

# LIFT: A Functional Data-Parallel IR for High-Performance GPU Code Generation

Michel Steuwer    Toomas Rimmelg    Christophe Dubach

University of Edinburgh

{michel.steuwer, toomas.remmelg, christophe.dubach}@ed.ac.uk

## Abstract

Parallel patterns (*e.g.*, map, reduce) have gained traction as an abstraction for targeting parallel accelerators and are a promising answer to the performance portability problem. However, compiling high-level programs into efficient low-level parallel code is challenging. Current approaches start from a high-level parallel IR and proceed to emit GPU code directly in one big step. Fixed strategies are used to optimize and map parallelism exploiting properties of a particular GPU generation leading to performance portability issues.

We introduce the LIFT IR, a new data-parallel IR which encodes OpenCL-specific constructs as functional patterns. This functional nature simplifies the exploration of optimizations and mapping of parallelism from portable high-level programs using rewrite-rules, as shown in prior work [19].

This paper describes how LIFT IR programs are compiled into efficient OpenCL code. This is non-trivial as many performance sensitive details such as memory allocation, array accesses or synchronization are not explicitly represented in the LIFT IR. We present techniques which overcome this challenge by exploiting the pattern’s high-level semantics. Our evaluation shows that the LIFT IR is flexible enough to express GPU programs with complex optimizations achieving performance on par with manually optimized code.

## 1. Introduction

GPUs (Graphics Processing Units) and other parallel accelerators are now commonplace in computing systems. Their performance is orders of magnitude higher than traditional CPUs making them attractive for many application domains. However, achieving their full performance potential is extremely hard, even for experienced programmers. This requires ultra-specialized kernels written in low-level languages such as OpenCL. This inevitably leads to code that is not performance portable across different hardware.

High-level languages such as Accelerate [16], Delite [20], StreamIt [21] or Halide [17] have been proposed to ease programming of GPUs. These approaches are all based on *parallel patterns*, a concept developed in the late 80’s [7]. Parallel patterns are deeply rooted in functional programming concepts such as function composition and nesting, and absence of side-effects. Using parallel patterns, programs are expressed without committing to a particular implementation which is *the* key to achieving performance portability.

From the compiler point of view, the semantic information associated with parallel patterns offers a unique opportunity for optimization. These abstractions make it easier to reason about parallelism and apply optimizations without the need for complex analysis. However, designing an IR (Internal Representation) that preserves this semantic information throughout the compilation pipeline is difficult. Most existing approaches either lower the parallel primitives into loop-based code, losing high-level semantic information, or directly produce GPU code using fixed optimization strategies. This inevitably results in missed opportunities for optimizations or performance portability issues.

In this paper, we advocate the use of a functional data-parallel IR which expresses OpenCL-specific constructs. Our functional IR is built on top of lambda-calculus and can express a whole computational kernel as a series of nested and composed function calls. It is equipped with a dependent type system that reasons about array sizes and value ranges for variables, preserving important semantic information from the high-level patterns. The information available in the types is used at multiple stages such as when performing array allocation, index calculation, and even synchronization and control flow simplification.

One downside of a functional IR is that all operations are represented as functions which produce intermediate results. This issue is addressed by fusing chains of composed or nested functions which only affect the data layout (*e.g.*, *zip* or *gather*). This involves recording information about the accessed data in a *view* structure which is then used to emit the appropriate array access expression. These indexing expressions are then simplified using a symbolic algebraic simplifier that relies on type information (*e.g.*, array length and value ranges). This leads to the generation of highly efficient GPU code competitive with hand-tuned kernels.

To summarize, we make the following contributions:

- We present a new data-parallel functional IR named **LIFT** that targets the OpenCL programming model;
- We show how semantic information embedded in the IR is used in various phases such as *memory allocation*, *array access* generation and optimizations, *synchronization* minimization and *control-flow simplification*;
- We demonstrate that the generated GPU code performance is on par with manually optimized OpenCL code.

The rest of the paper is organized as follows: Section 2 reviews related work and motivates our approach. Sections 3 and 4 discuss our **LIFT** pattern based function abstraction of OpenCL and its internal representation. Section 5 discusses the compiler implementation and optimizations for high-performance code generation. Sections 6 and 7 present experimental setup and evaluation before section 8 concludes.

## 2. Related Work and Motivation

The problem of producing efficient GPU code has been well studied over the years. Figure 1 gives an overview of the different approaches related to this paper. Loop-based auto-parallelization techniques have been extensively studied for languages like C. Recent work on polyhedral compilation [1] for instance has pushed the boundaries of such techniques for GPU code generation. However, these techniques only operate on loops and requires certain property such as affine indices to work effectively.

In the last decade, there have been a shift towards algorithmic skeletons [8] and DSLs (Domain Specific Languages). These approaches offer the advantage of exploiting high-level and domain-specific information. The simplest approaches are based on parametric library implementation of skeletons such as Thrust [2] and SkelCL [18]. However, these approaches are not portable and more importantly, cannot perform optimizations across library calls.

A different approach consists of lowering the applications to a functional representation which is then compiled to GPU code. This process involves mapping of parallelism, performing optimizations such as fusion of operations and finally code generation. This approach is used by a many systems such as Copperhead [5], Delite [3], Accelerate [6, 16], LiquidMetal [10], HiDP [22], Halide [17] and NOVA [9].

The drawback of such approaches is that the mechanism to map parallelism and optimize code is performed within the code generator and in general uses a fixed strategy. This means that it might be difficult to achieve performance portability due to the large gap to fill between the functional IR and the GPU code that will eventually be produced. In contrast, we advocate the use of a low-level IR which encodes OpenCL-specific constructs as we will see in the next section. The decision of how to optimize code and map parallelism are taken during the conversion from the generic functional IR and the OpenCL-specific **LIFT** IR. This clearly

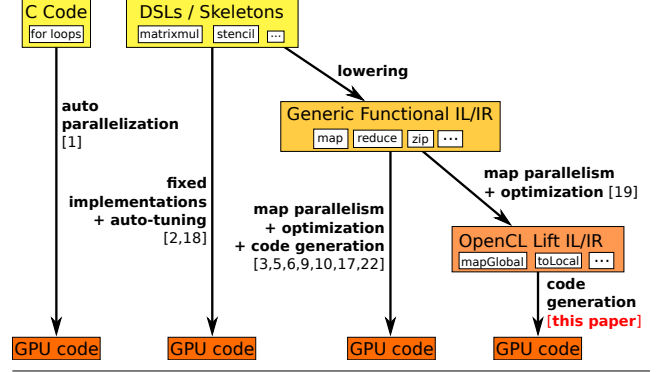


Figure 1. GPU code generation landscape.

separates the concerns of optimization and parallelism mapping from the actual process of code generation.

The **LIFT** IR introduced in this paper is capable of expressing many different OpenCL mappings and optimizations in a pattern-based and functional style. This complements prior work which studied the problem of deciding how to find optimal mapping using analysis-driven heuristics [13] or rewrite-rules [19]. The main contribution of this paper is to explain how the **LIFT** compiler produces efficient OpenCL code once the original program has been lowered and mapped into the **LIFT** IL. As we will see, generating high performance OpenCL code relies on using the parallel patterns’ semantic information encoded in the IR.

## 3. The **LIFT** Intermediate Language

This section presents **LIFT**, a functional data-parallel intermediate language designed as a target for high-level languages based on parallel patterns. It is similar in spirit to prior published work on data-parallel representation [4, 11, 12, 15, 19]. The **LIFT** IL (Intermediate Language) specifically targets OpenCL, although many concepts presented are more widely applicable.

### 3.1 Design Principles

High-level languages based on parallel patterns capture rich information about the algorithmic structure of programs. Take the computation of the dot product as an example:

```
dot(x, y) = reduce(+, 0, map(*, zip(x, y)))
```

Here the **zip** pattern captures that arrays **x** and **y** are accessed pairwise. Furthermore, the **map** pattern allows the compiler to perform the multiplication in parallel, as well as the final summation, expressed using the **reduce** pattern.

Our most important design goal is to preserve algorithmic information for as long as possible in the compiler. The **LIFT** intermediate language achieves this by expressing the OpenCL programming model functionally. One of the key advantages of the **LIFT** approach is that it is possible to decouple the problem of mapping and exploiting parallelism, which has been covered in prior work [13, 19], from the code generation process, which is what this paper is about.

### 3.2 Intermediate Language

The LIFT IL expresses program as compositions and nesting of functions which operate on arrays. The foundation of the LIFT IL is lambda calculus which formalizes the reasoning about functions, their composition, nesting and application.

In the following section we introduce the predefined patterns used as building blocks to express programs. Besides these patterns, the LIFT IL also supports *user functions* written in C operating on scalar values, which implement the application specific computations.

**Algorithmic Patterns** The LIFT IL supports four algorithmic patterns corresponding to sequential implementations of the well known *map* and *reduce* patterns, the identity and the iterate primitive. The iterate pattern applies a function  $f$   $m$  times by re-injecting the output of each iteration as the input of the next. The length of the output array is inferred as a function  $h$  of the number of iterations  $m$ , the input array length  $n$  and the change of the array length by a single iteration captured by the function  $g$ . We will discuss in section 5 how we automatically infer the length of the output array.

$$\begin{aligned}
 \text{mapSeq}(f, [x_n \dots x_2 x_1]) &= [f(x_1) f(x_2) \dots f(x_n)] \\
 \text{reduceSeq}(z, f, [x_n \dots x_2 x_1]) &= [f(\dots(f(z, x_1), x_2) \dots), x_n] \\
 \text{id}([x_n \dots x_2 x_1]) &= [x_n \dots x_2 x_1] \\
 \text{iterate}^m(f, [x_n \dots x_2 x_1]) &= \underbrace{f(\dots(f([x_n \dots x_2 x_1]))}_{m \text{ times}}
 \end{aligned}$$

**Data Layout Patterns** The LIFT IL defines a set of patterns that do not perform any computation but simply reorganize the data layout. The first two patterns, *split* and *join*, add or remove a dimension from the input array.

$$\begin{aligned}
 \text{split}^m([x_1 x_2 \dots x_n]) &= [x_1 x_2 \dots x_n] \\
 \text{join}([x_1 x_2 \dots x_n]) &= [x_1 x_2 \dots x_n]
 \end{aligned}$$

The *gather* and *scatter* patterns apply a permutation function  $f$  which remaps indices when reading from or writing to arrays respectively. Combined with *split* and *join*, for instance, these primitives can express matrix transposition:  $\text{split}^{\text{rows}} \circ \text{scatter}(i \rightarrow (i \% \text{ncols}) \cdot \text{nrows} + i / \text{ncols}) \circ \text{join}$

$$\begin{aligned}
 \text{gather}(f, [x_{f(1)} x_{f(2)} \dots x_{f(n)}]) &= [x_1 x_2 \dots x_n] \\
 \text{scatter}(f, [x_1 x_2 \dots x_n]) &= [x_{f(1)} x_{f(2)} \dots x_{f(n)}]
 \end{aligned}$$

The *zip* pattern is used to combine two arrays of elements into a single array of pairs while the *get* primitive projects a component of a tuple.

$$\begin{aligned}
 \text{zip}([x_1 x_2 \dots x_n], [y_1 y_2 \dots y_n]) &= [(x_1, y_1) (x_2, y_2) \dots (x_n, y_n)] \\
 \text{get}_i((x_1, x_2, \dots, x_n)) &= x_i
 \end{aligned}$$

Finally, *slide* applies a moving window to the input data and is used to express stencil computations. For instance,  $\text{mapSeq}(\text{reduceSeq}(0, +)) \circ \text{slide}(3, 1, \text{input})$  expresses a simple 3-point stencil. Multi-dimensional stencils are also expressible by composing several *slide* functions interleaved with transpositions (not shown for space constraint).

$$\begin{aligned}
 \text{slide}(\text{size}, \text{step}, [x_1 x_2 \dots x_n]) &= [x_1 x_2 \dots x_n]
 \end{aligned}$$

**Parallel Patterns** OpenCL provides a hierarchical organization of parallelism where threads are grouped into *work groups* of *local threads* or a flat organization where threads are simply *global*. This hierarchy is represented with three patterns, where a *mapLcl* must be nested inside of *mapWrg*:

$$\text{mapGlb}^{(0,1,2)} \quad \text{mapWrg}^{(0,1,2)} \quad \text{mapLcl}^{(0,1,2)}$$

OpenCL supports up to three thread dimensions, represented by the superscripts 0, 1, 2. The semantic of these patterns is identical to *mapSeq*, except that  $f$  is applied in parallel.

**Address Space Patterns** OpenCL distinguishes between *global*, *local* and *private* address spaces. The LIFT IL offers three corresponding primitives which wrap a function and influence the address space used to store the output:

**toGlobal toLocal toPrivate**

For example, a sequential copy of an array  $x$  into local memory is expressed as:  $\text{toLocal}(\text{mapSeq}(\text{id}))(x)$ . This design decouples the decision of *where* to store data (i.e., the address space) from the decision of *how* the data is produced (i.e., sequentially or in parallel).

**Vectorize Pattern** The LIFT IL supports two primitives that transforms data between scalar and vector types, and one pattern which applies a function to a vector.

$$\begin{aligned}
 \text{asVector}([x_1 x_2 \dots x_n]) &= \overrightarrow{x_1, x_2, \dots, x_n}, x_i \text{ is scalar} \\
 \text{asScalar}(\overrightarrow{x_1, x_2, \dots, x_n}) &= [x_1 x_2 \dots x_n] \\
 \text{mapVec}(f, \overrightarrow{x_1, x_2, \dots, x_n}) &= \overrightarrow{f(x_1), f(x_2), \dots, f(x_n)}
 \end{aligned}$$

During code generation, the function  $f$  is transformed into a vectorized form. This transformation is straightforward for functions based on simple arithmetic operations since OpenCL already defines vectorized forms for these operation. In the other more complicated cases, the code generator simply applies  $f$  to each scalar in the vector.

```

1 partialDot(x: [float]N, y: [float]N) =
2   (join ◦ mapWrg0(
3     join ◦ toGlobal(mapLcl0(mapSeq(id))) ◦ split1 ◦
4     iterate6( join ◦
5               mapLcl0( toLocal(mapSeq(id)) ◦
6                       reduceSeq(add, 0) ) ◦
7               split2 ) ◦
8     join ◦ mapLcl0( toLocal(mapSeq(id)) ◦
9                   reduceSeq(multAndSumUp, 0) ) ◦ split2
10  ) ◦ split128)( zip(x, y) )

```

**Listing 1.** LIFT IL implementation of partial dot product

### 3.3 Example: Dot product in the LIFT IL

Listing 1 shows one possible implementation of dot product expressed in the LIFT IL. The program is represented using a functional style, therefore, the program is read from right to left instead of the familiar left to right common in imperative programming. Furthermore, to simplify the notation we use the  $\circ$  symbol to denote sequential function composition, *i.e.*,  $(f \circ g)(x) = f(g(x))$ .

In the program of Listing 1 the input arrays  $x$  and  $y$  are combined using the `zip` pattern in line 10. The zipped array is then split into chunks of size 128 (line 10). A work group processes a single chunk using the `mapWrg` pattern (line 2) before combining the computed chunks using the `join` pattern (line 2). Inside of a work group we perform three steps to process a chunk of 128 elements: 1) we split the chunk further into pairs of two zipped elements, which we multiply and add up before copying the computed result into local memory (lines 8 and 9); 2) we iteratively reduce two elements a time in local memory (lines 5 and 7); 3) we copy the computed result back into global memory (line 3).

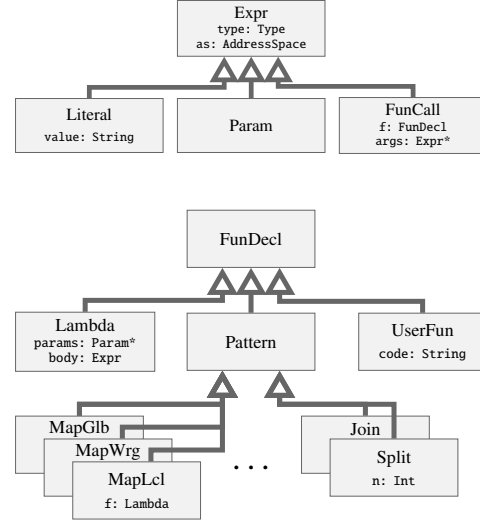
Note that the code shown here corresponds to a single OpenCL kernel which only computes a partial dot product. We focus on this OpenCL kernel and omit a second kernel which sums up all intermediate results, because the vast majority of the runtime is spent in the first kernel.

### 3.4 Summary

In this section we have discussed the design of the LIFT functional data-parallel intermediate language. It is similar in style to prior work [4, 15, 19] and is OpenCL specific. The LIFT IL expresses very precisely how programs are mapped to the OpenCL programming model, as we have seen for the dot product example. The following section describes how this language is represented in our compiler. Section 5 will then describe how efficient OpenCL code is produced.

## 4. The LIFT Intermediate Representation

This section introduces the LIFT Intermediate Representation. All programs expressible in the LIFT intermediate language can be represented by the LIFT IR. One of the key features of the LIFT IR is that it preserves a functional representation of the program all the way through.



**Figure 2.** Class diagram of the LIFT IR.

### 4.1 Organization of classes

Programs are represented as graphs where nodes are implemented as *objects*. The use of a graph-based representation avoids the problem of performing extensive renaming when transforming functional programs [14]. The class diagram of the LIFT IR in Figure 2 shows two main classes: *expressions* (`Expr`) and *function declarations* (`FunDecl`).

**Expressions** represent values and have a type associated with. Expressions are either literals, parameters or function calls. Literals represent compile time known constants such as `3.4f`, arrays or tuples. Parameters are used inside functions and their values are the arguments of a function call. Finally, function calls connect a function to be called (a `FunDecl`) with its arguments (`Exprs`).

**Function declarations** correspond to either a lambda, a predefined pattern or a user function. Lambdas are anonymous function declarations with parameters and a body which is evaluated when the lambda is called. A pattern is a built-in function such as `map` or `reduce`. The `UserFun` corresponds to user-defined functions expressed in a subset of the C language operating on non-array data types.

### 4.2 LIFT IR Example

Figure 3 shows the LIFT IR of the dot-product program from Listing 1. The plain arrows show how object reference each other. The top left node labeled *Lambda2* is the root node of the graph taking two parameters and its body implements dot-product as a sequence of function calls.

The dashed arrows visualizes the way the data flows through the IR. The inputs  $x$  and  $y$  are first used as an input to the `zip` function which is then fed into a call to `split(128)`. Then the results of the split is fed into the `mapWrg` function. The function which is applied to each chunk of 128

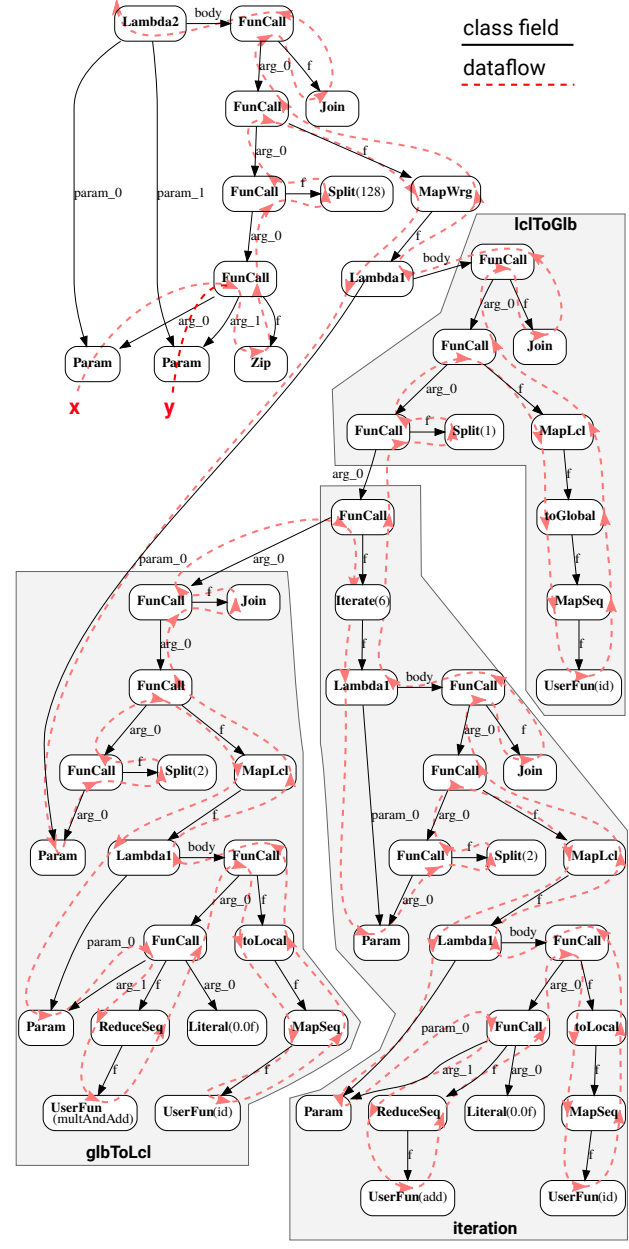


Figure 3. LIFT IR for dot-product example

elements is represented as a lambda which processes the input in three steps. First, the data is moved from global to local memory by performing a partial reduction (labeled *glbToLcl*). Then, the data flows to a function which iteratively reduces the elements in local memory (*iteration*). Finally, the data is moved back from local memory to global memory (*lclToGlb*), exits the *mapWrg* and the last join is applied to return the final result.

### 4.3 Lambda and Data Flow

Lambdas appear in several places in the IR graph and encode the data flow explicitly. For example, focusing on the *iteration* part of the graph, we see that a Lambda node is used



Figure 4. Overview of the LIFT compilation stages.

below the Iterate node. To understand what the lambda does, we can look back at Listing 1, lines 4–7 copied here:

```
iterate6( join ◦ mapLcl0( ... ) ◦ split2 )
```

This notation is only syntactic sugar for:

```
iterate6( λ p . join(mapLcl0(..., split2(p))) )
```

The purpose of the lambda ( $\lambda$ ) is to make the data flow explicit, i.e. the *iterate* pattern passing its input via the parameter  $p$  first to *split* which then passes it to *mapLcl* and *join*.

## 5. Compilation Flow

Figure 4 shows the stages involved in compiling the LIFT IR into efficient OpenCL code. The compilation starts by analyzing type information which is key for high-performance code generation. This information is used heavily in the subsequent memory allocation and array accesses generation passes. The barrier elimination stage minimizes the number of synchronizations required for correct parallel code. Finally, the OpenCL code generation performs a last final optimizations to simplify control flow.

### 5.1 Type System and Analysis

The LIFT compiler implements a *dependent type system* which keeps track of the length and shapes of nested arrays. Besides *array types*, the type system supports *scalar types* (e.g., *int*, *float*), *vector types*, and *tuple types*. While vector types correspond to OpenCL vector data types (e.g., *int2*, *float4*) tuples are represented as structs. Array types can be nested to represent multi-dimensional arrays. In addition, arrays carry information about the length of each dimension in their type. These length information are arithmetic expressions of operations on natural numbers larger than zero and named variables which are unknown at compiler time. For example, given an array  $x$  of length  $n$  where the type of the elements is *float* we write the type of  $x$  as  $[float]_n$ . Applying the array  $x$  to the *split* <sup>$m$</sup>  pattern results in the type  $[float]_{n/m}$ .

The types of function bodies are automatically inferred from the parameter types by traversing the graph following the data flow, as indicated by the dotted arrows in Figure 3.

### 5.2 Memory Allocation

A straightforward memory allocator would allocate a new output buffer for every single *FunCall* node. However, this would be very inefficient as data layout patterns, such as *split*, only change the way memory is accessed but do not modify the actual data. Memory allocation is, therefore, only performed for functions actually modifying data. These are *FunCall* nodes where the called function contains a *UserFun* node, such as the *UserFun(add)* node

```

input : Lambda expression representing a program
output: Expressions annotated with address space information

inferAddressSpaceProg(lambda)
1 foreach param in lambda.params do
2   if param.type is ScalarType then param.as = PrivateMemory;
3   else param.as = GlobalMemory;
4 inferASExpr(lambda.body, null)

inferASExpr(expr, writeTo)
5 switch expr.type do
6   case Literal expr.as = PrivateMemory;
7   case Param assert (expr.as != null);
8   case FunCall
9     foreach arg in expr.args do
10      inferASExpr(arg, writeTo)
11     switch expr.f.type do
12       case is UserFun
13         if writeTo != null then expr.as = writeTo;
14         else expr.as = inferASFromArgs(expr.args);
15       case is Lambda inferASFunCall(expr.f, expr.args, writeTo);
16       case is toPrivate
17         inferASFunCall(expr.f.lambda, expr.args, PrivateMemory);
18       case is toLocal
19         inferASFunCall(expr.f.lambda, expr.args, LocalMemory);
20       case is toGlobal
21         inferASFunCall(expr.f.lambda, expr.args, GlobalMemory);
22       case is Reduce
23         inferASFunCall(expr.f.f, expr.args, expr.f.init.as);
24       case is Iterate or Map
25         inferASFunCall(expr.f.f, expr.args, writeTo);
26       otherwise do expr.as = expr.args.as;

inferASFunCall(lambda, args, writeTo)
27 foreach p in lambda.params and a in args do p.as = a.as;
28 inferASExpr(lambda.body, writeTo)

```

**Algorithm 1:** Recursive address space inference algorithm

in Figure 3. For these nodes, the compiler uses the array length information from the type to compute the size of the memory buffer required. When a data layout pattern is encountered, an internal data structure called *view* is created, which remembers how memory should be accessed by the subsequent functions. Details of the *views* are discussed in subsection 5.3.

Memory is allocated in one of three OpenCL *address spaces*. Algorithm 1 determines the address space of each allocation. First, each parameters of the lambda expression which represents a program are processed where scalar are assigned to Private memory and Global memory is used for all others as required by OpenCL. Then, the body of the lambda is visited in line 4.

The function *inferASExpr* determines the address space of a given expression based on its second parameter *writeTo* or its function arguments in case *writeTo* is *null*. As we have exactly three subclasses of *Expr* we consider these as separate cases: *Literals* reside in the Private memory; *Params* have their address space set when their function is called, as we have already seen above; *FunCalls* determine the ad-

dress space of their arguments and then investigate which function was called. *UserFun* take their address space from the *writeTo* argument or inferred it from the address space of its arguments; if all arguments have the same address space, the user function will write into the same address space, otherwise it writes to global memory by default.

The *toPrivate*, *toLocal*, *toGlobal* functions change the *writeTo* argument before recursing within their nested function to produce the output in a specific address space. Finally, *Reduce* directly writes into the memory of the initializer expression and has, therefore, the same address space.

### 5.3 Multi-Dimensional Array Accesses

In the LIFT IR, arrays are not accessed explicitly but implicitly; the patterns determine which thread accesses which element in memory. This design simplifies the process of lowering high-level programs to the LIFT IR and guarantees that data races are avoided by construction since no arbitrary accesses into memory are permitted. However, this introduces two main challenges when compiling the LIFT IR: First, avoiding unnecessary intermediate results arising from function which change only the data layout; And, secondly, generating efficient accesses to multi-dimensional arrays which have a flat representation in memory.

**Example** Consider the following dot product example, partially copied again here for convenience:

```

1 (join ◦ mapWrg0( ...
2   join ◦ mapLcl0( ...
3     reduceSeq( $\lambda(a, xy) \mapsto a + (xy_0 * xy_1), 0$ ) ◦ split2
4   ) ◦ split128)( zip(x, y) )

```

We are interested in understanding how the arrays *x* and *y* are accessed inside the lambda in line 3 and, ultimately, how to generate code to express these accesses. This is not obvious, as the arrays are first combined using *zip* and then split into chunks of size 128 in line 4. When processing a single chunk inside a work group (*mapWrg* in line 1), the array is further split into smaller chunks of two elements (line 3) and every local thread (*mapLcl* in line 2) performs a sequential reduction. Individual elements of the arrays are accessed using the *xy* variable. The *xy<sub>0</sub>* indicates an access to the first element of the tuple, which is an element of array *x*.

**View construction** A *view* is a compiler internal data structure which stores information needed to generated array accesses. Functions that do not modify an array but merely change its data layout produce a *view* instead of allocating and writing to a new array.

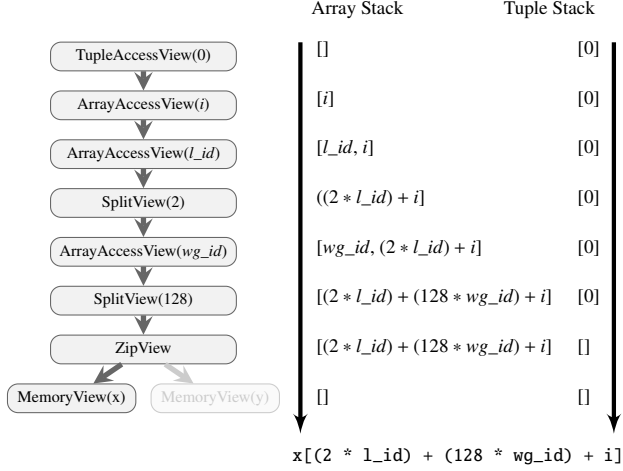
To generate the array access for the *xy<sub>0</sub>* expression from our example, we traverse the IR graph following the data flow. When visiting each node we construct a *view* representing how the particular node influences the array access. The resulting *view* structure is shown on the left hand side of Figure 5 where each view is connected to its predecessor view. For example, the *ZipView* has two predecessors, since the

```

1  (((wg_id*M+l_id)/M)+(((wg_id*M+l_id)%M)*N))/N)*N+(((wg_id*M+l_id)/M)+(((wg_id*M+l_id)%M)*N))%N
2  (( wg_id          +          l_id          *N) /N)*N+( wg_id          +          l_id          *N) %N
3  l_id          *N+ wg_id

```

**Figure 6.** Simplification process of automatically generated array indices.



**Figure 5.** Views constructed for the generation of the first memory access of dot product (on the left) and consumption of views to generate an array index (on the right).

two arrays  $x$  and  $y$  have been combined. Each map pattern results in a `ArrayAccessView` which emulates an access in one dimension of the array by the `map` function. Nested `ArrayAccessViews`, therefore, correspond to accesses to multi-dimensional arrays.

**View consumption** Once the view structure is constructed, all information required for generating accesses is available. An array index expression is calculated by consuming this information in the opposite order of construction, *i.e.*, top-to-bottom. This process is illustrated on the right hand side of Figure 5 with the resulting array access at the bottom. The constructed view is shown on left hand side. The *Tuple Stack* on the right side contains information about tuple access which determine which array is being accessed. The *Array Stack* in the middle records information about which element of the array is being accessed.

Starting from the top with two empty stacks, we process the `TupleAccessView(0)` pushing the first component of a tuple, *i.e.*, 0, onto the tuple stack. Then an `ArrayAccessView` pushes a new variable ( $i$ ) on the stack indexing the array in one dimension. Another `ArrayAccessView` pushes another index variable ( $l\_id$ ) on the stack. The `SplitView` pops two indices from the stack and combines them into a one dimensional index using the split factor, linearizing the array access. The `ZipView` pops from the tuple stack and uses this information to decide which view should be visited next: the `MemoryView(x)`. Finally, we reach a memory view which is used to emit the final index to the memory of input  $x$ .

**Simplifying array accesses** If we follow the approach described above we will obtain correct array indices, however, this can lead to long and complex expressions. We illustrate this issue using matrix transposition, expressed in LIFT as:

```

1  matrixTranspose(x: [[float]M]N) =
2  (mapWrg0(mapLcl0(id)) ◦
3  splitN ◦ gather (λ(i) → (i/M) + (i%M) × N) ◦ join)(x)

```

Here the *join*, *gather* and *split* patterns flatten the two-dimensional matrix, rearrange the indices with a stride before splitting the array in two dimensions. When generating the read accesses for the *id* function, following the methodology introduced above, we obtain the array index shown in Figure 6 line 1. While this array index expression is correct it is also quite long compared to the index a human could write for performing a matrix transposition, shown in line 3.

However, a standard compiler would be unable to simplify this expression since important information about value ranges is missing. In contrast, the LIFT compiler is able to derive the simplified form using a symbolic simplification mechanism exploiting domain knowledge. The simplification process follows a set of algebraic rules exploiting properties of arithmetic operators supported in the compiler (additions, multiplications, integer divisions, fractions, powers and logarithms). A small subset of the rules supported is shown below:

$$x/y = 0, \quad \text{if } x < y \text{ and } y \neq 0 \quad (1)$$

$$(x \times y + z)/y = x + z/y, \quad \text{if } y \neq 0 \quad (2)$$

$$x \% y = x, \quad \text{if } x < y \text{ and } y \neq 0 \quad (3)$$

$$(x + y) \% z = (x \% z + y \% z) \% z, \quad \text{if } z \neq 0 \quad (4)$$

$$(x/y) * y + x \% y = x, \quad \text{if } y \neq 0 \quad (5)$$

$$(x * y) \% y = 0, \quad \text{if } y \neq 0 \quad (6)$$

The type system exploits domain specific knowledge by inferring range information for every variable. For example, the `wg_id` variable corresponding to the `mapWrg`, has values ranging from 0 to  $M$ , which is the row length of the input matrix. Similarly, the `l_id` variable corresponding to the `mapLcl`, has values between 0 and  $N$  since it indexes an array split in chunks of  $N$ . The expression  $(wg\_id * M + l\_id) \% M$  can, therefore, be simplified to `l_id` using rule 4 to distribute the modulo followed by rules 6 and 3 to simplify the remaining modulo operations. A traditional OpenCL compiler would not be able to simplify this code, as it is missing the information that `wg_id` is positive and smaller than  $M$ . Lines 2 and 3 in Figure 6 shows the whole expression after a few simplification steps. This results in the same compact array index a human would write.

In one case, disabling the simplification led to the generation of several MB of OpenCL code. By applying arithmetic simplification we generate concise indices which reduce code size and speed up execution as costly operations such as modulo can often be simplified away. We will investigate the performance benefits in section 7.

#### 5.4 Barrier Elimination

When different threads access the same memory location they must synchronize their accesses to ensure memory consistency. When compiling the LIFT IR, this corresponds to generating an appropriate synchronization primitive after each occurrence of a parallel *map* pattern. A return is emitted after the *mapGlb* and *mapWrg* patterns to end the current kernel, since OpenCL does not support synchronization across work groups. For the *mapLcl* pattern a barrier is emitted synchronizing all threads in a work group.

Sometimes these barriers are not required, for example, when there is no sharing because all threads continue operating on the same memory locations. We take an approach of safety first, where a barrier is emitted by default and is only removed if we can infer from the context that it is not required. The key insight for this barrier elimination process is the fact that the LIFT IL only allows sharing of data by using the *split*, *join*, *gather* or *scatter* patterns. Therefore, we look for sequences of *mapLcl* calls which have no *split*, *join*, *gather*, or *scatter* between them and mark them specially. These marked *mapLcl* function calls will not emit a barrier in the OpenCL code generation stage. We also eliminate one barrier when two *mapLcl* appear inside of a *zip* since the two branches of a *zip* can be executed completely independently.

#### 5.5 OpenCL Code Generation

The final stage in the LIFT compilation pipeline is the OpenCL code generation where low-level optimizations are performed to precisely control the generated code. At this stage types are inferred, memory is allocated, array indices generated and unnecessary barriers eliminated. To generate the OpenCL code, the LIFT IR graph is traversed following the data flow and a matching OpenCL code snippets is generated for every pattern. The generated kernel for the dot product example is shown in Figure 7 with only minor cosmetic changes made by hand for presentation purpose (renamed variables, removed comments, removed extra parenthesis).

No OpenCL code is generated for patterns such as *split* and *toLocal* since their effect have been recorded in the views. For the different *map* patterns, for loops are generated, which for the parallel variations will be executed in parallel by multiple work groups or threads, such as the loop in line 7. For the *reduceSeq* pattern, a loop with an accumulation variable (e.g., in line 10) is generated calling its function in every iteration. The code generated for *iterate* spans lines 17 to 29 with the double buffering initializing two pointers in line 18 and swapping the pointers after each iteration in lines 27 and 28.

```

1  kernel void KERNEL(const global float *restrict x,
2                      const global float *restrict y,
3                      global float *z, int N) {
4      local float tmp1[64]; local float tmp2[64];
5      local float tmp3[32];
6      float acc1; float acc2;
7      for (int wg_id = get_group_id(0); wg_id < N/128;
8           wg_id += get_num_groups(0)) {
9          { int l_id = get_local_id(0);
10           acc1 = 0.0f;
11           for (int i = 0; i < 2; i += 1) {
12               acc1 = multAndSumUp(acc1,
13                                   x[2 * l_id + 128 * wg_id + i],
14                                   y[2 * l_id + 128 * wg_id + i]); }
15           tmp1[l_id] = id(acc1); }
16      barrier(CLK_LOCAL_MEM_FENCE);
17      int size = 64;
18      local float *in = tmp1; local float *out = tmp2;
19      for (int iter = 0; iter < 6; iter += 1) {
20          if (get_local_id(0) < size / 2) {
21              acc2 = 0.0f;
22              for (int i = 0; i < 2; i += 1) {
23                  acc2 = add(acc2, in[2 * l_id + i]); }
24              out[l_id] = id(acc2); }
25          barrier(CLK_LOCAL_MEM_FENCE);
26          size = size / 2;
27          in = (out == tmp1) ? tmp1 : tmp3;
28          out = (out == tmp1) ? tmp3 : tmp1;
29          barrier(CLK_LOCAL_MEM_FENCE); }
30      if (get_local_id(0) < 1) {
31          z[wg_id] = id(tmp3[l_id]); }
32      barrier(CLK_GLOBAL_MEM_FENCE); } }
```

**Figure 7.** Compiler-generated OpenCL kernel for the dot product example shown in Listing 1

**Control flow simplification** The LIFT compiler performs control flow simplification using the extra semantic information available in the pattern and in the type. A straightforward implementation would emit a for loop for every *map*, *reduce* and *iterate* pattern. Fortunately, the LIFT compiler can often statically infer if the number of threads assigned to a *map* is larger, equal or lower than the number of elements to process. This is the case in lines 20 and 30 which corresponds to the *mapLcl* in line 5 and 3 in the original Listing 1. This is possible because the compiler knows that `get_local_id(0)` returns a non-negative number. If we infer that the loop executes exactly once by every thread we eliminate the loop completely, which is the case in line 9 which corresponds to the *mapLcl* in line 8 in the original Listing 1.

Performing this control flow simplification is beneficial in two ways: first, execution time is improved as additional instructions from the loop are avoided; and, secondly, in general fewer registers are required when loops are avoided.

#### 5.6 Summary

In this section we have seen how LIFT IR is compiled to OpenCL. We followed the dot product computation as a running example, first introduced in Listing 1, and discussed how types are inferred, memory is allocated, concise array accesses are generated, barriers are eliminated, and, finally, the OpenCL kernel with simplified control flow shown in Figure 7 is generated. The next section investigates the performance achieved by the LIFT compiler as well as the impact of the optimizations discussed in this section.



Program	Source	Input Size (Small and Large)		Characteristics				
				Local memory	Private memory	Vectorization	Coalescing	Iteration space
N-Body	NVIDIA SDK	16K, 131K	particles	✓	✓	✓	✓	1D
N-Body	AMD SDK	16K, 131K	particles		✓	✓	✓	1D
MD	SHOC	12K, 74K	particles		✓		✓	1D
K-Means	Rodinia	0.2M, 0.8M	points				✓	1D
NN	Rodinia	8M, 34M	points				✓	1D
MRI-Q	Parboil	32K, 262K	pixels		✓		✓	1D
Convolution	NVIDIA SDK	4K <sup>2</sup> , 8K <sup>2</sup>	images	✓			✓	2D
ATAX	CLBlast	4K <sup>2</sup> , 8K <sup>2</sup>	matrices	✓			✓	1D
GEMV	CLBlast	4K <sup>2</sup> , 8K <sup>2</sup>	matrices	✓			✓	1D
GESUMMV	CLBlast	4K <sup>2</sup> , 8K <sup>2</sup>	matrices	✓				1D
MM	CLBlast, AMD	1K <sup>2</sup> , 4K <sup>2</sup>	matrices		✓	✓	✓	2D
MM	CLBlast, NVIDIA	1K <sup>2</sup> , 4K <sup>2</sup>	matrices	✓	✓	✓	✓	2D

**Table 1.** Overview of the benchmarks

## 6. Experimental Setup

Two GPUs are used for evaluation: an AMD Radeon R9 295X2 with AMD APP SDK 2.9.214.1 and driver version 1598.5, as well as an Nvidia GTX Titan Black with CUDA 8.0.0 and driver version 367.35. All experiments are performed using single precision floating point numbers. We report the median runtime of at least 10 executions for each kernel measured using the OpenCL profiling API. We want to focus on the quality of the kernel code and, therefore, ignore data transfer times. In the case of benchmarks with multiple kernels, we sum the kernel runtimes in each execution.

## 7. Experimental Evaluation

This section evaluates the quality of the LIFT code generator using 12 OpenCL hand-optimized kernels collected from various sources shown in Table 1. These represent GPU programs from different application fields such as physics simulations (N-Body, MD), statistics and machine learning (KMeans, NN), imaging (MRI-Q), stencil (Convolution), and universally useful linear algebra primitives (ATAX, GEMV, GESUMMV, Matrix Multiplication). The characteristics of the reference implementations for the benchmarks and the input sizes used are described in Table 1. Local and private memory denote that those address spaces are used for storing data that is going to be reused. Vectorization shows whether memory or compute operations have been vectorized and coalescing shows whether accesses to global memory are coalesced. Iteration space shows the dimensionality of thread organization when running the kernel.

### 7.1 Expressing OpenCL Optimizations in the LIFT IR

The benchmarks OpenCL implementations encode GPU specific optimizations. Each implementation is represented in the LIFT IR by mimicking the OpenCL reference code. We are interested in testing the ability to represent differently optimized programs using the LIFT patterns presented in section 3. This section gives a brief overview of different patterns of computation and communication are encoded.

The *N-Body* implementation from the NVIDIA SDK makes use of *local memory* to store particle locations accessed by multiple threads. In LIFT we represent this by copying the particle locations using *map(id)* nested inside the *toLocal* pattern. By selecting one of the *mapSeq*, *mapLcl*, and *mapGlb* patterns, we control how the data is copied in the local memory. The AMD implementation does not use local memory but vectorizes the operations expressed using a combination of *mapVec* and *asVector*.

The *Convolution* benchmark applies *tiling* to improve performance by exploiting locality. Overlapping tiles, required by stencil applications, are created using the *slide* pattern. Two-dimensional tiles are achieved by a clever composition of *slide* with *map* and matrix transposition, which itself is expressed using *split*, *join*, and *gather*. These 2D tiles are then cooperatively copied into the local memory using the *toLocal(mapLcl(id))* pattern composition.

The CLBlast implementation of matrix-vector multiplication (SGEMV) carefully loads elements from the global memory using *coalesced memory accesses*. In the LIFT IR the *gather* pattern is used to influence which thread loads which element from memory and by choosing the right permutation accesses to the global memory are coalesced.

The MM implementations from CLBlast applies slightly different optimizations for both GPUs. For NVIDIA CLBlast uses a combination of *tiling* in local memory, *register blocking*, and *vectorization* of global and local memory operations. For AMD it also uses register blocking and vectorization but not tiling in local memory. In the LIFT IR, tiling and register blocking are represented by compositions of the *split* and *map* patterns together with a matrix transposition, which is itself expressed as combination of *split*, *scatter/gather* and *join* as seen in section 3.2. The LIFT IR vectorize patterns are used for vectorization.

The LIFT IR has proven to be powerful and flexible enough to represent our set of benchmark applications and their versatile GPU optimizations. The next section investigates the performance obtained when generating OpenCL code from the LIFT IR.

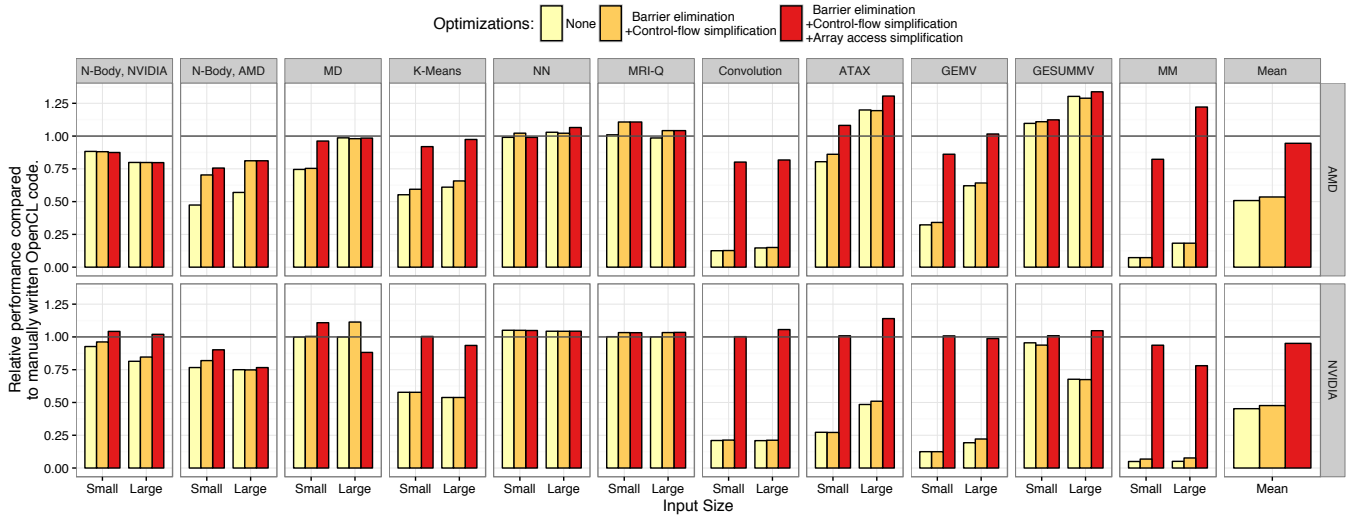


Figure 8. Speedup of generated code compared to OpenCL reference implementations.

## 7.2 Performance Evaluation

Figure 8 shows the relative performance of the LIFT generated code compared to the manually written OpenCL code on two GPUs. For each benchmark, we compare the performance of the hand-written OpenCL implementation with the performance of the generated kernel from the corresponding LIFT IR program. The different bars represent the performance obtained with different optimizations enabled and will be explain in the next section.

Concentrating on the right-most, dark red bar in each subplot, we notice that the code generator is able to achieve performance on-par with hand-written OpenCL kernels in most cases. This clearly demonstrates that the functional LIFT IR is able to express all the low-level details necessary to produce very efficient OpenCL code. The generated code is on average within 5% of the hand-written OpenCL implementation, which is quite a feat, considering how sensitive the underlying OpenCL compilers are. As anecdotal evidence, simply swapping statements around in our generated code for matrix multiplication can result in a performance difference of 3% on Nvidia for instance.

## 7.3 Evaluation of Optimization Impact

Figure 8 also shows the impact of each code generator optimization discussed in section 5. As can be seen, applying none of the optimizations discussed in this paper, leads to an average performance of only half the baseline. In extreme cases, such as matrix multiplication and convolution, the generated code can be as much as 10x or even 20x slower than the baseline. For convolution for instance, this is due to the complexity of the memory accesses expressions resulting from using the *slide* primitive. However, as can be seen on the figure, the effect of array access simplification on performance is very impressive, demonstrating the importance of this optimization. In addition, disabling array access sim-

plication generally leads to larger kernel code, up to 7MB of source code in the case of matrix multiplication.

Surprisingly, the barrier elimination and control-flow simplification seems to have little effect on performance on both machines. The largest impact is for the AMD version of N-Body where the simplification of control plays an important role since this AMD implementation does not use local memory. The control simplification is able to produce a kernel with a single loop (the reduction) which corresponds to the human-written implementation. On the other hand, without the simplification of control-flow enabled, three loops are produced which results in a 20% slowdown.

## 7.4 Summary

The experimental evaluation has shown that the optimizations presented in this paper have a significant impact on the performance of more complex applications. This can lead to potential performance losses of over 20 times if the optimizations discussed in this paper are disabled.

## 8. Conclusion

This paper has presented LIFT, a functional data-parallel intermediate representation for OpenCL. The LIFT IR abstracts away many of the OpenCL concepts and optimizations patterns typically found in hand-written code. The functional nature of LIFT makes it an ideal target for existing high-level approaches based on parallel patterns.

By design, LIFT preserves high-level semantic information which can be exploited by the LIFT compiler to generate efficient OpenCL code. However, as seen in this paper, generating efficient code is far from trivial and requires the careful application of optimizations such as array access simplification. Our evaluation shows that these optimizations are crucial to achieve high performance and produce code on par with hand-tuned kernels.

## References

- [1] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, J. Absar, S. van Haastregt, A. Kravets, et al. Pencil: A platform-neutral compute intermediate language for accelerator programming. PACT. ACM, 2015.
- [2] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for cuda. *GPU Computing Gems Jade Edition*, pages 359–371, 2011.
- [3] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. PACT, 2011.
- [4] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. D. Sa, C. R. Aberger, and K. Olukotun. Have abstraction and eat performance, too: optimized heterogeneous computing with parallel patterns. In B. Franke, Y. Wu, and F. Rastello, editors, *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pages 194–205. ACM, 2016.
- [5] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: Compiling an embedded data parallel language. PPoPP. ACM, 2011.
- [6] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. DAMP. ACM, 2011.
- [7] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press. ISBN 0-262-53086-4.
- [8] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, 1989.
- [9] A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. NOVA: A functional language for data parallelism. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, ARRAY, 2014*. doi: 10.1145/2627373.2627375.
- [10] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). PLDI. ACM, 2012.
- [11] T. Henriksen, M. Elsmann, and C. E. Oancea. A hybrid approach to size inference in futhark. FHPC, 2014.
- [12] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. INSPIRE: the insieme parallel intermediate representation. In C. Fensch, M. F. P. O’Boyle, A. Sez nec, and F. Bodin, editors, *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, pages 7–17. IEEE Computer Society, 2013.
- [13] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun. Locality-aware mapping of nested parallel patterns on GPUs. MICRO. IEEE, 2014.
- [14] R. Leißa, M. Köster, and S. Hack. A graph-based higher-order intermediate representation. CGO. IEEE, 2015.
- [15] P. Maier, M. Morton, and P. Trinder. Towards an adaptive skeleton framework for performance portability. IFL, 2015.
- [16] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP, 2013*. doi: 10.1145/2500365.2500595.
- [17] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. PLDI. ACM, 2013.
- [18] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A portable skeleton library for high-level GPU programming. IPDPSW. IEEE, 2011.
- [19] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. ICFP. ACM, 2015.
- [20] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM TECS*, 13(4s), 2014.
- [21] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO, 2009*.
- [22] Y. Zhang and F. Mueller. HiDP: A hierarchical data parallel language. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO, 2013*. doi: 10.1109/CGO.2013.6494994.