Design and GPGPU Performance of Futhark’s Redomap Construct

Troels Henriksen, Ken Friis Larsen, Cosmin E. Oancea
HIPERFIT, Department of Computer Science, University of Copenhagen (DIKU), Denmark
athas@sigkill.dk, kflarsen@di.ku.dk, cosmin.oancea@di.ku.dk

Abstract
This paper presents and evaluates a novel second-order operator, named redomap, that stems from map-reduce compositions in the context of the purely-functional array language Futhark, which is aimed at efficient GPGPU execution. Main contributions are: First, we demonstrate an aggressive fusion technique that is centered on the redomap operator. Second, we present a compilation technique for redomap that efficiently sequentializes the excess parallelism and ensures coalesced access to global memory, even for non-commutative reduce operators. Third, a detailed performance evaluation shows that Futhark’s automatically generated code matches or exceeds performance of hand-tuned Thrust code.

Our evaluation infrastructure is publicly available and we encourage replication and verification of our results.

Categories and Subject Descriptors D.1.3 [Concurrent Programming]: Parallel Programming; D.3.4 [Processors]: Compiler

Keywords GPGPU, map-reduce, autoparallelization, functional language

1. Introduction
Commodity many-core hardware is now mainstream, driven in particular by the evolution of general purpose graphics programming units (GPGPUs) that support thousands of cores, but commodity programming is still falling short of efficiently utilization this hardware: Principal reasons are that low-level programming APIs, such as CUDA and OpenCL, are difficult to use and the translation of a sequential program is quite tedious and requires specialized users.

For example, the implementation of parallel reduce and scan operators is not only algorithmically challenging, but also requires efficient use of the fast memory and barrier synchronizations. Libraries that provide generic implementations of common bulk-parallel operators, such as Thrust [15], enhance programmability, but at cost in performance, for example because:

1 A “naive” translation might not take advantage of the way in which, for example, map and reduce operators may be efficiently composed, and
2 Even when the library supports a wealth of such combined operators and the expert user take full advantage of those, performance still remains sub-optimal due to generality-related constraints, e.g., the reduce operator requires an array-of-tuples representation, which results in less-coalesced accesses to global memory than a tuple-of-arrays representation would.

This context opens the door for data-parallel array languages [3, 15, 20] to emerge as the mainstream environment for programming massively-parallel hardware. In essence:

1 the non-specialist user is encouraged to “naively” write the application in its clearest form, in terms of a small and easy to understand set of simple operators, such as map, reduce, which
2 are optimized automatically by compiler analysis that scales to program-level to reach efficiencies that outperform highly-tuned code that relies on generic parallel libraries.

This paper analyses in detail the simple but common, hence important, case of map-reduce composition in the context of Futhark [17]: a purely-functional array language that supports nested parallelism on regular multidimensional arrays. We remark that Futhark adopts a non-restricted semantics for the map and reduce operators, for example (i) the mapped function may receive an array argument and return an array, and (ii) the reduce operator is arbitrary (user defined), may be only associative or also commutative, and its supported input type may be a tuple of scalars or even arrays in some cases. The contributions of this paper are:

First, we demonstrate in Section 3 how map and reduce operators are composed aggressively (at all nest-levels in the program) by a combination of producer-consumer and horizontal fusion. The result of fusion is an operator, named redomap, that is not exposed to the user language because (i) its semantics is non-trivial/non-intuitive, and (ii) because in all map-reduce examples we encountered so far the compiler reliably fuses such compositions.

Second, we present in Section 4 how the high-level operator is translated to a lower-level representation in a manner that optimizes both (i) efficient sequentialization of the parallelism in excess, and (ii) coalesced access to global memory, which in particular may have appeared as an artifact of efficient sequentialization. Intuitively, the latter is achieved by reshaping the array to have an inner dimension of the sequential-chunk size, and by transposing this dimension to the outermost level.

Third, we present in Section 5 a detailed evaluation of performance results on a mini-benchmark that demonstrate that the automatically optimized Futhark code outperforms tuned Thrust programs by a geometric-mean factor of 1.75× (and as high as 8×).
2. A Brief and Informal Introduction to Futhark

Futhark\(^1\) is a monomorphic, statically typed, eagerly evaluated, pure functional language, in which nested parallelism is expressed via a set of second-order array combinator (SOACs), e.g., map, reduce, scan, filter, and which aims at efficient execution on GPUs. Figure 1 presents the abstract syntax of a subset of the Futhark core language, which is a restricted language used internally by the compiler. Whenever \( q \) is an object of some kind, we write \( q^{(n)} \) (or simply \( q \)) to range over sequences of \( n \) objects of that kind, that is, \( q_1, \ldots, q_n \). Important details are:

- Similarly to the A-normal form representation \(^{26}\), operands of compound expressions must be variables or values.
- The keyword \( in \) is optional before \( let \) (this is solely for aesthetic reasons).
- The source Futhark language uses the traditional array-of-tuples representation, supporting zip/unzip operators and in which, for example, map receives exactly one array argument.
- However, as shown in Figure 1 the core language does not support tuples because the program is (automatically) normalized to the tuple-of-arrays form. As a result, SOACs such as map take as input several arrays, and results in several arrays; our map can be seen as implicitly zipping its input and unzipping its output. This transformation is automatically performed early on in the compilation process, using a traditional array-of-tuples to tuple-of-array transformation \(^{4}\).
- For any array variable defined in a let-binding pattern, its type is parameterised with the exact shape information, i.e., at array-creation point. For example, \([\text{int}, n], n\) denotes a two-dimensional \( n \times m \) array of integers, where \( n \) and \( m \) must be constants or integer variables in scope.
- All arrays must be regular, that is, all rows of an array must have the same shape. For example array \([4], [1.0]\) is illegal; this is mostly verified through run-time checks.
- Futhark supports sequential do-loops, whose semantics is equivalent to a simple form of tail-recursive functions, as illustrated in Figure 2. The do-loop construct enables several important optimizations such as aggressive hoisting and map-reduce interchange (outside the scope of this paper). The latter may significantly enhance the amount of parallelism that can be statically extracted.

The compiler performs producer-consumer and horizontal fusion, which is demonstrated in Section 3 in the simple case of map-reduce compositions. Fusion is followed by extraction of flat-parallel kernels suitable for translation to GPU code. This step resembles the loop distribution performed by imperative approaches. Ultimately, several key lower-level optimizations are performed, such as changing the in-memory representation of arrays via transposition to promote coalesced access\(^2\) to global GPGPU memory.

### Figure 1: Syntax of a Subset of Futhark’s Core Language

\[
\begin{align*}
\phi & : \text{(fun/lambda type)} \\
\tau & : \text{(size-dep array types)} \\
\rho & : \text{(tuple-of-arrays) types} \\
\phi & : \text{fun} \, \rho \rightarrow \phi \\
\phi & : \text{(function-call)} \\
\text{let} & : \text{(let-binding)} \\
\text{map} & : \text{(n-ary map)} \\
\text{reduce} & : \text{(reduce with n-ary op)} \\
\text{scan} & : \text{(scan with n-ary op)} \\
\text{x with} & : \text{(in-place update)} \\
\text{loop} & : \text{(sequential do-loop)} \\
P & : \text{fun} \, \rho \, g(p) = e \\
\end{align*}
\]

### Figure 2: Loop to recursive function

\[
\begin{align*}
\text{fun} \, f & : (n, a_1, \ldots, a_n) = e \\
\text{for} & \, i < n \, \text{do} \, g(\text{tl} x, \ldots, \text{tl} x_n) \\
\text{else} & \, f(0, n, a_1, \ldots, a_n)
\end{align*}
\]
As such, in previous work, redomap had type:

\[
\text{redomap} : (\alpha \to \alpha \to \alpha) \to (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha
\]

in which the first argument is a binary associative operator \((\odot)\), the second is the folded function \((g)\), the third is the neutral element \((e)\) of the group induced by \(\odot\), and the fourth is the input array. The result type of \text{redomap} is \(\alpha\), that is, the result (and input) type of the binary associative operator \(\odot\). This is a significant restriction because, for example, it does not allow fusing the following code:

\[
\begin{align*}
\text{let } x &= \text{map}(f, a) \\
\text{let } r &= \text{reduce}(+, 0.0, x) \\
\text{in } (r, x)
\end{align*}
\]

If the result of the map is used outside of the reduce then they cannot be fused because \text{redomap}'s type disallows the return of \(x\).

Another significant restriction of the previous work was that horizontal fusion was not supported. The remaining of this section discusses our solution to solving the two (observed) limitations.

3.1 Extended Redomap: Type and Semantics

Denoting with \(z^{(n)}\) the sequence \(z_1, \ldots, z_n\), the type of the \text{redomap} (in the Futhark core language) was extended to:

\[
( (\bar{\alpha}^{(p)}, \bar{\alpha}^{(p)}) \to \bar{\alpha}^{(p)}), (\bar{\alpha}^{(p)}, \bar{\beta}^{(q)}, \bar{\gamma}^{(r)}) \to (\bar{\alpha}^{(p)}, \bar{\gamma}^{(r)}))
\]

1\(^{st}\) argument is a binary associative operator, of type \((\bar{\alpha}^{(p)}, \bar{\alpha}^{(p)}) \to \bar{\alpha}^{(p)}\), where \(\bar{\alpha}^{(p)}\) intuitively denotes a tuple of \(p\) elements.

2\(^{nd}\) argument is the folded function \(g\) of type: \((\bar{\alpha}^{(p)}, \bar{\beta}^{(q)}) \to (\bar{\alpha}^{(p)}, \bar{\gamma}^{(r)})\). That is, it receives the accumulator of type \(\bar{\alpha}^{(p)}\) and an arbitrary number \(q\) of (array) elements \(\bar{\beta}^{(q)}\) and produces a new accumulator \(\bar{\alpha}^{(p)}\) and an arbitrary number \(r\) of (array) elements \(\bar{\gamma}^{(r)}\).

3\(^{rd}\) argument is the neutral element of type \(\bar{\alpha}^{(p)}\),

- The remaining arguments are \(q\) arrays of equal-size outermost dimension \(n\) (i.e., \([\beta_1, n], \ldots, [\beta_q, n]\)).
- The result has two components: (i) the reduced part of type \(\bar{\alpha}^{(p)}\), and (ii) the mapped part \([\gamma_1, n], \ldots, [\gamma_q, n]\) which corresponds to \(r\) arrays of outermost size equal to \(n\) whose elements were produced by each invocation of \(g\).

The semantics of \text{redomap} is:

\[
\text{redomap}(\odot, g, e^{(p)}, \bar{\beta}^{(q)}, \bar{\gamma}^{(r)}) \equiv \text{let } (\bar{\alpha}^{(p)}, e^{(r)}) = \text{map}(g, \bar{\beta}^{(q)}) \text{ in } (\text{reduce}(\odot, e^{(p)}, \bar{\alpha}^{(p)}), e^{(r)}))
\]

except that in practice it is executed very similar to the OPENMP-style parallel loop with reduction pragmas. That is, each processor computes a partial accumulator from a chunk of the mapped arrays, and the partial accumulators are then reduced across processors.

3.2 Horizontal Fusion

The extension of the \text{redomap} operator opened the door to eliminating two important limitations of the previous fusion engine:

First, fusion is allowed between two \text{SOAC}s belonging to the same block of \text{let} statements even if the array produced by the first \text{SOAC} is used after the second\(^3\), i.e., as long as there is no unfused use of first \text{SOAC}'s result array in between the two \text{SOAC}s.

For example, even though in the code below \(x\) is used in the result:

\[
\begin{align*}
\text{let } x &= \text{map}(f, a) \text{ in } \\
\text{let } r &= \text{reduce}(+, 0.0, x) \text{ in } \\
\text{in } (r, x)
\end{align*}
\]

the \text{reduce} and the \text{map} are safely fused into:

\[
\text{let } (x, r) = \text{redomap}( + , \text{fn}(f32, f32) (f32 \text{ e}, f32 \text{ a}) => \\
\text{let } x = f(a) \text{ in } (e+x, x) \\
\text{in } (r, x)
\]

Second, we allow horizontal fusion (i.e., when the two \text{SOAC}s are not in a producer-consumer relation), whenever the two \text{SOAC}s belong to the same block of \text{let} statements, their outermost sizes are equal, and as before, there is no (unfused) use of the result array in between the two \text{SOAC}s. For example:

\[
\begin{align*}
\text{let } x &= \text{reduce}(+, 0.0, a) \\
\text{let } y &= \text{reduce}(+, 1.0, a) \\
\text{is fused horizontally as:}
\text{let } (x, y) = \text{reduce}(\text{fn}(f32, f32) (f32 \text{ x1}, f32 \text{ y1}, \\
\text{f32 \text{ x2}, f32 \text{ y2}) => \\
\text{let } r1 = x1 + x2 \\
\text{let } r2 = y1 + y2 \text{ in } (r1, r2) \\
\text{in } (0.0, a)
\end{align*}
\]

3.3 Demonstrating Redomap Fusion

Figure 3 demonstrates the fusion engine on a contrived example whose structure resembles code from FinPar’s Interest-Rate Calibration benchmark \([1]\). The original code is presented at the top, and \text{reduce\text{Comm}} denotes a \text{reduce} in which the binary operator is declared to be also commutative (besides being associative). The previous fusion implementation would have succeeded in fusing the first three maps, denoted \((7 - 9)\), but not the remaining \text{SOAC}s \((1 - 6)\). The rest of the figure demonstrates the enhanced fusion:

Step I: Since fusion proceeds bottom-up, the last two maps, denoted \((1)\) and \((2)\) are fused horizontally, resulting in the map denoted \((10)\), which receives two array arguments \(x\) and \(y\) and produces arrays \(v\) and \(w\).

Step II: The obtained map \((10)\) is fused horizontally with the \text{reduce} \((3)\) resulting in the \text{redomap} \((11)\), whose binary associative and commutative operator is \text{max}.

Step III: The obtained \text{redomap} \((11)\) is fused horizontally with the other two \text{reduces}, \((4 - 5)\), by extending the associative and commutative operator of the \text{redomap} to work over three-element tuples (and modifying accordingly the folded function).

Step IV: The resulting \text{redomap}, \((12)\) consumes the array \(t\) produced by the map \((16)\) and as such are fused together, and similarly for the maps producing arrays \(y, x\) and \(z\), which are \((7 - 9)\). At the end, the original program has been fused in one \text{redomap} construct, which is shown at the bottom of Figure 3.

The resulting code requires only the two result arrays \(v\) and \(w\) to be maintained in global memory. Because the input array \text{iotan(n)} is optimized away by reference to the thread index space, the rest of the computation involves only scalars. The next section describes in detail how the \text{redomap} second-order operator is mapped to efficient \text{GPGPU} code.

4. Optimizing Sequentialization and Coalescing

The efficient implementation of reductions on GPUs is well studied in the literature \([16, 21]\). This section gives a short introduction to the subject to enable discussion of the implementation and optimization of the \text{redomap} construct.

To execute a reduction in parallel on an idealised perfectly parallel machine, we might use tree reduction: to reduce an \(n\)-element array \([x_1, \ldots, x_n]\) using operator \(\odot\), we launch \(n/2\) threads, with thread \(i\) computing \(x_{2i} \odot x_{2i+1}\). The initial \(n\) elements are thus reduced to \(n/2\) elements. The process is repeated until just a
**Original Program:**

```haskell
fun (f32, f32, f32) [f32, f32]
main (f32 a, f32 b, int n) =
  let is = map (f32, iota(n)) in
  let x = map (*a, is) in
  let y = map (*b, x) in
  let t = map (+, zip(x, y)) in
  let t0 = reduceComm(+, 0.0f32, t) in
  let t1 = reduceComm(min, inf1(), x) in
  let t2 = reduceComm(max, 0.0f32, y) in
  t0; t1; t2;
```

**Figure 3:** Fusion demonstrated on a simplified example, resembling the structure of FinPar's Interest-Rate Calibration benchmark.

**Figure 4:** Tree Reduction

---

Single value is left–the final result of the reduction. We perform $O(\log(n))$ partial reductions, each of which is perfectly parallel, resulting in a work depth of $O(\log(n))$. Figure 4 shows a simple example.

Tree reduction is optimal on an idealised perfectly parallel machine, but on real hardware, such as GPUs, it is inefficient. The inefficiency is caused by exposing more parallelism than needed to fully exploit the hardware. This excess parallelism means we pay an unnecessary overhead due to communication cost between threads. Efficient parallel execution relies on exposing as much parallelism as is needed to saturate the machine, but no more.

On a GPU, a simple improvement is to have every thread in a `workgroup` thread a single value from the initial array, after which the threads cooperate in reducing their individual values to a partial result per `workgroup`. Thus, if we use size-`k` `workgroups`, each parallel level will result in a factor-`k` shrinkage, instead of the factor-2 performed by the naïve algorithm.

We can still do better: as mentioned earlier, a concrete GPU can only use up to a certain amount of parallelism. The mount depends on the hardware and exact form of the reduction, but suppose that `w` groups of size `k` each are sufficient. Then, instead of spawning a number of threads dependent on the input size `n`, we always spawn `w \times k` threads, organised into `w` `workgroups`. Each thread sequentially reduces a chunk of the input consisting of `n_w` elements, producing a per-thread intermediate result, which is then reduced inside each `workgroup` (using tree reduction) to one result per `workgroup`. If `w` is less than the maximum `workgroup` size, we can then launch a new reduction with one `workgroup` of size `w` that reduces the `w` per-`workgroup` results into the final result of the reduction. If `n` is not divisible by `w \times k`, not all threads will have the exact chunk size, but it is not hard to partition the array among the threads. One interesting consequence is that in the sequential stage, the function need not be a proper reduction operator, but can be a fold function with no associativity requirements - this is what we exploit in `redomap`.

This is an efficient algorithm: it uses only as much parallelism as is necessary, and it requires only two kernel launches in total. The only caveat is that we have to be careful about how the threads traverse their assigned chunk of the input. On a GPU, we must ensure that memory accesses are *coalesced* in order to fully utilise the memory bus. For reductions, which are bandwidth-bound for most operators, this is an important concern. If each thread sequentially accesses neighboring elements in the input array, as on figure 5a, the resulting memory accesses will be non-coalesced, severely impacting performance. The solution is to access with a stride, as on figure 5b, if the per-thread chunk size is `wk`, then thread `i` should access the input at indices `i, i + 2wk, i + 3wk`, and so forth. This means that in the same cycle, neighboring threads `i` and `i + 1` within the same `warps` will access elements at neighboring indices `i + jwk` and `i + 1 + jwk` (where `j` is the sequential iteration), thus obtaining coalesced memory access.

The strided access pattern just mentioned will result in the correct result if the operator is commutative, although we are in

---

\(^4\) This paper follows the OpenCL terminology. In NVIDIA’s CUDA, a `workgroup` is called a `thread block`.
effect accessing the array in a transposed form. Even if the input array is one-dimensional, we can pretend that it is a \( w_k \times \frac{1}{w_k} \) array which is then transposed — if the input size is not divisible by \( w_k \) we will have to pad the input, but we will ignore this issue in the present paper.

Many interesting operators are only associative, and swapping the order of application will produce a different (wrong) result. This is for example the case in the maximum segment sum problem, which is discussed in section 5. In such cases, we must first transpose the array in memory prior to the reduction — the strided access in the reduction will then “undo” the first transposition and restore the original evaluation order. Whilst transposition is not a free operation, it can be implemented on the GPU to run at essentially the speed of the memory bus.

This procedure is sketched on figure 6a where we assume that the reduction operator \( f \) is not commutative. The \( \text{redomap} \) on figure 6a is being turned into two kernels, the first of which is shown on figure 6b. The second, single-workgroup kernel is not particularly interesting, as the per-thread chunk size is always 1. In our intermediate language, we represent a reduction kernel via a \( \text{reduceKernel} \) construct. This construct is much like a \( \text{redomap} \), except that instead of a fold function operating on an accumulator and single element, we have a function that processes an entire chunk of the input array with an explicit sequential loop.

In Futhark, the transposition operation is by default delayed (i.e., not made manifest in memory) — instead, the code generator will replace accesses to a transposed array with accesses to the original array, with the indices swapped. This is exploited in figure 6b — the sequential loop accesses to \( \text{chunk}[\cdot] \), really access \( x s'[t, \cdot] \) (where \( t \) is the thread index), corresponding to \( x s_{\text{tr}}\_\text{manifested}[t, \cdot] \). This array is itself a manifested version of \( x s_{\text{tr}}\_\text{delayed} \), which is a (delayed) transposed version of the original \( x s \). Thus, \( x s'[t, i] == x s[t, i] \), but with \( x s' \) represented in transposed form in memory.

The mapped part of the \( \text{redomap} \) is always produced in transposed form by the \( \text{reduceKernel} \), again to ensure coalescing. We explicitly transpose the mapped part to restore the intended order. If the reduction operator is commutative, and there is no mapped part in the \( \text{redomap} \), the transposition is not actually performed in memory — this is for example the case when compiling particularly simple reductions such as \( \text{reduce}(+, 0, \text{as}) \).

During the parallel part of the reduction, intermediate results are kept in arrays stored in fast local memory on the GPU. We use one such array for every input array. Due to the array-of-tuples transformation mentioned in section 5, a reduction of an array of pairs turns into a reduction of two arrays, meaning that we use two local buffers during the reduction.

\( \text{reduce}(+, 0, \text{as}) \) is for example the case in the Black-Scholes and MSSP maximum segment sum problem.

Figure 6a is being turned into two kernels, the first of which is shown on figure 6b. The second, single-workgroup kernel is not particularly interesting, as the per-thread chunk size is always 1. In our intermediate language, we represent a reduction kernel via a \( \text{reduceKernel} \) construct. This construct is much like a \( \text{redomap} \), except that instead of a fold function operating on an accumulator and single element, we have a function that processes an entire chunk of the input array with an explicit sequential loop.

In Futhark, the transposition operation is by default delayed (i.e., not made manifest in memory) — instead, the code generator will replace accesses to a transposed array with accesses to the original array, with the indices swapped. This is exploited in figure 6b — the sequential loop accesses to \( \text{chunk}[\cdot] \), really access \( x s'[t, \cdot] \) (where \( t \) is the thread index), corresponding to \( x s_{\text{tr}}\_\text{manifested}[t, \cdot] \). This array is itself a manifested version of \( x s_{\text{tr}}\_\text{delayed} \), which is a (delayed) transposed version of the original \( x s \). Thus, \( x s'[t, i] == x s[t, i] \), but with \( x s' \) represented in transposed form in memory.

The mapped part of the \( \text{redomap} \) is always produced in transposed form by the \( \text{reduceKernel} \), again to ensure coalescing. We explicitly transpose the mapped part to restore the intended order. If the reduction operator is commutative, and there is no mapped part in the \( \text{redomap} \), the transposition is not actually performed in memory — this is for example the case when compiling particularly simple reductions such as \( \text{reduce}(+, 0, \text{as}) \).

During the parallel part of the reduction, intermediate results are kept in arrays stored in fast local memory on the GPU. We use one such array for every input array. Due to the array-of-tuples transformation mentioned in section 5, a reduction of an array of pairs turns into a reduction of two arrays, meaning that we use two local buffers during the reduction.

Figure 5: Strided and Nonstrided Chunked Reduction

![Figure 5: Strided and Nonstrided Chunked Reduction](image)

5 Futhark by default represents arrays in row-major form, so we could also say that the transposed array is column-major.
1. **RedomapNT**

   ```c
   struct RedomapNT
   struct Mini-Benchmark used for evaluating the Redomap concept.
   ```

2. **ReduceMax**

   ```c
   fun int main([int] a) = reduce(+, 0, a)
   ```

3. **IndexOfMax**

   ```c
   fun int main([int] as) = reduceComm(max, 0, as)
   ```

4. **IndexOfMaxPack**

   ```c
   fun int main([int] as) =
       let (ix, iy) = map(roimixin, 0, as) in
       i
   ```

5. **Reduce2x2MM**

   ```c
   fun int main([int] a) =
       loop(s = 1) = for i < 40 do
         let a' = map(+, a)
          in reduce(twoByTwoMult, 0xF000F000, a')
       in s
   ```

6. **MSSP**

   ```c
   fun int main([int] x) =
       let (s, _ , _ , _ ) = map(maxWithIndex, 0, x) in
       i
   ```

7. **ReducePlus**

   ```c
   fun int main([int] a) = reduce(+, 0, a)
   ```

8. **ScanPlus**

   ```c
   fun [int] main([int] a) = scan(+, 0, a)
   ```

9. **RedomapNT resembles fusion’s running example**

   ```c
   fun int main([int] a) = scan(+, 0, a)
   ```

10. **BlackScholes**

    ```c
    BlackScholes = 20
    ```

11. **Mini-Benchmark Evaluation**

    Figure 5 shows the running time (in miliseconds) of the Futhark and Thrust code of the eight mini programs, when the input arrays have sizes: 10^2, 5×10^2, 10^3, 5×10^3, 10^4, 5×10^4, and 10^5. Important observations are:

    - **ReducePlus and ReduceMax** operates on 32-bit integer scalars and exhibit very similar behavior: Futhark is about 5× faster on sizes up to one million, and the gap narrows after that, e.g., only 1.55× faster on the largest dataset. **IndexOfMaxPack** operates on 64-bit integers and shows a similar trend: Futhark is about 1.3× faster on the largest array size.

    - **IndexOfMax** operates on two-integer tuples which seems to significantly affect Thrust’s performance: on arrays of size fifty thousand, Futhark is about 4× faster, and the speedup reaches about 1.4× for the largest size. This is likely due to genericity-related constraints, such as using an array of tuple representation (we use Thrust’s zip-iterator). **ThrustOpt** denotes the version of the code that uses Thrust built-in, hence optimized, max_element operator, but this is still 2× slower than Futhark.

    - **MSSP** performs a reduction with a non-commutative operator on a four-integer tuple. We have implemented this as a transform_inclusive_scan, which fuses together the map and the scan and selects the last element of the scanned array.

    5.3 Mini-Benchmark Evaluation

    Figure 8 shows the running time (in miliseconds) of the Futhark and Thrust code for epoch six of the eight mini programs, when the input arrays have sizes: 10^2, 5×10^2, 10^3, 5×10^3, 10^4, 5×10^4, and 10^5. Important observations are:

    - **ReducePlus** and **ReduceMax** operates on 32-bit integer scalars and exhibit very similar behavior: Futhark is about 5× faster on sizes up to one million, and the gap narrows after that, e.g., only 1.55× faster on the largest dataset. **IndexOfMaxPack** operates on 64-bit integers and shows a similar trend: Futhark is about 1.3× faster on the largest array size.

    - **IndexOfMax** operates on two-integer tuples which seems to significantly affect Thrust’s performance: on arrays of size fifty thousand, Futhark is about 4× faster, and the speedup reaches about 1.4× for the largest size. This is likely due to genericity-related constraints, such as using an array of tuple representation (we use Thrust’s zip-iterator). **ThrustOpt** denotes the version of the code that uses Thrust built-in, hence optimized, max_element operator, but this is still 2× slower than Futhark.

    - **MSSP** performs a reduction with a non-commutative operator on a four-integer tuple. We have implemented this as a transform_inclusive_scan, which fuses together the map and the scan and selects the last element of the scanned array.

    5.3 Mini-Benchmark Evaluation

    Figure 8 shows the running time (in miliseconds) of the Futhark and Thrust code for epoch six of the eight mini programs, when the input arrays have sizes: 10^2, 5×10^2, 10^3, 5×10^3, 10^4, 5×10^4, and 10^5. Important observations are:

    - **ReducePlus** and **ReduceMax** operates on 32-bit integer scalars and exhibit very similar behavior: Futhark is about 5× faster on sizes up to one million, and the gap narrows after that, e.g., only 1.55× faster on the largest dataset. **IndexOfMaxPack** operates on 64-bit integers and shows a similar trend: Futhark is about 1.3× faster on the largest array size.

    - **IndexOfMax** operates on two-integer tuples which seems to significantly affect Thrust’s performance: on arrays of size fifty thousand, Futhark is about 4× faster, and the speedup reaches about 1.4× for the largest size. This is likely due to genericity-related constraints, such as using an array of tuple representation (we use Thrust’s zip-iterator). **ThrustOpt** denotes the version of the code that uses Thrust built-in, hence optimized, max_element operator, but this is still 2× slower than Futhark.

    - **MSSP** performs a reduction with a non-commutative operator on a four-integer tuple. We have implemented this as a transform_inclusive_scan, which fuses together the map and the scan and selects the last element of the scanned array.
quired when executing non-commutative reductions. Futhark is about 8.8× faster on an array of size fifty thousand and speedup decreases to only 3.6× on the largest size. This suggests that Thrust’s inefficiencies are related to a larger extent with the use of tuples rather than to the use of scan.

• In fact, ScanPlus shows that Thrust’s scan operator is up to 2.4× faster than Futhark’s. This is because Futhark’s scan is implemented in a manner similar to redomap (i.e., efficiently sequentializes the computation at the cost of transposing both the input and result arrays (to preserve coalesced accesses)). The overhead introduced the transposition is however too high, as it reaches two thirds of the total scan runtime.

• On RedomapNT the Futhark compiler has fused all operators into one redomap, without duplicating any computation. Since Thrust does not support an operator capable of expressing this, the Thrust optimized version corresponds to the code being fused in two kernels: semantically a redomap that “consumes” the input arrays and another map that duplicates some of the computation performed in redomap. Thrust unoptimized version corresponds to the unfused code. We observe that Futhark is about 5× faster on the largest dataset, which seems to indicate that the Thrust’s code manifests some of the intermediary arrays (i.e., it is memory bound rather than compute bound as in Futhark).

Finally, the code version denoted NoRedomap is produced by the Futhark compiler when the redomap fusion is disabled. Redomap fusion has significant impact (∼4×) in the cases of indexOfMaxPack, MSSP and Reduce2by2MM, and smaller impact for RedomapNT and Black-Scholes.

6. Related Work

One strand of related work is embedded domain-specific languages targeting GPGPU execution, such as Nikola [19], Accelerate [20], Obsidian [6], and SPL [10]. Such embedded solutions often suffer significant limitations imposed by the host language, for example they do not support nested parallelism. Perhaps more related to Futhark is the work on SAC [11, 12], which is a standalone functional language that, like Futhark, also seeks a common ground with imperative approaches, but, unlike Futhark, does not support tuple types and operators such as filter and scan in the intermediate representation. The with-loops of SAC can fulfill the same
role as redomap [3], although the fusion algorithm in which they are used is very different: in SAC, producer-consumer and horizontal fusion is combined in a general framework of with-loop fusion.

Another strand of related work is concerned with automatic parallelization of imperative code. Solution include (i) static dependency analyses by polyhedral analysis [14, 23], and (ii) dynamic extraction of parallelism, for example by means of thread-level speculation [7, 24]. The latter, however, have not been found suitable for GPGPU execution.

Finally, a rich body of work has studied interoperability solutions between array languages, for example (i) for inter-operating across computer-algebra systems [5, 23] or (ii) for efficient integration of option pricing in banks’ IT infrastructure [22], or (iii) for integrating a contract-specification language [2] with dynamical graphical user interface [9]. Since Futhark is intended as an accelerator language, work is in progress to extend the compiler with various backends and friendly foreign function interfaces for various mainstream languages, such as Python, Java, C++, etc, as well as using Futhark in conjunction with TAIL [8] as an intermediate language for an APL compiler.

7. Conclusions
This paper has presented a new data-parallel core construct, redomap, that can be used to perform both horizontal and producer-consumer fusion in a variety of complex cases, and which enables efficient mapping to parallel hardware.

Furthermore, we have validated our full-automated compiler on a range of microbenchmarks that perform both simple and nontrivial reductions, several with non-commutative operators. We show that we obtain performance comparable to or exceeding that of the Thrust library in most cases involving reductions, thus empirically demonstrating the potential for efficient compilation of the redomap construct.

We have also shown the value of supporting reductions with non-commutative operators, as well as demonstrating that such reductions can be compiled efficiently on modern GPU hardware.

Acknowledgments
This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center ‘HIPERFIT: Functional High Performance Computing for Financial Information Technology’ (http://hiperfit.dk) under contract number 10-092299.

References