High Performance Stencil Code Generation with Lift

Bastian Hagedorn
University of Münster
Münster, Germany
b.hagedorn@wwu.de

Larisa Stoltzfus
The University of Edinburgh
Edinburgh, Scotland, United Kingdom
larisa.stoltzfus@ed.ac.uk

Michel Steuwer
University of Glasgow
Glasgow, Scotland, United Kingdom
michel.steuwer@glasgow.ac.uk

Sergei Gorlatch
University of Münster
Münster, Germany
gorlatch@wwu.de

Christophe Dubach
The University of Edinburgh
Edinburgh, Scotland, United Kingdom
christophe.dubach@ed.ac.uk

Abstract

Stencil computations are widely used from physical simulations to machine-learning. They are embarrassingly parallel and perfectly fit modern hardware such as Graphic Processing Units. Although stencil computations have been extensively studied, optimizing them for increasingly diverse hardware remains challenging. Domain Specific Languages (DSLs) have raised the programming abstraction and offer good performance. However, this places the burden on DSL implementers who have to write almost full-fledged parallelizing compilers and optimizers.

Lift has recently emerged as a promising approach to achieve *performance portability* and is based on a small set of reusable parallel primitives that DSL or library writers can build upon. Lift's key novelty is in its encoding of optimizations as a system of extensible rewrite rules which are used to explore the optimization space. However, Lift has mostly focused on linear algebra operations and it remains to be seen whether this approach is applicable for other domains.

This paper demonstrates how complex multidimensional stencil code and optimizations such as tiling are expressible using compositions of simple 1D Lift primitives. By leveraging existing Lift primitives and optimizations, we only require the addition of two primitives and one rewrite rule to do so. Our results show that this approach outperforms existing compiler approaches and hand-tuned codes.

CCS Concepts • **Software and its engineering → Parallel programming languages**; **Compilers**;

Keywords Code Generation, Stencil, GPU Computing, Performance Portability, Lift

CGO'18, February 24–28, 2018, Vienna, Austria

ACM Reference Format:
Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3168824

1 Introduction

Stencils are a class of algorithms which update elements in a multi-dimensional grid based on neighboring values using a fixed pattern. They are used extensively in various domains such as medical imaging (*e.g.*, SRAD), numerical methods (*e.g.*, Jacobi) or machine-learning (*e.g.*, convolution neural networks). Stencils are part of the original "seven dwarfs" [2] and are considered one of the most relevant classes of high-performance computing applications.

However, efficient programming of stencils for parallel accelerators such as Graphics Processing Units (GPUs) is challenging even for experienced programmers. Hand-optimized high-performance stencil code is usually written using low-level programming languages like OpenCL or CUDA. Achieving high-performance requires expert knowledge to manage every hardware detail. For instance, special care is required on how parallelism is mapped to GPUs or how data locality is exploited with local memory to maximize performance.

Domain Specific Languages (DSLs) and high-level library approaches have been successful at simplifying HPC application development. These approaches are based on algorithmic skeletons [9] which are recurring patterns of parallel programming. While these raise the abstraction level, they rely on hard-coded, not performance portable implementations. Alternative approaches are based on code generation, which places a huge burden on the implementers who have to reinvent the wheel for each new application domain.

Lift [38] is a novel code generation approach based on a high-level, data-parallel intermediate language whose central tenet is performance portability. It is designed as a target for DSLs and library authors, and exploits functional

principles to produce high-performance GPU code. Applications are expressed using a small set of functional primitives and optimizations are all encoded as formal, semantics-preserving rewrite rules. These rules define an optimization space which is automatically searched for high-performance code [42]. This approach liberates programmers and DSL implementers from the tedious process of re-writing and tuning their code for each new domain or hardware.

This paper shows how stencil code and optimizations are expressible in Lift, reusing its existing machinery for managing parallelism, memory hierarchy and optimizations. Surprisingly, only two new primitives were added to Lift for neighborhood gathering and boundary condition handling. By composing these 1D primitives, complex multi-dimensional stencils are expressible, demonstrating the extensibility of the Lift approach to new application domains.

The paper also shows how stencil-specific optimizations are expressible using existing rewrite-rules in Lift such as overlapped tiling. This only requires the addition of one new rule that handles the newly introduced primitives. By reusing the existing exploration mechanism, we can automatically generate high-performance stencil code for AMD, NVIDIA and ARM GPUs. Our results show that this approach is highly competitive with hand-written implementations and with the state-of-the-art PPCG polyhedral GPU compiler.

This paper makes the following contributions:

1. We show how complex multi-dimensional stencils are expressible using Lift's existing primitives with the addition of only two new primitives;
2. We formalize and implement a stencil-specific optimization – overlapped tiling – as a rewrite rule;
3. We demonstrate that this approach generates high-performance code for several stencil benchmarks.

The paper is organized as follows. Section 2 motivates our work. Section 3 shows how stencil computations are expressed in Lift. Section 4 presents stencil-specific optimizations in Lift expressed as rewrite-rules. Section 5 explains how Lift expressions are compiled to efficient OpenCL code. Section 7 provides experimental evidence that this approach produces high-performance code on a selection of GPUs. Finally, Sections 8 and 9 present related work and conclude.

2 Motivation

The advent of Graphics Processing Units over the past decade have been the first sign of an increasing trend of diversity in computer hardware. The end of Dennard scaling and Moore's law forces computer architects to specialize their design for increased performance and efficiency. Traditional multi-core CPUs from Intel and AMD are now challenged by more energy-efficient designs by ARM, massively parallel architectures such as GPUs, and accelerators such as the Xeon Phi. This diversity in hardware requires massive changes for software as traditional, sequential implementations are hard to automatically adapt to this zoo of modern architectures.

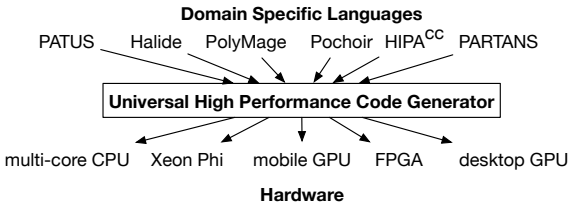

Figure 1. Vision of a high performance code generator used as a universal interface between DSLs and hardware.

2.1 A Solved Problem: High-Level Programming Abstractions for Stencils

Domain specific languages (DSLs) and libraries help application developers target modern hardware, shielding them from the ever changing landscape. They are commonly accepted as being part of the solution to address the performance portability challenge. They are widely used in stencil computations, which has been extensively – and successfully – studied in terms of application-specific optimizations in the high performance computing community. High-level framework such as Halide [33] are designed specifically to express stencil computations in a functional style, fuse multiple operations and generate parallel GPU code automatically. Similarly, PolyMage [31] fuses multiple stencil operations and uses the polyhedral model to produce parallel CPU code. These approaches are particularly good at optimizing long pipelines of stencil operations typically found in image processing applications.

While the use of DSLs provides a nice solution for the end user, they are costly in terms of compiler development. Each new DSL needs to implement its own backend compiler and optimizer with its own approach to parallelization. This is clearly not sustainable given the number of application domains and the ever growing hardware diversity.

2.2 The Real Challenge: Universal High Performance Code Generation

What is needed is a compiler approach which can be reused over a wide range of domains and deliver high performance on a broad set of devices. Figure 1 shows the vision of a universal compiler between DSLs and hardware which was first proposed by Delite [45]. Delite advocates the use of a small set of parallel functional primitives upon which DSL are implemented. A single backend takes care of compiling and optimizing these primitives down to high performance GPU code, enabling all the DSLs implemented on top of Delite to benefit from these optimizations. This type of approach can lead to good performance for many domains on a specific parallel device and in particular for stencil code.

Lift [38, 41, 42] has recently emerged as a novel approach to address the performance portability challenge. It follows

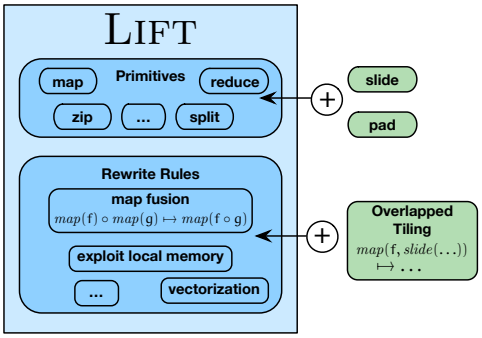

Figure 2. Additions to Lift proposed for supporting stencils. Only two new primitives and a rewrite rule enabling the overlapped tiling optimization are added.

a similar philosophy as Delite by offering a small set of data-parallel patterns used to implement higher-level abstractions. In contrast to Delite, Lift generates high performance code by encoding algorithmic choices and device-specific optimizations as provably correct rewrite rules. This design makes it easy to extend and add new optimizations into the compiler, in contrast to Delite where optimizations are hard-coded for each backend.

Lift has demonstrated that high performance is achievable for linear algebra operations [41]. This paper takes the Lift approach a step further and shows how it can also be applied, with few modifications, to stencil computations. In particular, we show how complex multi-dimensional stencils are expressible by composing a handful of simple 1D primitives. Additionally, we strive to leverage existing functionality in Lift, inheriting the benefits of automatic exploration of algorithmic and device-specific optimizations.

3 Extending Lift for Stencil Computations

Figure 2 shows the extension to Lift for supporting stencil computations which we describe in this and the following section. Only minor additions are required to support stencils and generate high-performance code across multiple parallel devices. We begin by describing the existing Lift primitives we reuse, before introducing the two new primitives *slide* and *pad* which allow us to express stencil computations in a functional style. After discussing a 1D example, we introduce the handling of multi-dimensional stencils which are expressed by composition of the fundamental 1D primitives.

3.1 Existing High-Level Lift Primitives

Lift was introduced in [38], offering a collection of data-parallel functional primitives. Prior work has shown that it is possible to compile these effectively to the GPU [42] using rewrite-rules. The most relevant primitives for stencil applications are shown below with their types which explain how these primitives can be composed.

$$
\begin{aligned}
\textbf{\textit{map}} &: (f : T \to U,\ in : [T]_n) \to [U]_n \\
\textbf{\textit{reduce}} &: (init : U,\ f : (U, T) \to U,\ in : [T]_n) \to [U]_1 \\
\textbf{\textit{zip}} &: (in1 : [T]_n,\ in2 : [U]_n) \to [\{T, U\}]_n \\
\textbf{\textit{iterate}} &: (in : [T]_n,\ f : [T]_n \to [T]_n,\ m : \texttt{Int}) \to [T]_n \\
\textbf{\textit{split}} &: (m : \texttt{Int},\ in : [T]_n) \to [[T]_m]_{n/m} \\
\textbf{\textit{join}} &: (in : [[T]_m]_n) \to [T]_{m \times n} \\
\textbf{\textit{at}} &: (i : Cst,\ in : [T]_n) \to T \\
\textbf{\textit{get}} &: (i : Cst,\ in : \{T_1, T_2, \ldots\}) \to T_i \\
\textbf{\textit{array}} &: (n : \texttt{Int},\ f : (i : \texttt{Int},\ n : \texttt{Int}) \to T) \to [T]_n \\
\textbf{\textit{userFun}} &: (s1 : ScalarT,\ s2 : ScalarT', \ldots) \to ScalarU
\end{aligned}
$$

We write $[T]_n$ for an array with n elements of type T. Note that arrays can be nested and carry their size in their type. We write $\{T_1, T_2, \ldots\}$ for a tuple with component types T_i. Finally, $T \to U$ denotes a function type expecting a value of type T and returning a resulting value of type U.

Map, Reduce, Iterate The *map* primitive applies a function f to all elements of an array and produces a new array of the same length. In Lift, this is the only primitive that expresses data parallelism. *reduce* applies a reduction operator f to an array by traversing it, applying f to the elements and an accumulator variable initialized with the given *init* value. *iterate* performs m iterations of a function f reusing the output produced as an input for the next iteration. While this paper purely evaluates single iteration stencils, the *iterate* primitive can be used to perform multiple iterations.

Zip, Split, Join *zip* creates an array of tuples $\{T, U\}$ by combining two input arrays of the same length. *split* introduces an additional dimension, by splitting the input array into chunks of size m, where m is a positive number evenly dividing the input size n. *join* performs the opposite operation.

Array and Tuple accesses The *at* primitive enables the indexing of arrays with constant (literal) indices. For stencils this is useful for accessing the elements which define the stencil shape. For the rest of this paper, we will write $in[3]$ as syntactic sugar for $at(3, in)$. Similarly, the *get* primitive indexes into the components of a tuple. For instance, $get(2, in)$ returns the second component of tuple in. For the rest of this paper, we will write $in.2$ as syntactic sugar for $get(2, in)$.

Array Constructor This primitive constructs array elements lazily by invoking the function f with index i and array length n. Later we will show this primitive being used for creating masks which can be useful for certain stencils.

UserFun Finally, *userFuns* define arbitrary functions which operate on scalar values. These functions are written in C and are embedded in the generated OpenCL code.

```
1 for(int i = 0; i < N; i++) {
2   int sum = 0;
3   for(int j = -1; j <= 1; j++) { // (a)
4     int pos = i+j;
5     pos = pos < 0   ? 0   : pos; // (b)
6     pos = pos > N-1 ? N-1 : pos;
7     sum += A[pos]; }             // (c)
8   B[i] = sum; }
```

Listing 1. Simple 3-Point Jacobi Stencil in C

3.2 Extensions for Supporting Stencils

It is not possible to express stencil computations in Lift using solely the primitives just described as they are too restrictive. Instead of expressing stencil computations using a single high-level *stencil* primitive, as often seen in other high-level approaches, e.g. [7, 39], in Lift we aim for composability and instead express stencil computations using smaller fundamental building blocks. Using an example, we can show that stencil computations can be decomposed in three steps: consider the 3-point stencil shown in Listing 1 applied on a 1D array A of length N that sums the elements of each neighborhood. As denoted in the comments, stencil computations consist of three fundamental parts:

(a) for every element of the input, a *neighborhood* is accessed specified by the shape of the stencil (line 3);
(b) *boundary handling* is performed which specifies how to handle neighboring values for elements at the borders of the input grid (lines 5 and 6);
(c) finally, *for each* neighborhood their elements are used to compute an output element (line 7).

We have added new primitives to perform the first two steps. Following Lift's design goal, each primitive expresses a single concept and complex functionality is achieved by composition. The first new primitive handles boundary conditions and the second one expresses element grouping.

Boundary Handling with pad *pad* adds l and r elements at the beginning and end of the input array *in*, respectively. A first variant reindexes into the input array, a second variant appends values computed by a user-specified function. For stencil computations, these primitives are used to express what happens when we reach the edge of the data boundary.

Step 1 in Figure 3 visualizes boundary handling with *pad*. The input array on the top is enlarged with one element on each side as highlighted with dashed lines.

The ***pad*** primitive for reindexing has the following type:

$$\textbf{\textit{pad}} : \Big(\begin{array}{l} l : \texttt{Int},\ r : \texttt{Int}, \\ h : (i : \texttt{Int},\ len : \texttt{Int}) \to \texttt{Int}, \\ in : [T]_n \end{array} \Big) \to [T]_{l+n+r}$$

It uses the index function h to map indices from the range $[0, l+n+r]$ into the smaller range of the input array $[0, n]$. The elements added at the boundaries are, thus, elements of the input array and h is used to determine which elements this will be. For instance, by defining the following function:

```
clamp(i, n) = (i < 0) ? 0 : ((i >= n) ? n-1 : i)
```

it is possible to express a clamping boundary condition which artificially extends the original *input* array by two elements to the left and three to the right by repeating the value at the boundary. In the extended version of Lift we write: $\textbf{\textit{pad}}(2, 3, clamp, input)$.

Indexing functions implementing mirroring or wrapping are similarly defined. Indexing functions must not reorder the elements of the input array, but only map indices from outside the array boundaries into a valid array index.

The *pad* primitive which appends values has a very similar type (not shown for brevity), where the function h produces a value which is added to the edges of the array. This variation of *pad* is used to implement constant boundary conditions, or dampening boundary conditions where the out-of-bound value decreases with the distance to the boundary.

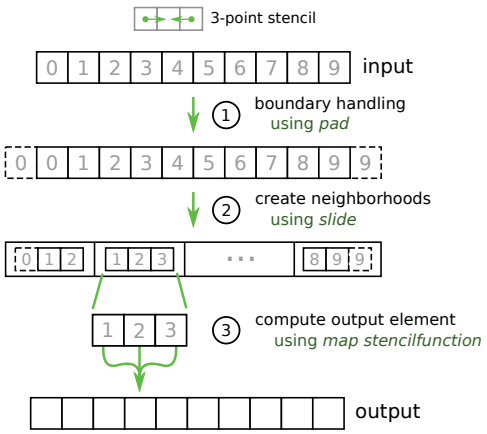

Figure 3. Expressing a stencil in Lift using *pad* for boundary handling, *slide* for creating the neighborhood and *map* to compute the output elements. These three logical steps are compiled into a single efficient OpenCL kernel by Lift.

Creating Neighborhoods with slide The *slide* primitive applies a sliding window of length *size* which traverses past *step* elements. For a one-dimensional 3-point stencil we write: $\textbf{\textit{slide}}(3, 1, input)$.

This creates a nested array, as shown after step 2 in Figure 3, where each element of the outer array is itself an array of three elements. The second element of the first inner array is hereby also the first element of the second array, which corresponds to the notion that we group the first three elements together before we move the sliding window by one element. The type of *slide* can be described as:

$$\textbf{\textit{slide}} : (size : \texttt{Int},\ step : \texttt{Int},\ in : [T]_n) \to [[T]_{size}]_{\frac{n-size+step}{step}}$$

We will later show how this primitive is used to create multi-dimensional neighborhoods.

```
1 val sumNbh = fun(nbh => reduce(add, 0.0f, nbh))
2
3 val stencil =
4   fun( A: Array(Float, N) =>
5     map(sumNbh,                    // (c)
6       slide(3, 1,                  // (a)
7         pad(1, 1, clamp, A))))     // (b)
```

Listing 2. 3-Point Jacobi Stencil expressed in Lift

Computing the Stencil for each Neighborhood with map The *map* primitive is the only way in Lift to express data parallelism. As stencils are naturally data-parallel, we express the last step of the stencil computation using the *map* primitive. This step takes arrays of neighborhood as its input and performs the stencil computation to produce a single output value for each neighborhood.

3.3 One-dimensional Stencil Example in Lift

Listing 2 shows a simple 3-Point Jacobi Stencil expressed in Lift. This is the same example we saw as C code in Listing 1. Due to the functional style of nested function calls, the Lift expression reads from bottom to top. We can see the decomposition in three logical steps: first, boundary handling is performed (line 7) using *pad*; then, the neighborhoods are created (line 6) using *slide*; finally, *map* is used (line 5) to perform the computation for every created neighborhood. The computation is defined as function `sumNbh` in line 1.

It is important to emphasize that the logical distinction of these three steps will not be echoed in the generated OpenCL code. The boundary handling and creation of neighborhoods are not performed by copying elements in memory, but are combined with map in a single step by creating a compiler-internal data structure, called *view* in Lift [42], which influences how data will be read from memory. This is discussed in more detail in Section 5.

3.4 Multi-Dimensional Stencils in Lift

One of the crucial concepts in this paper is the ability to express complex multi-dimensional stencils as compositions of simple one-dimensional primitives. We will now show how we define n-dimensional versions of pad_n and $slide_n$ as compositions of the simple *pad*, *slide*, and *map* primitives we have just seen.

Multi-dimensional stencils are then expressed following the same structure as one-dimensional stencils:

$$\boldsymbol{map_n}(f,\ \boldsymbol{slide_n}(size,\ step,\ \boldsymbol{pad_n}(l,\ r,\ h,\ input)))$$

Boundary handling is performed via $\boldsymbol{pad_n}$ using the function h. Here we present the simple case where the same boundary handling strategy is performed in each dimension. It is straightforward – and supported by our implementation – to do different boundary handlings in each dimension. The $\boldsymbol{slide_n}$ creates a n-dimensional neighborhood, which is then processed by $\boldsymbol{map_n}$.

Multi-Dimensional Boundary Handling Boundary handling in multiple dimensions follows the same idea as in the one-dimensional case. Using nested *map*s, we apply *pad* to inner dimensions. Thus, pad_n is defined recursively:

$$pad_1(l,\ r,\ h,\ input) = pad(l,\ r,\ h,\ input)$$
$$pad_n(l,\ r,\ h,\ input) = map_{n-1}(pad(l,\ r,\ h),\ pad_{n-1}(l,\ r,\ h,\ input))$$

where map_n are n nested *map*s:

$$map_1(f,\ input) = map(f,\ input)$$
$$map_n(f,\ input) = map_{n-1}(map(f),\ input)$$

While the base case is the one-dimensional *pad*, for each higher dimension a *pad* primitive is added where nested *map*s are used to apply it to the innermost dimension.

We provide a simple 2D example for pad_2 using the *clamp* boundary handling which repeats the values at the boundary:

$$pad_2(1,\ 1,\ clamp, \begin{bmatrix}[a,\ b],\\ [c,\ d]\end{bmatrix}) =$$
$$map(pad(1,1,clamp),\ pad(1,1,clamp, \begin{bmatrix}[a,\ b],[c,\ d]\end{bmatrix})) =$$
$$map(pad(1,1,clamp), \begin{bmatrix}[a,\ b],[a,\ b],[c,\ d],[c,\ d]\end{bmatrix}) =$$
$$\begin{bmatrix}[a,\ a,\ b,\ b],\\ [a,\ a,\ b,\ b],\\ [c,\ c,\ d,\ d],\\ [c,\ c,\ d,\ d]\end{bmatrix}$$

After expanding the definition of pad_2, we first apply *pad* to the outer dimension of the two-dimensional array, resulting in an enlarged array where the first and last element – themselves both arrays – are prepended and appended. Then, in the second step, *pad* is applied to the nested dimension using the *map* primitive, which applies *pad* to every nested array resulting in the final two-dimensional array.

Multi-Dimensional Neighborhood Creation The creation of multi-dimensional neighborhoods is more complex than the boundary handling, but follows a similar idea.

For the two-dimensional case, $slide_2$ is defined as:

$$slide_2(size,\ step,\ input) =$$
$$map(transpose,\ slide(size,\ step,\ map(slide(size,\ step),\ input)))$$

We explain this definition using an example:

$$slide_2(2,1, \begin{bmatrix}[a, & b, & c],\\ [d, & e, & f],\\ [g, & h, & i]\end{bmatrix}) =$$
$$map(transpose,\ slide(2,1,\ map(slide(2,1), \begin{bmatrix}[a,\ b,\ c],\ [d,\ e,\ f],\ [g,\ h,\ i]\end{bmatrix}))) =$$
$$map(transpose,\ slide(2,1,\ \begin{bmatrix}[[a,\ b],\ [b,\ c]],\ [[d,\ e],\ [e,\ f]],\ [[g,\ h],\ [h,\ i]]\end{bmatrix})) =$$

$$map(transpose, \begin{bmatrix} [\,[\,[a,\ b],\ [b,\ c]\,],\ [\,[d,\ e],\ [e,\ f]\,]\,], \\ [\,[\,[d,\ e],\ [e,\ f]\,],\ [\,[g,\ h],\ [h,\ i]\,]\,] \end{bmatrix}) = \begin{bmatrix} \begin{bmatrix} \begin{bmatrix} a & b \\ d & e \end{bmatrix}, \begin{bmatrix} b & c \\ e & f \end{bmatrix} \end{bmatrix}, \\ \begin{bmatrix} \begin{bmatrix} d & e \\ g & h \end{bmatrix}, \begin{bmatrix} e & f \\ h & i \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The resulting four-dimensional array is created out of four 2×2 neighborhoods. These are created by applying *slide* to the inner and then the outer dimension, before using *map(transpose)* to switch the two inner dimensions.

We can generalise the definition of $slide_2$ to $slide_n$ for creating n-dimensional neighborhoods. The general structure is similar to the two-dimensional case:

$$slide_n(size,\ step,\ input) = reorderingDimensions(slide(size,\ step, map(slide_{n-1}(size,\ step,\ input))))$$

We first recursively apply the sliding in one inner dimension for the nested dimension of our n-dimensional input with $map(slide_{n-1})$. Then we apply *slide* to the outermost dimension, so that we now have applied *slide* exactly once to all dimensions. In the last step, we now have to reorder the dimensions, so that the nested dimensions created by the slides are the innermost ones. This is best understood by looking at the types involved. For a three-dimensional array, after applying *slide* in each dimension we obtain an array of this type: $[[[[[[T]_{s_o}]_o]_{s_n}]_n]_{s_m}]_m$, where s_m and m are the two dimensions resulting from applying *slide* to the outermost dimension. By rearranging the dimensions, we obtain an array of type: $[[[[[[T]_{s_o}]_{s_n}]_{s_m}]_o]_n]_m$, which corresponds to the desired result: a three-dimensional neighborhood. The rearranging is realized purely as a combination of *map* and *transpose* calls which swap individual dimensions.

3.5 A Complex Stencil: Room Acoustics Simulation

Listing 3 shows a real stencil application for modeling room acoustics which was developed by HPC physicists [49] and models the behavior of a sound wave propagating from a source to a receiver in an enclosed 3-dimensional space.

The two inputs used in this benchmark ($grid_{t-1}$ and $grid_t$ in lines 1- 2) indicate previous and current time steps in order to update the state of the room across time. This type of inputs is often found in real-world physical simulations, which span three dimensions for physical space and one for time. The first grid is taken point-by-point, however the second grid uses $slide_3$ to form stencil neighborhoods. The number of neighborhoods correctly matches up to the size of the $grid_t$ input array as the $grid_{t-1}$ input is padded using pad_3 first, so that no out-of-bounds accesses occur. These inputs are then zipped together with their number of neighbors resulting in a tuple of: {VALUE$_{t-1}$, NEIGHBORHOOD$_t$, NUMNEIGHBORS} as seen in lines 12- 14.

```
1  acousticStencil(grid_{t-1}:[[[Float]_m]_n]_o,
2                  grid_t:[[[Float]_m]_n]_o) {
3   map_3(m -> {
4     val sumGrid_{t-1} =
5      m.1[0][1][1] + m.1[1][0][1] + m.1[1][1][0] +
6      m.1[1][1][2] + m.1[1][2][1] + m.1[2][1][1]
7     val numNeighbor = m.2
8     return getCF(m.2, CSTloss1, 1.0f) * ( (2.0f -
9        CST_{l2} * numNeighbor) * m.1[1][1][1] +
10       CST_{l2} * sumGrid_{t-1} - getCF(m.2,
11       CSTloss2, 1.0f) * m.0) },
12  zip_3(grid_t,
13        slide_3(3, 1, pad_3(1,1,1,zero,grid_{t-1})),
14        array_3(m,n,o,computeNumNeighbors))) }
```

Listing 3. Accoustic simulation expressed in LIFT

In lines 4- 6, the stencil is computed by accessing values in the neighborhood using the *at* primitive which is written as `[ ]`. The results are then combined with the other inputs in an equation to model the sound (lines 8–11).

A difficult problems for wave-based simulations is accurate handling of physical obstacles in the room. The variable coefficients (a. k. a. "loss") at the obstacles boundary are handled through the use of a mask, which returns a different value depending on whether it is on an obstacle or not. In LIFT, this mask is calculated on the fly using the *array3d* generator and contains a value at each point in the grid.

3.6 Summary

In this section, we have demonstrated how stencils are expressed in LIFT by extending the set of primitives by two new additions: *pad* and *slide*. Together with existing LIFT primitives, this allows for expressing multi-dimensional stencils which are built from the one-dimensional building blocks.

Crucially, the parallelism found in stencil applications is expressed using the existing *map* primitive, without introducing a special case for stencils. Rewrite rules explaining how to optimally leverage OpenCL hardware using *map* are then reusable for stencil applications as we show next.

4 Expressing Optimizations

This section discusses how stencil-specific optimizations can be expressed as rewrite rules. These new rules are then used together with LIFT's existing rules to explore the implementation space of stencil applications. By applying different rewrites, we can tailor programs to target different architectures, thus, achieving performance portability.

4.1 Exploiting Locality through Tiling

Stencil applications involve local computations which only access elements in a neighborhood. Furthermore, neighboring elements in a grid share large parts of their neighborhoods. Exploiting this locality is the most commonly used and most successful optimization for stencil computations.

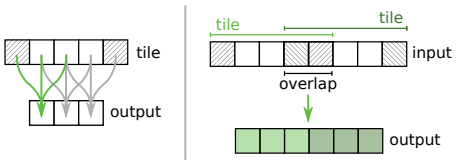

Figure 4. Overlapped tiling for a 3-point Jacobi stencil.

```
1 fun( A: Array(Float, N) =>
2   map(tile => map(sumNbh, slide(3, 1, tile),
3     slide(5, 3,
4       pad(1, 1, clamp, A)))) )
```

Listing 4. 3-Point Jacobi Stencil using overlapped tiling

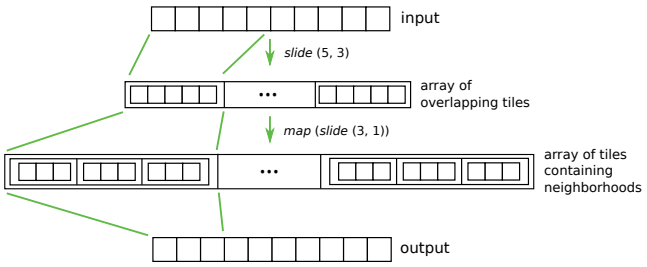

Figure 5. Expressing Overlapped Tiling using *slide*: Applying *slide* to the input creates overlapping tiles. Applying *slide* to every tile creates the required neighborhoods

On GPUs, the fast (but small) local memory can be used as a cache to store a set of neighborhoods where elements are loaded only once from the slow global memory, such that successive accesses are made from the fast local memory.

Traditionally, locality in stencils is exploited using overlapped tiling [17, 19, 52]. The input grid is divided into tiles which overlap at the edges to allow every grid element to access its neighboring elements. The size of the overlap is determined by the size of the neighborhood.

Figure 4 visualizes overlapped tiling for a 3-point one-dimensional Jacobi stencil. The left-hand side shows a single tile of five elements. Here we can see the reuse of data where the highlighted computation on the left shares two elements from the tile with the computation in the middle. On the right-hand side, we can see the overlap in between the left and right tile. These two elements are available in both tiles.

Representing Overlapped Tiling in Lift We reuse the *slide* primitive to represent overlapping tiles. Listing 4 shows the Lift expression of the 3-point Jacobi stencil using tiling.

The *slide* primitive is now used twice: in line 2 a neighborhood is created, as explained earlier, but in line 4 overlapping tiles are created instead of neighborhoods. Due to parameter choice (5 and 3 in this case), 5 elements are grouped in a tile, with 2 elements overlapping with the next tile.

Figure 5 shows the creation of tiles in the first step (by using *slide(5, 3)*), then for each tile we create the local neighborhoods using the *slide* primitive again.

Tiling as a Rewrite Rule Phrasing tiling as a rewrite rule makes it accessible to Lift's automatic exploration process.

Tiling in one dimension is expressible as follows:

$$map(f,\ slide(size, step, input)) \mapsto$$
$$join(map(tile \Rightarrow map(f, slide(size, step, tile)),$$
$$slide(u, v, input)))$$

The parameters u and v have to be selected appropriately, i.e., the difference between the *size* and *step* has to match the difference of u and v: $size\text{-}step = u - v$. Figure 4 visualizes this constraint for the one-dimensional 3-point Jacobi where the neighborhood *size* is 3 (and the *step* is 1). When choosing the size of the tile u, e.g. 5 in the example, v has be selected so that it matches the formula (i.e. 3 in this case) as $3 - 1 = 5 - 3$. This is the only valid choice for v as it determines the overlap created between the tiles which corresponds with the *size* of the original neighborhood. Choosing u and v according to the formula ensures that we end up with the same number of neighborhoods on both sides of the rewrite rule.

To see that this rule is semantics preserving, we can decompose it into two smaller rules:

$$map(f,\ join(input)) \mapsto join(map(map(f), input))$$

and

$$slide(size, step, input) \mapsto$$
$$join(tile \Rightarrow map(slide(size, step, tile)),$$
$$slide(u, v, input))$$

Here it can be seen that the first rule is semantics preserving, as on both sides function f is applied to each element of the two-dimensional *input*. On the left-hand side, this is done by flattening the input and then applying the function, whereas on the right-hand side the function is first applied to each element of the input and then flattened afterwards.

Assuming that u and v are valid parameter choices as described above, the correctness of the second rule is also straightforward. Starting on the right-hand side, we create tiles using the first *slide* primitive. Then, we perform the second *slide* for each created tile, before the *join* removes the outermost dimension and, therefore, resolves the tiles, leaving us with a two-dimensional array equivalent to the array produced by only applying the second *slide*.

Overlapped Tiling in Lift in Multiple Dimensions Our extension to Lift fully supports tiling in higher dimensions. Figure 6 visualizes overlapped tiling in two dimensions.

The optimization rules for tiling higher-dimensional stencils are expressed by reusing the one-dimensional primitives. The rewrite rule covering two-dimensional tiling looks similar to the one-dimensional case when written with the map_2

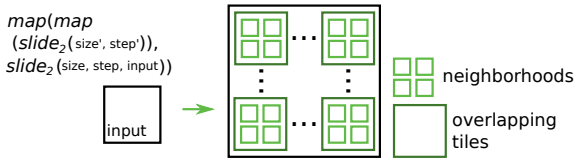

Figure 6. Applying overlapped tiling in two dimensions

and $slide_2$ primitives introduced previously:

$$map_2(f,\ slide_2(size, step, input)) \mapsto map(join,\ join(map(transpose,\ map_2(tile \Rightarrow map_2(f, slide_2(size, step, tile)),\ slide_2(u, v, input)))))$$

However, it is difficult to ensure that the correct dimensions are combined at the very end on the right-hand side.

4.2 Usage of Local Memory

Tiling is used to exploit locality. Modern GPUs have fairly small caches and rely on the programmer explicitly using the fast scratchpad memory, called local memory in OpenCL. This can be cumbersome and does not always provide better performance. Whether or not the local memory is beneficial depends on the hardware architecture and the amount of data reuse in the stencil application.

In Lift, we address these issues by expressing the local memory usage as a rewrite rule. When exploring the optimization space, this rule will be one of many optimization choices applied in the automatic optimization process.

Besides the high-level primitives introduced in Section 3, Lift also defines OpenCL-specific low-level primitives [38] to exploit particular features of OpenCL, such as the use the local memory. The *toLocal* primitive wraps around a function to indicate that this function should write its result into local memory. To copy a single scalar value into local memory, we can use the identity user function *id*, as in: *toLocal(id)*. For copying arrays, we wrap the *map(id)* function in *toLocal*.

As copying data into local memory is always legal inside an OpenCL workgroup, we introduce this rewrite rule:

$$map(id) \mapsto toLocal(map(id))$$

Together with a rule which introduces $map(id)$ at any position, this allows the exploration of copying to local memory as an optimization. Currently, heuristics are used to prevent applying this rule at unfavorable places.

4.3 Loop Unrolling

Unrolling loops is a traditional low-level optimization which can greatly increase performance for certain cases. To explore loop unrolling as an optimization for stencil applications, we make use of a variation of the *reduce* primitive which is unrolled by the Lift compiler. As we saw in the 3-Point Jacobi example in Listing 2, the *reduce* pattern is often used in stencil computations to sum up the values in a neighborhood. The unrolled variation of the reduction is called *reduceUnroll* and has a matching rewrite rule providing it as an optimization choice during exploration. Unrolling is only legal if the size of the input array has a length which is known at compile time. For stencils this is usually the case, as the reduction is applied to a neighborhood which almost always consists of a fixed number of elements.

4.4 Summary

In this section we have shown how stencil optimizations are expressed as rewrite rules, which are then applied by the Lift exploration process. Overlapped tiling in multiple dimensions is expressed by reusing ideas of the simple one-dimensional case. Together with low-level optimizations, such as usage of local memory, we can automatically explore a variety of optimizations for stencil applications.

5 Code Generation

A stencil program expressed using *pad*, *slide*, and *map* is rewritten into a Lift expression of low-level, OpenCL-specific primitives which explicitly encode implementation and optimization choices like the use of local memory. The OpenCL code generation is largely unchanged from the compilation of Lift programs as described in [42].

Lift uses so-called *views* [42] when implementing primitives, which modify data layout without performing computations themselves. These operations are not performed in memory, but define how primitives read input data,

Pad and *slide* are implemented using this same approach. These primitives are integrated with Lift's view system and are not directly compiled to OpenCL code. Instead, the reindexing of computations introduced with *pad* are performed when the padded array is read for the first time. Similarly, the *slide* primitive does not physically copy created neighborhoods into memory. *Slide* guides accesses to elements in a neighborhood to the original array, so that accesses to the same element in different neighborhoods result in memory accesses from the same physical location.

This technique means Lift can build complex - potentially multi-dimensional - abstractions which simplify the implementation of stencil applications compiled to efficient OpenCL code.

6 Experimental Setup

Platforms and Measurement Experiments are conducted using single precision floats on: a Tesla K20c with CUDA 8.0 driver version 367.48; an AMD Radeon HD 7970 with OpenCL version 1.2 AMD-APP (1912.5); and the SAMSUNG Exynos 5422 ARM Mali GPU with OpenCL 1.2 v1.r17p0. The medians of 100 executions are reported measured using the OpenCL profiling API. Data transfer times are ignored since the focus is on the quality of the generated kernel code.

Benchmark	Dim	Pts	Input size	#grids
Stencil2D [10]	2D	9	4098×4098	1
SRAD1 [6]	2D	5	504 × 458	1
SRAD2 [6]	2D	3	504 × 458	2
Hotspot2D [6]	2D	5	8192×8192	2
Hotspot3D [6]	3D	7	512×512×8	2
Acoustic [43]	3D	7	512×512×404	2
Gaussian [36]	2D	25	4096^2 / 8192^2	1
Gradient [36]	2D	5	4096^2 / 8192^2	1
Jacobi2D [36]	2D	5/9	4096^2 / 8192^2	1
Jacobi3D [36]	3D	7/13	256^3 / 512^3	1
Poisson [35]	3D	19	256^3 / 512^3	1
Heat [35]	3D	7	256^3 / 512^3	1

Table 1. Benchmarks used in the evaluation.

Benchmarks The Lift-generated kernels are compared against hand-tuned and automatically-generated kernels from the PPCG [48] state-of-the-art OpenCL polyhedral compiler. We collected hand-written kernels from SHOC (v1.1.5), Rodinia (v3.1) and an OpenCL version of the acoustics simulation code discussed in Section 3.5. We hard-coded each benchmark to perform a single iteration of the stencil computation. We also collected a series of single-kernel C codes that work with the PPCG compiler from a recent study [35, 36], provided by the authors. Table 1 lists these benchmarks along with their key characteristics.

Auto-Tuning Lift exposes optimization choices via rewrite rules which leads to several low-level Lift expressions per benchmark. Each low-level expression contains many parameters that are tunable, controlling for instance: local/global thread counts, tile sizes, how much work a thread performs or how memory accesses are reordered. The parameters of each Lift expression are fine-tuned using the ATF auto-tuning framework [34] which builds on top of OpenTuner [1] and additionally allows constraint specification in the parameter space. The auto-tuner was used for a maximum of three hours for a single program for tuning all expressions.

The PPCG compiler used in our comparison exposes global/local thread counts and tile sizes as tunable parameters in each dimension. We also use ATF and OpenTuner for finding the best combination of these parameters, with again a maximum tuning time of three hours per benchmark. For both Lift and PPCG, the auto-tuner has been enhanced to take into account OpenCL specific constraints (*e.g.*, global thread counts should be a multiple of local thread counts).

7 Evaluation

7.1 Performance Results

This section presents the results of the exploration and auto-tuning process. It also shows the performance achieved by hand-written optimized kernels from the benchmark suites or from HPC experts, as explained previously. Performance is expressed in elements updated per second, which we define simply as the output size divided by the execution time.

Figure 7 shows the performance for the six benchmarks of which we have hand-written implementations. As can be seen, in most cases the Lift generated kernels are comparable to their hand-written counter parts, showing that our compiler approach generates high-performance kernels.

The benchmarks `srad1` and `srad2` seem to under-perform compared to the other benchmarks on the AMD and Nvidia platforms. This is due to the input sizes being too small to saturate these large GPUs (on the smaller ARM GPU, these benchmarks perform as good as the others).

The `Hotspot2D` benchmark is also a clear outlier on the AMD and ARM platforms. On the ARM GPU, the Lift generated version is 2× faster than the hand-written version. On the AMD platform, the performance of the hand-written version is clearly under-performing, especially compared to the performance of the other benchmarks. The Lift generated kernel achieves similar performance than the other benchmarks while being 15× faster than the hand-written version which was originally written for an Nvidia platform. This clearly illustrates the need for code-generation techniques which compile generated code specific to a particular device.

7.2 Performance Comparison of Lift versus PPCG

This section compares Lift with the state-of-the-art PPCG polyhedral GPU compiler [48]. Similar to Lift, PPCG is an approach for compiling data-parallel algorithms starting from a single program and generating optimized code.

Figure 8 shows the relative performance of Lift-generated kernels over PPCG-generated kernels. As explained in section 6, both Lift and PPCG use the same auto-tuning mechanism for a fair comparison. As can be seen, in most cases, the Lift generated code is on-par or clearly outperforms PPCG.

On Nvidia, many benchmarks achieve a speedup of up to 4× over PPCG, such as the `Heat` program with large size, where Lift is 4.3× faster. In this case, the best Lift kernel performs no tiling and each thread only computes 2 elements. On the contrary, the PPCG version looks very different and uses tiling, with each thread processing 512× more elements sequentially than Lift. For `Gradient` on the small size, the PPCG performance is almost as good as Lift. In this case, both versions are similar, use tiling and the difference between the amount of sequential work is only 4×.

On AMD, the results look more uniform, with the exception of the `Poisson` benchmark on the large input. Here again, the best Lift kernel does not use tiling, while the PPCG compiler generates a tiled version of the benchmarks. On the ARM GPU, the results of Lift and PPCG are much closer than on the other platforms, with most of the gain coming again from not using tiling.

Interestingly, none of the Lift kernels generated for the ARM or AMD GPU use tiling, however on Nvidia 33% of the best Lift versions use the tiling optimization. This confirms that different optimization strategies are required for varying program/input sizes as well as for different hardware.

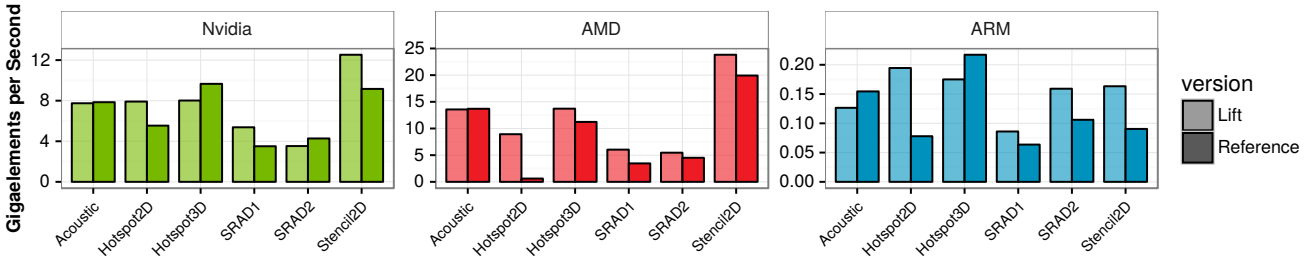

Figure 7. Performance of the Lift generated code and hand-optimized kernels expressed as Giga-elements updated per second.

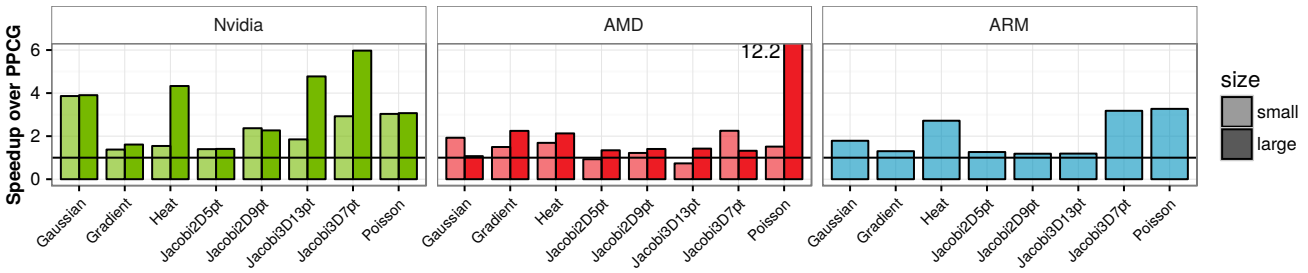

Figure 8. Performance of Lift-generated kernels compared to PPCG-generated kernels. Both approaches auto-tune the kernels for up to three hours per benchmark/input/device. Large input sizes did not fit onto the ARM GPU.

8 Related Work

Stencil-Specific High-Level Programming Approaches There exist many approaches aiming to simplify the programming of stencils. These include stencil-specific DSLs (Domain Specific Languages) or EDSLs (Embedded DSLs) like HLSF [12], Pochoir [46], Halide [33], PolyMage [31] and many others [3, 8, 20, 22, 30]. [35, 36] discuss a stencil-specific compiler with a focus on fusion of stencil operations to minimize data movements. Even more specialized solutions exist for Partial Differential Equations [4, 5] and image processing [14]. Skeleton libraries providing stencil skeletons include SkePU [13], SkelCL [40], MUESLI [24], and PASTHA [25]. Most of these approaches rely on hard-coded and stencil specific implementations.

Optimizations for Stencil Computations There are also many works detailing stencil optimization strategies. These include blocking [32, 50, 51] and tiling approaches [17–19, 23, 26, 37], and other collections of optimizations [11, 15, 28, 44]. Furthermore, multiple auto-tuning frameworks aim to automatically optimize stencils [16, 21, 27].

High Perfomance Code Generation Languages like Accelerate [29], StreamIt [47] or Halide [33] aim to simplify the programming of GPUs through parallel patterns.

Delite [45] is the closest related work. A small set of parallel patterns is compiled and optimized by a single backend into high-performance code. This approach lacks performance portability as device-specific optimizations have to be implemented separately for each platform. In contrast, Lift goes a step further by encoding optimizations in an extensible system of rewrite rules.

9 Conclusions

This paper has shown how stencils and their optimizations are expressible in the data-parallel, hardware-agnostic intermediate language Lift. The language has been extended by two primitives to gather neighboring elements (*slide*) and define boundary conditions (*pad*). Lift can now express complex stencils (like acoustic simulations), which will allow higher-level DSLs to be defined on top of these primitives.

Stencil-specific optimizations are encoded as rewrite rules. Due to the general nature of Lift, we are able to leverage existing Lift optimizations which are also directly applicable to stencil computations. This demonstrates that Lift is easily extensible to new domains with little effort required.

Finally, experimental results provide evidence that this approach generates high-performance stencil code on GPUs. On three platforms, we see that performance is on par with hand-optimized reference implementations. We also compare our approach to the PPCG polyhedral GPU compiler, showing that Lift outperforms it in many cases.

Acknowledgments

We would like to thank the Lift team; Prashant Singh Rawat for help with the PPCG comparison; Ari Rasch and students of the University of Münster for help with the ATF framewok and its integration with Lift. The first author was supported by a HiPEAC collaboration grant. This work was also supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh.

References

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *PACT '14*. ACM, 303–316.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. 2006. *The Landscape Of Parallel Computing Research: A View From Berkeley*. Technical Report. UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[3] Olivier Aumage, Denis Barthou, and Alexandre Honorat. 2016. A Stencil DSEL For Single Code Accelerated Computing With SYCL. In *SYCL 2016 (Workshop) at ACM SIGPLAN PPoPP*.

[4] Peter Bastian, Markus Blatt, Christian Engwer, Andreas Dedner, Robert Klöfkorn, S Kuttanikkad, Mario Ohlberger, and Oliver Sander. 2006. The Distributed And Unified Numerics Environment (DUNE). In *Proc. Of The 19th Symposium On Simulation Technique In Hannover*.

[5] Tobias Brandvik and Graham Pullan. 2010. SBLOCK: A Framework For Efficient Stencil-Based PDE Solvers On Multi-Core Platforms. In *CIT 2010*. IEEE, 1181–1188.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite For Heterogeneous Computing. In *IISWC 2009*. IEEE, 44–54.

[7] Matthias Christen, Olaf Schenk, and Helmar Burkhart. 2011. PATUS: A Code Generation And Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures. In *IPDPS*. IEEE, 676–687.

[8] Milosz Ciznicki, Michal Kulczewski, Piotr Kopta, and Krzysztof Kurowski. 2016. Scaling The GCR Solver Using A High-Level Stencil Framework On Multi-And Many-Core Architectures. In *Parallel Processing And Applied Mathematics*. Springer, 594–606.

[9] Murray I Cole. 1988. *Algorithmic Skeletons: A Structured Approach To The Management Of Parallel Computation*. Ph.D. Dissertation. University of Edinburgh.

[10] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings Of The 3rd Workshop On General-Purpose Computation On Graphics Processing Units*. ACM, 63–74.

[11] Usman Dastgeer and Christoph Kessler. 2012. A Performance-Portable Generic Component For 2D Convolution Computations On GPU-Based Systems. In *MULTIPROG Workshop at HiPEAC-2012*. 1–12.

[12] Fabian Dütsch, Karim Djelassi, Michael Haidl, and Sergei Gorlatch. 2014. HLSF: A High-Level; C++-Based Framework For Stencil Computations On Accelerators. In *Proceedings Of The Second Workshop On Optimizing Stencil Computations*. ACM, 41–4.

[13] Johan Enmyren and Christoph W Kessler. 2010. SkePU: A Multi-Backend Skeleton Programming Library For Multi-GPU Systems. In *Proceedings Of The Fourth International Workshop On High-Level Parallel Programming And Applications*. ACM, 5–14.

[14] Thomas L Falch and Anne C Elster. 2016. ImageCL: An Image Processing Language For Performance Portability On Heterogeneous Systems. *arXiv preprint arXiv:1605.06399* (2016).

[15] Matteo Frigo and Volker Strumpen. 2005. Cache Oblivious Stencil Computations. In *ICS 2005*. ACM, 361–366.

[16] Joseph D Garvey. 2015. *Automatic Performance Tuning Of Stencil Computations On Graphics Processing Units*. Ph.D. Dissertation. University of Toronto.

[17] Tobias Grosser, Albert Cohen, Paul HJ Kelly, J Ramanujam, P Sadayappan, and Sven Verdoolaege. 2013. Split Tiling For GPUs: Automatic Parallelization Using Trapezoidal Tiles. In *Proceedings Of The 6th Workshop On General Purpose Processor Using Graphics Processing Units*. ACM, 24–31.

[18] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P Sadayappan. 2014. The Relation Between Diamond Tiling And Hexagonal Tiling. *Parallel Processing Letters* 24, 03 (2014).

[19] Jia Guo, Ganesh Bikshandi, Basilio B Fraguela, and David Padua. 2009. Writing Productive Stencil Codes With Overlapped Tiling. *Concurrency and Computation: Practice and Experience* 21, 1 (2009), 25–39.

[20] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2013. A Stencil Compiler For Short-Vector SIMD Architectures. In *ICS 2013*. ACM, 13–24.

[21] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. 2010. An Auto-Tuning Framework For Parallel Multicore Stencil Computations. In *IPDPS 2010*. IEEE, 1–12.

[22] Shoaib Kamil, Derrick Coetzee, Scott Beamer, Henry Cook, Ekaterina Gonina, Jonathan Harper, Jeffrey Morlan, and Armando Fox. 2012. Portable Parallel Performance from Sequential, Productive, Embedded Domain-specific Languages. In *PPoPP 2012*. ACM, 303–304.

[23] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. 2007. Multi-Level Tiling: M For The Price Of One. In *SC 2007*. ACM, 51.

[24] Herbert Kuchen. 2002. A Skeleton Library. In *Euro-Par 2002*. Springer, 620–629.

[25] Michael Lesniak. 2010. PASTHA: Parallelizing Stencil Calculations In Haskell. In *Proceedings Of The 5th ACM SIGPLAN Workshop On Declarative Aspects Of Multicore Programming*. ACM, 5–14.

[26] Tareq Malas, Georg Hager, Hatem Ltaief, and David Keyes. 2015. Multi-Dimensional Intra-Tile Parallelization For Memory-Starved Stencil Computations. *arXiv preprint arXiv:1510.04995* (2015).

[27] Azamat Mametjanov, Daniel Lowell, Ching-Chen Ma, and Boyana Norris. 2012. Autotuning Stencil-Based Computations On GPUs. In *CLUSTER 2012*. IEEE, 266–274.

[28] Naoya Maruyama and Takayuki Aoki. 2014. Optimizing Stencil Computations For NVIDIA Kepler GPUs. In *Proceedings Of The 1st International Workshop On High-Performance Stencil Computations, Vienna*. 89–95.

[29] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *ICFP 2013*. ACM, New York, NY, USA, 49–60.

[30] Richard Membarth, Frank Hannig, Jürgen Teich, and Harald Köstler. 2012. Towards Domain-Specific Computing For Stencil Codes In HPC. In *SCC 2012*. IEEE, 1133–1138.

[31] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. In *ASPLOS 2015*. ACM, New York, NY, USA, 429–443.

[32] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D Blocking Optimization For Stencil Computations On Modern CPUs And GPUs. In *SC 2010*. IEEE Computer Society, 1–13.

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language And Compiler For Optimizing Parallelism, Locality, And Recomputation In Image Processing Pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[34] Ari Rasch, Michael Haidl, and Sergei Gorlatch. 2017. ATF: A Generic Auto-Tuning Framework. In *HPCC*. IEEE.

[35] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noel Pouchet, Atanas Rountev, and P. Sadayappan. 2016. Resource Conscious Reuse-Driven Tiling for GPUs. In *PACT 2016*. ACM, 99–111.

[36] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, and P. Sadayappan. 2016. Effective Resource Management for Enhancing Performance of 2D and 3D Stencils on GPUs. In *GPGPU 2016*. ACM, New York, NY, USA, 92–102.

[37] Lakshminarayanan Renganarayana, Manjukumar Harthikote-Matha, Rinku Dewri, and Sanjay Rajopadhye. 2007. Towards Optimal Multi-Level Tiling For Stencil Computations. In *IPDPS 2007*. IEEE, 1–10.

[38] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions To High-Performance OpenCL Code. In *ICFP*. ACM, 205–217.

[39] Michel Steuwer, Michael Haidl, Stefan Breuer, and Sergei Gorlatch. 2014. High-Level Programming Of Stencil Computations On Multi-GPU Systems Using The SkelCL Library. *Parallel Processing Letters* 24, 03 (2014), 1441005.

[40] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. SkelCL - A Portable Skeleton Library For High-Level Gpu Programming. In *Parallel And Distributed Processing Workshops And Phd Forum (IPDPSW), 2011 IEEE International Symposium On*. IEEE, 1176–1182.

[41] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2016. Matrix Multiplication Beyond Auto-Tuning: Rewrite-Based GPU Code generation. In *CASES*. ACM, 15:1–15:10.

[42] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-Parallel IR For High-Performance GPU Code generation. In *CGO*. ACM, 74–85.

[43] Larisa Stoltzfus, Alan Gray, Christophe Dubach, and Stefan Bilbao. 2017. Performance Portability For Room Acoustics Simulations.

[44] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. 2011. Cache Accurate Time Skewing In Iterative Stencil Computations. In *ICPP*. IEEE, 571–581.

[45] Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture For Performance-Oriented Embedded Domain-Specific Languages. *TECS* (2014), 134.

[46] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. 2011. The Pochoir Stencil Compiler. In *SPAA*. ACM, 117–128.

[47] Abhishek Udupa, R Govindarajan, and Matthew J Thazhuthaveetil. 2009. Software Pipelined Execution Of Stream Programs On GPUs. In *CGO*. IEEE, 200–209.

[48] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (Jan. 2013), 23 pages.

[49] Craig Jonathan Webb. 2014. PArallel COmputation TEchniques For VIrtual ACoustics And PHysical MOdelling SYnthesis. (2014).

[50] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. 2009. Efficient Temporal Blocking For Stencil Computations By Multicore-Aware Wavefront Parallelization. In *COMPSAC*, Vol. 1. IEEE, 579–586.

[51] Markus Wittmann, Georg Hager, and Gerhard Wellein. 2010. Multicore-Aware Parallel Temporal Blocking Of Stencil Codes For Shared And Distributed Memory. In *IPDPSW*. IEEE, 1–7.

[52] Xing Zhou. 2013. *Tiling Optimizations For Stencil Computations*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

A Artifact Description

Submission and reviewing guidelines and methodology: *http://cTuning.org/ae/submission.html*

A.1 Abstract

The artifact contains the high performance stencil code generator Lift which is described in the CGO 2018 paper *High Performance Stencil Code Generation with Lift*. Furthermore, this artifact contains the scripts and reference implementations required to reproduce the performance results presented in the paper. To validate the results build Lift and the software used as comparison with the provided scripts, run the benchmarks and, finally, the plotting script to reproduce the results from Figure 7 and 8 in the paper. We also provide scripts to generate OpenCL kernels from high-level Lift expressions, as well as scripts for performing a time-intensive auto-tuning for finding the best performing OpenCL kernel and its numerical parameters on the target platform.

A.2 Description

A.2.1 Check-List (Artifact Meta Information)

- **Program:** The Lift stencil code generator implemented in Scala; The PPCG polyhedral compiler; Benchmark programs implemented in C/C++ using OpenCL
- **Compilation:** With provided scripts
- **Data set:** Provided with the corresponding benchmarks
- **Run-time environment:** Linux with OpenCL
- **Hardware:** Any OpenCL enabled GPU
- **Output:** Runtime in CSV files and plots as PDFs
- **Experiment workflow:** Git clone; build software; run benchmarking scripts; observe performance results
- **Publicly available?:** Yes

A.2.2 How Delivered

The artifact is publicly available and hosted on `gitlab` at: https://gitlab.com/larisa.stoltzfus/liftstencil-cgo2018-artifact/

A.2.3 Hardware Dependencies

An OpenCL enabled GPU is required. In the paper we used a Nvidia Tesla K20c, an AMD Radeon HD 7970, and a ARM Mali GPU on an ODROID-XU4 developer board.

A.2.4 Software Dependencies

Lift requires Java 8, OpenCL, OpenTuner, the Auto Tuning Framework (ATF), and PPCG as its main dependencies.

The software dependencies are listed on the `gitlab` page.

A.2.5 Datasets

Datasets are provided as part of the artifact.

A.3 Installation

After cloning the repository, build scripts are provided for Lift, OpenTuner, the ATF, PPCG, and the used benchmarks.

Detailed installation descriptions are given on `gitlab`.

A.4 Experiment Workflow

After the installation, provided scripts should be used for running all benchmarks and plotting the results.

Detailed descriptions for the experiment workflow are provided on the `gitlab` page.

A.5 Evaluation and Expected Result

The main results of the artifact evaluation is to reproduce the performance comparison given in Figure 7 and 8. Depending on the precise GPU used for evaluation we expect the results to show a similar performance trend as reported in the paper between the Lift generated OpenCL kernels compared to the reference implementations.

The reviewers are invited to investigate the implementation of the Lift compiler, re-generate OpenCL kernels with Lift and perform the auto-tuning process for finding the best OpenCL kernel and parameters for a target hardware.