Tiling Optimizations for Stencil Computations Using Rewrite Rules in Lift

LARISA STOLTZFUS, The University of Edinburgh, United Kingdom
BASTIAN HAGEDORN, University of Münster, Germany
MICHEL STEUWER, University of Glasgow, United Kingdom
SERGEI GORLATCH, University of Münster, Germany
CHRISTOPHE DUBACH, The University of Edinburgh, United Kingdom

Stencil computations are a widely used type of algorithm, found in applications from physical simulations to machine learning. Stencils are embarrassingly parallel, therefore fit on modern hardware such as Graphic Processing Units perfectly. 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 method places the burden on DSL implementers to write almost full-fledged parallelizing compilers and optimizers.

Lift has recently emerged as a promising approach to achieve *performance portability* by using a small set of reusable parallel primitives that DSL or library writers utilize. 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.

This article demonstrates how complex multi-dimensional stencil code and optimizations are expressed using compositions of simple 1D Lift primitives and rewrite rules. We introduce two optimizations that provide high performance for stencils in particular: classical overlapped tiling for multi-dimensional stencils and 2.5D tiling specifically for 3D stencils. We provide an in-depth analysis on how the tiling optimizations affects stencils of different shapes and sizes across different applications. Our experimental results show that our approach outperforms existing compiler approaches and hand-tuned codes.

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

Additional Key Words and Phrases: Code generation, stencil, GPU computing, performance portability, lift

Extension of Conference Paper *High performance stencil code generation with Lift* published at CGO 2018 [22]. This paper presents an extended in-depth discussion of a real-world stencil application, the representation of a optimization specific for 3-dimensional stencils - 2.5D tiling - as a rewrite rule, and additional performance results analyzing the performance characteristics of 2.5D tiling, in particular with respect to different stencil sizes and shapes.

This work was supported by the following: EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (Grant No. EP/L01503X/1), the University of Edinburgh, HiPEAC collaboration grant, and Google Faculty grant.

Authors' addresses: L. Stoltzfus and C. Dubach, The University of Edinburgh, 10 Crichton St, Edinburgh EH8 9AB, UK; emails: {larisa.stoltzfus, christophe.dubach}@ed.ac.uk; B. Hagedorn and S. Gorlatch, University of Münster, Einsteinstraße 62, 48149 Münster, Germany; emails: {b.hagedorn, gorlatch}@wwu.de; M. Steuwer, University of Glasgow, 18 Lilybank Gardens, Glasgow G12 8RZ, UK; email: michel.steuwer@glasgow.ac.uk.

1544-3566/2019/12-ART52
https://doi.org/10.1145/3368858

ACM Reference format:
Larisa Stoltzfus, Bastian Hagedorn, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2019. Tiling Optimizations for Stencil Computations Using Rewrite Rules in Lift. *ACM Trans. Archit. Code Optim.* 16, 4, Article 52 (December 2019), 25 pages.
https://doi.org/10.1145/3368858

1 INTRODUCTION

Stencil algorithms update elements in a multi-dimensional grid based on neighboring values using a fixed pattern. They are part of the "seven dwarfs" [2] and are one of the most relevant classes of high-performance computing applications. Domains such as medical imaging (e.g., SRAD), numerical methods (e.g., Jacobi), or machine learning (e.g., convolutional neural networks) are all heavily dependent on stencils.

Programming of stencils for parallel accelerators such as Graphics Processing Units (GPUs) remains challenging. High-performance stencil code is usually hand-written using low-level programming languages like OpenCL or CUDA and achieving high-performance requires expert knowledge to map parallelism to GPUs or to exploit data locality with local memory.

Domain-specific Languages (DSLs) and high-level libraries drastically simplify application development. Many of these approaches are based on algorithmic skeletons [10], which are recurring patterns of parallel programming. While these solutions raise the abstraction level, they rely on hard-coded, not performance portable implementations. Alternative code generation approaches place a great burden on their implementers who have to reinvent the wheel for each new domain.

Lift [44] 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 writers, and exploits functional principles to produce high-performance GPU code. Applications are expressed using a small set of composable functional primitives and optimizations are all encoded as semantics-preserving rewrite rules. These rules define the optimization space, which is automatically searched [48]. This approach frees programmers and DSL implementers from the tedious process of re-writing and tuning their code for each new domain or hardware.

This article shows how stencil codes and their optimizations are expressible in Lift, reusing its existing machinery wherever possible to manage parallelism, memory hierarchy and optimizations. Two new primitives were added to Lift to allow to express stencils functionally: one primitive for neighborhood gathering and another one for boundary condition handling. By composing these simple 1D primitives, complex multi-dimensional stencils are expressible, demonstrating the extensibility of Lift to new application domains.

Two distinct tiling optimizations have also been added to Lift by through the addition of rewrite rules, a new primitive and additional code generation techniques. The first tiling optimization is classical overlapped tiling, which exploits data reuse and memory locality. The second tiling optimization is 2.5D tiling specifically optimizing 3D stencil codes by exploiting register reuse to provide high performance. Both tiling optimizations are expressed as rewrite rules allowing them to be automatically selected by Lift's optimization process.

This article extends our prior conference publication [22] by extensively discussing the implementation of the new tiling optimization and analyzing its performance characteristics. The 2.5D tiling optimization is implemented using a new primitive, which ensures the precise code generation required for the optimization. To generate efficient OpenCL code, we added new functionality to the Lift code generator with respect to array unrolling and inlining of structs. A detailed analysis of the performance impact of the 2.5D tiling optimization using varying sizes, shapes and

applications demonstrates the importance of a flexible code generator such as Lift capable of applying the best optimization in different settings.

Experimental performance results show that our approach is highly competitive with hand-written implementations and with the state-of-the-art PPCG polyhedral GPU compiler. We are also able to show how the additional 2.5D tiling optimization further improves on our previous results. By reusing Lift's existing exploration mechanism, we automatically generate high-performance stencil code for AMD, NVIDIA, and ARM GPUs.

This article makes the following contributions:

(1) We show how complex multi-dimensional stencils are expressible using Lift's existing primitives with the addition of two fundamental primitives;
(2) We formalize and implement two stencil-specific optimizations—overlapped tiling and 2.5D tiling—as rewrite rules;
(3) We analyze the performance characteristics of the 2.5D tiling optimization on a range of different types of stencils, in particular those relevant to 3D wave model simulations;
(4) We demonstrate this approach generates high-performance code for several stencil codes.

The formalization and implementation of 2.5D tiling and the analysis of its performance characteristics are additional contributions that were not discussed in the original conference publication [22].

2 MOTIVATION

The advent of Graphics Processing Units has been the first sign of an increasing trend of hardware diversity. 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 are now challenged by massively parallel architectures. This diversity in hardware requires massive changes for software as traditional, sequential implementations are hard to automatically adapt to this zoo of architectures.

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

Domain-specific languages 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 have been extensively—and successfully—studied in terms of application-specific optimizations in the high performance computing community. High-level frameworks such as Halide [38] are designed specifically to express stencil computations in a functional style, fuse multiple operations and generate parallel GPU code automatically. Similarly, PolyMage [35] fuses multiple stencil operations and uses the polyhedral model to produce parallel CPU code.

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 reusable compiler approach that covers a wide range of domains and delivers high performance across devices. Figure 1 shows the vision of a universal compiler, which was first proposed by Delite [51]. Delite advocates the use of a small set of parallel functional primitives upon which DSLs are implemented. A single backend takes care of compiling and optimizing these

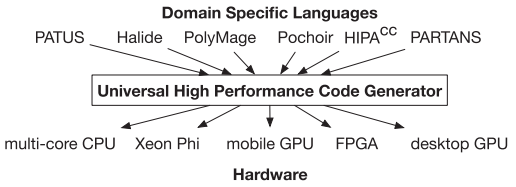

Fig. 1. Vision of a high performance code generator used as a universal interface between DSLs and hardware.

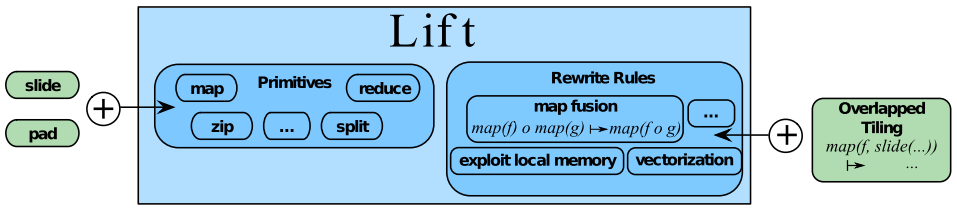

Fig. 2. Additions to Lift proposed for the minimal support of stencils. Only two main new primitives and a rewrite rule enabling general tiling optimization are added.

primitives down to the hardware, enabling all the DSLs implemented on top of Delite to benefit from these optimizations and reach good performance on a specific parallel device.

Lift [44, 47, 48] is a novel approach to achieve performance portability. Similarly to Delite, a small set of data-parallel patterns are used to implement higher-level abstractions. However, in contrast to Delite, Lift generates 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, whereas in Delite optimizations are hard-coded for each backend. A more detailed discussion about this process can be found in our previous work [42].

Lift has demonstrated that high performance is achievable for linear algebra [47]. This article takes Lift a step further and shows how it is also applicable, with few modifications, to stencil computations. 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 extensions to Lift for the general support of stencil computations. 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 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 the composition of fundamental 1D primitives. We finish with a real-world 3D stencil example simulating room acoustics expressed with the introduced high-level primitives.

3.1 Existing High-level Lift Primitives

Lift [44] offers data-parallel primitives which are efficiently compiled to GPUs using rewrite-rules [48]. Primitives relevant to stencils are shown in Figure 3 with their types. $[T]_n$ is an array with n elements of type T, $\{T_1, T_2, \ldots\}$ represents tuple types while $T \to U$ is a function type from T to U.

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

Fig. 3. High-level Lift primitives and their types.

Map, Reduce, Iterate. Map applies function f to all array elements and is the only primitive that expresses data parallelism. *Reduce* applies a reduction operator f to an array, using an accumulator initialized with $init$. *Iterate* performs m iterations of f reusing output as input at the next iteration.

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 number evenly dividing the input size n. *Join* performs the opposite operation.

Array and Tuple accesses. The *at* primitive indexes an array with constant index. This is used for accessing elements from the stencil shape. In this article, we write $in[3]$ as syntactic sugar for $at(3, in)$. *get* indexes into tuples, written as $in.i$ in the rest of this article.

Array Constructors. Arrays can be built lazily by invoking a function f on index i and length n. Section 3.5 shows this primitive is used for creating masks, which is useful for certain stencils.

UserFun. Finally, *userFun*s define arbitrary functions that operate on scalar values and that do not have side effects. These functions are written in C and are embedded in the generated OpenCL code.

3.2 Extensions for Supporting Stencils

It is not possible to express stencil computations in Lift using solely the primitives above. Instead of expressing stencils using a single high-level *stencil* primitive, as is often seen in other high-level approaches, e.g., References [7, 45], in Lift we aim for composability and thus express stencil computations using smaller, fundamental building blocks. Every stencil computation is decomposable into 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 explained below, stencil computations consist of three fundamental parts:

```
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 < 0) ? 0 : ((i+j > N-1) ? N-1 : i+j); // (b)
5     sum += A[pos]; }                                      // (c)
6   B[i] = sum; }
```

Listing 1. Simple 3-Point Jacobi Stencil in C.

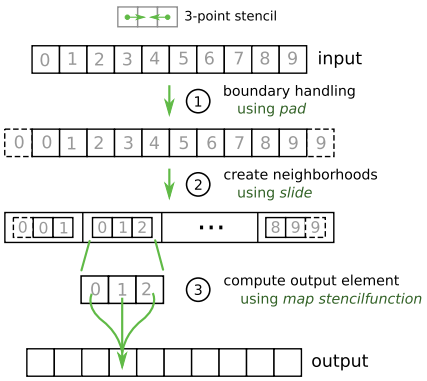

Fig. 4. Expressing a stencil in Lift using *pad* for boundary handling, *slide* for creating the neighborhood and *map* to compute the output elements. These three steps are compiled into a single OpenCL kernel by Lift.

(a) for every input element, a *neighborhood* is accessed specified by the stencil shape (line 3);
(b) *boundary handling* is performed, which specifies how to handle neighboring values for elements at the borders of the input grid (line 4);
(c) finally, *for each* neighborhood, its elements are used to compute an output element (line 5).

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. One variant reindexes into the input array, while 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 4 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:

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

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 that 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 ***pad***$(2, 3, clamp, input)$.

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

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

Creating Neighborhoods with Slide. The *slide* primitive applies a sliding window of length *size* traversing past *step* elements. For a one-dimensional 3-point stencil, we write ***slide***$(3, 1, input)$.

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

Listing 2. 3-Point Jacobi Stencil expressed in Lift.

This creates a nested array, as shown in step 2 in Figure 4, where each element of the outer array is itself an array of three elements. The second element of the first inner array is 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* is

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

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 the array of neighborhoods 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 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 bottom-up. We can see the decomposition in three steps: first, boundary handling is performed (line 5) using *pad*; then, neighborhoods are created (line 4) using *slide*; finally, *map* (line 3) performs the computation for every 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 [48], 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 of this article is the ability to express complex multi-dimensional stencils as compositions of simple 1D primitives. We now show how we define n-dimensional versions of pad_n and $slide_n$ as compositions of the simple *pad*, *slide*, and *map* primitives.

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

$$\textbf{\textit{map}}_{\textbf{\textit{n}}}(f, \textbf{\textit{slide}}_{\textbf{\textit{n}}}(size, step, \textbf{\textit{pad}}_{\textbf{\textit{n}}}(l, r, h, input))).$$

Boundary handling is performed via 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 $slide_n$ creates a n-dimensional neighborhood, which is then processed by map_n.

Multi-dimensional Boundary Handling. This 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 as

$$\textbf{\textit{pad}}_{\textbf{1}}(l, r, h, input) = \textbf{\textit{pad}}(l, r, h, input),$$
$$\textbf{\textit{pad}}_{\textbf{n}}(l, r, h, input) = \textbf{\textit{map}}_{\textbf{n-1}}(\textbf{\textit{pad}}(l, r, h), \textbf{\textit{pad}}_{\textbf{n-1}}(l, r, h, input)),$$

where map_n are n nested *map*s:

$$\begin{aligned}\boldsymbol{map_1}(f, input) &= \boldsymbol{map}(f, input),\\ \boldsymbol{map_n}(f, input) &= \boldsymbol{map_{n-1}}(map(f), input).\end{aligned}$$

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 an example for pad_2 using the *clamp*ing, which repeats the values at the boundary:

$$\begin{aligned}\boldsymbol{pad_2}\left(1, 1, clamp, \begin{bmatrix}[a, & b],\\ [c, & d]\end{bmatrix}\right) &= \boldsymbol{map}(\boldsymbol{pad}(1, 1, clamp), \boldsymbol{pad}(1, 1, clamp, [\,[a, b], [c, d]\,]))\\ &= \boldsymbol{map}(\boldsymbol{pad}(1, 1, clamp), [\,[a, b], [a, b], [c, d], [c, d]\,]) = \begin{bmatrix}[a, & a, & b, & b],\\ [a, & a, & b, & b],\\ [c, & c, & d, & d],\\ [c, & c, & d, & d]\end{bmatrix}.\end{aligned}$$

After expanding pad_2, *pad* is applied to the 2D array outer dimension, resulting in an enlarged array where the first and last element—themselves both arrays—are prepended and appended. Then, *pad* is applied using *map*, which pads every nested array resulting in the final 2D array.

Multi-dimensional Neighborhood Creation. Neighborhood creation is more complex than boundary handling, but follows a similar idea. For the two-dimensional case, $slide_2$ is defined as

$$\boldsymbol{slide_2}(size, step, input) = \boldsymbol{map}(transpose, \boldsymbol{slide}(size, step, \boldsymbol{map}(\boldsymbol{slide}(size, step), input))).$$

We explain this definition using an example:

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

The resulting 4D array is created out of four 2×2 neighborhoods. These are created by applying *slide* to the inner and outer dimensions, before using *map(transpose)* to switch the two inner dimensions.

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

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

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, *slide* is applied to the outermost dimension, so that we now have applied *slide* exactly once to all dimensions. In the last step, we 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: $[[[[[[T]_{s_o}]_{s_n}]_{s_m}]_o]_n]_m$, which corresponds to 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

Lift can handle complex real-world stencils. Listing 3 shows a stencil application for modeling room acoustics developed by HPC physicists [55] in Lift. A sound wave propagates from a source to a receiver in a three-dimensional space using Finite-difference Time-domain methods. The coefficients used in calculating the physical properties of the sound wave are adjusted according to the reflection when encountering a physical boundary (the walls are the boundaries of the grid). The computation is based on a discretized version of the 3D wave equation to simulate the energy at different points.

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

Listing 3. Acoustic simulation expressed in Lift.

The two inputs used in this benchmark ($grid_{t-1}$ and $grid_t$ on line 1) indicate previous and current time steps 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 $grid_{t-1}$ input is padded using pad_3, so that no out-of-bounds accesses occur. These grids are zipped together along with an on-the-fly array generator, which calculates the number of neighbors for a given point (`computeNumNeighbors`) as seen on line 8 (the zip_3 and $array_3$ primitives used are similar versions to the one-dimensional primitives that work on 3D data). This results in a tuple of: $\{\text{VALUE}_{t-1}, \text{NEIGHBORHOOD}_t, \text{NUMNEIGHBORS}\}$.

On lines 3 and 4, the stencil is computed by accessing neighboring values using the *at* primitive (written `[ ]`), which are combined with other inputs in an equation to model the sound (lines 6 and 7).

A difficult problem for wave-based simulations is handling physical obstacles in the room. This simplified version uses state-free boundary conditions, which involve variable coefficients, however the same ideas could be applied to more complicated state conditions. 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. The coefficients are then calculated using the *getCF* function as can be seen on line 6. For those values that are on the border (i.e., *numNeighbors* < 6), a lossy coefficient is used in the equation (*CSTloss1* or *CSTloss2*).

This 7-point stencil benchmark represents the most simplistic version of an acoustics simulation, but there are other schemes that bring improved accuracy [23]. These schemes use either "leggy" (i.e., higher-order) or dense schemes that involve more memory accesses, thus modeling the wave more accurately. In Section 8.2, we will take a closer look at these variations of stencil shapes.

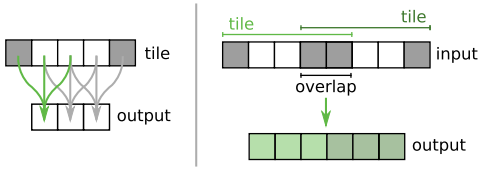

Fig. 5. Overlapped tiling for 3-point Jacobi Stencil.

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

Listing 4. Algorithm for Overlapped Tiling.

3.6 Summary

This section has shown how stencils are expressed in Lift by adding two new primitives: *pad* and *slide*. Together with existing Lift primitives, this allows for expressing multi-dimensional stencils built from the one-dimensional building blocks. We have also discussed stencils in a physics simulation and the importance of building tools and optimizations that work for these codes.

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 stencil-specific optimizations and how they are expressed as rewrite rules in Lift. These new rules are used together with Lift's existing rules to explore the implementation space of stencil applications. By applying different rewrites, programs can be tailored to target different architectures, thus, achieving performance portability.

4.1 Exploiting Locality through Classical Overlapped Tiling

Stencil applications involve local computations that only access elements in a neighborhood. Nearby elements in a grid share large parts of their neighborhoods. Exploiting this locality is the most commonly used and successful optimization for stencil computations. On GPUs, the fast (but small) local memory is used 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.

Locality is traditionally exploited using overlapped tiling [19, 21, 60]. Although this optimization is most often used for time tiling, performance benefits can still be seen when analyzing single-step stencil computations. However, the same approach can be used for iterative stencils. An input grid is divided into overlapping tiles allowing grid elements to access neighboring values. The overlap size is determined by neighborhood size. Figure 5 shows overlapped tiling for a 3-point 1D Jacobi stencil. The left-hand side shows a tile of five elements. The reuse of data can be seen where the highlight on the left shares two elements from the tile with the middle computation. On the right-hand side, overlap in between the left and right tile can be seen. These two elements are available in both tiles.

Representing Overlapped Tiling in Lift. The *slide* primitive is reused to represent overlapping tiles. Listing 4 shows the Lift expression of the 3-point Jacobi stencil using tiling. The *slide* primitive is used twice: in line 3 a neighborhood is created, but in line 5 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 6 shows the creation of tiles in the first step (using *slide(5, 3)*), then for each tile we create the local neighborhoods using *slide* again.

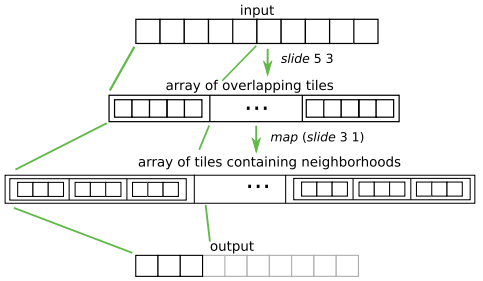

Fig. 6. Applying *slide* to the input creates tiles. Applying *slide* to tiles creates neighborhoods.

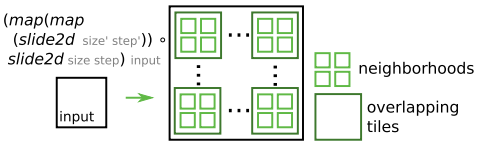

Fig. 7. Applying overlapped tiling in 2D.

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:

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

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 5 visualizes this constraint for a neighborhood *size* of 3 and where *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.

Decomposing this rule into two smaller rules shows that it is semantics preserving:

$$\textbf{\textit{map}}(f, \textbf{\textit{join}}(input)) \mapsto \textbf{\textit{join}}(\textbf{\textit{map}}(\textbf{\textit{map}}(f), input)),$$
$$\textbf{\textit{slide}}(size, step, input) \mapsto \textbf{\textit{join}}(tile \Rightarrow \textbf{\textit{map}}(\textbf{\textit{slide}}(size, step, tile)), \textbf{\textit{slide}}(u, v, input)).$$

Here it can be seen that the first rule preserves semantics 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 7 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 and $slide_2$ primitives introduced previously:

$$\textbf{\textit{map}}_2(f, \textbf{\textit{slide}}_2(size, step, input)) \mapsto$$
$$\textbf{\textit{map}}(join, join(\textbf{\textit{map}}(transpose, \textbf{\textit{map}}_2(tile \Rightarrow \textbf{\textit{map}}_2(f, slide_2(size, step, tile)), \textbf{\textit{slide}}_2(u, v, input))))).$$

4.1.1 Usage of Local Memory. Local memory is key to gaining high performance for overlapped tiling. Modern GPUs have small caches and the programmer must explicitly use the fast scratchpad memory (called local memory) in OpenCL. This can be cumbersome and does not always provide

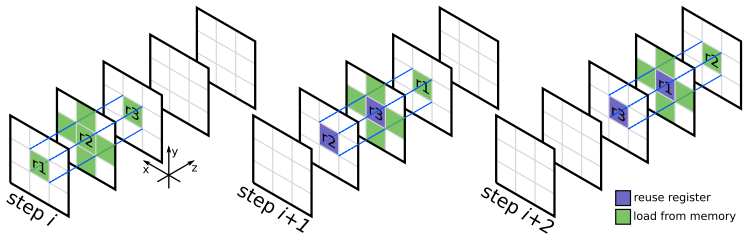

Fig. 8. A visual representation of the 2.5D tiling optimization for a single thread loading memory into registers `r1`, `r2`, and `r3`, which are reused over multiple iteration steps.

better performance. The benefit of local memory depends on the hardware architecture and how much data reuse there is in the stencil application. We address these issues in Lift by using a rewrite rule to express local memory use, which can be used on its own or in conjunction with tiling. This rule is 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 [44] 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*.

Copying into local memory is legal inside workgroups as described by the following rewrite rule:

$$\boldsymbol{map}(id) \mapsto toLocal(\boldsymbol{map}(id)).$$

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

4.2 Exploiting Spatial Locality of 3D Stencils Through 2.5D Tiling

2.5D tiling is an optimization for 3D stencils, which iterates over two spatial dimensions in parallel and the third dimension sequentially [36, 59]. This optimization allows 3D stencils to exploit locality in the third dimension and avoid costly redundant memory accesses. Parallelism is thus only exploited for the XY plane. Figure 8 shows this technique for a single thread. The highlighted squares represent the values accessed by a thread at each iteration of the stencil. The dark centered squares show the values that are reused across iteration and are, therefore, re-loaded from registers and not each time from memory. Because threads only iterate over two dimensions, instead of the typical three for 3D stencils, parallelism is essentially traded for fewer memory accesses (but increased register pressure). We will discuss the impact of register pressure more in Section 7.

Representing 2.5D Tiling in Lift. Supporting this optimization in Lift is achieved by combining the existing functionality of *map* and *slide*. In particular, we use the *mapseq* primitive, a version of *map* that iterates sequentially. To support a moving window, we added a third primitive: *mapseqslide*. This primitive has the following type:

$$\boldsymbol{mapseqslide} : \left(f : T \to U, size\text{: } \texttt{Int}, step\text{: } \texttt{Int}, in : [T]_n\right) \to [U]_{\frac{n-size+step}{step}}.$$

Mapseqslide implements a moving window to iterate over tiles sequentially. The primitive can be used on any number of dimensions and the loop in dimension *N-1* will be sequential. That is, in 1D (or ".5 tiling") there will only be a single sequential loop, for 2D (or "1.5 tiling") there will be one parallelizable loop and one sequential loop. However, the remainder of this article focuses on the optimization for 3D stencils with 2.5D tiling, as this is where speedups have been known to occur.

2.5D Tiling as a Rewrite Rule. The *mapseqslide* primitive can replace any *mapseq* following a *slide*. We explain the addition of these rewrite rules for the 2.5D case in particular. To add 2.5D tiling as a rewrite rule requires matching against three nested *maps* and three nested *slides*. The rewrite rule in three dimensions is described as follows:

$$\boldsymbol{map_3}\,(f,\ \boldsymbol{slide_3}(size,step,input)) \mapsto \boldsymbol{map_2}\,(\boldsymbol{mapseqslide}\,(f,size,step,\boldsymbol{slide_2}(size,step,input))).$$

Memory Coalescing. For the new *mapseqslide* primitive to generate performant code, care must be taken when data is passed to the *mapseqslide* primitive. Doing so in dimensions greater than one requires reordering the data through one or more transposes for the outer-most dimension to be the first one from the perspective of the primitive. The outer dimension is then iterated over first, which retains memory coalescing and ensures the same stencil shape is retained as a normal operation. Similar to the process of multi-dimensional slides described in Section 3.4, transposes are added to the data input to the primitive in the same fashion relative to the number of dimensions used. As we focus on 2.5D tiling in particular for three dimensional stencils, we describe the approach for ensuring the correct ordering for the 3D case below:

$$\boldsymbol{mapseqslide}(size,\ step,\ f,\ \boldsymbol{transpose}\,(\boldsymbol{map}\,(\boldsymbol{transpose}\,(\boldsymbol{slide_2}\,(size,\ step,\ \boldsymbol{map}\,(\boldsymbol{transpose}\,(\boldsymbol{transpose}\,(input)))))))).$$

This approach takes an input array of type$[[[T]_m]_n]_o$, transposes it to create $[[[T]_m]_o]_n$, then maps a transpose to create $[[[T]_o]_m]_n$. A $slide_2$ on the input then produces $[[[[[T]_o]_{s_m}]_{s_n}]_m]_n$. Once inside the *mapseqslide*, the matrices of size $s_m \times s_n$ iterate over columns of length O (i.e., $[[[T]_o]_{s_m}]_{s_n}]$). This is then transposed twice more with another *map(transpose)* followed by a *transpose* resulting in $[[[T]_{s_m}]_{s_n}]_o$, which can then be slided and mapped over in the correct direction. All transposes are then undone before the resulting data is written to the output.

Loop Unrolling. Loop unrolling is a traditional low-level optimization that can greatly increase performance for certain cases. It is crucial for ensuring high performance for the 2.5D tiling optimization, as we will discuss further in Section 5.2. However, we can also utilize loop unrolling as an isolated optimization, similar to local memory. To explore this for stencil applications, we use a variation of the *reduce* primitive that is unrolled by the Lift compiler. As seen in the 3-Point Jacobi example in Listing 2, the *reduce* pattern is often used in stencil computations to sum up values in a neighborhood. The unrolled variation of reduction is called *reduceUnroll* and has a matching rewrite rule making it an optimization choice during exploration. Unrolling is only legal if the size of the input array has a known length at compile time. For stencils, the reduction is applied to a neighborhood that almost always consists of a fixed number of elements.

4.3 Summary

This section has 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. 2.5D tiling is built using an extension of existing primitives and uses a rolling window over the input array. Together with low-level optimizations, such as usage of local memory and loop unrolling, Lift is capable of automatically exploring a variety of optimizations for stencil applications.

5 CODE GENERATION

Lift's exploration process automatically rewrites a stencil program expressed using *pad*, *slide*, and *map* into a Lift expression of low-level, OpenCL-specific primitives, which explicitly encode implementation and optimization choices, such as tiling. In addition to the OpenCL code generation described in [48], we describe in this section additions to the code generator for producing

efficient OpenCL code for the new primitive and ensuring good performance for the new tiling optimization.

5.1 Views

Lift uses an intermediate compiler data-structure called a *view* [48] when implementing primitives, which modifies the data layout without performing any computation itself. These operations are not performed in memory, but influence how successive primitives read input data.

Pad and *slide* are implemented using this 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 accessed 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.

5.2 Generating Efficient Code for 2.5D Tiling

There are three stages in the code generation for *mapseqslide*: (1) initialize values, (2) update values, and (3) swap values. The code generator implements a rolling window within *mapseqslide* where each value is stored in private memory (registers). This provides the data reuse, which leads to performance benefits. However, this necessitates that arrays in private memory are automatically unrolled into scalar values that are guaranteed to be stored in private memory.

Unrolling arrays in private memory. Array unrolling is achieved by creating private scalar variables for each value in the array. The array size must be statically known for this optimization. The code generator automatically unrolls arrays until there are no more dimensions to unroll. An example of the resulting code for 2.5D tiling generated with the *mapseqslide* can be seen in Listing 5, which shows the generated C code for a 2D example for a 5-point stencil when rolling a window over the Y dimension. On lines 2–4 the unrolled values of the rolling window are initialized, line 6 loads the new value into the window, and finally on line 8 the rolling window values swap.

```
1 for (int i = get_global_id(0); i < M ; i = i + get_global_size(0)) {
2    float w10 = in[3]
3    float w11 = in[4]
4    float w12;
5    for (int j = 1; j < N; j = j+1) {
6       w12 = in[i+j*M+M]
7       out[i] = multSumUp(in[i+j*M-1],w10,w11,w12,in[i+j*M+1]);
8       w10 = w11; w11 = w12; /* ... */ } /* ... */}
```

Listing 5. Example of 1.5D Tiling generated C code for a 5-point stencil in 2D.

Inlining structs. In addition to unrolling arrays, structs are inlined into independent private variables as well to ensure these values are stored in registers separately. When inlined, a variable of the `struct Tuple2_int_float { int intValue; float floatValue; }` would result in the individual variables `int tup_i; float tup_f;`.

The algorithm for private memory array unrolling and struct inlining follows subsequent recursive passes of the C AST in the compiler. First a pre-order traversal of the tree is performed to unroll private arrays. Then another pre-order traversal of the tree is performed to inline structs. Passes continue until all private memory has been unrolled and all structs have been inlined.

Table 1. Platforms and Hardware Metrics Used in the Evaluation

Platform	Memory Bandwidth (GB/s)	Peak Performance (Single Precision GFLOPS)	Ridge Point (FLOPS/Byte)
NVIDIA K20c	208	3,524	16.9
AMD Radeon HD 7,970	288	4,096	14.2
ARM Mali T-628	14.9	68	4.5

Table 2. Benchmarks Used in the Evaluation

Benchmark	Dim	Pts	Input Size	#FLOP	#Grids
Stencil2D [11]	2D	9	$4{,}098 \times 4{,}098$	17	1
SRAD1 [6]	2D	5	504×458	35	1
SRAD2 [6]	2D	3	504×458	13	2
Hotspot2D [6]	2D	5	$8{,}192 \times 8{,}192$	4	2
Hotspot3D [6]	3D	7	$512 \times 512 \times 8$	17	2
Acoustic [49]	3D	7	$256 \times 256 \times 202$	13	2
Gaussian [41]	2D	25	$4{,}096^2$ / $8{,}192^2$	50	1
Gradient [41]	2D	5	$4{,}096^2$ / $8{,}192^2$	18	1
Jacobi2D [41]	2D	5/9	$4{,}096^2$ / $8{,}192^2$	10/18	1
Jacobi3D [41]	3D	7/13/19/27	256^3 / 512^3	13/25/37/54	1
Poisson [40]	3D	19	256^3 / 512^3	21	1
Heat [40]	3D	7	256^3 / 512^3	15	1

Benchmark is the benchmark name, *Dim* is the dimensions of the input grids, *Pts* is the number of points in the stencil, *Input Size* is the number of grid points, *#FLOP* is the number of floating point operations in the stencil, and *#Grids* is the number of input grids.

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. More information about the hardware used (including their ridge points from the Roofline Model [57] showing the ratio of computation and memory bandwidth) can be found in Table 1.

Benchmarks. The Lift-generated kernels are compared against hand-tuned and automatically-generated kernels from the PPCG [54] state-of-the-art OpenCL polyhedral compiler. We also 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 [40, 41], provided by the authors. Table 2 lists these benchmarks along with their key characteristics. Subsequent benchmarks used in Section 8.1, include two additional Jacobi 3D stencils using 19 points (with 3 points in each direction) and 27 points (which accesses all points in a $3 \times 3 \times 3$ neighborhood). The 19-point stencil used is the same as the Poisson stencil and the 13-point Jacobi stencil is the same as the PPCG version.

Exploration and Auto-Tuning. Our exploration process is divided into two phases: (1) Rewriting, and (2) Auto-tuning. For this evaluation, our existing rewriting strategy [42] was used without

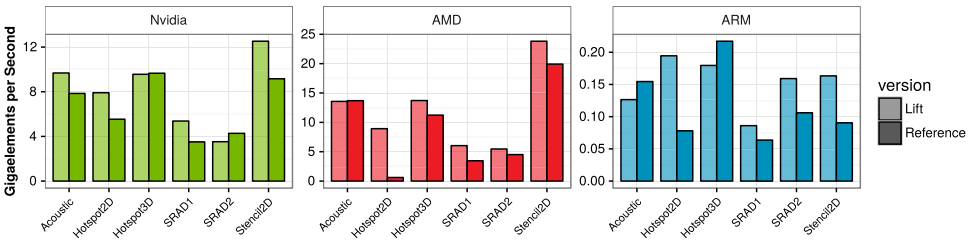

Fig. 9. Performance of the Lift generated code and hand-optimized kernels.

making any adjustments. In the first phase, a derivation tree was created by applying multiple potentially applicable rules, each creating a separate branch. For some optimizations, like overlapped-tiling, multiple "small" rewrite rules are gathered into a larger macro rewrite rule, which encodes a specific optimization sequence. Instead of using the smaller rules in the rewriting phase, these macro rules are used, which additionally helps to limit the number of leaves in the derivation tree, i.e., the low-level expression that serves as input to the auto-tuning phase.

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 [39], 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. ATF and OpenTuner were also used for finding the best combination of these parameters, with the same 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 for hand-tuned kernels. 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 9 shows the performance for six benchmarks of which there are hand-written implementations. In most cases the Lift generated kernels are comparable to their hand-written counterparts, 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.

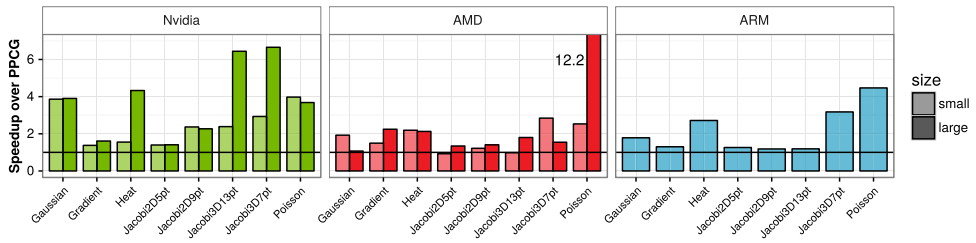

Fig. 10. Performance of Lift-generated kernels compared to PPCG-generated kernels. Both approaches auto-tune kernels for three hours per benchmark/input/device. Large sizes did not fit onto the ARM GPU.

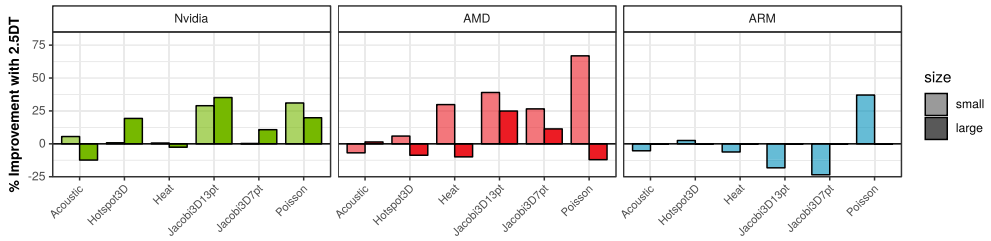

Fig. 11. Effects of using 2.5D Tiling on 3D Benchmarks from Table 2.

7.2 Performance Comparison of Lift versus PPCG

This section compares Lift with the state-of-the-art PPCG polyhedral GPU compiler [54]. Similar to Lift, PPCG generates optimized code for data-parallel algorithms starting from a single program.

Figure 10 shows the relative performance of Lift-generated kernels over PPCG-generated kernels. As described in Section 6, both Lift and PPCG use the same auto-tuning mechanism for a fair comparison. In nearly all 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 uses 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`, small size, the PPCG performance is almost as good as Lift. 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 ARM or AMD GPU use classical tiling, however on Nvidia, 33% of the best Lift versions use this tiling. This confirms that different optimization strategies are required for varying program/input sizes as well as for different hardware.

8 ANALYSIS OF PERFORMANCE CHARACTERISTICS OF 2.5D TILING

8.1 Effects of 2.5D Tiling

This section compares the effects of the 2.5D tiling optimization for 3D stencils found in Table 2. Figure 11 shows the performance impact of using 2.5D tiling compared to the best found kernel without 2.5D tiling. Overall, the most consistent improvement can be seen for stencils with larger numbers of accesses (`Jacobi3D13pt` and `Poisson`). This will be investigated further in Section 8.2.

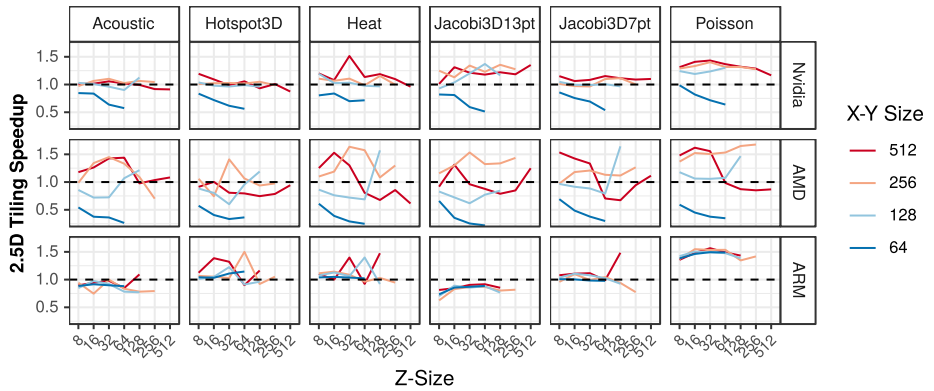

Fig. 12. Speedup using 2.5D Tiling on 3D Benchmarks from Table 2 across z-sizes and xy-domain sizes.

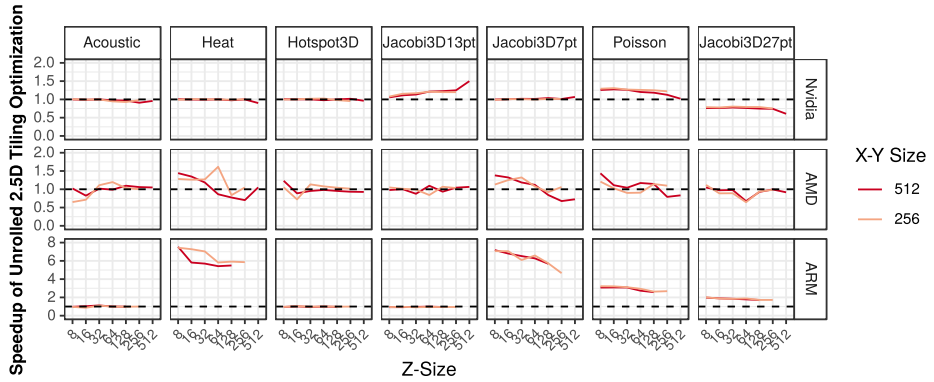

Fig. 13. Performance benefit of Lift-generated when unrolling private arrays and inlining structs.

2.5D tiling achieves the most speedup on the Nvidia and AMD GPUs for both stencil sizes; however, on the Mali GPU, we only see minimal improvement (or decrease in performance) for most stencils, apart from a large improvement for the `Poisson` stencil. This trend is supported by the ridge points in Table 1 showing that the AMD and Nvidia hardware require significantly more computations per byte to unfold their full potential. Therefore, these devices should benefit more from optimizations that save memory loads, such as 2.5D tiling, which is confirmed by Figure 11 showing the Nvidia and AMD hardware gain higher performance than the ARM GPU.

2.5D tiling across different input sizes. Figure 12 shows an in-depth study of the same 3D stencils and how 2.5D tiling affects performance for different input domain sizes by varying the z-sizes and x-y sizes. The speedup shown is again in comparison to the best version found in the space that does not use 2.5D tiling. Although each benchmark behaves uniquely, we note that the speedup tends to improve with domain size on Nvidia and AMD across the benchmarks as also has been reported in [59]. Focusing on the z-size, for example, on Nvidia the performance decreases for larger z values. Threads perform more sequential work at higher z-sizes compared to the baseline (without 2.5D tiling), which uses parallelism in all three dimensions, which explains this behaviour.

Effect of unrolling of private arrays and struct inlining. Figure 13 shows the impact of the code generation optimisations of unrolling private arrays and inlining of structs. This graph shows how the additional compiler passes affect different stencil benchmarks across different platforms and

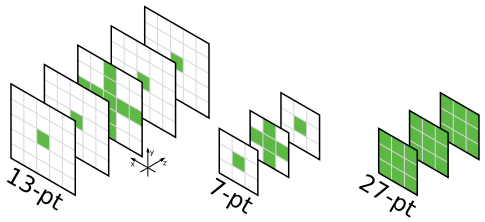

Fig. 14. A comparison of different stencils shapes when applying 2.5D Tiling.

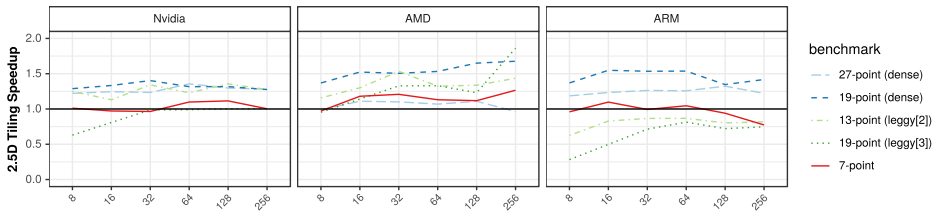

Fig. 15. Speedup using 2.5D Tiling for different z-sizes where domain size = 256 for leggy and dense stencils.

different input sizes. The benchmarks shown are the same as in Figure 12 apart from the addition of a Jacobi 27 point stencil. In particular, it can be seen that the Jacobi 27 point benchmark shows a slowdown on NVIDIA and AMD, while others show similar or much improved performance. Analysis of the NVIDIA PTX code shows a difference in the ordering of instructions, which could explain the performance difference for the stencils with larger numbers of memory accesses. The particularly large performance differences for Heat and Jacobi3D7pt on Mali are due to a much larger number of registers being used in the non-unrolled version as can be seen using the Mali Offline Compiler.

8.2 Stencil Shape Study

This section discusses how performance changes for different stencil shapes. A *leggy* (or higher order) stencil accesses more points along the same axes as the 7-point stencil (a 13-point example can be seen on the left in Figure 14). This has the potential to cut spatial error by half for each additional point used in these schemes [9, 23].

A *dense* stencil accesses more points within the same *3×3×3* neighborhood as a 7-point stencil (a 27-point example is shown on the right in Figure 14) and produces physical simulation results with more evenly distributed errors [23, 37]. Simulations, such as room acoustics, produce more realistic models using either of these schemes. However this additional accuracy comes at the cost of more computation, thus optimizations that work for these types of stencils are critical.

Figure 15 shows the speedups of using the 2.5D tiling optimization for leggy and dense stencils with 13 points, 19 points (dense), 19 points (leggy) and 27 points, as compared to a 7-point stencil. A 13-point stencil represents a leggy stencil with two points of memory access in each direction instead of one. Similarly, a 19-point (leggy) stencil has three memory accesses in each direction. In Figure 15 and Table 3, we denote these as *leggy[n]*. A 19-point (dense) stencil accesses 19 points in a symmetric configuration and a 27-point scheme accesses all points in a *3×3×3* cuboid.

In Figure 15, the 13-point leggy stencil slightly benefits more than the 19-point leggy stencil from 2.5D tiling on AMD and Nvidia, though the difference between them is close on the AMD and ARM GPUs. ARM generally also shows less performance benefits of 2.5D tiling for both of these stencils, as previously observed in Section 8.1.

Table 3. Jacobi3D Benchmark Characteristics Used in the Evaluation

Type		# Memory accesses w/ 2.5DT	Memory access reduction	# Registers w/o	# Registers w/ 2.5DT	Occupancy w/o	Occupancy w/ 2.5DT	OI [FLOPS/Byte] w/o	OI [FLOPS/Byte] w/ 2.5DT
7pt	original	5	1.40 ×	26	29	100.00%	100.00%	0.18	0.25
13pt	leggy[2]	9	1.44 ×	30	59	100.00%	50.00%	0.44	0.62
19pt	dense	9	2.10 ×	43	42	62.50%	62.50%	0.26	0.52
19pt	leggy[3]	13	1.60 ×	38	118	75.00%	25.00%	0.46	0.66
27pt	dense	9	3.00 ×	51	62	56.25%	50.00%	0.38	1.07

The first column shows the type of stencil, the second shows the number of points in the stencil, the third shows the number of points read from memory when using 2.5D tiling, the fourth column shows the overall reduction in memory accesses, the fifth and sixth columns show the profiled number of registers reused across iterations (with / without 2.5D tiling), the seventh and eighth columns show the occupancy of each multiprocessor on NVIDIA K20c (with / without 2.5D tiling for the configuration XY = 256 and Z = 32), and the last two columns show the Operational Intensity (OI) for each of the benchmarks (with / without 2.5D tiling).

For the dense stencils with 19 and 27 points, the performance benefits of 2.5D tiling are larger. 19-point stencils benefit the most from 2.5D tiling across all platforms (as also seen in Figure 12 for Poisson), however the 27-point stencil does not benefit as much as one might expect based on the number of reductions of memory accesses seen in Table 3.

To explain this behaviour, we investigated the reduction in memory accesses and register pressure with and without 2.5D tiling. This information is shown in Table 3. The last two columns show the operational intensity (as proposed by [57]) measured in FLOPS/byte. By comparing this number to the ridge points shown in Table 1, we can see that all stencils are heavily memory bound (which is well known in the community [13]). As the number of reduction of memory accesses is greatest for the 27-point dense stencil, one would expect to see the greatest benefit of 2.5D tiling for this stencil; however, it is the 19-point dense stencil that benefits the most from this optimization. Investigating the register usage of the kernels on the Nvidia GPU showed that the number of registers increases in almost all cases with 2.5D tiling and with number of points in the stencil (from 26 registers up to 118). On GPUs, high numbers of registers limit the number of parallel executing threads compared to the theoretical maximum, which results in a reduced occupancy of the multicore. For the 7-point stencil a perfect occupancy of 100% without and with 2.5D tiling is achieved while the occupancy is lower for the dense 19-point (62.5% without and with tiling) and dense 27-point (56.25% without 2.5D tiling) stencils. For the 27-point stencil 2.5D tiling increases the register usage and results in a reduced occupancy of only 50% limiting the amount of parallelism that is exploited. The 19-point stencil strikes a good balance between the amount of parallelism exploited in the hardware that is not negatively affected by the 2.5D tiling optimization and the number of memory accesses saved leading to the best overall performance. On AMD, profiling has shown that for the 19-point dense stencil the memory unit is saturated the highest of all versions confirming the prior observations and highlighting the importance of the memory systems performance for these memory bound stencil codes.

9 RELATED WORK

High-perfomance Code Generation. Languages like Accelerate [33], StreamIt [53] or Halide [38] aim to simplify the programming of GPUs through parallel patterns. However, all of these approaches are compiled to low-level loop-based code at an early stage of the compilation process. Accelerate allows users to write high-level functional code in a DSL that compiles down to NVIDIA GPUs. StreamIt aims to exploit parallelism for streaming applications and also lifts

the abstraction level for users. Halide focuses on developing parallel pipelines for image processing. These frameworks all rely on hard-coded optimizations or heuristics and are limited in the backends they can target, while Lift is able to optimize specifically for a particular architecture.

Delite [51] is the closest related work to Lift. A small set of parallel patterns is compiled and optimized by a single backend into high-performance code, enabling DSLs implemented on top of Delite to benefit from these optimizations. 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.

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 [14], Pochoir [52], PolyMage [35], and many others [3, 8, 24, 26, 34]. These DSLs are often limited to a particular domain or rely on heuristics or hard-coded implementations. [40, 41] 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 [16]. Skeleton libraries providing stencil skeletons include SkePU [15], SkelCL [46], MUESLI [28], and PASTHA [29]. Most of these approaches rely on hard-coded and stencil specific implementations. Lift is designed instead to be a middle layer between high-level abstractions and low-level optimisations, capable of handling different domains beyond stencils. None of these solutions create a separation of concerns in this way, which means they will always be strained in what they are capable of achieving.

Optimizations for Stencil Computations. There are also many works detailing stencil optimization strategies. However, these optimizations must be hard-coded, and as we have shown in this article, it is crucial to be able to only apply optimizations where they work. These include blocking [36, 56, 58, 59] and tiling approaches [19–21, 27, 30, 43], and other collections of optimizations [12, 17, 32, 50]. Furthermore, multiple auto-tuning frameworks aim to automatically optimize stencils [18, 25, 31]. However, none of these approaches formalize these optimizations as provably correct rewrite rules. This enables their exploration systematically using an optimizing compiler, instead of applying them in a ad-hoc manner using imprecise rules of thumb.

The 2.5D tiling optimization has been explored in other papers [36, 59]. These papers only investigate the optimization on NVIDIA GPUs or CPUs, whereas this article reports in detail on its benefits across three GPU platforms. Additionally, we have analyzed the performance characteristics of 2.5D tiling for different stencil shapes and sizes where related work had reported only on a single 7-point Jacobi stencil for limited input sizes. The shape and size of stencils is important in physical simulations as well as other disciplines, particular for the accuracy of the simulation results [9, 37]. Reference [23] did a large exploration of stencil performance on NVIDIA GPUs for a range of different shapes and sizes. However, this did not include investigating any optimizations such as 2.5D tiling.

10 FUTURE WORK

Lift's strength stems from its ability to generate complex codes from small building blocks and the addition of stencil functionality has enabled further possible progress in multiple directions.

Iterative Stencils are an interesting area for continued work, which are pervasive in HPC and image processing. Handling such cases requires the ability to generate code on the host side to run the kernel multiple times, which is mostly an engineering effort. More interestingly, separate handling of boundary conditions would be required to support more complicated stencils.

Time Tiling is another interesting tiling optimization that could be added. As this work focused on single iteration and single kernel stencils, this article did not explore this further. However, all the building blocks for this type of optimization already exist in Lift. Similar to what can be seen

in this article, where overlapped tiling and 2.5D tiling required only one additional rule for the Lift rewrite system, time tiling could also be encoded with the addition of a single rule.

11 CONCLUSIONS

This article 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 for stencil functionality: 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.

This article has also discussed how stencil-specific tiling optimizations are encoded as rewrite rules. We have covered code generation additions required to supplement a new 2.5D tiling optimization. We have also discussed the addition of rewrite rules for tiling optimizations, which allow for automatic optimization on platforms that could benefit from it. Due to the general nature of Lift, we are also able to able to apply existing Lift optimizations, which are also applicable to stencil computations. This demonstrates that Lift is easily extensible to new domains with little effort.

Finally, experimental results provide evidence that this approach generates high-performance stencil code on GPUs. In particular, we have shown how the added 2.5D tiling optimization affects stencils varying by shape, size and domain space and how a reuse metric helps explains the results we see. 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.

APPENDIX

```
1  typedef struct coeffs_type {
2    double l2;
3    double loss1;
4    double loss2;
5  } coeffs_type;
6
7  void UpdateStencil(global float* t1, global float* t,
8                     __constant struct coeffs_type* cf_d) {
9    int X = get_global_id(0);
10   int Y = get_global_id(1);
11   int Z = get_global_id(2);
12
13   if( (X>0) && (X<(Nx-1)) && (Y>0) && (Y<(Ny-1)) && (Z>0) && (Z<(Nz-1)) ){
14     int cp   = Z*area+(Y*Nx+X);
15
16     double cf  = 1.0;
17     double cf2 = 1.0;
18
19     int K = (0||X-1)+(0||X-Nx-2)+(0||Y-1)+(0||Y-Ny-2)+(0||Z-1)+(0||Z-Nz-2);
20
21     if(K < 6) {
22       cf   = cf_d[0].loss1;
23       cf2  = cf_d[0].loss2;
24     }
25
26     double S = t1[cp-1]+t1[cp+1]+t1[cp-Nx]+t1[cp+Nx]+t1[cp-area]+t1[cp+area];
27     t[cp]    = cf*( (2.0-K*cf_d[0].l2)*t1[cp] + cf_d[0].l2*S - cf2*u[cp] );
28   }
29 }
```

Listing 6. Acoustic Stencil implementation in C [55].

ACKNOWLEDGMENTS

We thank the Lift team; Prashant Singh Rawat for help with PPCG comparisons; Ari Rasch and students of the University of Münster for help with the ATF framewok and integrating it with Lift.

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 *Proceedings of the 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 *Proceedings of the SYCL 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 *Proceedings of the SST'06*.

[5] Tobias Brandvik and Graham Pullan. 2010. SBLOCK: A framework for efficient stencil-based PDE solvers on multi-core platforms. In *Proceedings of the CIT'10*. 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 *Proceedings of the IISWC'09*. 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 *Proceedings of the 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] G. C. Cohen and G. C. Gaunaurd. 2002. Higher-order numerical methods for transient wave Equations. Scientific computation. *Appl. Mech. Rev.* 55 (2002), B85.

[10] Murray I. Cole. 1988. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. Ph.D. Dissertation. University of Edinburgh.

[11] 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 GPGPU'10*. ACM, 63–74.

[12] Usman Dastgeer and Christoph Kessler. 2012. A performance-portable generic component for 2D convolution computations on GPU-based systems. In *Proceedings of the MULTIPROG Workshop at HiPEAC'12*. 1–12.

[13] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine A. Yelick. 2009. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.* 51, 1 (2009), 129–159.

[14] 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 WOSC'14*. ACM, 41–4.

[15] Johan Enmyren and Christoph W. Kessler. 2010. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the HLPP'10*. ACM, 5–14.

[16] Thomas L. Falch and Anne C. Elster. 2016. ImageCL: An image processing language for performance portability on heterogeneous systems. *arXiv preprint arXiv:1605.06399*.

[17] Matteo Frigo and Volker Strumpen. 2005. Cache oblivious stencil computations. In *Proceedings of the ICS'05*. ACM, 361–366.

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

[19] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. 2013. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles. In *Proceedings of the GPGPU'13*. ACM, 24–31.

[20] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P. Sadayappan. 2014. The relation between diamond tiling and hexagonal tiling. *Parallel Process. Lett.* 24, 3 (2014).

[21] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, and David Padua. 2009. Writing productive stencil codes with overlapped tiling. *Concurr. Comput.: Pract. Exper.* 21, 1 (2009), 25–39.

[22] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with Lift. In *Proceedings of the CGO*. ACM, 100–112.

[23] Brian Hamilton, Craig J. Webb, Alan Gray, and Stefan Bilbao. 2015. Large stencil operations for GPU-based 3-D acoustics simulations. *Proceedings of the Conference on DAFx'15*. 292–299.
[24] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2013. A stencil compiler for short-vector SIMD architectures. In *Proceedings of the ICS'13*. ACM, 13–24.
[25] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. 2010. An auto-tuning framework for parallel multicore stencil computations. In *Proceedings of the IPDPS'10*. IEEE, 1–12.
[26] 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 *Proceedings of the PPoPP'12*. ACM, 303–304.
[27] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. 2007. Multi-level tiling: M for the price of one. In *Proceedings of the SC'07*. ACM, 51.
[28] Herbert Kuchen. 2002. A skeleton library. In *Proceedings of the Euro-Par'02*. Springer, 620–629.
[29] Michael Lesniak. 2010. PASTHA: Parallelizing stencil calculations in haskell. In *Proceedings of the DAMP'10*. ACM, 5–14.
[30] Tareq M. Malas, Georg Hager, Hatem Ltaief, and David E. Keyes. 2017. Multidimensional intratile parallelization for memory-starved stencil computations. *ACM Trans. Parallel Comput.* 4, 3, Article 12 (Dec. 2017). DOI: https://doi.org/10.1145/3155290
[31] Azamat Mametjanov, Daniel Lowell, Ching-Chen Ma, and Boyana Norris. 2012. Autotuning stencil-based computations on GPUs. In *Proceedings of the CLUSTER'12*. IEEE, 266–274.
[32] Naoya Maruyama and Takayuki Aoki. 2014. Optimizing stencil computations for NVIDIA kepler GPUs. In *Proceedings of the HiStencils'14*. 89–95.
[33] Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising purely functional GPU programs. In *Proceedings of the ICFP'13*. ACM, New York, NY, 49–60.
[34] Richard Membarth, Frank Hannig, Jürgen Teich, and Harald Köstler. 2012. Towards domain-specific computing for stencil codes in HPC. In *Proceedings of the SCC'12*. IEEE, 1133–1138.
[35] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the ASPLOS'15*. ACM, New York, NY, 429–443.
[36] 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 *Proceedings of the SC'10*. IEEE Computer Society, 1–13.
[37] Michael Patra and Mikko Karttunen. 2006. Stencils with isotropic discretization error for differential operators. *Numer. Methods Partial Differ. Equat.: Int. J.* 22, 4 (2006), 936–953.
[38] 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.
[39] Ari Rasch, Michael Haidl, and Sergei Gorlatch. 2017. ATF: A generic auto-tuning framework. In *Proceedings of the HPCC*. IEEE.
[40] 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 *Proceedings of the PACT'16*. ACM, 99–111.
[41] 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 *Proceedings of the GPGPU'16*. ACM, New York, NY, 92–102.
[42] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. 2016. Performance portable GPU code generation for matrix multiplication. In *Proceedings of the GPGPU'16*. ACM, New York, NY, 22–31. DOI: https://doi.org/10.1145/2884045.2884046
[43] Lakshminarayanan Renganarayana, Manjukumar Harthikote-Matha, Rinku Dewri, and Sanjay Rajopadhye. 2007. Towards optimal multi-level tiling for stencil computations. In *Proceedings of the IPDPS'07*. IEEE, 1–10.
[44] 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 *Proceedings of the ICFP*. ACM, 205–217.
[45] 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 Process. Lett.* 24, 3 (2014), 1441005.
[46] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. 2011. SkelCL - A portable skeleton library for high-level GPU programming. In *Proceedings of the IPDPSW'11*. IEEE, 1176–1182.
[47] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2016. Matrix multiplication beyond auto-tuning: Rewrite-based GPU code generation. In *Proceedings of the CASES*. ACM, 15:1–15:10.

[48] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the CGO*. ACM, 74–85.
[49] Larisa Stoltzfus, Alan Gray, Christophe Dubach, and Stefan Bilbao. 2017. Performance portability for room acoustics simulations. In *Proceedings of the DAFx'17*. 367–374.
[50] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. 2011. Cache accurate time skewing in iterative stencil computations. In *Proceedings of the ICPP*. IEEE, 571–581.
[51] 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. *Trans. Embed. Comput. Syst.* 13, 4s (2014), 134.
[52] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir stencil compiler. In *Proceedings of the SPAA*. ACM, 117–128.
[53] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2009. Software pipelined execution of stream programs on GPUs. In *Proceedings of the CGO*. IEEE, 200–209.
[54] 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).
[55] Craig Jonathan Webb. 2014. *Parallel Computation Techniques for Virtual Acoustics and Physical Modelling Synthesis*. Ph.D. thesis. University of Edinburgh.
[56] Gerhard Wellein, Georg Hager, Thomas Zeiser, Markus Wittmann, and Holger Fehske. 2009. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *Proceedings of the COMPSAC*, Vol. 1. IEEE, 579–586.
[57] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.
[58] Markus Wittmann, Georg Hager, and Gerhard Wellein. 2010. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. In *Proceedings of the IPDPSW*. IEEE, 1–7.
[59] Guangwei Zhang and Yinliang Zhao. 2016. Modeling the performance of 2.5D blocking of 3D stencil stencil code on GPUs. In *Proceedings of the HPEC*. IEEE Computer Society.
[60] Xing Zhou. 2013. *Tiling Optimizations for Stencil Computations*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Received April 2019; revised October 2019; accepted October 2019