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)
athas@sigkill.dk, kflarsen@di.ku.dk, cosmin.oancea@diku.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:

- demonstrating an aggressive fusion technique that is centered on the `redomap` operator.
- a compilation technique for `redomap` that efficiently sequentializes the excess parallelism and ensures coalesced access to global memory, even for non-commutative `reduce` operators.
- a detailed performance evaluation showing that Futhark's automatically generated code matches or exceeds performance of hand-tuned Thrust code.

Our evaluation infrastructure is publicly available[1] 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 utilizing 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 (Hoberock and Bell 2016), 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 (Guo et al. 2011; Bergstrom and Reppy 2012; McDonell et al. 2013) 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 (Henriksen and Oancea 2013): 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, i.e., user defined, may be only associative or also commutative, and its supported input type may be tuple of scalars or even arrays in some cases. Main 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 pro-

[1] https://github.com/HIPERFIT/futhark-array16

$\bar{q}^{(n)}$	::=	$q_1, \ldots, q_n$	(notation)
s, x	::=	`id`	(variable names, s for scalar)
k	::=	`Ct` | $[k_1, \ldots, k_n]$	(scalar or array value)
z	::=	`id` | `Ct`	(variable name or constant)
g, h	::=	`id`	(function names)
p	::=	$\tau_1\ x_1$	(typed variable)

t	::=	`int` | `bool` | `f32` | `f64`	(basic types)
τ	::=	t | $[\tau, s]$	(size-dep array types)
ρ	::=	$\{\tau_1, \ldots, \tau_n\}$	(tuple (of arrays) types)
ϕ	::=	$\rho_1 \rightarrow \rho_2$	(fun/lambda type)

l	::=	`fn` $\bar{t}^{(n)}$ `(`$\bar{p}^{(m)}$`) =>` e	(anonymous fun)

e	::=	z	(Variable or Value)
	|	$\{\bar{y}^{(n)}\}$	(n-tuple exp)
	|	x`[`y_1`, ..., `y_n`]`	(array indexing)
	|	$y_1 \odot y_2$	(binop-call)
	|	g`(`$\bar{y}$`)`	(function-call)
	|	`if` s `then` e_1 `else` e_2	(if binding)
	|	`let {` $\bar{p}$ `} =` e_1 `in` e_2	(let-binding)
	|	`iota(`s`)`	($[0,\ldots,s-1]$)
	|	`map(`l`,` $\bar{x}^{(n)}$`)`	(n-ary map)
	|	`reduce(`l`,` $\bar{z}^{(n)}$`,` $\bar{x}^{(n)}$`)`	(reduce with n-ary op)
	|	`scan(`l`,` $\bar{z}^{(n)}$`,` $\bar{x}^{(n)}$`)`	(scan with n-ary op)
	|	x `with [`s_1`,...,`s_n`] <-` y	(in-place update)
	|	`loop (`$\bar{p}^{(n)}$ `=` $\bar{y}^{(n)}$`) =`	(sequential do-loop)
		`for` $s < S$ `do` e	(next iter $\bar{p}^{(n)} \leftarrow e$)

P ::= `fun` ρ g`(`p`) =` e`;` P | ϵ (named function def)

Figure 1: Syntax of a Subset of Futhark's Core Language.

grams by a geometric-mean factor of $1.75\times$ (and as high as $8\times$). Finally, Section 2 briefly introduces Futhark informally, Section 6 reviews related work and Section 7 concludes the paper.

2. A Brief and Informal Introduction to Futhark

Futhark[2] is a monomorphic, statically typed, strictly evaluated, purely functional language, in which nested parallelism is expressed via a set of second-order array combinators (SOAC), e.g., `map`, `reduce`, `scan`, `filter`, and which aims at efficient execution on GPGPUs. Figure 1 presents the abstract syntax of a subset of Futhark *core language*, which is a restricted language used internally by the compiler. Whenever q is an object of some kind, we write $\bar{q}^{(n)}$ (or simply $\bar{q}$) 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 (Sabry and Felleisen 1992), 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 `zip`ping its input and `unzip`ping its output.
- 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, `[[int,m],n]` denotes a two-dimensional $n \times m$ array of integers, where n and m must be constants or integer variables in scope.
- While not in the scope of this paper, our core language also supports existential types (Henriksen et al. 2014) to express the type of the result arrays whose shapes cannot be known in advance. In practice, optional user-annotated sizes allows the compiler to eliminate existential types in most cases.
- 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-loop` interchange (outside the scope of this paper). The latter may significantly enhance the amount of parallelism that can be statically extracted.

```
loop ({x1,...,xn} =          -- Equivalent function
      {a1,...,an}) =         fun t f( int i
  for i < n do                        , int n
    g(i, x1,...,xn)  ⇒                , t1 x1
                                      ..., tn xn) =
-- Equivalent to:              if i >= n then x
f(0, n, a1, ..., an)           else f(i+1, n, g(i,x1,...,xn))
```

Figure 2: Loop to recursive function

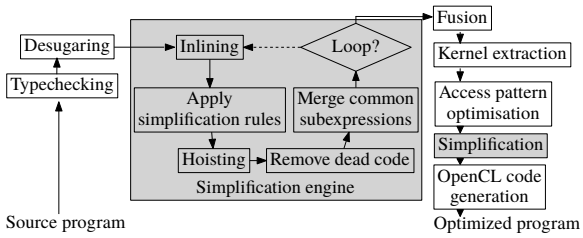


Figure 3: Compiler pipeline.

Finally, Figure 3 shows the compilation stages: Type checking operates on the original program to allow meaningful error messages, then desugaring uniquely renames bindings, changes the representation to tuple-of-arrays (Blelloch et al. 1994), (i.e., removes `(un)zip`s, and normalizes the program to A-norm form (Sabry and Felleisen 1992)). This results in a core-language program whose syntax was shown in Figure 1. Then functions are aggressively inlined and a mixture of rewrite rules, CSE, hoisting, copy/constant propagation, constant folding and dead-code removal optimize away simple inefficiencies. The next step is producer-consumer and horizontal fusion, which is demonstrated in Section 3 in the simple(r) 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, but also supports interchanging an outer `map` with an inner `do`-loop in the attempt to create more parallelism. Ultimately, several key lower-level optimizations are performed,

[2] The language is named after the first six letters of the runic alphabet, (i.e., "Fuþark"). The `filter` operator is not covered in this paper.

such as changing the in-memory representation of arrays via transposition to promote *coalesced accesses*[3] to global GPGPU memory.

3. Redomap: Type, Semantics and Fusion Rules

The Futhark source language does not support the `redomap` second-order array combinator (SOAC), because this is synthesized automatically by fusion from `map-reduce` compositions.

In previous work (Henriksen and Oancea 2013) we have proposed an aggressive technique for fusing producer-consumer SOACs, without duplicating computation, even when the produced array is used in several places. Semantically, this is achieved by a bottom-up traversal of program's dependency graph, in which SOACs are fused whenever a T2 reduction is possible (i.e., a SOAC can be fused if it is the source of only one dependency edge, and the target of the dependence is another (compatible) SOAC). The technique was further refined to solve some hindrances to fusion, for example, to move intervening transpositions before/after the fused kernel and to interchange an outer `reduce` with an inner `map` in order to create more fusion opportunities.

Using a Haskell notation, the `redomap` construct was inspired by the following rewriting of a `map-reduce` composition:
$\texttt{red} \odot \texttt{e} \,.\, \texttt{map f} \;\equiv\; \texttt{red} \odot \texttt{e} \,.\, \texttt{map}\,(\texttt{red} \odot \texttt{e} \,.\, \texttt{map f}) \,.\, \texttt{split}_p$
$\equiv\; \texttt{red} \odot \texttt{e} \,.\, \texttt{map}\,(\texttt{foldl g e}) \,.\, \texttt{split}_p \;\equiv\; \texttt{redomap} \odot \texttt{g e}$

- the input array is `split` into a number of chunks that roughly equals the number of available processors
- each processor efficiently sequentializes the computation on its chunk by rewriting the inner $(\texttt{red} \odot \texttt{e})\,.\,(\texttt{map f})$ as $\texttt{foldl g e}$,
- and finally, the partial results are reduced across processors.

As such, in previous work, `redomap` had type:
$\texttt{redomap}:: (\alpha \to \alpha \to \alpha) \to (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \;\to\; \alpha$
in which the first argument is a binary associative operator ($\odot$), the second is the `fold`ed 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 `redomap` is α, 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:

```
let x = map(f, a) in
let r = reduce(+, 0.0, x) in
{r,x}
```

That is, if the result of the `map` is used outside of the `reduce` then they cannot be fused because `redomap`'s type disallows the return of `x`.

Another significant restriction of the previous work was that horizontal fusion was not supported (i.e., two SOACs were fused only when an array produced by the first SOAC was consumed by the second SOAC). The remaining of this section discusses our solution to solving the (two) observed limitations.

3.1 Extended Redomap: Type and Semantics

Denoting with $\bar{z}^{(n)}$ the sequence $z_1, \ldots, z_n$, the type of the `redomap` (in the Futhark core language) was extended to:
$(\,((\bar{\alpha}^{(p)}, \bar{\alpha}^{(p)}) \to \bar{\alpha}^{(p)}),\; ((\bar{\alpha}^{(p)}, \bar{\beta}^{(q)}) \to (\bar{\alpha}^{(p)}, \bar{\gamma}^{(r)})),$
$\bar{\alpha}^{(p)}, [\beta_1, n], \ldots, [\beta_q, n]\,) \;\to\; (\bar{\alpha}^{(p)},\; [\gamma_1, n], \ldots, [\gamma_r, n])$

- 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 $[\gamma_1, n], \ldots, [\gamma_r, n]$ which corresponds to r arrays of outermost size equal to n whose elements were produced by each invocation of g.

The semantics of `redomap` is:
$\texttt{redomap}(\oplus, g, \bar{e}^{(p)}, \bar{b}^{(q)}) \;\equiv\; \texttt{let}\,\{\bar{a}^{(p)},\, \bar{c}^{(r)}\} = \texttt{map}(g,\, \bar{b}^{(q)})$
$\texttt{in}\;(\texttt{reduce}(\oplus, \bar{e}^{(p)}, \bar{a}^{(p)}),\; \bar{c}^{(r)})$
except that in practice it is executed very similar to the OPENMP-style parallel loop with reduction pragmas, that is, each processor produces its chunk of mapped arrays and private accumulator) and the partial accumulators are then reduced across processors.

3.2 Horizontal Fusion

The extension of the `redomap` operator opened the door to eliminating two important limitations of the previous fusion engine:

First, fusion is allowed between two SOACs belonging to the same block of `let` statements even if the array produced by the first SOAC is used after the second[4], i.e., as long as there is no unfused use of first SOAC's result array in between the two SOACs. For example, even though in the code below `x` is used in the result:

```
let x = map(f, a) in
let r = reduce(+, 0.0, x)
in {r, x}
```

the `reduce` and the `map` are safely fused into:

```
let {x,r}=redomap( +
                 , fn {f32, f32}
                      (f32 e, f32 a) =>
                      let x = f(a) in {e+x,x}
                 , 0.0, a)
in {r, x}
```

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

```
let x = reduce(+, 0.0, a)
let y = reduce(*, 1.0, a)
```

is fused horizontally as:

```
let {x,y}=reduce( fn  { f32, f32 }
                      ( f32 x1, f32 y1,
                        f32 x2, f32 y2 ) =>
                      let r1 = x1 + x2 in
                      let r2 = y1 * y2 in
                      {r1, r2}
                , 0.0, a, a)
```

3.3 Demonstrating Redomap Fusion

Figure 4 demonstrates the fusion engine on a contrived example, but whose structure resembles code from FinPar's Interest-Rate

[3] Coalesced access requires that neighboring threads access neighboring memory words in a SIMD instruction.

[4] This can be further relaxed by using a dependency-graph representation of block of `let` statements.

```
-- Original Program:
fun {f32, f32, f32, [f32], [f32]}
main(f32 a, f32 b, int n) =
  let is= map(f32, iota(n))                          -- (9)
  let x = map( *a, is )                              -- (8)
  let y = map( *b, x  )                              -- (7)
  let t = map( +, zip(x,y) )                         -- (6)
  in
  let t0 = reduceComm( +,   0.0f32,   t )            -- (5)
  let t1 = reduceComm( min, inf(),    x )            -- (4)
  let t2 = reduceComm( max, 0.0f32,   y )            -- (3)
  in
  let v = map(*a, x )                                -- (2)
  let w = map(*b, y )                                -- (1)
  in {t0, t1, t2, v, w}

-- I. horizontal fusion (1) and (2) results in (10):
  let {[f32,n] v, [f32,n] w} =
    map(fn {f32, f32} (f32 xi, f32 yi) =>            -- (10)
          let {f32 res_1} = xi * a
          let {f32 res_2} = yi * b
          in {res_1, res_2},
        , x, y)

-- II. horizontal fusion (10) and (3) results in (11):
let {f32 t2, [f32,n] v, [f32,n] w} =
    redomapComm( max                                 -- (11)
               , fn {f32, f32, f32}
                    (f32 acc, f32 yi, f32 xi) =>
                    let {f32 res} = max(acc, yi)
                    let {f32 vi } = xi * a
                    let {f32 wi } = yi * b
                    in {res, vi, wi}
               , {0.0f32}, y, x)

-- III. horizontal fusion (11) and (4,5) results in (12):
let {f32 t0, f32 t1, f32 t2, [f32,n] v, [f32,n] w} =
    redomapComm(fn { f32, f32, f32 }                 -- (12)
                   ( f32 x1, f32 x2, f32 x3
                   , f32 y1, f32 y2, f32 y3 ) =>
                  let {f32 z1} = x1 + y1
                  let {f32 z2} = min(x2, y2)
                  let {f32 z3} = max(x3, y3)
                  in {z1, z2, z3},
                fn {f32, f32, f32, f32, f32}
                   ( f32 acc1, f32 acc2, f32 acc3
                   , f32 ti,   f32 xi,   f32 yi  ) =>
                  let {f32 res1} = acc1 + ti
                  let {f32 res2} = min(acc2, xi)
                  let {f32 res3} = max(acc3, yi)
                  let {f32 vi}   = xi * a
                  let {f32 wi}   = yi * b
                  in {res1, res2, res3, vi, wi},
               {0.0f32, Infinityf32, 0.0f32}, t, x, y)

-- IV. Producer-consumer fusion between (12) and (6),
--     then (7) then (8) then (9) results in (13):
let {f32 t0, f32 t1, f32 t2, [f32,n] v, [f32,n] w} =
    redomapComm(fn { f32, f32, f32 }                 -- (13)
                   ( f32 x1, f32 x2, f32 x3
                   , f32 y1, f32 y2, f32 y3 ) =>
                  let {f32 z1} = x1 + y1
                  let {f32 z2} = min(x2, y2)
                  let {f32 z3} = max(x3, y3)
                  in {z1, z2, z3},
                fn {f32, f32, f32, f32, f32}
                   (f32 acc1,f32 acc2,f32 acc3, int i) =>
                  let {f32 iflt} = f32(i)
                  let {f32 xi  } = iflt * a
                  let {f32 yi  } = xi   * b
                  let {f32 ti  } = xi   + yi
                  let {f32 res1} = acc1 + ti
                  let {f32 res2} = min(acc2, xi)
                  let {f32 res3} = max(acc3, yi)
                  let {f32 vi}   = xi * a
                  let {f32 wi}   = yi * b
                  in {res1, res2, res3, vi, wi},
               {0.0f32, Infinityf32, 0.0f32}, iota(n) )
```

Figure 4: Fusion demonstrated on a simplified example, resembling the structure of FinPar's Interest-Rate Calibration benchmark.

Calibration benchmark (Andreetta et al. 2016). The original code is presented on the top part of the Figure and `reduceComm` denotes a `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 `map`s, denoted $(7-9)$, but not the remaining SOACs $(1-6)$. The rest of the figure demonstrates the enhanced fusion engine:

Step I: Since fusion proceeds bottom up, the last two `map`s, 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 `reduce` numbered (3) resulting in the `redomap` numbered (11), whose binary associative and commutative operator is `max`.

Step III: The obtained `redomap` (11) is fused *horizontally* with the other two `reduce`s, numbered $(4-5)$, by extending the associative and commutative operator of the `redomap` to work over three-float tuples (and modifying accordingly the folded function).

Step IV: The resulted `redomap`, numbered (12) consumes the array `t` produced by the `map` numbered (16) and as such are fused together, and similarly for the `map`s producing arrays `y`, `x` and `is`, which are numbered $(7-9)$. At the end, the original program has been fused in one `redomap` construct, which is shown at the bottom of Figure 4.

We note that the resulted code requires only the two result arrays `v` and `w` to be maintained in global memory, and the rest of computation involves only scalars (because the input array `iota(n)` is optimized away as a normalized-iteration space). The next section describes in detail how the `redomap` second-order operator is mapped to efficient GPGPU code.

4. Optimizing Sequentialization and Coalescing

The efficient implementation of reductions on GPUs is well studied in the literature (Harris et al. 2007; Nickolls et al. 2008). Nevertheless, we shall give a small introduction to establish a foundation on which we can discuss the implementation and optimisation of the `redomap` construct.

There are several different ways to execute a reduction in parallel. On an idealised perfectly parallel machine, we would probably opt for *tree reduction*: to reduce an n-element array $[x_1, \ldots, x_n]$ using operator $\oplus$, we launch $n/2$ threads, with thread i computing $x_{2i} \oplus x_{2i+1}$. The result is that the initial n elements are reduced to $n/2$ elements, a process that we continue until we are left with just a single value - the final result of the reduction. We have to perform $O(\log(n))$ partial reductions, each of which is perfectly parallel, resulting in a work depth of $O(\log(n))$. A simple example is shown on figure 5. On an idealised perfectly parallel machine, this is the best we can do, but on real hardware, such as GPUs, tree reduction is not efficient. The problem is that we expose more parallelism than is needed to fully exploit the hardware, thereby paying an unnecessary overhead in the form of communication cost between threads due to this *excess parallelism*. Efficient parallel execution relies on exposing as much parallelism as is needed to saturate the machine, but no more.

On GPUs, we have to be aware of further restrictions. A GPU program (typically called a *kernel*) is executed by specifying a number of equally-sized *workgroups*, where each workgroup consists of a number of parallel threads.[5] The maximum size of a workgroup is limited to a relatively small number, typically 1024 on present hardware. Threads can communicate with each other within

[5] This paper follows the OpenCL terminology. In NVIDIAs CUDA, a workgroup is called a *block*.

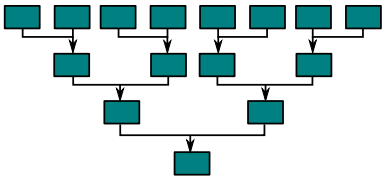
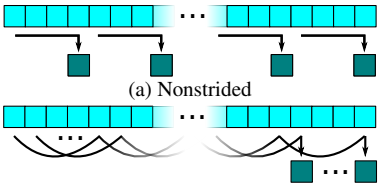


Figure 5: Tree Reduction



(a) Nonstrided

(b) Strided

Figure 6: Strided and Nonstrided Chunked Reduction

a single workgroup via group-wide *barriers*, but there is no way for threads in different workgroups to communicate. The only way to obtain synchronisation between workgroups (barring very expensive atomic global memory operations) is to write intermediary results to global memory, wait for the entire kernel to finish, and then launch a new kernel that can then read the results back from global memory. Thus, the tree reduction from figure 5 could be implemented by having each layer be a single kernel invocation where each thread writes its result to an array in global memory, which is then passed on to the next kernel invocation. This requires no synchronisation between threads within a kernel, but it does require us to write back intermediate results to global memory, which is very inefficient.

A simple improvement is to have every thread in a workgroup read a single value from the initial array, and then have the threads in the workgroup cooperate in reducing their individual values to a single result. 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 naive algorithm.

We can still do better: as mentioned earlier, there is a maximum amount of parallelism that a concrete GPU can take advantage of. The specific number is dependent on the hardware and exact form of the reduction, but assume 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 then sequentially reduces a chunk of the input consisting of $\frac{n}{w \times k}$ elements, producing a per-thread intermediate result, which is then reduced inside each workgroup (using tree reduction) to one result per workgroup. Then, if we have picked w to be less than the maximum workgroup size, we can 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 same chunk size, but it is not hard to divide 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 a good 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 6a, the resulting memory accesses will be non-coalesced, severely impacting performance. The solution is to access with a stride, as on figure 6b: 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 and $i + 1$ within the same warp 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 effect accessing the array in a *transposed* form. Even if the input array is one-dimensional, we can pretend that it is a $wk \times \frac{n}{wk}$ array which is then transposed – if the input size is not divisible by wk 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, which is an easy tradeoff if the alternative is non-coalesced memory access in the reduction kernel.

This procedure is sketched on figure 7, where we assume that the reduction operator `f` is not commutative. The `redomap` on figure 7a is being turned into two kernels, the first of which is shown on figure 7b. 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 `reduceKernel` construct. This construct is much like a 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 `transpose` operation is by default *delayed* (i.e., not made manifest in memory) – instead, the code generator will replace acceses to a transposed array with accesses to the original array, with the indices properly transformed. This is exploited in figure 7b – when the sequential loop accesses `chunk[i]`, it is really accessing `xs'[t,i]` (where `t` is the thread index), which corresponds to `xs_tr_manifested[i,t]`. This array is itself an explicitly manifested version of `xs_tr_delayed`, which is a (delayed) transposed version of the original `xs`. Thus, `xs'[t,i]==xs[t,i]`, but with `xs'` being represented in a transposed form in memory.[6]

Furthermore, the mapped part of the `redomap` is always produced in transposed form by the `reduceKernel`, again to ensure coalescing. Thus, 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 `redomap`, we do not need the first transposition and its manifestation, as it does then not matter that the input is read in a permuted order. In this case, the transposition is not actually performed in memory – this is for example the case when compiling particularly simple reductions such as `reduce(+, 0, as)`.

During the parallel part of the reduction, intermediate results are kept in arrays stored in fast memory on the GPU (termed *local memory* in OpenCL and *shared memory* in CUDA). We store one such array for every input array being reduced. Due to array-of-tuples to tuple-of-array transformation mentioned in section 2, a reduction of an array of pairs will be turned into a reduction of two

[6] Futhark by default represents arrays in row-major form, so we could also say that the transposed array is column-major.

```
let {int p, [int,n] vs} =
  redomap(f,
      fn {int, int} (int acc, int x) =>
        let {int res} = f(acc, x)
        let {int v} = x * a
        {res, v},
      {0}, xs)
```

(a) The original `redomap`

```
let xs_tr_delayed = transpose(xs)
let xs_tr_manifested = manifest(xs_tr_delayed)
let xs' = transpose(xs_tr_manifested)
let {[int,w] group_res, [int,n] vs_trans} =
  reduceKernel(f, -- reduction operator
    fn {int,[int,c]} (int c, [int,c] chunk) =>
      let init_ws = replicate(c, 0) in
      loop ({acc, ws'} = {0, init_ws}) =
        for i < c do
          let x = chunk[i]
          let {int res} = f(acc, x)
          let {int v} = x * a
          let {[int,c] ws''} =
            ws' with [i] <- v
          in {res, ws''},
      {0}, xs')
let vs = transpose(vs_trans)
```

(b) The first of the two resulting kernels

Figure 7: Compilation of `redomap` to GPU kernels

arrays, which means that we use two local memory arrays during the reduction, thus minimizing the number of bank conflicts.

5. Performance Analysis

5.1 Experimental Methodology and Hardware

Reported running times are averaged across one hundred runs within a loop. An initial iteration is executed without timing in order to avoid warmup and delayed initialization affecting the measured runtime. Running times represent core computation time. That is, it does not account for initial device-host memory transfers, GPU context creation, build time, reading/writing data from/to files, etc. Futhark programs are compiled and run with no special compiler flags (the default) (i.e., no fine tuning). Thrust programs are compiled with `nvcc -O3` and `-arch=sm_35` where this did not hurt performance[7], and we use the Git version of Thrust[8] at commit ID `00315ad`, as this was substantially faster than the preinstalled version.

Running times are measured on a a GeForce GTX 780 Ti with 3GiB of global memory and 2880 cores running at 1.08GHz.

We use GCC 4.8.4 and CUDA 6.0 to compile `C++/C` and `OpenCL/CUDA` code (although as mentioned above, we use a newer version of Thrust than the one bundled with CUDA 6.0).

Our experimental setup is publicly available at the following URL:

https://github.com/HIPERFIT/futhark-array16

We believe strongly in the value of reproducible results, and have attempted to automate the reproduction of our experiments. We have also documented common technical problems with running the experiments.

[7] Most of the benchmarks required the `-arch=sm_35` option in order to compile succesfully, but `BlackScholes` and `MSSP` suffered significant slowdown when this option was used. We have no explanation for this. The reported runtimes for these two benchmarks are without the option `-arch=sm_35`.

[8] https://github.com/thrust/thrust

5.2 Mini-Benchmark Description

Figure 8 shows the Futhark code of the nine simple programs we have used for performance evaluation:

1 `ReducePlus` sums up the elements of an array of integers.

2 `ReduceMax` computes the maximum element of an array of integers.

3 `IndexOfMax` computes the index of the maximal element of an integral array. The difference with the previous benchmarks is that the `reduce` operator is defined over index-value *tuples*, rather than basic-type scalars.

4 `IndexOfMaxPack` uses a 64-bit integer to pack together the index and the value into a scalar.

5 `Reduce2x2MM` consists of a loop of count 42, in which the loop-variant integral scalar s is added to each element of an input array, and then the array is reduced with the 2-by-2-matrix-multiplication operator, where a 32-bit integer is assumed to pack the four byte-size elements of the matrix. The result of the `reduce` becomes the value of `s` for the next iteration. Since the reduce operator is not associative, the Thrust version is implemented as a `transform_inclusive_scan`, which fuses together the `map` and the `scan` and selects the last element of the scanned array.

6 `MSSP` is the maximum-segment-sum problem: it looks for the segment of consecutive array elements whose element-wise sum is the largest across all possible segments, and returns the sum. The `reduce` operator is not commutative and operates on four-integer tuples, hence the Thrust implementation uses a `transform_inclusive_scan`.

7 `ScanPlus` is a prefix-sum computation on an array of integers.

8 `RedomapNT` is a code whose structure resembles the one in Figure 4, except that the `mapped` functions are more computationally expensive, e.g., involve division and `exp`. We note that Thrust does not support an operator that can express the fully-fused code, as in Futhark.

9 `BlackScholes` (McDonell et al. 2013) implements a simple pricing engine for European options, and can be represented by one `redomap` in which the `mapped` arrays are consumed by the `reduce` (i.e., no arrays are returned).

5.3 Mini-Benchmark Evaluation

Figure 9 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, $5 \cdot 10^4$, 10^5, $5 \cdot 10^5$, 10^6, $5 \cdot 10^6$, and 10^7. Important observations are:

- `ReducePlus` and `ReduceMax` operates on 32-bit-integer scalars and exhibit very similar behavior: Futhark is about $3\times$ faster on sizes up to one million, and the gap narrows after that, e.g., only $1.55\times$ faster on the largest dataset. `IndexOfMaxPack` operates on 64-bit integers and shows a similar trend: Futhark is about $1.3\times$ 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 $3.4\times$ faster, and the speedup reaches about $14\times$ 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). `ThurstOpt` denotes the version of the code that uses Thrusts built-in, hence optimized, `max_element` operator, but this is still $2\times$ slower than Futhark.
- `MSSP` performs a reduction with a *non-commutative* operator on a four-integer *tuple*. We have implemented this as a `scan` in

```
-- 1. ReducePlus --
--------------------
fun int main([int] a) = reduce(+, 0, a)

-- 2. ReduceMax --
------------------
fun int main([int,n] as)  = reduceComm(max, 0, as)
fun int max(int x, int y) = if x < y then y else x

-- 3. IndexOfMax --
--------------------
fun int main([int,n] as) =
  let {_,i}=reduceComm(maxWithIndex,{0,0},zip(as,iota(n)))
  in i
fun {int,int} maxWithIndex({int,int} x, {int,int} y) =
  let {xv, xi} = x in let {yv, yi} = y
  in if xv < yv then y
     else if yv < xv then x
     else if xi < yi then x else y

-- 4. IndexOfMaxPack --
-----------------------
fun int main([int,n] as) =
  let i = reduceComm(maxWithIndex, 0i64,
                     map(|, zip( map(<<32i64,map(i64,as))
                                 , map(i64,iota(n) ) ) ) )
  in int(i)
fun i64 maxWithIndex(i64 x, i64 y) =
  let {xv, xi} = {int(x >> 32i64) & 0xFFFFFFFF, int(x)}
  let {yv, yi} = {int(y >> 32i64) & 0xFFFFFFFF, int(y)}
  in if xv < yv then y
     else if yv < xv then x
     else if xi < yi then x else y

-- 5. Reduce2x2MM --
--------------------
fun int main([int] a) =
  loop(s = 1) = for i < 42 do
    let a' = map(+s, a)
    in reduce(twoByTwoMult, 0xFF00FF00, a')
  in s

fun int twoByTwoMult(int x, int y) =
  let {x11, x12, x21, x22} = unpackInt(x)
  let {y11, y12, y21, y22} = unpackInt(y)
  let z11 = x11 * y11 + x12 * y21
  let z12 = x11 * y12 + x12 * y22
  let z21 = x21 * y11 + x22 * y21
  let z22 = x21 * y12 + x22 * y22
  in packInt(z11, z12, z21, z22)

fun {i8, i8, i8, i8} unpackInt(int x) =
  {i8(x >>> 24), i8(x >>> 16), i8(x >>> 8), i8(x >>> 0)}
fun int packInt(i8 a, i8 b, i8 c, i8 d) =
  ((i32(a)&0xFF) << 24) | ((i32(b)&0xFF) << 16) |
  ((i32(c)&0xFF) << 8)  | ((i32(d)&0xFF) << 0)

-- 6. MSSP (maximum segment sum problem) --
-------------------------------------------
fun int main([int,n] xs) =
  let {x, _, _, _} =
    reduce(redOp, {0,0,0,0}, map(mapOp, xs)) in
  x
fun {int,int,int,int} redOp({int,int,int,int} x,
                            {int,int,int,int} y) =
  let {mssx, misx, mcsx, tsx} = x in
  let {mssy, misy, mcsy, tsy} = y in
  { max(mssx, max(mssy, mcsx + misy))
  , max(misx, tsx+misy), max(mcsy, mcsx+tsy), tsx + tsy}

fun {int,int,int,int} mapOp (int x) =
  { max(x,0), max(x,0), max(x,0), x}

-- 7. ScanPlus --
-----------------
fun [int] main([int] a) = scan(+, 0, a)

-- 8. RedomapNT: resembles fusion's running example --
-- 9. Black Scholes: not shown --
```

Figure 8: Mini-Benchmark used for evaluating the Redomap construct. `RedomapNT` has a structure similar to the one in Figure 4.

Thrust. On smaller sizes, Thrust version is slightly faster, but Futhark is about $2.5\times$ faster on the largest dataset.

- `Reduce2by2MM` also performs a reduction with non-commutative operator, but on integer scalars. Also, the array being reduced is invariant to the 42-iteration outer loop, which permits Futhark to amortise the cost of the manifested transposition that is required when executing non-commutative reductions. Futhark is about $6.5\times$ faster on an array of size fifty thousand and speedup decreases to only $3.4\times$ on the largest size. This suggests that the Thrust inefficiencies are related to a larger extent with the use of tuples rather than to the use of `scan` (for non-commutative operators).
- In fact, `ScanPlus` shows that Thrust's `scan` operator is up to $2.4\times$ 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 $2.3\times$ faster than the unoptimized Thrust and only $1.1\times$ slower than the optimized Thrust code. This is due to some redundant-copy inefficiencies in our translation, but even so, Futhark would still win if the `map` computations would be heavier (as it is typically the case in real-world applications).
- Finally, on `Black-Scholes`, the unoptimized and optimized Thrust versions correspond to unfused and fully fused code. The latter is possible since the `reduce` consumes the `map`ped arrays, and is implemented in Thrust by the `transform_reduce` and `transform_iterator` operators. Futhark is about $8\times$ 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).

6. Related Work

One strand of related work are domain-specific languages targeting GPGPU execution, such as Nikola (Mainland and Morrisett 2010), Accelerate (McDonell et al. 2013), Obsidian (Claessen et al. 2012), TAIL (Elsman and Dybdal 2014), and SPL (Flænø Werk et al. 2012). 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 (Grelck and Scholz 2006; Grelck and Tang 2014), 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. To our knowledge, none of these approaches support a `redomap` operator that allows to fuse an intermediate array that is both reduced, but also `map`ped into a result array. SAC, however, supports producer-consumer and horizontal fusion, even when the array is explicitly indexed inside the mapped function (Scholz 2003); this is not supported in Futhark.

Another strand of related work is concerned with automatic parallelization of imperative code. Solution include (i) static dependency analyses by polyhedral analysis (Pouchet and et al. 2012; Grosser et al. 2014), and (ii) dynamic extraction of parallelism, for

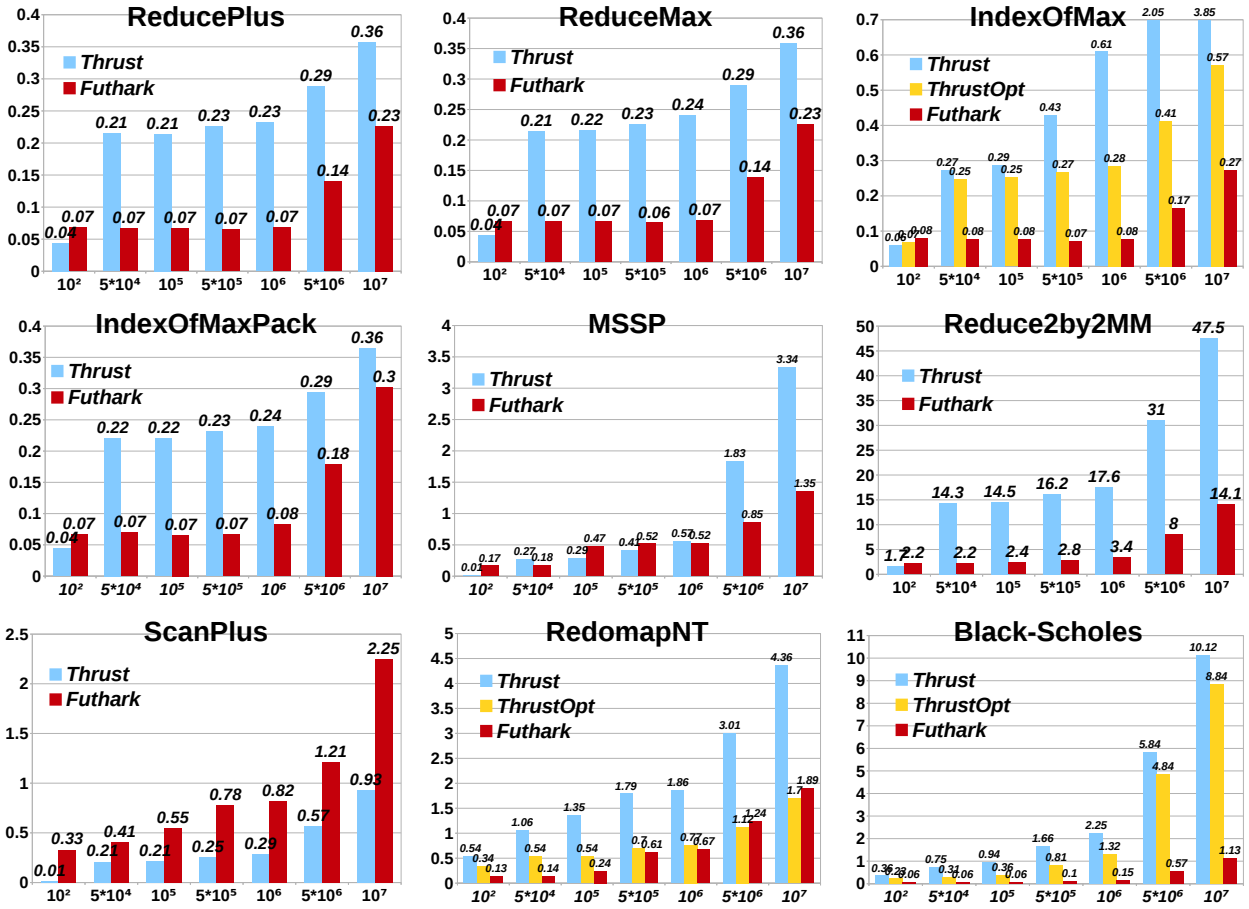


Figure 9: Running Times for Thrust and Futhark on GPGPU. X-Axis is the Array Size; Y-Axis is the runtime (in miliseconds).

example by means of thread-level speculation (Dang et al. 2002; Oancea et al. 2005). 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 (Chicha et al. 2004; Oancea and Watt 2005) or (ii) for efficient integration of option pricing in banks' IT infrastructure (Nord and Laure 2011), or (iii) for integrating a contract-specification language (Bahr et al. 2015) with dynamical graphical user interface (Elsman and Schack-Nielsen 2014). 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 Phython, Java, C++, etc.

7. Conclusions

This paper has presented a new data-parallel core language 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 fully-automatic 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 demonstrated 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

C. Andreetta, V. Bégot, J. Berthold, M. Elsman, F. Henglein, T. Henriksen, M.-B. Nordfang, and C. E. Oancea. Finpar: A parallel financial benchmark. In *ACM TACO*, 2016.

P. Bahr, J. Berthold, and M. Elsman. Certified symbolic management of financial multi-party contracts. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2015.

L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. *SIGPLAN Not.*, 47(9):247–258, Sept. 2012. ISSN 0362-1340. doi: 10.1145/2398856.2364563. URL http://doi.acm.org/10.1145/2398856.2364563.

G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing*, 21(1):4–14, 1994.

Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. Parametric Polymorphism for Computer Algebra Software Components. In *Proc. 6th International*

Symposium on Symbolic and Numeric Algorithms for Scientific Comput., pages 119–130. Mirton Publishing House, 2004.

K. Claessen, M. Sheeran, and B. J. Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Work. on Decl. Aspects of Multicore Prog DAMP*, pages 21–30, 2012.

F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Int. Par. and Distr. Processing Symp. (PDPS)*, pages 20–29, 2002.

M. Elsman and M. Dybdal. Compiling a Subset of APL Into a Typed Intermediate Language. In *Procs. Int. Workshop on Lib. Lang. and Compilers for Array Prog. (ARRAY)*. ACM, 2014.

M. Elsman and A. Schack-Nielsen. Typelets - A Rule-Based Evaluation Model for Dynamic, Statically Typed User Interfaces. In *Int. Symp. on Practical Aspects of Declarative Languages (PADL)*, 2014.

M. Flænø Werk, J. Ahnfelt-Rønne, and K. F. Larsen. An embedded dsl for stochastic processes: Research article. In *Procs. Funct. High-Perf. Comp. (FHPC)*, pages 93–102. ACM, 2012.

C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *Int. Journal of Parallel Programming*, 34(4): 383–427, 2006.

C. Grelck and F. Tang. Towards Hybrid Array Types in SAC. In *7th Workshop on Prg. Lang., (Soft. Eng. Conf.)*, pages 129–145, 2014.

T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544160. URL http://doi.acm.org/10.1145/2544137.2544160.

J. Guo, J. Thiyagalingam, and S.-B. Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *Procs. Workshop Decl. Aspects of Multicore Prog. (DAMP)*, pages 15–24. ACM, 2011.

M. Harris et al. Optimizing parallel reduction in cuda. *NVIDIA Developer Technology*, 2(4), 2007.

T. Henriksen and C. E. Oancea. A t2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 47–58. ACM, 2013.

T. Henriksen, M. Elsman, and C. E. Oancea. Size slicing: a hybrid approach to size inference in futhark. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 31–42. ACM, 2014.

J. Hoberock and N. Bell. Thrust: A parallel template library, 2016. URL http://thrust.github.io/.

G. Mainland and G. Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Int. Symp. on Haskell*, pages 67–78, 2010.

T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *Procs. of Int. Conf. Funct. Prog. (ICFP)*, 2013.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500.

F. Nord and E. Laure. Monte Carlo Option Pricing with Graphics Processing Units. In *Int. Conf. ParCo*, 2011.

C. E. Oancea and S. M. Watt. Domains and Expressions: An Interface between Two Approaches to Computer Algebra. In *Proceedings of the ACM ISSAC 2005*, pages 261–269, 2005. URL http://www.csd.uwo.ca/~coancea/Publications.

C. E. Oancea, J. W. A. Selby, M. Giesbrecht, and S. M. Watt. Distributed Models of Thread-Level Speculation. In *Proceedings of the PDPTA'05*, pages 920–927, 2005. URL http://www.csd.uwo.ca/~coancea/Publications.

L. Pouchet and et al. Loop Transformations: Convexity, Pruning and Optimization. In *Int. Conf. Princ. of Prog. Lang. (POPL)*, 2012.

A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *SIGPLAN Lisp Pointers*, V(1):288–298, Jan. 1992. ISSN 1045-3563.

S.-B. Scholz. Single assignment c: Efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, Nov. 2003. ISSN 0956-7968. doi: 10.1017/S0956796802004458. URL http://dx.doi.org/10.1017/S0956796802004458.