Accelerating Nested Data Parallelism: Preserving Regularity

Accelerating Nested Data Parallelism: Preserving Regularity

Lars B. van den Haak[1,2][0000−0002−0330−5016], Trevor L. McDonell[2][0000−0001−7806−9751], Gabriele K. Keller[2][0000−0003−1442−5387], and Ivo Gabe de Wolff[2][0000−0002−4731−2234]

[1] Eindhoven University of Technology l.b.v.d.haak@tue.nl
[2] Utrecht University {t.l.mcdonell,g.k.keller,i.g.dewolff}@uu.nl

Abstract. Irregular nested data-parallelism is a powerful programming model which enables the expression of a large class of parallel algorithms. However, it is notoriously difficult to compile such programs to efficient code for modern parallel architectures. Regular data-parallelism, on the other hand, is much easier to compile to efficient code, but too restricted to express some problems conveniently or in a manner to exploit the full parallelism. We extend the regular data-parallel programming model to allow for the parallel execution of array-level conditionals and iterations over irregular nested structures, and present two novel static analyses to optimise the code generated for these programs which reduces the costs of this more powerful irregular model. We present benchmarks to support our claim that these extensions are effective as well as feasible, as they enable to exploit the full parallelism of an important class of algorithms, and together with our optimisations lead to an improvement in absolute performance over an implementation limited to exploiting only regular parallelism.

1 Introduction

The collection-oriented approach to data-parallel programming, where computations are expressed in terms of higher-order functions—such as maps, folds, and scans—over (multi-dimensional) arrays provides a powerful and convenient programming model. By allowing programmers to identify the parallelism of an algorithm explicitly, yet in an abstract, architecture-independent way, languages such as Futhark [7,11,12,13], Manticore [9], Lift [22], and Accelerate [3,6,16,17] have demonstrated that it is possible to achieve performance comparable to hand-optimised, low-level code on a range of concrete hardware architectures, such as GPUs and multi-core CPUs.

These languages are typically restricted to *regular* data-parallelism: they support executing nested loops in parallel only if the inner loop bounds are independent of the indices of the outer loops, thereby limiting the kinds of parallel algorithms which can be conveniently expressed. While it is possible to transform any *irregular* data-parallel computation into one containing only regular data-parallelism [2], doing so efficiently in practice has still, in general, not

been achieved. Instead, techniques to add limited support for irregular computations to regular data-parallel languages—without compromising performance—are used [6,7]. This work is a further step in this direction. We discuss our approach in the context of the language Accelerate [3], but it applies to any similarly structured language. The main contributions of the paper are:

1. An extension of the regular data-parallel programming model to allow for the parallel execution of array-level conditionals and iterations over irregular nested structures, enabling a larger class of problems using irregular nested parallelism to be executed efficiently (Section 3)
2. A shape analysis to detect at compile time when shapes of nested arrays are equal (Section 4)
3. A program analysis to identify the regular subcomputations of an irregular program (Section 4)
4. Benchmarks demonstrating the effect of the optimisations enabled by the shape equality and regularity detection analyses (Section 5)

We defer the discussion of related work to Section 6.

2 Background

In this section, we give an overview of Accelerate as a representative of the collection-oriented programming model. We discuss the difference between regular and irregular data parallelism, and why the expressiveness of the latter significantly complicates the mapping of the high-level operations to efficient code.

2.1 Accelerate's programming model

The collective operations with which we express parallel computations in Accelerate are based on the scan-vector model [4,21], and consist of multi-dimensional array variants of familiar Haskell list operations such as `map` and `fold`, as well as array-specific operations such as index permutations. In this paper we use a slightly simplified syntax for Accelerate programs for the sake of readability: Accelerate is deeply embedded in Haskell, so the types of expressions are wrapped in a type constructor, which we omit here, as we do with class constraints. For example, to compute the dot product of two arrays, we write:

$$\texttt{dotp} :: \texttt{Array}\ \texttt{DIM}_{n+1}\ \texttt{Float} \rightarrow \texttt{Array}\ \texttt{DIM}_{n+1}\ \texttt{Float} \rightarrow \texttt{Array}\ \texttt{DIM}_{n}\ \texttt{Float}$$
$$\texttt{dotp}\ \texttt{xs}\ \texttt{ys} = \texttt{fold}\ (+)\ 0\ (\texttt{zipWith}\ (*)\ \texttt{xs}\ \texttt{ys})$$

Accelerate is rank-polymorphic, meaning that operations work on arrays of arbitrary rank, or dimensionality. The rank of an array is encoded in the type, denoted by a subscript in our example code. The function `dotp` consumes two multidimensional arrays of rank $n+1$ and produces an array of rank n as output, by folding along the innermost dimension. The type of `fold` is:

```
fold :: (a → a → a) → a → Array DIM_{n+1} a → Array DIM_n a
```

There are two sources of parallelism here: the actual reduction can be done in parallel in $\log n$ steps using a tree fold, and for arrays of rank two and higher, we can do all the tree folds in parallel. In Accelerate, both sources of parallelism are exploited. This is a limited form of nested parallelism—regular nested parallelism—where the size of the inner parallel loop is the same for all iterations.

The `generate` operation is a parallel loop construct which takes as input a shape descriptor of type $\texttt{DIM}_n$ specifying the extent of the resulting array, and a function that will be applied to each index of that shape to compute the value at that index. In the regular data-parallel model, the operation passed to `generate` is restricted to a sequential function returning a single scalar value:

```
generate :: DIM_n → (DIM_n → a) → Array DIM_n a
```

All parallel operations so far have the property that the extent of the output array is independent of the values of the array elements. Unfortunately, there are useful operations for which this is not the case. For example, consider the function `filter`, which removes elements of an array which do not satisfy a given predicate. A rank-polymorphic `filter`, where the output array has the same nesting depth as the input array, requires that the shape of the innermost nesting level is ragged. We use the type $\texttt{IArray DIM}_n\ \texttt{DIM}_m\ \texttt{a}$ for an array of nesting depth $\texttt{n+m}$, where the outermost $\texttt{n}$ dimensions are guaranteed to be regular, and the inner $\texttt{m}$ are potentially irregular. The type of the `filter` operation becomes:

```
filter :: (e → Bool) → Array DIM_{n+1} e → IArray DIM_n DIM_1 e
```

We also have segmented versions of parallel operations, which take irregular arrays as input. For example, the segmented fold calculates the sum of each of the innermost segments of an irregular array in parallel, and has the type:

```
foldSeg :: (a → a → a) → a → IArray DIM_n DIM_{m+1} a → IArray DIM_n DIM_m a
```

Apart from collection-oriented operations, we have an array-level conditional operator $\texttt{cond c es}_1\ \texttt{es}_2$, and an iteration construct $\texttt{while p}_f\ \texttt{b}_f\ \texttt{es}$, which iteratively applies the function $\texttt{b}_f$ to initial array `es` as long $\texttt{p}_f$ applied to the current iteration value is `True`. For example, assume `bubble` is a (potentially) parallel implementation of the inner loop of Bubblesort, and `notSorted` a function which check whether an array is sorted, then we can write:

```
bubbleSort :: Array DIM_1 Float → Array DIM_1 Float
bubbleSort xs = while notSorted bubble xs
```

In this example, the size of the resulting array is the same as the size of the input array, independent of the number of iterations, as `bubble` should not change the size of its input. In general, however, this is not the case. Similarly, the two branches of a conditional do not need to evaluate to arrays of the same shape.

We extend the regular data-parallel programming model by allowing both `cond` and `while` to occur inside of regular nested parallel loops. This generalisation introduces the possibility for this previously regular operation to introduce irregular nested parallelism, but affords the programmer more flexibility to ex-

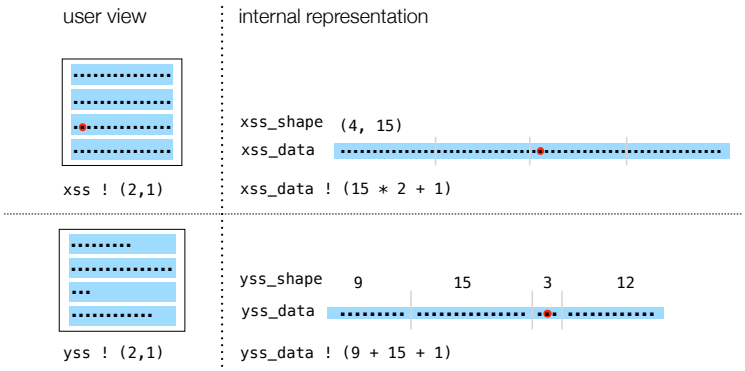

Figure 1: Representation and indexing for regular and irregular arrays

press a larger range of applications. The techniques we present in this paper are aimed at minimising the costs of irregular parallelism arising from these constructs.

Continuing our previous example, if we want to apply the parallel bubble sort program in parallel to a collection of arrays, we can now write:

$$\texttt{bubbleSortAll} :: \texttt{IArray}\ \texttt{DIM}_\texttt{n}\ \texttt{DIM}_1\ \texttt{Float} \rightarrow \texttt{IArray}\ \texttt{DIM}_\texttt{n}\ \texttt{DIM}_1\ \texttt{Float}$$
$$\texttt{bubbleSortAll}\ \texttt{xss} = \texttt{generate}\ (\texttt{extent}\ \texttt{xss})\ (\lambda\texttt{i} \rightarrow \texttt{bubbleSort}\ (\texttt{xss}\,!\,\texttt{i}))$$

where `extent` returns the outer regular shape of an array, which in this example is the number of inner arrays n. Since we know that `bubbleSort` leaves the size of its input unchanged, we also know that `bubbleSortAll` will return an array with the same shape as its input. In particular, if the input array happens to be regular, then the output array will also be regular. The aim of our shape analysis (Section 4.2) is exactly to check whether shapes stay the same. Our regularity detection (Section 4.3) can then use this information to find regular subcomputations.

3 Preserving Regularity

Flat arrays of primitive type are, for the majority of parallel architectures, the most efficient representations, and in case of GPUs, actually the only structure which is supported. Therefore, we need to represent the nested arrays of our source language by flat data arrays with the shape information stored separately.

For regular nested arrays of rank `n`, that is not a problem as they can be represented efficiently by storing the elements in a flat data vector in row-major order, and we can store the size of each dimension as an array of integer values of length n. For example, consider the two arrays `xss` and `yss` in Figure 1. The former is regular, and the shape can be represented compactly, whereas we have to store the size of each segment for the latter, which can incur a significant memory overhead, especially if there are many small or even empty

segments. Operations like indexing into the array are also more expensive for irregular representations. To index into the third subarray in Figure 1, we have to calculate the sum of the sizes of all the preceding subarrays in the irregular case, whereas for the regular, calculating the offset is just a simple multiplication.

Programs manipulating irregular arrays are therefore more expensive, both in terms of the additional memory required to store the size of each segment on every level, as well as the extra processing required to maintain and manipulate the segment descriptor. This is exactly why we want to use the regular array representation whenever possible.

3.1 Statically Determining Regularity

Clifton-Everest [6] added irregular arrays to Accelerate with support for a single level of nested parallelism. In that formulation, the regularity of an expression is evident from the type of the operators used; for example, the `map` operation preserves the regularity of its input and `fold` removes the innermost dimension while preserving the regularity of the outer $n - 1$ dimensions.

We extend that work by adding support for regularity preservation in the presence of nested *control-flow* operators, `cond` and `while`.

Let us go back to our `bubbleSortAll` example (see Figure 2), and replace the function `bubble` with the function `double`, which returns an array twice the size of the input array. If we don't know how many iterations the `while`-loop performs on each subarray, we can't ensure that the resulting nested array is regular. However, if we know that the number of iterations depends only on the size of the input array, then we again know that regularity is preserved as the termination condition function returns the same value for each row of a regular array, even if we do not statically know its exact shape.

The parallel application of the `cond` operation preserves regularity, for example when either all conditionals take the same branch, or the subarrays of the true and false branch have the same shape. In the first case we will end up with one of the two regular branches, thus will stay regular. In the second case, subcomputations may take different branches, but the output shape has the same shape as the two branches and therefore remains regular.

Both parallel `while`-loops and conditionals occur in many applications, so it is worthwhile to detect the cases where their use preserves the regularity of their inputs, and use regular code and array representations in these cases. The next section formalises the analyses we use to detect patterns such as those mentioned above, where regularity is preserved.

4 Program Analyses

The goal of our two program analyses is to identify, at compile time, the regular (sub-)computations of the program, so that the more efficient regular data-parallel operations and array representation can be used for those computations. Our analysis consists of two parts: *regularity detection* generalises vectorisation

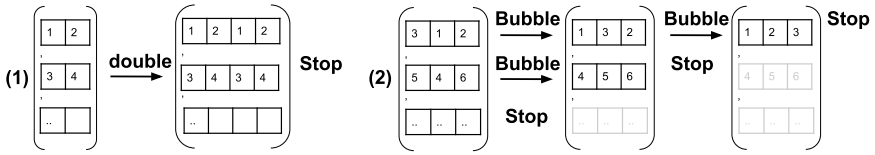

Figure 2: The parallel `while` function preserves regularity if (1) the iteration is applied the same number of times to all subarrays or (2) the iteration function does not alter the shape of its input.

$$
\begin{aligned}
t &::= \texttt{Int} \mid \texttt{Bool} \mid \texttt{(}t, \ldots, t\texttt{)} \\
&\quad \mid \texttt{Array DIM}_l\ t \\
l &::= 0 \mid 1 \mid \ldots \\
b &::= \texttt{True} \mid \texttt{False} \\
c &::= l \mid b \mid [c, \ldots]_{(l,\ldots,l)} \\
p &::= (+) \mid (*) \mid (-) \mid \ldots
\end{aligned}
\qquad
\begin{aligned}
e &::= v \mid c \mid e\ \texttt{!}\ e \mid p\ e\ e \mid (e, \ldots, e) \mid \pi_l(e) \\
&\quad \mid \texttt{let}\ v = e\ \texttt{in}\ e \mid \texttt{extent}\ e \\
&\quad \mid \texttt{generate}\ e\ (\lambda v \to e) \\
&\quad \mid \texttt{fold}\ (\lambda v\ v \to e)\ e\ e \\
&\quad \mid \texttt{cond}\ e\ e\ e \\
&\quad \mid \texttt{while}\ (\lambda v \to e)\ (\lambda v \to e)\ e
\end{aligned}
$$

Listing 1: Grammar of the nested data-parallel core language.

avoidance [15] and identifies sub-expressions that are independent of their surrounding parallel context, using information from our *shape analysis* that determines equivalence of array shapes. We discuss these analyses in the following section in the context of a small nested data-parallel core language.

4.1 Core Language

Listing 1 gives the grammar of the core language which we use to describe our analyses. This language is a generalisation of the core language of Clifton-Everest [6], allowing for arbitrary nesting depth and with the addition of control flow constructs `cond` and `while`. Expressions e consist of variables v; scalar and array constants c (subscript $_{(l,\ldots,l)}$ indicates the exact dimensions); array indexing !; application of primitive operators p; let-expressions; as well as tuples and projections from tuples π_l, where l is the index of an element in the tuple. The operator `extent` returns the outer regular shape of an array; `generate` constructs an array of the given regular shape by applying the function to every index of that shape in data-parallel; `fold` performs a parallel tree-reduction over the inner-most dimension of an array using the supplied binary function and initial element; `cond` and `while` are conditional and iteration constructs as described in Section 2.1. Irregular nested arrays are introduced as array constants, or constructed via `generate`, which—in contrast to Accelerate—does not limit the result type of the generator function to scalar values.

4.2 Shape Analysis

Before we formalise our shape analysis, first consider the following example, which illustrates a common pattern we wish to detect:

```
yss = generate (extent xss)
        (λsh → generate (extent (xss ! sh))
          (λsh' → (xss ! sh) ! sh'+1))
```

This term uses nested applications of `generate` to add one to every element of the array `xss` :: `Array` $\texttt{DIM}_\texttt{N}$ (`Array` $\texttt{DIM}_\texttt{M}$ `Int`). The goal of the analysis is to determine that the shape of `xss` and `yss` are, in fact, identical.

We use the shape analysis in Section 4.3 to identify regular subcomputations, but it can be used to enable other optimisations, such as preventing redundant recomputation of segment descriptors, array recycling, and identifying opportunities to use destructive updates.

Formalisation Our shape analysis proceeds by first building an abstract shape descriptor for every array in the program, and then simplifying these descriptors so that they can be compared for equivalence. We write $ns_1 = ns_2$ to denote the shape equivalence. Note that this comparison is not exact; since we do not have all information available to us at compile time, the equivalence test is necessarily conservative: if two shape descriptors are found to be equal, their associated arrays will definitely have the same shape at runtime, but the reverse is not necessarily true.

Our shape descriptors are constructed using the following grammar:

$$\begin{aligned} s &::= \langle \Sigma_j ; e \rangle \mid \mathit{Folded}\ s \mid \mathit{Outer}\ ns \\ ns &::= \mathbb{S} \mid s \rhd ns \mid (ns, \ldots, ns) \mid \pi_l(ns) \mid \mathit{Inner}\ ns\ \langle \Sigma_j ; e \rangle \mid u_l \end{aligned}$$

A shape s is either an expression e of type $\texttt{DIM}_\texttt{N}$, which may contain free variables bound in environment Σ_j; *Folded* s, which drops the innermost dimension of s; or *Outer* ns, the outermost shape of the nested shape ns. Nested shapes ns are $\mathbb{S}$-terminated lists of s; tuples of nested shapes; the result of projections; or the result of indexing into a shape list with an expression e of type `Int`, thus taking the *Inner* shape. Complex (irregular) shapes or shapes for which we don't have any static information are represented by a unique label u_l.

The judgement $\Sigma_j \vdash e :_S ns$ denotes the derivation of shape descriptor ns for the expression e under environment Σ_j according to the rules in Listing 2. The environment maps every variable v to its shape descriptor (ns) and nesting level (i) of the `generate` whose function bound v. Variables not introduced via a `generate` function—that is, are not a potential source of nested parallelism—have nesting level $\emptyset$. The environment is annotated with an index j, denoting the number of `generate` calls we entered. This index is used as the nesting level of the variable introduced by the next `generate` combinator and we thus increment the index when entering its body.

Returning to our initial example, assuming environment $\Sigma = [\texttt{xss} : (u_0, \emptyset)]$ containing only the array `xss`, about which we have no static information, we

$$\frac{v : (ns, i) \in \Sigma_j}{\Sigma_j \vdash v :_S ns}\ [\textsc{Var}] \qquad \frac{\Sigma_j \vdash e :_S ns}{\Sigma_j \vdash e\ !\ e_{ix} :_S \mathit{Inner}\ ns\ \langle \Sigma; e_{ix} \rangle}\ [\textsc{Index}]$$

$$\frac{e \text{ has a scalar type}}{\Sigma_j \vdash e :_S \mathbb{S}}\ [\textsc{Scalar}] \qquad \frac{\Sigma_j \vdash \overline{e} :_S \overline{ns}}{\Sigma_j \vdash (\ \overline{e}\ ) :_S (\overline{ns})}\ [\textsc{Tuple}]$$

$$\frac{\Sigma_j \vdash \overline{c} :_S ns}{\Sigma_j \vdash [\ \overline{c}\ ]_{\overline{l}} :_S \langle \Sigma_j; (\ \overline{l}\ ) \rangle \triangleright ns}\ [\textsc{Const-Shape}] \qquad \frac{\Sigma_j \vdash e :_S ns}{\Sigma_j \vdash \pi_l(e) :_S \pi_l(ns)}\ [\textsc{Project}]$$

$$\frac{\Sigma_j \vdash e_1 :_S ns_1 \quad \Sigma_j, v : (ns_1, \emptyset) \vdash e_2 :_S ns_2}{\Sigma_j \vdash \texttt{let}\ v = e_1\ \texttt{in}\ e_2 :_S ns_2}\ [\textsc{Let}]$$

$$\frac{\Sigma_{j+1}, v : (\mathbb{S}, j) \vdash e_2 :_S ns}{\Sigma_j \vdash \texttt{generate}\ e_1\ (\lambda v \to e_2) :_S \langle \Sigma_j; e_1 \rangle \triangleright ns}\ [\textsc{Generate}]$$

$$\frac{\Sigma_j \vdash e_3 :_S ns \quad ns' = \mathit{Folded}\ (\mathit{Outer}\ ns) \triangleright \mathbb{S}}{\Sigma_j \vdash \texttt{fold}\ (\lambda\ v_0\ v_1 \to e_1)\ e_2\ e_3 :_S ns'}\ [\textsc{Fold}]$$

$$\frac{\Sigma_j \vdash e_2 :_S ns_2 \quad \Sigma_j \vdash e_3 :_S ns_3 \quad ns_2 = ns_3}{\Sigma_j \vdash \texttt{cond}\ e_1\ e_2\ e_3 :_S ns_2}\ [\textsc{Cond-shape}]$$

$$\frac{\Sigma_j \vdash e_3 :_S ns_3 \quad \Sigma_j, v : (ns_3, \emptyset) \vdash e_2 :_S ns_2 \quad ns_2 = ns_3}{\Sigma_j \vdash \texttt{while}\ (\lambda\ v.\ e_1)\ (\lambda\ v.\ e_2)\ e_3 :_S ns_3}\ [\textsc{While-shape}]$$

$$\frac{l \text{ is a fresh label}}{\Sigma_j \vdash e :_S u_l}\ [\textsc{Fallback}]$$

Listing 2: Inference rules of shape analysis.

can then deduce:

$$\Sigma_0 \vdash \texttt{xss} :_S u_0$$
$$\Sigma_0 \vdash \texttt{yss} :_S (\langle \Sigma_0; \texttt{extent xss} \rangle \triangleright \langle \Sigma_1, sh : (\mathbb{S}, 0); \texttt{extent (xss!sh)} \rangle \triangleright \mathbb{S}$$

where we apply the [Generate] rule twice and subsequently the [Scalar] rule.

Although the two shape descriptors for `xss` and `yss` are equivalent, they are not syntactically equal. We thus introduce shape equivalence, denoted by $ns_1 = ns_2$, which compares shape descriptors after partially evaluating the shape descriptors, for example by applying projections to tuples. Furthermore we simplify certain patterns which we found to occur frequently. Note that other domain-specific simplification rules may also be possible. The following steps can always be applied to a single shape descriptor:

S1. $\langle \Sigma_j; \texttt{extent e} \rangle$: we apply shape analysis on e, $\Sigma_j \vdash e :_S ns$, and take the outer shape as a result, $\mathit{Outer}\ ns$. This is exactly the semantics of `extent`.
S2. $\mathit{Outer}\ ns \triangleright \mathbb{S}$: if ns is not nested, which can be determined from type information, it simplifies to ns.
S3. $\mathit{Outer}\ (s \triangleright ns)$ simplifies to s.
S4. $\mathit{Outer}\ ns_1 \triangleright \mathit{Inner}\ ns_2\ \langle \Sigma_j; v \rangle$: this pattern arises from nested `generate`s, e.g. `generate (extent xss) (λv → generate (xss ! v) e)`. If $ns_1 = ns_2$ and v's nesting level matches the nesting depth of the shape, the shape descriptor

simplifies to ns_1. The nesting depth of the shape denotes how many $\triangleright$ are in front of the shape in the $\triangleright$-separated list. For instance, in $s' \triangleright s \triangleright ns$, the whole shape has a nesting depth of 0, $s \triangleright ns$ has depth 1, and ns has 2.

After the simplification, when two shapes are compared, we check on syntactical equivalence. However, when comparing shape expressions we have a few more equivalence rules:

E1. $\langle \Sigma; e \rangle$: any variables inside `e` that were introduced by a `generate` only have to match by their nesting level. This is stored in the shape environment Σ_j.
E2. $\langle \Sigma_j; e \rangle$: when we encounter `extent` $\mathtt{e}_1$ (as a subexpression) in `e`, we apply shape analysis on $\mathtt{e}_1$.

Using these simplification rules, the shape descriptor `yss` can be rewritten to be equal to the shape descriptor of `xss` in the following steps:

$$= \mathit{Outer}\ u_0 \triangleright \mathit{Outer}\ (\mathit{Inner}\ u_0\ \langle \Sigma, sh : (\mathbb{S}, 0); sh \rangle) \triangleright \mathbb{S} \qquad \texttt{(S1, S1)}$$
$$= \mathit{Outer}\ u_0 \triangleright \mathit{Inner}\ u_0\ \langle \Sigma, sh : (\mathbb{S}, 0); sh \rangle \qquad \texttt{(S2)}$$
$$= u_0 \qquad \texttt{(S4)}$$

The shape analysis can be parameterised by which simplification rules to apply; it depends on the application context of the shape analysis which rules are worthwhile. One additional rule which we do use is inlining of all let-bound variables in expressions, for which we have another environment containing the definitions for all variables which are in scope.

4.3 Regularity Detection

Regularity detection identifies (sub-)expressions which are either constant or produce regular parallelism with respect to the surrounding parallel context. For example, if we `map` the function λ`x` $\rightarrow$ `x+(6*7)` over an array in parallel, then the value of `x` depends on the parallel context, but the expression `6*7` is constant with respect to that context.

Keller [15] provide an algorithm to identify these subexpressions in the presence of arbitrarily nested contexts, however that work does not take regularity information into account, and therefore misses important optimisation opportunities. Take for example the term λ`sh` $\rightarrow$ `extent (xss ! sh)`; if `xss` is a regular nested structure this function returns the same result for all values `sh`, so the term as a whole is constant even though it depends on the parallel context. In the remainder of the section we formalise our generalisation of the vectorisation avoidance [15] algorithm to take this information into account.

Formalisation Listing 3a presents the grammar for the analysis, where regularity information is stored as a triple d, with i denoting whether the full term is totally independent ($\top$) or not ($\bot$); n records for each nesting level whether it is regular (R) or irregular (Ir); and k tracks the nesting level of the variables introduced by `generate` (only `generate` can introduce nested computations which

$$
\begin{aligned}
d &::= \langle n, i, k\rangle \mid (d, \ldots, d)\\
n &::= \mathbb{S} \mid r \rhd n\\
r &::= R \mid Ir\\
i &::= \top \mid \bot\\
k &::= \infty \mid l
\end{aligned}
$$

(a) Grammar

$$
\begin{aligned}
ir(\langle n, i, l\rangle) &= \langle ir_n(n), i, k\rangle\\
ir(d_1, \ldots, d_n) &= (ir(d_1), \ldots, ir(d_n))\\
ir_n(\_ \rhd n) &= Ir \rhd ir_n(n)\\
ir_n(\mathbb{S}) &= \mathbb{S}
\end{aligned}
$$

(b) The ir helper function

$$
\begin{aligned}
\top \wedge \top &= \top\\
\_ \wedge \_ &= \bot\\
r_1 \rhd n_1 \wedge r_2 \rhd n_2 &= r_1 \wedge r_2 \rhd n_1 \wedge n_2\\
\mathbb{S} \wedge n &= n\\
n \wedge \mathbb{S} &= n\\
R \wedge R &= R\\
\_ \wedge \_ &= Ir\\
\langle n_1, i_1, l_1\rangle \wedge \langle n_2, i_2, l_2\rangle &= \langle n_1 \wedge n_2, i_1 \wedge i_2,\\
&\qquad \min(l_1, l_2)\rangle\\
(d_1, \ldots, d_n) \wedge (d'_1, \ldots, d'_n) &= (d_1 \wedge d'_1, \ldots, d_n \wedge d'_n)\\
(d_1, \ldots, d_n) \wedge d &= (d_1 \wedge d, \ldots, d_n \wedge d)\\
d \wedge (d_1, \ldots, d_n) &= (d_1 \wedge d, \ldots, d_n \wedge d)
\end{aligned}
$$

(c) Lattice definitions

Listing 3: The annotation used for regularity detection.

are dependent on the parallel context). Merging of regularity information is done via the operator $\wedge$ given in Listing 3c. The analysis uses the results of shape analysis and thus passes around a shape environment Σ_j besides the regularity environment Γ, mapping variables to their regularity. The judgement $\Gamma; \Sigma_j \vdash e :_R d$ denotes that expression e has regularity d under environments Σ_j and Γ. We present the rules of regularity detection in Listings 4 and 5, where we use $\Gamma; \Sigma_j \vdash e : (d, ns)$ to denote the results of both analyses:

$$\frac{\Gamma; \Sigma_j \vdash e :_R d \quad \Sigma_j \vdash e :_S ns}{\Gamma; \Sigma_j \vdash e : (d, ns)} \tag{1}$$

The rules for `cond` and `while` must check whether the shapes are respectively fixed (rules [...-Shape]), independent ([...-Indep]), or whether we must assume that they may be irregular ([...-Irr]). Rule [Extent-Regular] checks whether the argument array is regular, in which case it always returns the same extent and is therefore independent. We have three rules for `generate`, rule [Generate-1] checks whether the nesting level k of the function is greater than or equal to the current level; if so the function is independent of any outer `generate` operations. Rule [Generate-2] checks whether the outer shape of the `generate` is independent, meaning the operation as a whole is regular, and [Generate-3] is the fallback case.

We want to conclude with a more interesting example, we modified our previous example from Section 4.2 to contain a conditional.

```
yss = generate (extent xss)
        (λsh → let c = ((xss ! sh) ! 0)>10 in
```

$$\frac{}{\Gamma;\Sigma_j \vdash l :_R \langle \mathbb{S}, \top, \infty \rangle}\ [\textsc{Literal}] \qquad \frac{\Gamma;\Sigma_j \vdash e :_R (d_1, \ldots, d_l, \ldots, d_n)}{\Gamma;\Sigma_j \vdash \pi_l(e) :_R d_l}\ [\textsc{Project}]$$

$$\frac{}{\Gamma;\Sigma_j \vdash b :_R \langle \mathbb{S}, \top, \infty \rangle}\ [\textsc{Bool}] \qquad \Gamma;\Sigma_j \vdash e :_R \langle \_ \rhd n, i, k \rangle$$

$$\frac{\Gamma;\Sigma_j \vdash \overline{e} :_R \overline{d}}{\Gamma;\Sigma_j \vdash (\ \overline{e}\ ) :_R (\ \overline{d}\ )}\ [\textsc{Tuple}] \qquad \frac{\Gamma;\Sigma_j \vdash e\ !\ e_{ix} :_R \langle n, i, k \rangle \wedge d}{\Gamma;\Sigma_j \vdash e\ !\ e_{ix} :_R \langle n, i, k \rangle \wedge d}\ [\textsc{Index}]$$

$$\frac{\Gamma;\Sigma_j \vdash e_1 : (d_1, ns_1) \quad \Gamma, v : d_1; \Sigma_j, v : (ns_1, \emptyset) \vdash e_2 :_R d_2}{\Gamma;\Sigma_j \vdash \texttt{let}\ v = e_1\ \texttt{in}\ e_2 :_R d_2}\ [\textsc{Let}] \qquad \frac{\Gamma;\Sigma_j \vdash e_1 :_R d_1 \quad \Gamma;\Sigma_j \vdash e_2 :_R d_2}{\Gamma;\Sigma_j \vdash p\ e_1\ e_2 :_R d_1 \wedge d_2}\ [\textsc{Op}]$$

$$\frac{v : d \in \Gamma}{\Gamma;\Sigma_j \vdash v :_R d}\ [\textsc{Var}]$$

$$\frac{\Gamma;\Sigma_j \vdash \overline{c} : (\langle n, \top, \infty \rangle, ns)}{\Gamma;\Sigma_j \vdash [\ \overline{c}\ ]_{\overline{l}} :_R \langle R \rhd n, \top, \infty \rangle}\ [\textsc{Const-Regular}]$$

$$\frac{\Gamma;\Sigma_j \vdash \overline{c} :_R \overline{d} \quad \langle n, \top, \infty \rangle = \bigwedge \overline{d}}{\Gamma;\Sigma_j \vdash [\ \overline{c}\ ]_{\overline{l}} :_R \langle R \rhd ir(n), \top, \infty \rangle}\ [\textsc{Const-Irr}]$$

$$\frac{\Gamma;\Sigma_j \vdash e :_R \langle R \rhd \_, \_, \_ \rangle}{\Gamma;\Sigma_j \vdash \texttt{extent}\ e :_R \langle \mathbb{S}, \top, \infty \rangle}\ [\textsc{Extent-Regular}]$$

$$\frac{\Gamma;\Sigma_j \vdash e :_R \langle Ir \rhd \_, i, k \rangle}{\Gamma;\Sigma_j \vdash \texttt{extent}\ e :_R \langle \mathbb{S}, i, k \rangle}\ [\textsc{Extent-Irr}]$$

$$\frac{\Gamma;\Sigma_j \vdash e_1 :_R d_1 \quad \Gamma;\Sigma_j \vdash e_2 : (d_2, ns_2) \quad ns_2 = ns_3 \quad \Gamma;\Sigma_j \vdash e_3 : (d_3, ns_3)}{\Gamma;\Sigma_j \vdash \texttt{cond}\ e_1\ e_2\ e_3 :_R d_1 \wedge d_2 \wedge d_3}\ [\textsc{Cond-Shape}]$$

$$\frac{\Gamma;\Sigma_j \vdash e_1 :_R \langle \mathbb{S}, \top, k \rangle \quad \Gamma;\Sigma_j \vdash e_2 :_R d_2 \quad d = d_2 \wedge d_3 \quad \Gamma;\Sigma_j \vdash e_3 :_R d_3}{\Gamma;\Sigma_j \vdash \texttt{cond}\ e_1\ e_2\ e_3 :_R \langle \mathbb{S}, \top, k \rangle \wedge d}\ [\textsc{Cond-Indep}]$$

$$\frac{\Gamma;\Sigma_j \vdash e_1 :_R d_1 \quad \Gamma;\Sigma_j \vdash e_2 :_R d_2 \quad \Gamma;\Sigma_j \vdash e_3 :_R d_3}{\Gamma;\Sigma_j \vdash \texttt{cond}\ e_1\ e_2\ e_3 :_R d_1 \wedge ir(d_2 \wedge d_3)}\ [\textsc{Cond-Irr}]$$

$$\frac{\Gamma;\Sigma_j \vdash e_3 : (d_3, ns_3) \quad \Gamma, v : d_3; \Sigma_j, v : (ns_3, \emptyset) \vdash e_2 : (d_2, ns_2) \quad ns_2 = ns_3 \quad \Gamma, v : (d_2 \wedge d_3); \Sigma_j, v : (ns_3, \emptyset) \vdash e_1 :_R d_1}{\Gamma;\Sigma_j \vdash \texttt{while}\ (\lambda\ v.\ e_1)\ (\lambda\ v.\ e_2)\ e_3 :_R d_1 \wedge d_2 \wedge d_3}\ [\textsc{While-Shape}]$$

$$\frac{\Gamma, v : d_3; \Sigma_j, v : (ns_3, \emptyset) \vdash e_2 : (d_2, ns_2) \quad \Gamma;\Sigma_j \vdash e_3 : (d_3, ns_3) \quad \Gamma, v : (d_2 \wedge d_3); \Sigma_j, v : (ns_2, \emptyset) \vdash e_1 :_R \langle \mathbb{S}, \top, k \rangle}{\Gamma;\Sigma_j \vdash \texttt{while}\ (\lambda\ v.\ e_1)\ (\lambda\ v.\ e_2)\ e_3 :_R \langle \mathbb{S}, \top, k \rangle \wedge d_2 \wedge d_3}\ [\textsc{While-Indep}]$$

$$\frac{\Gamma, v : d_3; \Sigma_j, v : (ns_3, \emptyset) \vdash e_2 : (d_2, ns_2) \quad \Gamma;\Sigma_j \vdash e_3 : (d_3, ns_3) \quad \Gamma, v : (d_2 \wedge d_3); \Sigma_j, v : (ns_3, \emptyset) \vdash e_1 :_R d_1}{\Gamma;\Sigma_j \vdash \texttt{while}\ (\lambda\ v.\ e_1)\ (\lambda\ v.\ e_2)\ e_3 :_R d_1 \wedge ir(d_2 \wedge d_3)}\ [\textsc{While-Irr}]$$

Listing 4: First set of inference rules for regularity detection. (1/2)

$$\frac{\begin{array}{l}\Gamma;\Sigma_j \vdash e_1 :_R \langle \_, \top, k_1\rangle \\ \Gamma, v : \langle \mathbb{S}, \bot, j\rangle; \Sigma_{j+1}, v : (\mathbb{S}, j) \vdash e_2 :_R \langle n, \_, k_2\rangle \quad k_2 \geqslant j\end{array}}{\Gamma;\Sigma_j \vdash \texttt{generate}\ e_1\ (\lambda v \to e_2) :_R \langle R \rhd n, \top, \min(k_1, k_2)\rangle}\ [\textsc{Generate-1}]$$

$$\frac{\Gamma;\Sigma_j \vdash e_1 :_R \langle \_, \top, k_1\rangle \quad \Gamma, v : \langle \mathbb{S}, \bot, j\rangle; \Sigma_{j+1}, v : (\mathbb{S}, j) \vdash e_2 :_R \langle n, i, k_2\rangle}{\Gamma;\Sigma_j \vdash \texttt{generate}\ e_1\ (\lambda v \to e_2) :_R \langle R \rhd n, i, \min(k_1, k_2)\rangle}\ [\textsc{Generate-2}]$$

$$\frac{\Gamma;\Sigma_j \vdash e_1 :_R \langle \_, \bot, k_1\rangle \quad \Gamma, v : \langle \mathbb{S}, \bot, j\rangle; \Sigma_{j+1}, v : (\mathbb{S}, j) \vdash e_2 :_R \langle n, \_, k_2\rangle}{\Gamma;\Sigma_j \vdash \texttt{generate}\ e_1\ (\lambda v \to e_2) :_R \langle Ir \rhd n, \bot, \min(k_1, k_2)\rangle}\ [\textsc{Generate-3}]$$

$$\frac{\begin{array}{l}\Gamma;\Sigma_j \vdash e_2 :_R d_2 \quad \Gamma;\Sigma_j \vdash e_3 :_R d_3 \\ d = d_2 \wedge d_3 \qquad \Gamma, v_0 : d, v_1 : d; \Sigma_j, v_0 : (\mathbb{S}, \emptyset), v_1 : (\mathbb{S}, \emptyset) \vdash e_1 :_R d_1\end{array}}{\Gamma;\Sigma_j \vdash \texttt{fold}\ (\lambda\ v_0\ v_1 \to e_1)\ e_2\ e_3 :_R d_1 \wedge d_3}\ [\textsc{Fold}]$$

Listing 5: Second set of inference rules for regularity detection. (2/2)

```
let t = xss ! sh in
let e = generate (extent (xss ! sh))
          (λsh' → (xss ! sh) ! sh'+1)
in cond c t e)
```

The shape analysis can detect that `yss` has the same shape as `xss`, but it will also show that `t` and `e` have the same shape. Suppose we know that `xss` is a regular nested array. We show that the regularity detection detects that the sub-computations stay regular. Formally, we now have the environment $\Gamma;\Sigma = [\texttt{xss} : \langle R \rhd R \rhd \mathbb{S}, \top, \infty\rangle, \texttt{sh} : \langle \mathbb{S}, \bot, 0\rangle]; [\texttt{xss} : (u_0, \emptyset), \texttt{sh} : (\mathbb{S}, 0)]$, where we added the variable `sh` introduced by the outer generate. Let us inspect the result of `c`, `t` and `e`, which we need for `cond c t e`.

$$\Gamma;\Sigma_1 \vdash \texttt{c} :_R \langle \mathbb{S}, \bot, 0\rangle$$
$$\Gamma, c : ...; \Sigma_1, c : ... \vdash \texttt{t} :_R \langle R \rhd \mathbb{S}, \bot, 0\rangle$$
$$\Gamma, c : ..., t : ...; \Sigma_1, c : ..., t : ... \vdash \texttt{e} :_R \langle R \rhd \mathbb{S}, \bot, 0\rangle$$

The results of `c` and `t` are a simple application of a combination of the [Index], [Op], [Var] and [Literal] rules and you can view them as a scalar and a regular array respectively. Both are also dependent ($\bot$) on the outer generate (0). The result of `e` is the same as `t`, but the [Extent-Regular] [Generate-2] rules are used.

With the above results and the fact that `t` and `e` have the same shape, we use the [Cond-Shape] rule to get:

$$\Gamma, ...; \Sigma_1, ... \vdash \texttt{cond c t e} :_R \langle R \rhd \mathbb{S}, \bot, 0\rangle$$

Thus the body of the `generate` has sub-computations that are dependent, but regular. Finally, using [Generate-1] on the outer generate, gets us that `yss` is a nested regular array that is independent of any other parallel context.

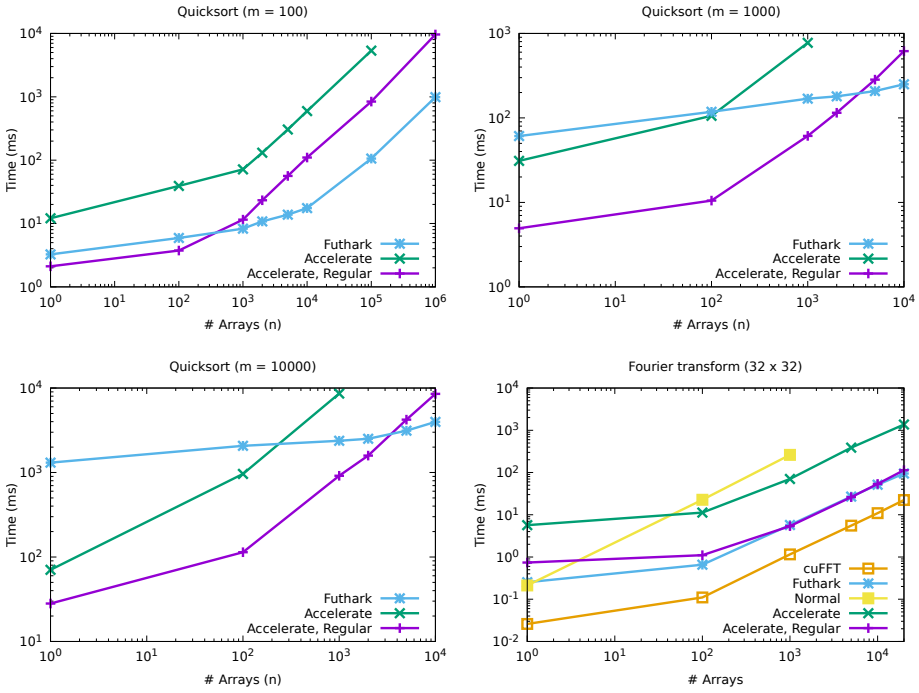

Figure 3: Weak scaling of benchmark programs. The results of this work are shown in purple (Accelerate, Regular), compared to unoptimised Accelerate programs in green (Accelerate).

5 Evaluation

The objective of this work is to extend the data-parallel programming model to efficiently execute array-level conditionals and iterations over irregularly nested structures. In this section we evaluate the effectiveness of our work through a number of benchmarks. Our benchmarks are conducted using a GeForce RTX 2080 Ti (compute capability 7.0, 68 multiprocessors = 4352 cores at 1.65GHz, 11GB GDDR6) backed by a 16-core Threadripper 2950X (1.9GHz, 64GB RAM, hyperthreading is enabled) running GNU/Linux (Ubuntu 19.10). We used GHC-8.6.3, LLVM-9, and CUDA-10.1.

Our implementation in the deeply embedded language Accelerate means that the analyses presented here, as well as all other compiler stages such as optimisation and code generation, occur during the runtime of the host language program. In order to focus on the effectiveness of the optimisations presented in this paper, which are generally applicable and not related to our specific implementation, we report total kernel execution time on the GPU including memory transfer overhead rather than overall application runtime.

5.1 Quicksort

To evaluate the overhead of irregularity we use a loop-based implementation of the Quicksort [14] algorithm,[3] which is representative of irregular divide-and-conquer algorithms that require both the intra- and inter-routine parallelism to achieve optimal parallel work complexity. This benchmark is chosen because it has minimal computation: the runtime of the algorithm is entirely dominated by data movement, so the cost of managing segment descriptors for irregular arrays cannot be hidden.

The benchmark sorts each row of an $n \times m$ matrix in parallel, so each row of the matrix iterates a different number of times over its subarray. Our analysis detects that the iteration leaves the shapes of the subarrays unchanged so can be optimised as regular nested parallelism. The results are shown in Figure 3, showing that our optimised version is 6 to 13 times faster than the unoptimised Accelerate program.

Futhark uses a different method to support nested parallelism [13]. For small arrays the GPU is not fully utilised, but for larger arrays their approach has lower overhead and overtakes our implementation. The incremental flattening approach of Futhark is orthogonal to this work, so it would be possible to utilise both approaches simultaneously.

5.2 Fast Fourier Transform (FFT)

We benchmark three versions of the Split-Radix FFT algorithm in Accelerate: *normal*, where we do not exploit the nested parallelism; *regular* with nested parallelism and optimisations switched on; and *irregular*, a nested parallel implementation without optimisation. The Split-Radix FFT algorithm consists of an outer `while` loop which operates over successively smaller arrays. Futhark does not support irregular nested parallelism and their algorithm can not detect that the computation is regular, so their compiler is unable to compile this program. We instead benchmark the Stockham algorithm in Futhark, which it is able to compile.

Figure 3 shows the results of execution a number of 32×32 Fourier transforms. As a baseline we also compare against the highly optimised cuFFT library, which we call via Accelerate's foreign function interface [5]. The unoptimised *irregular* code is more than an order of magnitude slower than our optimised *regular* implementation. The program *normal* is unable to execute multiple FFTs in parallel and thus performs poorly at large array sizes—even compared to the unoptimised *irregular* implementation—as it does not expose enough parallelism to properly utilise the GPU.

[3] `https://github.com/AccelerateHS/accelerate-examples/tree/master/examples/quicksort`

6 Related Work

Languages like NESL [2] and Data Parallel Haskell [18] support fully irregular nested data parallelism, but they struggle to achieve good performance. Nessie [20] is ongoing work on a NESL compiler which targets GPUs. Manticore [9] also supports irregular nested data parallel computations on CPU multicores by flattening the data structures [1], but not the parallel computations.

To control excessive parallelism due to regular nested parallelism, *incremental flattening* [13] executes the inner parallelism sequentially of a nested computation in some circumstances, which allows for better use of shared memory in GPUs. More recently Futhark added more support for a certain kind of irregular nested parallelism [7], but this has not been integrated into the backend yet [8]. Futhark also performs shape analysis [11] to symbolically determine the exact shape of arrays if possible but switches to dynamic handling if necessary. Our analysis aims to compare shapes, not determine them exactly, so it can be done completely statically. We can capture some irregular structures of arrays, whereas Futhark only works with regular structures.

Other data parallel languages, like Halide [19], Obsidian [23], Lift [22] and SaC [10] all aim at producing high performing code for CPUs and/or GPUs. We believe they could benefit from the work presented here, in the implementation of irregular nested data parallelism or to allow more programs which expose regular subcomputations.

7 Conclusion

We presented two analyses for irregular nested parallel array languages, and demonstrated how this analyses can be used to identify and specialise code for regular sub-computations within nested irregular computations. We extended the Accelerate language with two constructs to enable expressing a limited form of irregular nested parallelism, together with our regularity optimisations, and provide benchmarks demonstrating the effect of these optimisations. Our work is open source and available at `https://github.com/sakehl/accelerate/tree/feature/sequences`.

Acknowledgements We would like to thank the reviewers for their detailed feedback and suggestions also with respect to possible future work directions, and Manuel Chakravarty for his comments on an earlier version of this paper.

References

1. Bergstrom, L., Fluet, M., Rainey, M., Reppy, J., Rosen, S., Shaw, A.: Data-only flattening for nested data parallelism. In: Principles and Practice of Parallel Programming (2013)
2. Blelloch, G.E., Sabot, G.W.: Compiling collection-oriented languages onto massively parallel computers. Parallel and Distributed Computing **8**(2), 119–134 (1990)

3. Chakravarty, M.M.T., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating haskell array codes with multicore GPUs. In: Declarative Aspects of Multicore Programming (2011)
4. Chatterjee, S., Blelloch, G.E., Zagha, M.: Scan primitives for vector computers. In: Supercomputing (1990)
5. Clifton-Everest, R., McDonell, T.L., Chakravarty, M.M.T., Keller, G.: Embedding foreign code. In: Practical Aspects of Declarative Languages (2014)
6. Clifton-Everest, R., McDonell, T.L., Chakravarty, M.M.T., Keller, G.: Streaming irregular arrays. In: Haskell (2017)
7. Elsman, M., Henriksen, T., Serup, N.G.W.: Data-parallel flattening by expansion. In: Libraries, Languages and Compilers for Array Programming (2019)
8. Elsman, M., Larsen, K.F.: Efficient translation of certain irregular data-parallel array comprehensions. In: Trends in Functional Programming (2020)
9. Fluet, M., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Manticore: A heterogeneous parallel language. In: Declarative aspects of multicore programming (2007)
10. Grelck, C.: Single assignment C (SAC): High productivity meets high performance. In: Central European Functional Programming School (2012)
11. Henriksen, T., Elsman, M., Oancea, C.E.: Size slicing: A hybrid approach to size inference in futhark. In: Functional High-performance Computing (2014)
12. 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: Programming Language Design and Implementation (2017)
13. Henriksen, T., Thorøe, F., Elsman, M., Oancea, C.: Incremental flattening for nested data parallelism. In: Principles and Practice of Parallel Programming (2019)
14. Hoare, C.A.: Quicksort. The Computer Journal **5**(1), 10–16 (1962)
15. Keller, G., Chakravarty, M.M.T., Leshchinskiy, R., Lippmeier, B., Peyton Jones, S.: Vectorisation avoidance. In: Haskell (2012)
16. McDonell, T.L., Chakravarty, M.M.T., Grover, V., Newton, R.R.: Type-safe runtime code generation: Accelerate to LLVM. In: Haskell (2015)
17. McDonell, T.L., Chakravarty, M.M.T., Keller, G., Lippmeier, B.: Optimising purely functional GPU programs. In: International Conference on Functional Programming (2013)
18. Peyton Jones, S., Leshchinskiy, R., Keller, G., Chakravarty, M.M.T.: Harnessing the multicores: Nested data parallelism in Haskell. In: Foundations of Software Technology and Theoretical Computer Science (2008)
19. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optizing parallelism, locality, and recomputation in image processing pipelines. In: Programming Language Design and Implementation (2013)
20. Reppy, J., Sandler, N.: Nessie: A NESL to CUDA compiler. In: Compilers for Parallel Computing (2015)
21. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Graphics Hardware (2007)
22. Steuwer, M., Remmelg, T., Dubach, C.: Lift: A functional data-parallel ir for high-performance gpu code generation. In: Code Generation and Optimization (2017)
23. Svensson, B.J., Sheeran, M., Claessen, K.: Obsidian: A domain specific embedded language for parallel programming of graphics processors. In: Implementation and Application of Functional Languages (2008)