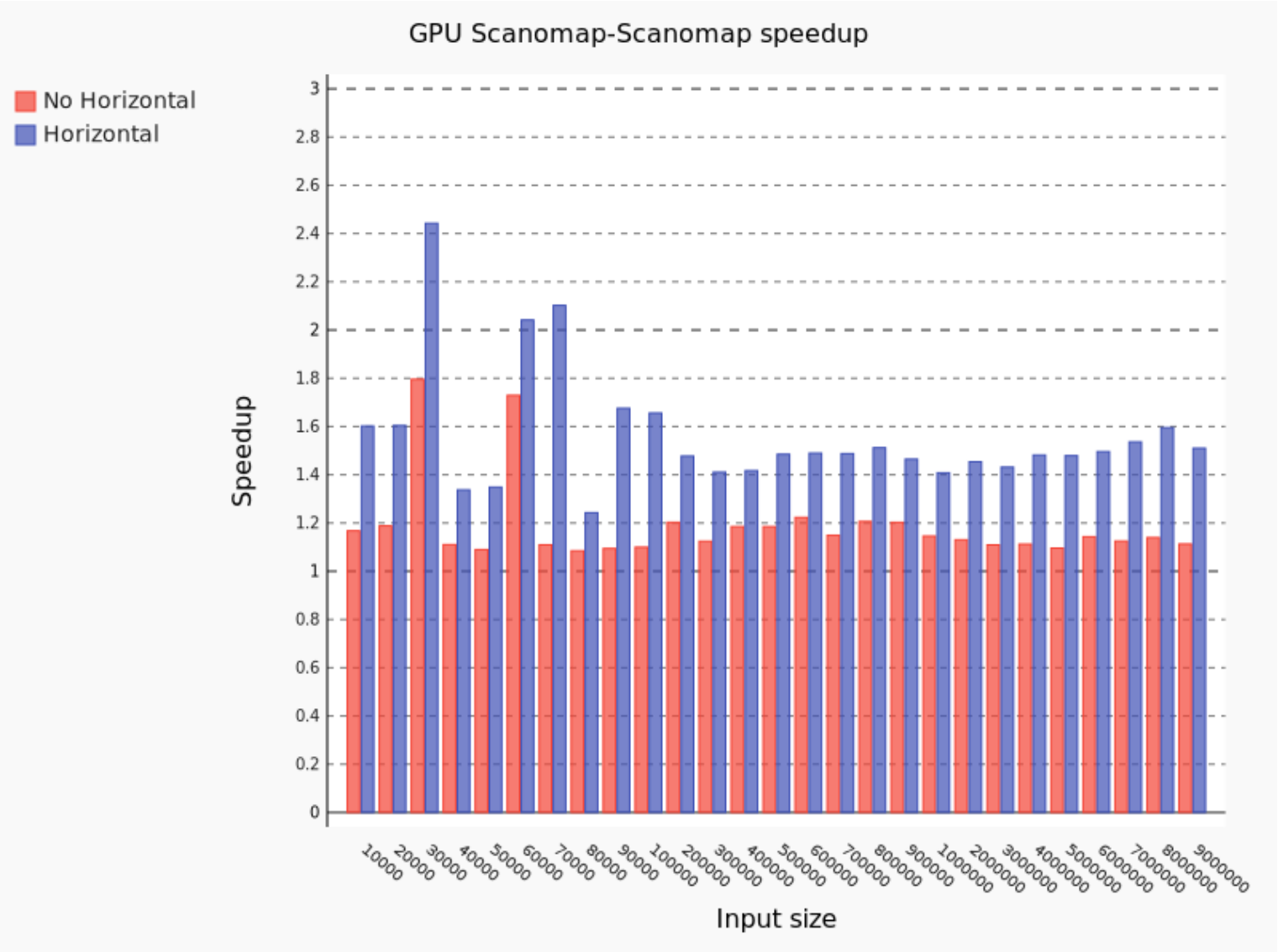
Benchmark: Simple Map-Scan



Benchmark: Horizontal Fusion

```
1 fun [int, n] main([int, n] inp) =
2     let a  = map(+10, inp) in
3     let b1 = scan(+, 0, a) in
4     let a2 = map(+1, a) in
5     let b2 = scan(+, 0, a2) in
6     map(fn int (int x,int y) => x + y, zip(b1,
          b2))
```

Benchmark: Horizontal Fusion



Benchmark: Radix Sort

```
1  fun [u32, n] radix_sort_step([u32, n] xs, i32
       digit_n) =
2    let bits = map(fn i32 (u32 x) => i32((x >>
         u32(digit_n)) & 1u32), xs)
3    let bits_inv = map(fn i32 (i32 b) => 1 - b,
         bits)
4    let ps0 = scan(+, 0, bits_inv)
5    let ps0_clean = map(*, zip(bits_inv, ps0))
6    let ps1 = scan(+, 0, bits)
7    let ps0_offset = reduce(+, 0, bits_inv)
8    let ps1_clean = map(+ ps0_offset, ps1)
9    let ps1_clean' = map(*, zip(bits, ps1_clean)
       )
10   let ps = map(+, zip(ps0_clean, ps1_clean'))
11   let ps_actual = map(fn i32 (i32 p) => p - 1,
       ps)
12   in write(ps_actual, xs, copy(xs))
```

Benchmark: Radix Sort

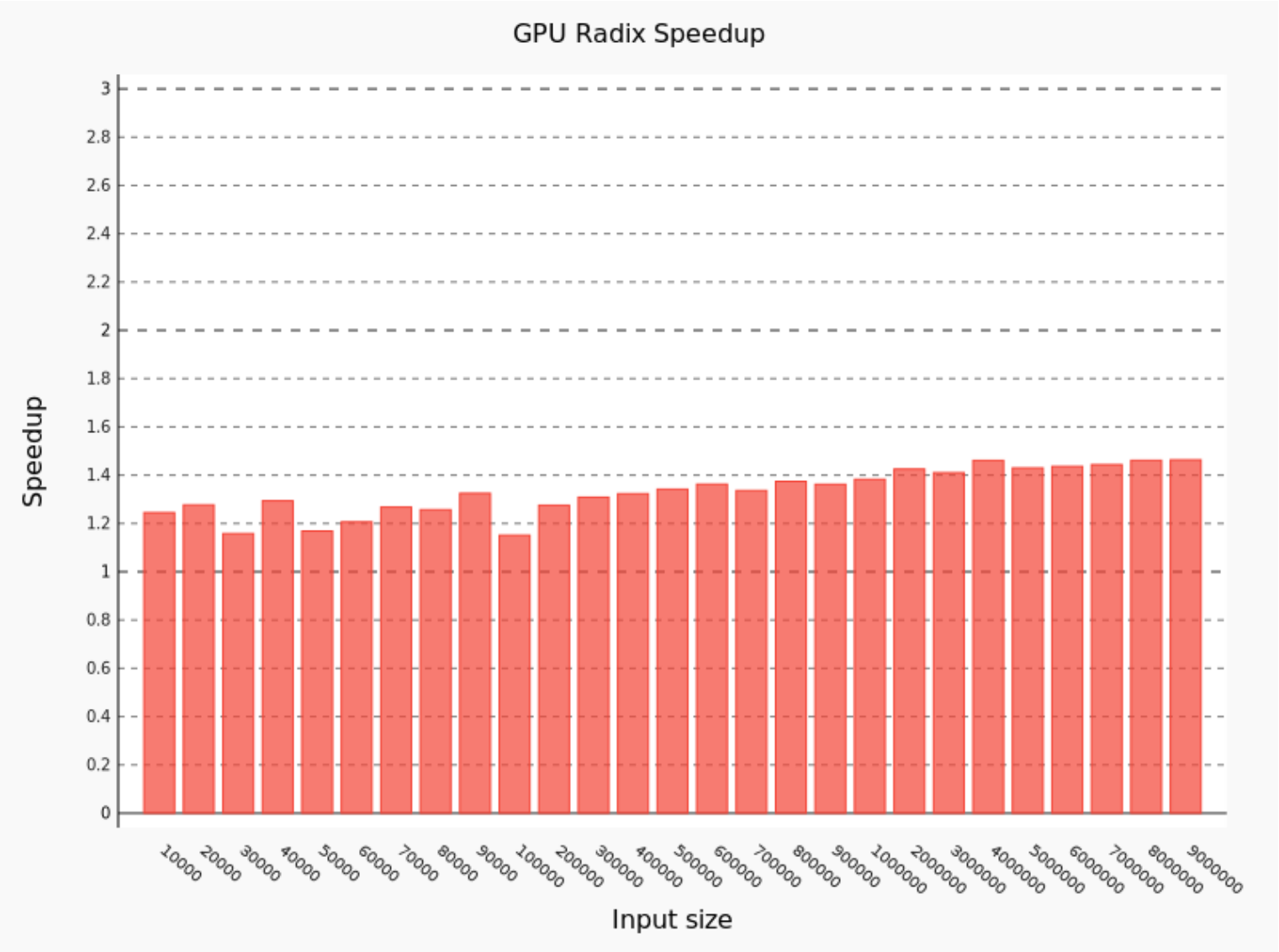


Table of Contents

Future work

Scanomap-Map fusion

- Scanomap doesn't natively fuse as a producer.
- Extend Scanomap with a third function.
- Fuse with Map using the techniques from earlier.

Should result in similar speedup.

Conclusion

What have we achieved?

- Performed Map-Scan fusion using Scanomap.
- Implemented further fusion for Scanomap.
- Uncovered opportunities for further optimisations.
- Demonstrated significant speedup.

End of Presentation.