Efficient Translation of Certain Irregular Data-Parallel Array Comprehensions[⋆] (*Extended Abstract*)

Martin Elsman [0000−0002−6061−5993]   Ken Friis Larsen ✉ [0000−0002−0990−5127]

Department of Computer Science, University of Copenhagen, Denmark
`{mael,kflarsen}@diku.dk`

Abstract. We present an array comprehension language that features execution on single-instruction multiple-data (SIMD) architectures even in cases where a comprehension leads to certain kinds of non-regular nested parallelism. The comprehension language naturally supports features such as nesting, zipping, and filtering, and it also supports more advanced features such as sorting. We demonstrate how the language is compiled into efficient Futhark code featuring calls to the Futhark `segmented` library. Moreover, we give a number of examples showing that the array comprehension language provides the programmer with effective abstractions for expressing queries and computations to be executed on SIMD architectures, such as GPUs.

Keywords: List comprehensions · Irregular nested parallelism · Data-parallelism.

1 Introduction

SIMD architectures, such as GPUs, have been known to be notoriously difficult to program as they are exposed to programmers through low-level APIs such as OpenCL [30] and CUDA [27]. Recently, however, a number of research groups have made progress in the development of high-level data-parallel functional language technology with particular focus on targeting SIMD architectures through compiler technology such as fusion [6,8,13,17,26,32], vectorisation and flattening [1,7,21,23,24], and multi-versioning [15,19,29].

Functional languages, on the other hand, are recognised for their state-of-the-art abstraction features, including support for higher-order programming, type polymorphism, advanced module systems, and constructs for controlling and reasoning about computational effects. Whereas some of these techniques carry over to data-parallel functional languages, some techniques are closely tight to

⋆ This research has been partially supported by an Independent Research Fund Denmark grant under the research project *FUTHARK: Functional Technology for High-performance Architectures.*

the concept of computations on lists, a simple inductive data structure embraced by the functional language community for its simplicity and associated straightforward reasoning principles. One such abstraction technique is the notion of *list comprehensions*, a syntactic notation that elegantly allows for expressing complex computations on lists [22,34].

In this paper, we show how comprehensions carry over to data-parallel programming on arrays and how a high-level flattening technique can be used to turn certain kinds of non-regular nested parallelism, which is often introduced by the compilation of array comprehensions, into flat data-parallel code.

The contributions of this paper are the following.

1. We demonstrate how a class of array comprehensions can be compiled into data-parallel computations and, in particular, how a technique for flattening, called flattening-by-expansion, leads to flat data-parallel code.
2. We give a number of examples demonstrating processing and computations on data and examples demonstrating pure computations.
3. We provide evidence that the proposed compilation scheme leads to efficient code through the use of Futhark, a data-parallel functional array language that targets GPUs.

The remainder of this paper is organised as follows. In Sect. 2, we introduce the concept of array comprehensions and give examples of comprehensions with particular emphasis on the motivation for executing non-regular nested comprehensions on parallel architectures. In Sect. 3, we briefly present the Futhark language and describe the flattening-by-expansion technique, which is exposed to the Futhark programmer in the Futhark `segmented` library [11]. In Sect. 4, we present the translation from array comprehensions to Futhark expressions and in Sect. 5, we present a larger example, demonstrating that array comprehensions can be used for writing certain irregular data-parallel algorithms. In Sect. 6, we describe related work and in Sect. 7, we conclude and suggest future work.

2 Motivating Examples

Simple array comprehensions, consisting of an expression e followed by a *binding qualifier* `x` $\leftarrow e'$, where `x` is a variable and e is an array expression, take the form `[` e `|` `x` $\leftarrow e'$ `]`, which is semantically equivalent to a `map` expression of the form `map (`λ`x` $\rightarrow e$`)` e'. Array comprehensions may include multiple binding qualifiers and filtering qualifiers, each of which are separated by commas. To give an example, to compute the first 50 Pythagorean triples,[1] we can use the array comprehension:

```
[ (x,y,z) | x ← [1..<100], y ← [x..<100], z ← [y..<(x+y)],
            z < 100, x*x + y*y == z*z]
```

[1] A Pythagorean triple is a triple of natural numbers x, y, and z that satisfy the properties $x^2 + y^2 = z^2$ and $x \leq y \leq z$.

An expression `[`e`..<`e'`]`, where e and e' are integer expressions, is a Futhark *range expression*, which denotes an array containing the integer values between e (inclusive) and e' (non-inclusive). Notice that binding qualifiers allow the programmer to refer to bound variables in subsequent binding and filter qualifiers.

Traditionally, list comprehensions on the above form [22,34] are translated into nested `map` expressions, but because the nested iteration space is non-regular (i.e., inner `map` expressions are applied to arrays of different sizes), we cannot use this approach for translating into a data-parallel functional language such as Futhark, as Futhark does not support such irregular nested computations.

There are a number of possibilities for solving this issue. First, if we were targeting a data-parallel language supporting full flattening, such as NESL [4,2], non-regular map nests would be translated into fully flattened code in which arrays would be represented as a flattened structure with additional flag arrays specifying row splitting. Unfortunately, the overhead of such an approach can be significant and it is the topic of current research to have a language like NESL provide acceptable performance also in cases that make only limited use of irregular nested parallelism [28]. As a second possibility, we could choose to split non-regular map nests, keep the inner parallelism, and sequentialise the outer parallelism. Such an approach, however, will work well only in cases where there is sufficient inner regular nested parallelism. Yet a possibility would be to use a padding scheme, which, unfortunately could lead to work inefficient code, for which the sequential work complexity is superior to the work complexity for the padded parallel code version.

In this paper, we will instead follow a *flattening-by-expansion* approach [12], which can be implemented as a library functionality and which Futhark exposes through the Futhark `segmented` library. Before presenting this approach, we first introduce the Futhark language.

3 The Futhark Language and Compiler

Futhark is a data-parallel functional array language and compiler aimed at targeting GPUs. It features a number of parallel second-order array combinators (SOACs), including `map`, `reduce`, `filter`, and `scan`, and supports regular nested parallelism and regular multi-dimensional arrays [18]. Futhark provides the programmer with type polymorphism, a restricted notion of higher-order functions [20], and a higher-order module system [10]. These abstraction features are all eliminated at compile time using specialisation techniques, which together with fusion, regular flattening, code versioning, and tiling techniques leave the programmer with efficient executables to be executed on GPUs.[2]

3.1 Flattening by Expansion

The Futhark `segmented` library exposes the following library function:

[2] Futhark also generates fairly efficient code for execution on CPUs.

```
val expand 'a 'b : (a → i32) → (a → i32 → b) → []a → []b
```

The function expands a source array into a target array given (1) a function that determines, for each source element, how many target elements it expands to and (2) a function that computes a particular target element based on a source element and the target element number associated with the source. As an example, the expression `expand (λx → x) (*) [2,3,1]` returns the array `[0,2,0,3,6,0]`. Other examples of use include a work-efficient data-parallel flattened version of Eratosthenes' sieve, a flattened version of sparse matrix-vector multiplication, a way of computing the set of points that make up a line segment and a way of computing the set of line segments making up filled 2D geometric objects, such as circles or triangles [12].

4 Array Comprehension Syntax and Translation

We use z, y, and z to range over program variables. Fig. 1 shows the syntax for expressions and comprehensions, including a construct for zipping (also elsewhere called parallel comprehensions) following [22].

e, f, g	$::=$	$d \mid x \mid (e_1, \cdots, e_n) \mid e.i$	Expressions
	$\mid$	$\texttt{let}\ w = e\ \texttt{in}\ e' \mid \lambda w \to e \mid e_1\ e_2 \mid \ldots$	
	$\mid$	$\texttt{iota}\ e \mid \texttt{map}\ f\ e \mid \texttt{filter}\ f\ e \mid \texttt{zip}\ e\ e'$	
	$\mid$	$\texttt{expand}\ f\ g\ e$	
	$\mid$	$[\ e \mid q\ ]$	Comprehensions
w	$::=$	$x \mid (w_1, \cdots, w_n)$	Patterns
p, q, r	$::=$		Qualifiers
		$w \leftarrow e$	Binding
	$\mid$	e	Filtering
	$\mid$	$p\ ,\ q$	Sequence
	$\mid$	$p \mid q$	Zip

Fig. 1. Syntax of expressions and comprehensions.

The syntax for expressions is a subset of the syntax for Futhark expressions but extended with a syntactic construct for array comprehensions. An expression of the form `iota` e results in an integer array containing the integers from 0 to the integer resulting from computing e (not inclusive). We assume that the syntax for array range expressions, as used in the introduction, have been compiled into expressions using `map` and `iota`.

A zipping comprehension on the form `[` e `|` q_1 `|` q_2 `]` is translated into the expression `map` $(\lambda(w_1,w_2) \to e)$ $(\texttt{zip}\ e_1\ e_2)$, where e_1 and e_2 are the results of translating `[` w_1 `|` q_1 `]` and `[` w_2 `|` q_2 `]`, respectively, and w_1 and w_2 are the patterns containing the variables bound by q_1 and q_2, respectively.

Before we show how comprehensions are translated into data-parallel Futhark expressions, we first define a few helper functions. When w and w' are patterns, the result of *appending* w and w', written $w \oplus w'$, is defined by the following equations:

$$
\begin{array}{rcl}
() \oplus w & = & w \\
w \oplus () & = & w \\
(w_1, \cdots, w_n) \oplus (w'_1, \cdots, w'_m) & = & (w_1, \cdots, w_n, w'_1, \cdots, w'_m) \\
x \oplus (w_1, \cdots, w_n) & = & (x, w_1, \cdots, w_n) \\
(w_1, \cdots, w_n) \oplus x & = & (w_1, \cdots, w_n, x) \\
x \oplus y & = & (x, y)
\end{array}
$$

We use *Exp* to denote the set of expressions and *Pat* to denote the set of patterns. For convenience, we sometimes treat a pattern as an expression, which is always possible as $Pat \subset Exp$.

Qualifiers $\boxed{[\![\ q\ ]\!] : Pat \times Exp \to Pat \times Exp}$

$$
\begin{array}{rcl}
[\![\ w \leftarrow e\ ]\!]\ ((),\ \_) & = & (w,\ e) \\
[\![\ w \leftarrow e\ ]\!]\ (w_0,\ v) & = & (w_0 \oplus w \\
 & & ,\ \texttt{expand}\ (\lambda w_0 \to Size\ e) \\
 & & \qquad (\lambda w_0 \to \lambda x \to \texttt{let}\ y = Fetch\ e\ x \\
 & & \qquad\qquad \texttt{in}\ w_0 \oplus y\ ) \\
 & & \qquad v \\
 & &) \\
[\![\ p\ ,\ q\ ]\!]\ (w,\ v) & = & [\![\ q\ ]\!]\ ([\![\ p\ ]\!]\ (w,\ v)) \\
[\![\ p \mid q\ ]\!]\ (w,\ v) & = & \text{let}\ (w_p, e_p) = [\![\ p\ ]\!]\ (w,\ v) \\
 & & \quad (w_q, e_q) = [\![\ q\ ]\!]\ (w,\ v) \\
 & & \text{in}\ ((\ w_p\ ,\ w_q\ ),\ \texttt{zip}\ e_p\ e_q) \\
[\![\ e\ ]\!]\ ((),\ \_) & = & \texttt{Failure "expecting non-filtering"} \\
[\![\ e\ ]\!]\ (w,\ v) & = & (w,\ \texttt{filter}\ (\lambda w \to e)\ v)
\end{array}
$$

Comprehensions $\boxed{[\![\ \texttt{[}\ e \mid p\ \texttt{]}\ ]\!]_c : Exp}$

$$
\begin{array}{rcl}
[\![\ \texttt{[}\ e \mid p\ \texttt{]}\ ]\!]_c & = & \text{let}\ (w,\ v) = [\![\ p\ ]\!]\ ((),\ ()) \\
 & & \text{in}\ \texttt{map}\ (\lambda w \to e)\ v
\end{array}
$$

Fig. 2. Translating comprehensions.

The translation of comprehensions and qualifiers are given in Fig. 2. Translating a qualifier q takes as arguments, besides q, a pair of a pattern w, defining variables that are in scope in q, and an array expression that defines an array of elements that each matches the pattern w. The translation then returns a pair of a pattern and an array expression that together define the context for possibly further qualifiers.

The translation functions make use of two compile-time functions, $Size : Exp \to Exp$ and $Fetch : Exp \to Exp \to Exp$, which we have not yet defined.

These functions analyse Futhark expression syntax trees. The functions are partial in the sense that they depend on their first argument to have a certain form. In particular, the *Size* function takes an array-computing expression and returns, if possible, an expression representing the size of the given array without constructing the actual array. For instance, if the expression is an expression of the form `iota` e, the *Size* function will return the expression e as this expression represents the size of the array. Definitions of the *Size* and *Fetch* functions are given in Fig. 3.

$$\begin{array}{lcl}
\mathit{Size} : \mathit{Exp} \to \mathit{Exp} & & \\
\mathit{Size}\ (\texttt{iota}\ e) & = & e \\
\mathit{Size}\ (\texttt{map}\ f\ e) & = & f(\mathit{Size}\ e) \\
\mathit{Size}\ \_ & = & \text{undefined}
\end{array}
\qquad
\begin{array}{lcl}
\mathit{Fetch} : \mathit{Exp} \to \mathit{Exp} \to \mathit{Exp} & & \\
\mathit{Fetch}\ (\texttt{iota}\ e)\ e' & = & e' \\
\mathit{Fetch}\ (\texttt{map}\ f\ e)\ e' & = & f(\mathit{Fetch}\ e\ e') \\
\mathit{Fetch}\ \_\ \_ & = & \text{undefined}
\end{array}$$

Fig. 3. Simple versions of the *Size* and *Fetch* functions.

The feature that the *Size* and *Fetch* functions rely on is that it should be possible to determine the size of the array and the i'th element of the array without actually materialising the array, as such a materialisation would result in irregular structures. Notice, however, that we can extend the definitions of *Size* and *Fetch* to query (and index into) arrays that are invariant to the guarding pattern (i.e., w_0 in the translation.)

4.1 The Regular Case and the Very Irregular Case

When the *Size* function returns an expression that is independent of the variables defined in the guarding pattern, the translation can instead generate a simple regular `map`-nest. Futhark is not particularly clever regarding the generated code for the `expand` library function, thus, it does not itself recognise that it sometimes will be possible to replace a call to the `expand` function with a simpler `map`-nest.

As mentioned in the introduction, we could apply a fall-back implementation in the case that *Size* and *Fetch* are undefined on the given input. In these cases, we could choose to sequentialise the outer parallelism and thereby still allow Futhark to compute correct results. We have not yet implemented such a possibility.

5 More Examples

As a more involved example we define an array comprehension that defines the set of points that make up a set of lines. We first give the type definitions for points and lines and a few utility functions for computing the number of points that make up a line, the slope of a line, and a function that computes the i'th point in a line:[3]

[3] A thorough discussion of these definitions are given in [12].

```
type point = (i32,i32)
type line = (point,point)
let points_in_line ((x1,y1),(x2,y2)) =
  i32.(1 + max(abs(x2-x1)) (abs(y2-y1)))
let compare (v1:i32) (v2:i32) : i32 =
  if v2 > v1 then 1 else if v1 > v2 then -1 else 0

let slope (x1,y1) (x2,y2) : f32 =
  if x2 == x1 then if y2 > y1 then 1 else -1
  else r32 (y2-y1) / f32.abs (r32(x2-x1))

let get_point_in_line ((p1,p2):line) (i:i32) : point =
  if i32.abs (p1.1-p2.1) > i32.abs (p1.2-p2.2)
  then let dir = compare p1.1 p2.1
       let sl = slope p1 p2
       in ( p1.1 + dir*i,
            p1.2 + t32 (f32.round (sl*r32 i)))
  else let dir = compare p1.2 p2.2
       let sl = slope (p1.2, p1.1) (p2.2, p2.1)
       in ( p1.1 + t32 ( f32.round (sl*r32 i)),
            p1.2 + i*dir )
```

The code utilises a number of library functions available in the modules `i32` and `f32` and general utility functions for converting between values of basic types (e.g., the function `t32` has type `f32`→`i32`). Another feature is the module local opening syntax; in the expression `i32.(`*e*`)`, identifiers defined by the module `i32` can be accessed unqualified within the expression *e*.

We can now define an array comprehension that computes the array of points that make up a set of lines:

```
let points_of_lines (lines: []line) : []point =
  [ p | line ← lines, i ← [0..<get_points_in_line line],
        p ← get_point_in_line line i ]
```

Notice that the reason the technique works is that the array range computation is a shorthand for an `iota` expression, which can be dealt with by the *Size* and *Fetch* functions, defined in the previous section. The result is a work-efficient data-parallel definition of line drawing.

The other flattening-by-expansion examples defined in [12], in particular, drawing of filled 2D geometric objects, sparse matrix-vector multiplication, and the Sieve of Eratosthenes, can also be expressed as array comprehensions.

6 Related work

Most related to this work is previous work on supporting nested parallelism, including the work on flattening of nested parallelism in NESL [4,3], which was extended to operate on a richer set of values in Data-parallel Haskell [5,7], and the work on data-only flattening [36]. These approaches focus on maximising expressed parallelism, but has proven challenging to implement efficiently in practice, particularly on GPUs [2]. Other promising attempts at compiling NESL to GPUs include Nessie [28], which is still under development, and CuNesl [36], which aims at mapping different levels of nested parallelism to different levels of parallelism on the GPU, but lacks critical optimisations such as fusion.

Other data-parallel languages include Obsidian [32,8,31] and Accelerate [6], which are both embedded in Haskell, and do not feature arbitrary nested parallelism. In Accelerate, programmers may easily write manually flattened programs in the `expand` style, as segmented scans and scatter operations are readily available. We can therefore conjecture that support for array comprehensions in the style suggested here could also be made available in a language such as Accelerate. Accelerate also supports certain forms of irregular arrays by supporting a notion of irregular stream scheduling [9].

Also related to this work is the work on the array language SaC [14], which also supports a kind of comprehension syntax, in particularly, in the context of SaC's `with-loop` construct.

Other attempts at supporting nested irregular parallelism on GPUs are the more dynamic approaches, such as dynamic thread block launching [35] and dynamic parallelism, which are extensions to the GPU execution model involving runtime and micro architecture changes. These approaches to supporting irregular parallelism does, however, often come with a significant overhead [33]. Other dynamic approaches include a partial flattening approach, implemented using thread stealing, which also introduce a significant overhead [21].

7 Conclusion and Future Work

We have shown how the notion of array comprehensions can be used for specifying certain kinds of irregular data-parallel algorithms and that such array comprehensions can be translated into flat data-parallel code using a library approach to flattening.

There are a number of possibilities for future work. First, it would be interesting to investigate the possibility of extending the approach to cover a larger class of irregular nested data-parallel problems. Second, we are currently investigating the possibility of extending the approach to support sorting, which is straightforward using one of Futhark's parallel sorting routines, and group-by constructs [22] using segmented reductions [25]. Moreover, it would be interesting to investigate various optimisations. In particular, one can apply identities involving `expand` to obtain regular map nests when possible. Similarly, for group-by constructs, it should be possible to turn generated segmented reductions into more efficient regular reductions [16].

References

1. Bergstrom, L., Fluet, M., Rainey, M., Reppy, J., Rosen, S., Shaw, A.: Data-only Flattening for Nested Data Parallelism. In: Procs. of the 18th ACM SIGPLAN

Symp. on Principles and Practice of Parallel Programming. pp. 81–92. PPoPP '13, ACM, New York, NY, USA (2013)

2. Bergstrom, L., Reppy, J.: Nested data-parallelism on the GPU. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. pp. 247–258. ICFP '12, ACM, New York, NY, USA (2012)
3. Blelloch, G.E.: Vector models for data-parallel computing, vol. 75. MIT press Cambridge (1990)
4. Blelloch, G.E., Greiner, J.: A provable time and space efficient implementation of NESL. In: Proceedings of the First ACM SIGPLAN Int. Conf. on Functional Programming. pp. 213–225. ICFP '96, ACM, New York, NY, USA (1996)
5. Chakravarty, M., Leshchinskiy, R., Jones, S.P., Keller, G., Marlow, S.: Data Parallel Haskell: A Status Report. In: Int. Work. on Decl. Aspects of Multicore Prog. (DAMP). pp. 10–18 (2007)
6. Chakravarty, M.M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: Proc. of the sixth workshop on Declarative aspects of multicore programming. pp. 3–14. ACM (2011)
7. Chakravarty, M.M., Leshchinskiy, R., Jones, S.P., Keller, G.: Partial vectorisation of haskell programs. In: Proc ACM Workshop on Declarative Aspects of Multicore Programming, San Francisco (2008)
8. Claessen, K., Sheeran, M., Svensson, B.J.: Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In: Work. on Decl. Aspects of Multicore Prog DAMP. pp. 21–30 (2012)
9. Clifton-Everest, R., McDonell, T.L., Chakravarty, M.M.T., Keller, G.: Streaming irregular arrays. In: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. pp. 174–185. Haskell 2017, ACM, New York, NY, USA (2017)
10. Elsman, M., Henriksen, T., Annenkov, D., Oancea, C.E.: Static interpretation of higher-order modules in Futhark: Functional GPU programming in the large. Proceedings of the ACM on Programming Languages **2**(ICFP), 97:1–97:30 (Jul 2018)
11. Elsman, M., Henriksen, T., Oancea, C.E.: Parallel Programming in Futhark. DIKU (November 2018), https://futhark-book.readthedocs.io
12. Elsman, M., Henriksen, T., Serup, N.G.W.: Data-parallel flattening by expansion. In: Proceedings of the 6th ACM SIGPLAN Int. Work. on Libraries, Lang. and Comp. for Array Prog. pp. 14–24. ARRAY '19, ACM, New York, NY, USA (2019)
13. Grelck, C., Hinckfuss, K., Scholz, S.B.: With-loop fusion for data locality and parallelism. In: Proc. of the 17th Int. Conf. on Impl. and Appl. of Functional Languages. pp. 178–195. IFL '05, Springer-Verlag, Berlin, Heidelberg (2006)
14. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. Int. Journal of Parallel Programming **34**(4), 383–427 (2006)
15. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with Lift. In: Procs. of Int. Symp. on Code Gen. and Opt. pp. 100–112. CGO 2018, ACM, New York, NY, USA (2018)
16. Henriksen, T., Larsen, K.F., Oancea, C.E.: Design and GPGPU performance of futhark's redomap construct. In: Proceedings of the 3rd ACM SIGPLAN Int. Work. on Libraries, Languages, and Comp. for Array Prog. pp. 17–24. ARRAY 2016, ACM, New York, NY, USA (2016)
17. Henriksen, T., Oancea, C.E.: A T2 graph-reduction approach to fusion. In: Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing. pp. 47–58. FHPC '13, ACM, New York, NY, USA (2013)
18. Henriksen, T., Serup, N.G.W., Elsman, M., Henglein, F., Oancea, C.E.: Futhark: Purely functional gpu-programming with nested parallelism and in-place array

updates. In: Proceedings of the 38th ACM SIGPLAN Conf. on Prog. Language Design and Impl. pp. 556–571. PLDI 2017, ACM, New York, NY, USA (2017)
19. Henriksen, T., Thorøe, F., Elsman, M., Oancea, C.: Incremental flattening for nested data parallelism. In: Proc. of the 24th Symposium on Principles and Practice of Parallel Prog. pp. 53–67. PPoPP '19, ACM, New York, NY, USA (2019)
20. Hovgaard, A.K., Henriksen, T., Elsman, M.: High-performance defunctionalization in Futhark. In: Symposium on Trends in Functional Programming (TFP'18) (September 2018)
21. Huang, M.H., Yang, W.: Partial flattening: A compilation technique for irregular nested parallelism on GPGPUs. In: Procs. of the 45th Int. Conf. on Parallel Processing. pp. 552–561. ICPP '16 (08 2016)
22. Jones, S.P., Wadler, P.: Comprehensive Comprehensions: Comprehensions with "order by" and "group by". In: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop. pp. 61–72. Haskell '07, ACM, New York, NY, USA (2007)
23. Keller, G., Chakravarty, M.M., Leshchinskiy, R., Lippmeier, B., Peyton Jones, S.: Vectorisation avoidance. In: Proceedings of the 2012 Haskell Symposium. pp. 37–48. Haskell '12, ACM, New York, NY, USA (2012)
24. Keller, G., Simons, M.: A calculational approach to flattening nested data parallelism in functional languages. In: Proceedings of the Second Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security. pp. 234–243. ASIAN '96, Springer-Verlag, Berlin, Heidelberg (1996)
25. Larsen, R.W., Henriksen, T.: Strategies for regular segmented reductions on GPU. In: Proceedings of the 6th ACM SIGPLAN Int. Work. on Functional High-Performance Comp. pp. 42–52. FHPC 2017, ACM, New York, NY, USA (2017)
26. McDonell, T.L., Chakravarty, M.M., Keller, G., Lippmeier, B.: Optimising Purely Functional GPU Programs. In: Procs. of Int. Conf. Funct. Prog. (ICFP) (2013)
27. NVIDIA: CUDA API Reference Manual, 8.0 edn. (Oct 2017), http://docs.nvidia.com/cuda
28. Reppy, J., Sandler, N.: Nessie: A NESL to CUDA compiler. *Compilers for Parallel Comp. Work. (CPC '15)* (Jan 2015), imperial College, London, UK
29. Steuwer, M., Remmelg, T., Dubach, C.: Lift: A Functional Data-parallel IR for High-performance GPU Code Generation. In: Procs. of Int. Symp. on Code Gen. and Opt. pp. 74–85. CGO'17, IEEE Press, Piscataway, NJ, USA (2017)
30. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. IEEE Des. Test **12**(3), 66–73 (May 2010)
31. Svensson, J.: Obsidian: GPU Kernel Programming in Haskell. Ph.D. thesis, Chalmers University of Technology (2011)
32. Svensson, J., Sheeran, M., Claessen, K.: Obsidian: A domain specific embedded language for parallel programming of graphics processors. In: Proc. of the 20th Int. Conf. on Impl. and Appl. of Func. Lang. IFL'08, Springer-Verlag (2011)
33. Tang, X., Pattnaik, A., Jiang, H., Kayiran, O., Jog, A., Pai, S., Ibrahim, M., Kandemir, M., Das, C.: Controlled kernel launch for dynamic parallelism in gpus. In: Proc. of the 23rd Symp. on High Perf. Comp. Arch. HPCA '17, IEEE (5 2017)
34. Wadler, P.: List comprehensions. In: Peyton Jones, S. (ed.) The Implementation of Functional Programming Languages. pp. 127–138. Prentice Hall (January 1987)
35. Wang, J., Rubin, N., Sidelnik, A., Yalamanchili, S.: Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs. In: Procs. of the 42nd Int. Symp. on Comp. Arch. ISCA '15, ACM (2015)
36. Zhang, Y., Mueller, F.: CuNesl: Compiling nested data-parallel languages for SIMT architectures. In: Proc. of the 41st Int. Conf. on Parallel Processing. pp. 340–349. ICPP'12, IEEE, Washington, DC, USA (2012)