Position-Dependent Arrays and Their Application for High Performance Code Generation

Citation for published version:
Pizzuti, F, Steuwer, M & Dubach, C 2019, Position-Dependent Arrays and Their Application for High Performance Code Generation. in *FHPNC 2019: Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing.* ACM Association for Computing Machinery, pp. 14-26, FHPNC 2019 Functional High-Performance and Numerical Computing
, Berlin, Germany, 18/08/19. https://doi.org/10.1145/3331553

Digital Object Identifier (DOI):
10.1145/3331553

Published In:
FHPNC 2019: Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing

Position-Dependent Arrays and Their Application for High Performance Code Generation

Federico Pizzuti
The University of Edinburgh
Edinburgh, Scotland, United Kingdom
federico.pizzuti@ed.ac.uk

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

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

Abstract

Modern parallel hardware promises unprecedented performance, for the gifted few experts who can program it correctly. Code generators from high-level languages provide an attractive alternative, promising to deliver high performance automatically. Existing projects such as Accelerate, Futhark, Halide, or Lift show that this approach is feasible. Unfortunately, existing efforts focus on computations over tensors: regularly shaped higher dimensional arrays. This limits the expressiveness of these approaches and excludes many interesting data structures that are commonly encoded manually in memory, such as trees or triangular matrices.

This paper presents an extended array type that lifts this restriction. For multidimensional arrays, the size of a nested array might depend on its position in the surrounding arrays, which enables the expression of computations over less regularly shaped data structures. However, these position-dependent arrays bring new challenges for high-performance code generation, as determining the position of the elements in memory becomes more challenging.

This paper shows how these challenges are addressed by extending the existing Lift type system and compiler. The experimental results show that this approach enables the efficient code generation of triangular matrix-vector multiplication, with performance improvements over cuBLAS on an Nvidia GPU by up to 2×. Furthermore, we show a use case for a low-level optimization for avoiding unnecessary out-of-bound checks in stencils, leading to up to 3× improvements over already optimized generated stencil codes.

1 Introduction

Domain specific code generators enables the generation of efficient parallel code from high-level abstractions. These code generators attempt to fulfill the high-performance needs of many domains, such as machine learning, that crucially rely on the efficient exploitation of high-performance hardware such as GPUs. It is extremely challenging, even for experts, to write correct and efficient programs in low-level programming approaches such as CUDA or OpenCL. Code generation from high-level abstractions offers an attractive alternative and recent research has provided significant advances with projects such as Accelerate [13], Futhark [11], Halide [14], and Lift [17, 18]. There has been particular interest from industry in high performance code generations for deep learning with projects such as Tensor Comprehensions [19], Glow [16] for compiling PyTorch networks, as well as XLA [8] for compiling TensorFlow graphs.

The focus of existing work in this area has been on computations over regularly shaped higher-dimensional arrays, know as *tensors*. Many important application domains fall into this category, but there also exist many important applications that require more irregularly shaped data structures. For instance, triangular matrices are extremely important in many fields, as they are commonly used to perform efficient inversion of symmetric matrices [3]. Moreover, many physical phenomena can be modelled using matrices that have unusual characteristics, such as banded matrices, or even more exotic type of matrices [7].

For such applications, irregularly shaped data must currently be encoded manually in the provided regular-shaped arrays. This leads to increased complexity for the programmer, possible inefficiencies in memory usage and compute time, and ultimately defies the purpose of a high-level code generator. Encoding irregularly shaped data explicitly in memory is well known to low-level programmers who are forced to manually encode higher-level data structures in flat memory buffers.

In this paper, we present our approach to generate efficient parallel code for irregularly shaped data by extending the functional Lift intermediate representation and code generator. Crucially, we show how an extension of the type system with a limited form of dependent types enables us to generate efficient parallel code for computations over irregularly shaped data, without much changes in the way the Lift code generator operates. We also tackle the challenges that such irregularly shaped data bring for memory access calculation when indexing elements.

High-level Lift programs are composed of well known high-level primitives such as *map* or *reduce*. Such programs are transformed by the Lift compiler into a low-level form using a set of rewrite rules that are applied in an automated optimization process. Finally, efficient parallel code is generated from an optimized low-level program. Our extension to Lift ensures that existing primitives continue to work

Conference'17, July 2017, Washington, DC, USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

over irregularly shaped arrays. Furthermore, we add a new primitive — *partition* — to break down a one-dimensional array into an irregularly shaped two-dimensional structure. This primitive is useful for encoding lower level optimizations, such as avoiding out-of-bound checks for stencil codes. Our implementation carefully extends Lift to reuse as much existing infrastructure as possible.

Our experimental results demonstrates that the extended Lift code generator is capable of generating efficient parallel GPU code for a number of important use-cases operating on irregular data. For triangular matrix-vector multiplication, a crucial numerical kernel included in BLAS, we achieve performance on par with cuBLAS on an Nvidia GPU and even a speedup of 2× for certain input sizes. By using the *partition* primitive, we improve the performance of GPU stencil code by avoiding out-of-bound checks resulting in up to 3× improvements for certain stencil sizes.

To summarize the contributions of this paper:

- We present a generalization of array types capable of representing irregularly shaped data such as triangular arrays and discuss our design and implementation, including a *partition* primitive for introducing irregularity into regular arrays (Section 4);
- We show how we generate efficient array indices using symbolic simplification extended to deal with the position dependent arrays (Section 5);
- We present performance results for two case studies demonstrating that our approach generates efficient parallel code with performance improvements of up to 2× compared to the *tmrv* kernel in cuBLAS as well as performance improvements of up to 3× over already optimized stencil code by automatically avoiding unnecessary boundary checks (Section 6).

We first start with a motivation (Section 2) and background information about traditional array types (Section 3).

2 Motivation

The ever growing demand of increased performance is fueling the development of domain specific code generators to automatically generate high performance code from high level notations. Academic projects such as Accelerate [13], Futhark [11], or Lift [17, 18] use functional languages as compiler intermediate representation for high performance code generators. This approach has the advantage that code generation is guided by a strong formal foundation, including a type system and formal semantics.

Lift, for example, tracks the size of multi-dimensional array dimensions in the type system and uses this information for the generation of loop bounds and array indices. This allows for a high level notation that omits details such as indexing arrays while typing guarantees that the information required for generating correct array indices in the low level program are always accessible. Similarly, Accelerate and Futhark track the *shape* of multidimensional arrays.

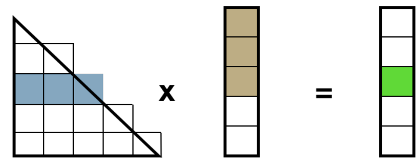

Figure 1. Triangular matrix vector multiplication

The representation of regularly shaped multidimensional arrays in a type system is well known in the functional programming community and it is well understood how such arrays are represented when flattened in memory. However, the type systems used for functional intermediate representations are so far not expressive enough to represent other useful less regular data multi-dimensional structures such as *triangular matrices*. Triangular matrices for instance, are used when dealing with some classes of partial differential equations and least square problems, and are commonly used to represent systems of equations, as discussed by de Castro Martins et al. [6].

In this paper, we investigate how multi-dimensional arrays representing less regularly shaped data are represented at the type level and how to generate high performance code for them. We look at two use cases of computations in detail: *triangular matrix vector multiplication* and the use of irregularly shaped array as a compiler internal data structure to *optimize avoidance of out-of-bound checks for stencils*. These two very different use cases have been chosen to highlight the potential of our generic approach.

2.1 Use-case 1: Triangular matrix vector multiplication

Triangular matrices naturally appear in different areas of mathematics, for example when solving linear equations. Triangular matrix vector multiplication is a fundamental building block included in the basic linear algebra subroutines (BLAS) API in form of the *trmv* kernel. Figure 1 shows a visualization of the operation. Since the matrix has a triangular shape, for each row i the dot product is computed only with the first i elements of the vector.

In current systems such as Accelerate, Futhark, or Lift, it is unclear how a triangular matrix should be represented, as their type systems are not expressive enough to precisely represent a triangular matrix. Instead, the programmer is forced to change how to express the computation, how to represent the data or how to do both, *e.g.*, by flattening the matrix into a one dimensional array and use manual index computations, or by wastefully representing the data as a regular rectangular matrix filled with zeros.

As we will see, section 4 presents a type system that is capable of precisely capturing the shape of the triangular

```
val sumNbh = fun(nbh => reduce(add, 0.0f, nbh))
val stencil = fun( A: Array(Float, N) =>
  map(sumNbh, slide(3, 1, pad(1, 1, clamp, A))))
```

Listing 1. 3-Point Jacobi Stencil in LIFT. From [10].

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

Listing 2. Simple 3-Point Jacobi Stencil in C. From [10].

matrix, section 5 explains how we generate GPU code for the triangular matrix vector multiplication from the straightforward high level notation. Finally, section 6 will show that our automatically generated code outperforms a cuBLAS implementation up to 2×.

2.2 Use-case 2: Optimizing stencil boundary checks

Stencil computations are an important computational pattern occurring in many application domains ranging from image processing to convolution neural networks. Stencils update a point in a grid with a computation that depends on neighboring grid points. In LIFT, stencils are represented in a high level notation using a combination of primitives as shown in listing 1 and described in detail by Hagedorn et al. [10]. Here the *pad* primitive describes the boundary handling by applying a clamping boundary condition, the *slide* primitive creates a sliding window of neighboring grid points, and, finally, the *map* primitive applies the *sumNbh* function to all created neighborhoods.

Listing 2 shows the C pseudo code generated by LIFT. The boundary handling is done in lines 3 and 4 where an out-of-bound check is performed in every loop iteration. Human experts are able to produce a more optimal version, where the first and last few iterations of the outer loop are peeled away. The loop itself then becomes entirely free of bound checks, leading to increased performance.

Code generators such as LIFT are good at treating data in arrays uniformly, but currently struggle with optimizations where data and computations are treated *non-uniformly*.

What is required is a type system that is able to represents arrays whose elements are themselves nested arrays of varying size. Using such ability, we could partition the input array into three nested arrays each representing a differently sized portion of the input data. If we had a value of such a type, we could use LIFT's primitives such as *map* to generate a version similar to the human optimized one. Section 6 shows that this approach leads to improvements of up to 3× over already optimized GPU code generated by LIFT [10].

2.3 Summary

Current code generators do not support the generation of high performance code for computations with irregularly shaped multidimensional arrays. This section has motivated an extension of the type system in the existing LIFT compiler. We have discussed two particular and quite different use-cases. As we will see in section 4, the extension of the type system increase the expressiveness by representing less regularly shaped multidimensional arrays while still imposing structure that is exploited to generate efficient code.

The next section discusses the design of this extended type system which is inspired by a limited form of dependent types. We will first start by explaining the design of existing type system for regularly shaped multidimensional arrays.

3 Traditional Multidimensional Arrays

This section describes how traditional multidimensional arrays types are represented in existing code generators with functional intermediate representations like Accelerate, Futhark, or LIFT that all track the shape or even size of multidimensional arrays in the type.

Tracking the shape (i.e., dimensionality) and size of arrays in the type system has proven useful for efficient code generation from functional intermediate representations. Accelerate tracks the shape of arrays in the type: `Array sh a`. Here `sh` represents the *shape* of the array and `a` the type of the array elements [13]. Futhark and LIFT track the length of multidimensional arrays in the type system. In Futhark an array type is written: $[n]\rho$ where n is the number of elements and ρ the element type [11]. Similarly, but using a different notation, LIFT expresses an array type as: $[A]_n$ where A is the element type and n the number of elements [17].

To represent multidimensional arrays, both Futhark and LIFT use nesting. A two dimensional $n{\times}m$ matrix type is written as: $[\,[A]_m\,]_n$ where A is the element type. LIFT supports rich arithmetic expressions for the length of arrays beyond constants such as n. Operations such as addition, division, or modulo are used to represent the length of arrays [18]. This paper extends the LIFT type system which we discuss next.

3.1 Type System

The type system used by LIFT is shown in Figure 2 using the formulation used by Atkey et al. [1] and adapted for the presentation here. We distinguish between three different kinds (2a): natural numbers (nat) for the length of arrays; data types (datatype) for types that are stored in memory; data types together with function types and a limited form of dependent function types make up the final kind (type). As types may contain variables we use a kinding judgement $\Delta \vdash \tau : \kappa$ stating that type τ has kind κ in the kinding context Δ. As our types contain expressions of natural numbers type equality can not be assumed by syntactic equality. Figure 2c

$$\kappa ::= \mathsf{nat} \mid \mathsf{datatype} \mid \mathsf{type}$$

(a) Kinds

$$\frac{x : \kappa \in \Delta}{\Delta \vdash x : \kappa}$$

(b) Kinding Structural Rules

$$\frac{\models \forall \sigma : dom(\Delta) \to \mathbb{N}.\sigma(I) = \sigma(J)}{\Delta \vdash I \equiv J : \mathsf{nat}}$$

(c) Type Equality

$$\frac{}{\Delta \vdash \underline{n} : \mathsf{nat}} \qquad \frac{\Delta \vdash N : \mathsf{nat} \qquad \Delta \vdash M : \mathsf{nat}}{\Delta \vdash N + M : \mathsf{nat}} \qquad \frac{\Delta \vdash N : \mathsf{nat} \qquad \Delta \vdash M : \mathsf{nat}}{\Delta \vdash N \cdot M : \mathsf{nat}}$$

(d) Natural numbers

$$\frac{}{\Delta \vdash \mathsf{int} : \mathsf{datatype}} \qquad \frac{\Delta \vdash N : \mathsf{nat} \qquad \Delta \vdash \delta : \mathsf{datatype}}{\Delta \vdash [\delta]_N : \mathsf{datatype}} \qquad \frac{\Delta \vdash \delta_1 : \mathsf{datatype} \qquad \Delta \vdash \delta_2 : \mathsf{datatype}}{\Delta \vdash \delta_1 \times \delta_2 : \mathsf{datatype}}$$

(e) Data Types

$$\frac{\Delta \vdash \delta : \mathsf{datatype}}{\Delta \vdash \delta : \mathsf{type}} \qquad \frac{\Delta \vdash \theta_1 : \mathsf{type} \qquad \Delta \vdash \theta_2 : \mathsf{type}}{\Delta \vdash \theta_1 \to \theta_2 : \mathsf{type}} \qquad \frac{\kappa \in \{\mathsf{nat}, \mathsf{datatype}\} \quad \Delta, x : \kappa \vdash \theta : \mathsf{type}}{\Delta \vdash (x{:}\kappa) \to \theta : \mathsf{type}}$$

(f) Types

Figure 2. Well-formed Types of Lift

states that nats are equal when their interpretations as numbers are equal for all interpretations of their free variables.

Figure 2d defines well formed natural numbers which are either literals (indicated by $\underline{n}$) or expressions of natural numbers. We show here only addition and multiplication as possible binary operators, but in our implementation we support much richer expressions of natural numbers using additional operators such as division and modulo.

Figure 2e defines three different data types supported in Lift: scalar types such as `int`; array data types; and pair types. For each exists a direct representation in C, whereby pair types are mapped to structs. The design of the type system deliberately prevents function types inside arrays or pairs as OpenCL does not support function pointers.

Finally, figure 2f defines all well-formed Lift types. We consider all well-formed datatypes to be well-formed types and add the usual function type and a function type abstracting over data types and natural numbers at the type level.

3.2 Type checking

The typing judgment $\Delta|\Gamma \vdash P : \theta$ states that a program P is well typed with type θ in the contexts Δ and Γ. For this the type θ as well as all types in Γ must be well-kinded by Δ and P must be well-typed by Γ. Figure 3 shows the typing rules of Lift. The structural rules in figure 3a show the forming of well-typed variables, implicit conversion between equal types, and how the primitives of Lift integrate. The rules in figure 3b are the standard λ-calculus rules for abstraction and application for usual lambdas as well as the nat and datatype dependent lambdas (written as Λ) where for application the argument is substituted in the type of the lambda body.

3.3 Computations over multidimensional arrays

The most important Lift primitives are shown in figure 4. We usually infer the first arguments representing nat and datatype and omit them when we write Lift programs.

The primitives nest naturally: *map* applies a function to each array element - independent if the element is a scalar value or an array itself. Operations on higher dimensional data are expressible using familiar functional primitives. For example, matrix-matrix-multiplication can be expressed as:

$$map\,(\lambda\, r\,.\, map\,(\lambda\, c\,.\, reduce\,(+)\; 0\; (map\,(*)\; (zip\; r\; c)))\; B)\; A$$

While nesting of the presented array type enables the representation of regularly – or rectangularly – shaped multidimensional arrays, it is not sufficient to represent less regularly shaped arrays. In the two dimensional array type $[\,[\mathtt{int}]_m\,]_n$ the inner size m must be the same for all elements of the outer array, as arrays are homogeneous containers.

Our goal is to relax this strict notion of homogeneity to allow differently shaped multidimensional arrays to be represented. But we still insist on some form of homogeneity for multidimensional arrays: the underlying scalar data type (`int` in the example) must be the same for all elements in the multidimensional array. This ensures that arrays can be flatten efficiently in memory, *e.g.*, with a C-like row-major storage layout. The information in the type: m, n, and $sizeof(\mathtt{int})$, is sufficient to compute the index of each element.

4 Position Dependent Arrays

This section describes the proposed extension to the Lift type system for irregularly shaped multidimensional arrays. We first describe the extended array type, followed by an

$$\frac{x : \theta \in \Gamma}{\Delta \mid \Gamma \vdash x : \theta}\ \textsc{Var} \qquad \frac{\Delta \mid \Gamma \vdash P : \theta_1 \qquad \Delta \vdash \theta_1 \equiv \theta_2 : \mathsf{type}}{\Delta \mid \Gamma \vdash P : \theta_2}\ \textsc{Conv} \qquad \frac{\mathsf{prim} : \theta \in \textsc{Primitives}}{\Delta \mid \Gamma \vdash \mathsf{prim} : \theta}\ \textsc{Prim}$$

(a) Structural Rules

$$\frac{\Delta \mid \Gamma, x : \theta_1 \vdash P : \theta_2}{\Delta \mid \Gamma \vdash \lambda x.P : \theta_1 \to \theta_2}\ \textsc{Lam} \qquad \frac{\Delta \mid \Gamma_1 \vdash P : \theta_1 \to \theta_2 \qquad \Delta \mid \Gamma_2 \vdash Q : \theta_1}{\Delta \mid \Gamma_1, \Gamma_2 \vdash P\,Q : \theta_2}\ \textsc{App}$$

$$\frac{\Delta, x : \kappa \mid \Gamma \vdash P : \theta \qquad x \notin fv(\Gamma)}{\Delta \mid \Gamma \vdash \Lambda x.P : (x{:}\kappa) \to \theta}\ \textsc{TLam} \qquad \frac{\Delta \mid \Gamma \vdash P : (x{:}\kappa) \to \theta \qquad \Delta \vdash \tau : \kappa}{\Delta \mid \Gamma \vdash P\,\tau : \theta[\tau/x]}\ \textsc{TApp}$$

(b) Abstraction and Application Rules

Figure 3. Typing Rules of Lift

$$\begin{array}{lll}
\textbf{map} & : & (n : \mathsf{nat}) \to (\delta_1\ \delta_2 : \mathsf{datatype}) \to \\
 & & \quad (\delta_1 \to \delta_2) \to [\delta_1]_n \to [\delta_2]_n \\
\textbf{reduce} & : & (n : \mathsf{nat}) \to (\delta_1\ \delta_2 : \mathsf{datatype}) \to \\
 & & \quad (\delta_1 \to \delta_2 \to \delta_2) \to \delta_2 \to [\delta_1]_n \to \delta_2 \\
\textbf{zip} & : & (n : \mathsf{nat}) \to (\delta_1\ \delta_2 : \mathsf{datatype}) \to \\
 & & \quad [\delta_1]_N \to [\delta_2]_n \to [\delta_1 \times \delta_2]_n \\
\textbf{split} & : & (n\ m : \mathsf{nat}) \to (\delta : \mathsf{datatype}) \to [\delta]_{n \cdot m} \to [[\delta]_n]_m \\
\textbf{join} & : & (n\ m : \mathsf{nat}) \to (\delta : \mathsf{datatype}) \to [[\delta]_n]_m \to [\delta]_{n \cdot m}
\end{array}$$

Figure 4. Existing Lift primitives

example for triangular matrix and higher dimensions arrays. This is followed by a section on how to compute over data with such types. We end this section with a description of new primitives added to Lift useful for expressing low level optimizations such as avoiding out-of-bound accesses.

4.1 Position dependent array type

To describe the shape of a triangular matrix precisely in its type, for instance, we need to lift some restrictions of the traditional multidimensional array types. The homogeneity of arrays which ensures an efficient data representation is overly restrictive. It is obvious that a triangular matrix can be stored efficiently in memory following a row-major order into a flat representation. This shape can be precisely described statically, allowing to efficiently compute element indices.

Therefore, we can carefully extend the notion of an array type to allow the size of nested arrays to depend on its position. Figure 5 shows an array type where the element type δ is allowed to depend on the position i in the array. Since nat is only allowed to appear in the lengths of arrays, it is ensured that the underlying scalar type of the array remains the same. In other words, the position dependent arrays are still homogeneous, besides for the array lengths that might appear in the element type. This ensures that multi-dimensional arrays are stored efficiently as a flat representation of the underlying element type. This prevents, for

$$\frac{\Delta \vdash N : \mathsf{nat} \qquad \Delta, i : \mathsf{nat} \vdash \delta : \mathsf{datatype}}{\Delta \vdash [i \mapsto \delta]_N : \mathsf{datatype}}$$

Figure 5. Well formed position dependent array type

$$\begin{bmatrix} [0], \\ [1, 2], \\ [3, 4, 5], \\ [6, 7, 8, 9] \end{bmatrix} : \big[\, i \mapsto [\mathtt{int}]_{i+1} \,\big]_4$$

Figure 6. A triangular matrix value on the left and a type precisely describing it on the right.

$$\begin{bmatrix} \begin{bmatrix} [0], \\ [1, 2] \end{bmatrix}, \\ [\,[3, 4, 5]\,], \\ [\,[6, 7, 8, 9]\,] \end{bmatrix} : \big[\, i \mapsto \big[\, j \mapsto [\mathtt{int}]_{s_1(i,j)} \,\big]_{s_0(i)} \,\big]_3$$

where

$$s_0(i) = \begin{Bmatrix} 2 & \textit{if } i = 0 \\ 1 & \textit{otherwise} \end{Bmatrix} \quad \textit{and} \quad s_1(i, j) = \sum_{k=0}^{i-1} s_0(k) + j + 1$$

Figure 7. A three dimensional array, irregularly grouping rows of a triangular matrix. Type shown on the right.

instance, the expression of a type of a matrix which stores `floats` in the some rows and `doubles` in some other rows, as it wouldn't be clear how to compute efficiently the addresses of the individual elements in memory.

4.2 Example

To describe the type of a triangular matrix, we write: $\big[\, i \mapsto [\mathtt{int}]_{i+1} \,\big]_n$. This type indicates that the length of each row is equal to its position i in the array plus one (to accommodate the 0-based indexing): the first row has length 1, the second row has length 2, and so on with the last row (at position $n-1$) having length n. This is shown in figure 6 for a two dimensional array with four nested arrays of different size. Overall the data forms a lower triangular matrix that is reflected in its type.

Triangular Matrix For the triangular matrix example with the type $\left[\, i \mapsto [\texttt{int}]_{i+1} \,\right]_n$, δ is the nested type $[\texttt{int}]_{i+1}$ for which the length depends on i and can be described by this function: $s(i) = i+1$. This is a strict generalization of the classical array type seen in the previous section. If the element type δ does not depend on i then all elements of the arrays must have the exact same length, as for arrays with a classical array type. For example, a two dimensional matrix can be expressed in our extended array type as: $\left[\, i \mapsto [\texttt{int}]_m \,\right]_n$. If the array index i is not used in the type we might omit it: $\left[\, \_ \mapsto [\texttt{int}]_m \,\right]_n$. This type is equivalent to the classical multidimensional array type: $\left[\, [\texttt{int}]_m \,\right]_n$.

Higher Dimension Arrays For higher dimensional types, the sizes of nested arrays might depend on all positions of the surrounding arrays. See figure 7 for an example. Here the first two rows of a triangular matrix have been grouped together forming a nested array together with the third and fourth row. Such a representation could be useful to achieve some form of load balancing by grouping multiple shorter rows together to balance the number of elements in every group. The three dimensional type is interesting with two functions s_0 and s_1 describing the size of the nested array dimensions. For s_0 a *case* statement is used to define the function specifying that the first element of the outer array will have two elements while the other elements will all be of size one. The deeper nested array has a more complex computation of its size. s_1 depends on both positions i and j of the surrounding arrays. We can still see the same arithmetic expression used to represent the triangular matrix: $j+1$. In addition, the expression $\sum_{k=0}^{i-1} s_0(k)$ computes a prefix sum over the outer dimensions expressed in s_0. We will understand how to derive this expression from the triangular matrix type using a new primitive we will introduce in section 4.4 called *partition*.

4.3 Computations over irregularly shaped arrays

So far, we have seen how to represent irregularly shaped multidimensional arrays with a novel type. We will now investigate how to express computations over such data structures in the functional high-level notation of Lift. We will use as much of the existing Lift primitives as possible.

The Lift primitives shown before in figure 4 are overloaded to work on the new position dependent arrays as shown in figure 8 for this the type system is extended with type level functions that map natural numbers to data types (nat $\rightarrow$ datatype) or to natural numbers (nat $\rightarrow$ nat).

The types of the primitives using the position dependent array types are interesting. For *map* the elements in the input array are now described by the type level function

$$
\begin{array}{lll}
\textbf{map} & : & (n : \mathsf{nat}) \rightarrow (f_{\delta_1}\ f_{\delta_2} : \mathsf{nat} \rightarrow \mathsf{datatype}) \rightarrow \\
& & \quad ((k : \mathsf{nat}) \rightarrow (f_{\delta_1}\ k) \rightarrow (f_{\delta_2}\ k)) \rightarrow \\
& & \quad [i \mapsto (f_{\delta_1}\ i)]_n \rightarrow [j \mapsto (f_{\delta_2}\ j)]_n \\
\textbf{reduce} & : & (n : \mathsf{nat}) \rightarrow (f_{\delta_1} : \mathsf{nat} \rightarrow \mathsf{datatype}) \rightarrow \\
& & \quad (\delta_2 : \mathsf{datatype}) \rightarrow \\
& & \quad ((k : \mathsf{nat}) \rightarrow (f_{\delta_1}\ k) \rightarrow \delta_2 \rightarrow \delta_2) \rightarrow \\
& & \quad \delta_2 \rightarrow [i \mapsto (f_{\delta_1}\ i)]_n \rightarrow \delta_2 \\
\textbf{zip} & : & (n : \mathsf{nat}) \rightarrow (f_{\delta_1}\ f_{\delta_2} : \mathsf{nat} \rightarrow \mathsf{datatype}) \rightarrow \\
& & \quad [i \mapsto (f_{\delta_1}\ i)]_N \rightarrow [j \mapsto (f_{\delta_2}\ i)]_n \rightarrow \\
& & \quad [k \mapsto ((f_{\delta_1}\ k) \times (f_{\delta_2}\ k))]_n \\
\textbf{join} & : & (n : \mathsf{nat}) \rightarrow (f_n : \mathsf{nat} \rightarrow \mathsf{nat}) \rightarrow (\delta : \mathsf{datatype}) \rightarrow \\
& & \quad [i \mapsto [\delta]_{f_n(i)}]_n \rightarrow [\delta]_{\sum_{i=0}^{n-1} f_n(i)}
\end{array}
$$

Figure 8. Overloaded Lift primitives operating on position dependent array types

```
fun(matrix:[i → [float]_(i+1)]_N, vector:[float]_N) => {
 map(λ row → reduce(add, 0, map(mult,
   zip(row, slice(0, getLength(row), vector))))
 , matrix) }
```

Listing 3. Lift code for triangular matrix multiplication

f_{δ_1} mapping natural numbers to datatype. The first value of the array has type $(f_{\delta_1}\ 0)$ the second $(f_{\delta_1}\ 1)$ and so on. The function that *map* applies to each element of the input array is now parameterized by an additional natural number k that represents the index at which the function is applied. The k-th element of the input array has the type $(f_{\delta_1}\ k)$. A similar type level function f_{δ_2} describes the element types in the output array. *reduce*, *zip* and *join* generalize to position dependent arrays in a similar way to *map*.

We do not provide a version of *split* for position dependent arrays, as we will introduce a new primitive in the next section called *partition*, which is more general than *split*.

Using the overloaded Lift patterns together with the extended array type we can straightforwardly write the implementation of triangular matrix multiplication, as shown in listing 3. We start by applying *map* to the triangular matrix to perform a computation for every row. For each row we compute the dot product with the vector by combining them with *zip*, multiplying the resulting pairs and summing them up. Lift transforms high-level programs into efficient low-level code by applying a set of rewrite rules in an automated optimization process. This process rewrites the expression by, e.g., fusing patterns to avoid the generation of unnecessary temporaries and by mapping the computation to the different levels of parallelism offered by modern hardware.

The only difference compared to the matrix vector multiplication of a rectangular matrix is the use of *slice* and *getLength*. *Slice* is a pattern for accessing a subarray defined by start and end indices. It is implemented in terms of a more general pattern called *partition*, whose details are covered in

$$\left[\texttt{0,1,2,3,4,5,6}\right] : [\texttt{int}]_6$$

$$\Downarrow \quad \textit{partition}\ 3\ (i \mapsto i+1) \quad \Downarrow$$

$$\begin{bmatrix} [\texttt{0}], \\ [\texttt{1,2}], \\ [\texttt{3,4,5}] \end{bmatrix} : \left[\, i \mapsto [\texttt{int}]_{i+1} \,\right]_3$$

Figure 9. An example of *partition* used to transform a one dimensional array into a two dimensional triangle.

the next section The *getLength* primitive returns the length of the given array. For this it accesses the length represented at the type level.

Together, these primitives select the upper part of the vector up to an equal size to the current row. This part of the vector is then combined with the row to compute their dot product to produce an element in the output vector.

4.4 Partition

Type Partition has the following type:

$$\textbf{partition} : (n : \text{nat}) \to (f : (\text{nat} \to \text{nat})) \to (\delta : \text{datatype}) \to [\delta]_{\sum_{i=0}^{n-1} f(i)} \to [i \mapsto [\delta]_{f(i)}]_n$$

Here n represents the number of subarrays produced by *partition* and f is a type level function mapping each index (ranging from $[0, n-1]$) to the length of the corresponding subarray. The produced array is a two dimensional array of size m and with subarrays as elements where element i has a size of $f(i)$. A simple example visualizing the partitioning of an array into a triangle matrix is shown in figure 9. The dual operation of *partition* is the generalized *join* defined in figure 8. The following identity holds:

$$\textit{join}\ n\ f\ \delta\ (\textit{partition}\ n\ f\ \delta\ \textit{input}) = \textit{input}$$

The length of the input of *partition* can therefore also be seen as the length of the output array of the type of *join*. A similar duality exists for *split* and *join* for rectangular arrays.

Introducing Partition via Rewriting Rules One of the core ideas underpinning Lift is the use of an automated exploration system that uses rewriting rules to automatically generate high performance code. A rewrite rule is a semantic preserving transformation of expressions, and is Lift's way to express optimization choices that are automatically explored in the optimization process using stochastic methods, as explained by [15].

To make automatic use of the *partition* primitive as a low level optimization we design rewrite rules that introduce the primitive and, thus, expose it to Lift's exploration process. As mentioned before, *partition* can be seen as a generalization of *split*. There exists a few rewrite rules that include *split*, like a divide-and-conquer style rule that splits an input array into several parts that are then processed individually before the results are joined back together:

$$\textit{map}(f, \textit{input}) \mapsto \textit{join}(\textit{map}(\textit{map}(f), \textit{split}(n, \textit{input})))$$

These rules can simply be generalized by exchanging *partition* for *split*, e.g., to express a load balancing aspect when the work of applying f to every element of the input array is not uniformly distributed. In addition to these generalized rules, it is possible to express a more specific low level optimization: the use of *partition* for a more fine-grained handling of stencil boundary conditions.

As seen in section 2.2, stencil applications need to handle the boundary of its multidimensional input array specially, for example by padding the array with additional values or (as shown in listing 1) by clamping the index computation. This boundary handling introduce potentially expensive branches in the code, human experts often write their programs in such a way to handle these section as special corner cases.

Due to the uniformity of the Lift primitives and its regular array types, it is not possible to express this optimization without support for position dependent arrays. The introduction of *partition*, however, allows for the optimized handling of boundary conditions to be expressed and automatically introduced via a rewrite rule:

$$\begin{aligned} &\textit{map}(f, \textit{slide}(\textit{size}, \textit{step}, \textit{pad}(l, r, \textit{input}))) \mapsto \\ &\quad \textit{join}(\textit{map}(\textit{map}(f), \textit{partition}(3, \textit{caseSplit}(l, n-l-r, r), \\ &\qquad \textit{slide}(\textit{size}, \textit{step}, \textit{pad}(l, r, \textit{input}))))) \end{aligned}$$

The intuition behind this rule is as follows: we insert a *partition* right before executing f which represents the stencil computation performed over the stencil neighborhood. The *partition* splits the grid of neighborhoods that has been produced by *slide* in three distinct sections: a *prologue*, a central *body*, and an *epilogue*. The sizes of the prologue and epilogue correspond to the number of elements padded to the input. The central body takes up the remaining input size.

Due to the information available in the type the compiler is capable of removing the out-of-bound checks from the central body section of the code. We will discuss details how this is implemented in section 5.

Concerning the rule correctness, we can see that the rule is indeed semantic preserving: partition has the effect to splitting the input in chunks and add one layer of nesting. The addition of a subsequent map wrapping around the original body does not modify the number or order of the elements in the output, but simply influences how the computation is organized. Finally, the terminating join will undo the effects of partition on the output.

4.5 Summary

In this section we have introduced an array type that allows for the size of nested arrays to depend on their position in the outer array. This enhances the expressiveness allowing to represent data structures such has triangular matrices or trees. We have seen that computations over such structures are as naturally expressed using the same set of primitives already familiar to functional programmers.

Furthermore, we have introduced a new primitive to *partition* a regular array into a nested irregular one and we have discussed how an rewrite rule automatically exposes this transformation as an optimization choice for removing unnecessary boundary checks. In the next section we will discuss important implementation details.

5 Implementation and Code Generation

After describing the design of the extended multidimensional array type in the previous section we now discuss some of the implementation details. We first describe the implementation challenges faced, before discussing them individually. Finally, we will briefly discuss the code generation before evaluating the performance achieved in the next section.

5.1 Implementation challenges

Extending the existing LIFT OpenCL backend to support the extended array type presented a number of challenges:

- Allowing for the array size to depend on its position in the surrounding arrays significantly complicates the computation of the number of element in a multidimensional array. This is a fundamental operation necessary for computing array indices and to perform memory allocation.
- The implementation of efficient index computations in the generated OpenCL kernel is no longer straightforward. When done naively, many of the generated arithmetic expressions would require the introduction of loops and conditional branches when computing indices.
- The implementation of *partition* should not produce results directly but instead lazily influence the code generated for following patterns. We describe a solution using LIFT's *view* system.
- Finally, we will discuss some memory management and memory allocation challenges.

5.2 Calculating the Number of Array Elements

In the previous section, we have seen how LIFT provides support for multidimensional arrays, and that these arrays are essential to the compositional nature of LIFT programs. Multi-dimensionality in LIFT, however, exists purely as an abstraction: in order to generate high performance code, the OpenCL code generator flattens the multidimensional arrays into contiguous memory buffers.

This requires the compiler to calculate the number of elements contained in a potentially multidimensional array. In the context of regular arrays with a classical type, one can easily compute the linear index using the formula

$$dim_1 \times dim_2 \times \ldots \times dim_n$$

For an irregular sized arrays with an extended array type, the number-of-elements formula generalized as follows:

$$\sum_{i_1=0}^{dim_1-1} \ldots \sum_{i_{n-1}=0}^{dim_{n-1}(i_1,..,i_{n-2})-1} dim_n(i_1,..,i_{n-1})$$

An important feature of the LIFT compiler is the extensive use of symbolic algebra for reasoning about arithmetic expressions of natural numbers, such as array sizes and iteration ranges. The system was originally designed to only work with arrays of regular length and is introduced and described in [18]. With the introduction of the extended array types, it becomes, therefore, necessary to extend the symbolic algebra system to include a new $\sum$ construct. This constructs corresponds to the concept of an algebraic summation as commonly used in mathematics. We discuss next how we exploit the properties of algebraic summations to optimize many cases of index computations.

5.3 Optimizing Index Computations

Generating concise index computations is incredible important for achieving high performance. Prior work [18] reports massive performance losses for applications such as matrix matrix multiplication when index computations are not simplified by the compiler. In this section we describe the generation of optimized index computations that contain the newly introduced $\sum$ operator.

A possible naive implementation of $\sum$ would generate a sequential loop. Due to performance considerations however, this approach is in practice not viable. Instead, in the majority of cases encountered in practice, the index computation can be simplified to a close form formula without any further $\sum$ terms.

Example Consider the problem of indexing of an element of a triangular matrix flattened in memory. As see earlier, the matrix has type $\left[\, i \mapsto [\delta]_{i+1} \,\right]_n$, which means the length of each row is $i+1$. To compute the position in memory of element (rid, cid), we can compute a close form as follows using well-known properties of $\sum$:

$$\begin{aligned} memLocation(rid, cid) &= (\sum_{i=0}^{rid-1} i + 1) + cid \\ &= rid + (\sum_{i=0}^{rid-1} i) + cid = rid + \frac{(rid) \times (rid-1)}{2} + cid \\ &= \frac{(rid+1) \times rid}{2} + cid \end{aligned}$$

By implementing these algebraic simplification rules, the compiler is capable of generating efficient index computations for a large number of expressions. The following section list the rules implemented in the compiler.

Simplification rules for $\sum$ The rules presented here might look slightly different from their usual presentation in mathematics: since in this work their main use is to determine offsets in linear arrays, the rules are indexed from 0, as opposed to the more commonly used indexing from 1.

$$\sum_{i=0}^{N} c = N * c + 1 \tag{1}$$

$$\sum_{i=0}^{N} i = \frac{N \times (N-1)}{2} \tag{2}$$

$$\sum_{i=0}^{N} 2^i = 2^{n+1} - 1 \tag{3}$$

$$\sum_{i=a}^{b} f(i) = \sum_{i=0}^{b} f(i) - \sum_{i=0}^{a} f(i) \tag{4}$$

$$\sum_{i=0}^{N} c * f(i) = c * \sum_{i=0}^{N} f(i) \tag{5}$$

$$\sum_{i=0}^{N} f(i) + g(i) = \sum_{i=0}^{N} f(i) + \sum_{i=0}^{N} g(i) \tag{6}$$

$$\sum_{i=c}^{N} \begin{Bmatrix} f_1(1) & if\, i = c \\ f_2(i) & otherwise \end{Bmatrix} = \begin{Bmatrix} l_1(c) & if\, c \geq N \\ 0 & otherwise \end{Bmatrix} + \sum_{i=c+1}^{N} f_2(i) \tag{7}$$

The algebraic simplification rules can be roughly grouped in three categories, according to their main purpose:

- Rules (1) - (3) are used to compute the number of elements of one dimension of an irregular array, each matching a different *primitive shape* that the dimensions can take. (1) is used for fixed size dimension, (2) corresponds to linearly variable dimension, such as the rows of triangular matrices (3) corresponds to exponentially variable dimension, such as binary trees
- Rules (4) - (6) are auxiliary simplification rules, used to split composite sums into simpler parts.
- Rule (7) deals with the elimination of if-statements, usually generated by the length function of *partition*. This rule is an analogue of *loop peeling*, a classical compiler optimization in which loop iterations are extracted out of the loop into the loop's prologue and epilogue.

If-elimination rules We observe that programs containing non-trivial uses of *partition* could generate inefficient OpenCL programs. The main cause of this is the presence of a large number of conditional expressions - many generated by the *loop-peeling* simplification mentioned above.

To address this issue, we investigated *if-elimination* rules in the arithmetic expressions. By leveraging the strength of the Lift symbolic algebra system, which tracks the ranges for each arithmetic expression. It is possible to implement a simplifier that is often capable of identifying the conditional expressions that are guaranteed to be never taken. This is achieved by checking the difference between the minimum and maximum possible values of the expressions within the conditional. For example, a conditional of the form

$$\begin{Bmatrix} x & if\, a \geq b \\ y & otherwise \end{Bmatrix}$$

simplifies to x if $min(a) \geq max(b)$, and to y if $max(a) < min(b)$. Similar rules exist for other boolean operators.

5.4 Implementation of *Partition*

Some Lift primitives are lazy: instead of performing a computation and writing into memory, they influence the behavior of the following patterns, by creating a compiler intermediate data structure – called a *view* – over their inputs and outputs. An example of a lazy pattern is *split* that reshapes an input array by introducing another dimension. The *partition* also primitive falls into this category as it lazily influences the reading of memory of following patterns.

Partition's view performs the mapping of the indices ranging over the two dimensions of the output array to the one dimensional index into the input array. In particular, the indices i, for the outer dimension of the output array, and j, for the inner dimension, will be mapped to a one dimensional index: $(i, j) \mapsto \sum_{k=0}^{i-1} f(i) + j$. The offset is computed as the sum of the length of the first $i-1$ elements in the outer array and j provides the index into the inner dimension.

5.5 Code Generation

After addressing the challenges described here, there is no need to modify the Lift code generator which remains unchanged compared to the techniques described in [1].

5.6 Summary

In this section we have discussed a number of implementation challenges and how we overcome them. Particularly, the introduction of the extended array type has lead to changes in computing length of and indices into arrays. By introducing and optimizing $\sum$ as arithmetic expressions, we are able to generate low level code from familiar high level Lift expressions. The new *partition* primitive is implemented as a Lift view and maybe surprisingly, no other modification to the Lift code generator was necessary for implementing our work. The next section experimentally evaluate the implementation using the two case studies introduced earlier.

6 Experimental Evaluation

Experimental Setup We conducted an experimental evaluation using single precision floats on a GeForce GTX TITAN X with CUDA 8.0 and driver version 375.66. We report the median of at least 100 executions measured using the OpenCL profiling API. Data transfer times are ignored since the focus of the evaluation lies on the quality of the generated kernel code. For the triangle matrix vector benchmarks, we perform an automatic exploration of implementation parameters including the OpenCL local size. We report the runtimes for the best parameter configuration we found.

```
fun(matrix:[i → [float]_(i+1)]_N , vector:[float]_N ) → {
  mapGlobal(λrow →
  reduceSeq(λ(acc, t) → acc+(t.0*t.1))(
    zip(row,slice(N,0,getLength(row))(vector)) )
  )(matrix) } }
```

Listing 4. Lift code for the Basic implementation of triangular matrix-vector multiplication

```
fun(matrix:[i ↦ [float]_(i+1)]_N , vector:[float]_N ) → {
 mapWorkgroup(λrow →
  mapLocal(reduceSeq(λ(acc, t) → acc+(t.0*t.1))) o
    split(SPLIT_SIZE)(
      zip(padConstant(0,N-getLength(row),0.0)(row)),
          vector)) }
```

Listing 5. Lift code for the Best implementation of triangular matrix-vector multiplication

6.1 Triangle Matrix Vector Multiplication

We start with the triangular matrix vector multiplication for which we saw already the high level Lift code in listing 3.

Lift implementations We present two different versions of the triangular matrix vector multiplication in Lift. The first one is derived from the high-level implementation, referred to here as *basic* version. The other is an improved version, written to better exploit the parallel facilities of the GPU. This version is referred to as the *best*.

The code for *basic* is shown in listing 4. The program then follows the structure of a simple high-level matrix-vector multiplication. Different to the high level Lift program in listing 3 this version includes the OpenCL specific parallel versions of *map* indicating the parallelism mapping. Global threads are used to process each row of the matrix in parallel. The only divergence from a regular matrix vector multiplication lies in the $slice(N, 0, getLength(row))$ expression, which clips the vector to the length of the current row.

The code for *best* is show in listing 5. In this version, we assign each row to a workgroup and then, instead of clipping the vector, we extend the row up to the vector length using the *padConstant* primitive. The *padConstant* primitive creates a view to lazily extend the array with a constant value. This allows us to further split the row and column vectors and process each chunk in a separate thread.

One must take note that the code for *best* presented here comprises only the first part of the algorithm. The code shown in listing 5 computes a partial reduction for each row. A second reduction kernel is then necessary. As this is a common feature of many high performance GPU applications, the code is omitted. The runtime cost of the second kernel, while negligible, is included in the results.

Generated OpenCL code Listing 6 and listing 7 show the automatically generated OpenCL codes. The outer for loop of the basic version distributes the rows across the global threads. The input index in lines 15 - 17 is automatically derived from the extended array type. It is concise following the optimizations described in section 5.

```
kernel void BASIC(const global float* restrict matrix,
                  const global float* restrict vector,
                  global float* out) {
  float accum = 0.0f;
  for (int row_idx = get_global_id(0); (row_idx < 5);
        row_idx = (row_idx+get_global_size(0))){
   for (int i = 0; i < 1+row_idx; i = 1+i) {
    accum = multAndSumUp(accum,
      matrix[(row_idx + i + (((-1 * row_idx) +
        (row_idx * row_idx)) / 2))], vector[i]); }
    out[row_idx] = id(accum); } }
```

Listing 6. OpenCL code for Lift basic implementation of triangular matrix-vector multiplication.

```
1  kernel void BEST(const global float* restrict matrix,
2                   const global float* restrict vector,
3                   global float* out, int N) {
4   float accum = 0.0f
5   for (int row_idx = get_group_id(0); (row_idx < N);
6        row_idx = row_idx + get_num_groups(0)) {
7    for (int split_idx = get_local_id(0);
8         split_idx < ((N)/(SPLIT_SIZE));
9         split_idx = split_idx+get_local_size(0)){
10    for (int i = 0; (i < SPLIT_SIZE); i = 1+i){
11     accum = multAndSumUp( accum, (
12      (((i + (SPLIT_SIZE * split_idx)) < 0) ||
13       ((i + (SPLIT_SIZE * split_idx)) >=
14         (1 + row_idx)) ) ? 0.0f :
15      matrix[(i + row_idx +
16       (((-1*row_idx) + (row_idx*row_idx))/2) +
17       (SPLIT_SIZE * split_idx))]),
18     vector[(i + (SPLIT_SIZE * split_idx))]);  }
19   out[split_idx+(N*row_idx)/SPLIT_SIZE] = id(accum);}}}
```

Listing 7. OpenCL code for Lift best implementation of triangular matrix-vector multiplication.

The *best* version's alternative parallelization strategy is more complicated: the SPLIT_SIZE parameter in the Lift code controls the amount of work in each workgroup.

Performance Results Figure 10 presents the performance results measured for the triangular matrix vector multiplication expressed in Lift and compared against the equivalent BLAS kernel *trmv* implemented in cuBLAS. cuBLAS is the fastest known high performance linear algebra library for Nvidia hardware. As we can clearly see, the Lift generated code outperforms the *trmv* cuBLAS implementation clearly on all input sizes. The *best* Lift version is also significantly faster than the *basic* version. The largest input size represents a large triangular matrix of over 500 MB and when approaching this size, the performance of cuBLAS improves and the advantage of the Lift generated code becomes smaller. Still, the Lift generated code outperforms cuBLAS by up to 2.3×.

6.2 Boundary Conditions of Stencil code

In the second case study, we show how the introduction of irregular arrays is used as a means to express a low level optimization by considering the case of boundary checking in stencil codes. Stencil codes need special handling at the boundary, for example clamping array accesses with a check and re-index computation when out-of-bound. This handling requires the introduction of branches, human experts often prefer to write extra code for handling the boundary regions.

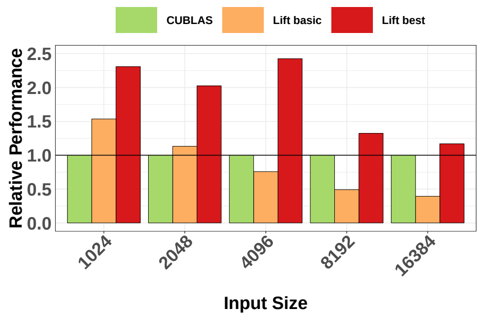

Figure 10. Relative performance of Lift triangle matrix vector multiplication implementations compared with NVIDIA cuBLAS *trmv* implementation.

```
fun(input:[[float]_N]_M, boundary:float) → {
 mapGlobal(mapGlobal(reduceSeq(add, 0) o join)) o
  slide2D(STENCIL, 1) o
    pad2D(STENCIL/2, STENCIL/2, boundary)(input) }
```

Listing 8. Jacobi stencil expressed in Lift

```
fun(input:[[float]_N]_M, boundary:float) → {
 join o mapSeq(mapGlobal((reduceSeq(add,0) o join))) o
 partition(3,caseSplit(STENCIL/2,M-STENCIL,STENCIL/2))o
 slide2D(STENCIL, 1) o
 pad2D(STENCIL/2, STENCIL/2, boundary)(input) }
```

Listing 9. Jacobi stencil expressed in Lift with specialized boundary handling

```
for (int i = get_global_id(0); i < 1+N;
     i = (i + get_global_size(0))) { acc = 0.0f;
 for (int j = 0; j < STENCIL_SIZE; j = 1+j) {
  acc = add(acc, input[
   (((-(STENCIL_SIZE/2)+i+j) >= 0)
    ? (((-(STENCIL_SIZE/2)+i+j) < N)
     ? (-(STENCIL_SIZE/2)+i+j) : -1+N) : 0)]); }
```

Listing 10. OpenCL code generated for a one-dimensional stencil without special handling of boundary conditions.

Classical approach Listing 8 shows a classic Lift implementation of a 2D Jacobi stencil. The program contains of three main steps: first, the input grid is padded, which is Lift's way of introducing specialized boundary handling. Next, *slide2D* is used to create an array of neighborhoods. Finally, the code within *mapGlobal* implements the actual stencil computation performed on each neighborhood. The main issue with this straightforward implementation lies in the result of *pad2D*, which in Lift is a *view* over the padded input array. Therefore, every access into this array needs to be guarded by boundary checks, resulting in index expressions with conditionals. But, these checks are only necessary for neighborhoods with elements falling outside the boundary. The use of regular arrays in the Lift expression prevents us to be able to express this specialized behavior.

Position dependent array approach We can solve this problem by applying the rewrite rule presented in section 4.4 that introduces the *partition* primitive. The code for the

```
// Prologue
int i = get_global_id(0);
if (i < STENCIL_SIZE/2) { float accum = 0.0f;
  for (int j = 0; j < STENCIL_SIZE; j = 1 + j) {
    accum = add(accum, input[
      (((-(STENCIL_SIZE/2) + i+j) >= 0)
        ? (-(STENCIL_SIZE/2) +i+j) : 0)]); }
...
// Body
for (int i = get_global_id(0); i < (N - STENCIL_SIZE);
     i = (i + get_global_size(0))) { float accum =0.0f;
  for (int j = 0; j < STENCIL_SIZE; j = 1 + j) {
    accum = add(accum, input[(i + j)]); }
...
// Epilogue
int i = get_global_id(0);
if (i < STENCIL_SIZE/2) { float accum = 0.0f;
  for (int j = 0; j < STENCIL_SIZE; j = 1 + j) {
    accum = add(accum, input[
        (((-(STENCIL_SIZE / 2) + i + j + N) < N)
          ? (-(STENCIL_SIZE / 2) + i + j + N)
            : (-1 + N))]); }
```

Listing 11. OpenCL code generated for a one-dimensional stencil with special handling of boundary conditions.

rewritten Lift program is shown in listing 9. In this version, a *partition* call has been introduced, splitting the input to the stencil in three - unequally sized - areas: left boundary, center, and right boundary. Since *partition* has a known constant number of partitions, the code generator will not produce a for-loop when mapping over it, but instead fully unroll it. This will yield three different sections, corresponding to the *prologue*, *body* and *epilogue* of the stencil computation.

Moreover, since the Lift compiler accurately tracks the ranges of iteration variables, it also infers that the *prologue* and *epilogue* sections are implemented with an `if` instead of a `for` loop, since there is at most one iteration per thread.

The effects of this transformation on the generated OpenCL code are visible by comparing the code snippets shown in listing 10 and listing 11. For sake of clarity, we are showing the code for a one-dimensional stencil: the principle is the same as for the 2d stencil used in the experimental evaluation one with additional loops in the OpenCL code and more complex index computations.

The traditional Lift stencil code has a single nested loop that is performing the entire computation. Every access into the input performs the costly out-of-bound checks.

In the rewritten stencil the computation has been split in three separate code sections, where only the prologue and epilogue containing the boundary checks. This is possible, as the arithmetic expression simplifier automatically infers closer bounds for the loop variables guaranteeing that such checks are redundant for the body of the stencil. It is therefore safe to omit them.

Performance Results Figure 11 presents the performance impact of applying the rewrite rule on a 2D Jacobi stencil when we vary the size of the stencil. The bars represent the speedup of the rewritten version compared with the non-rewritten baseline for a variety of stencil sizes. The data shown is relative to an grid-input size of 4096×4096 floating

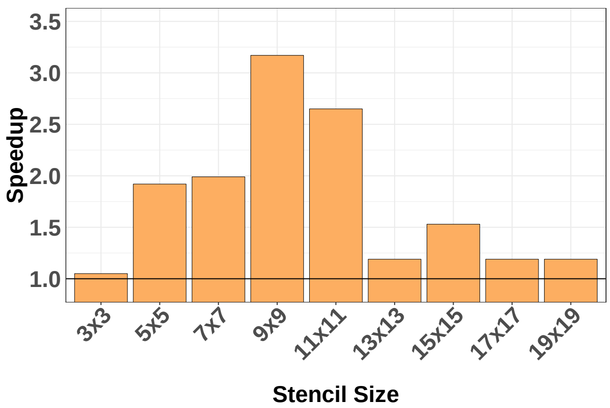

Figure 11. Relative performance benefits of specialized boundary handling over traditional boundary handling

point entries. As we can see, the effect on performance are positive, with a peak performance gain of approximately 3.2× for the 9 × 9 stencil size, with lesser gains as the sizes increase or decrease. In no case was a slowdown measured.

Concerning the uneven distribution of performance, we suspect this may be due to the behavior of the OpenCL compiler: heuristics-driven optimizations such as loop-unrolling and constant-propagation have a significant impact on the performance of stencil programs. As the rewritten kernel is slightly different for all these sizes, there might be some unexpected interactions between these two optimizations.

7 Related Work

High Level GPU Programming Languages such as Accelerate [13], Futhark [11], Halide [14], and Lift [17, 18] aim to simplify GPU programming through the use of parallel patterns, while at the same time allowing for the generation of efficient code. Each of these provides some approach to track the size of arrays at the type system level, in order to improve performance and correctness.

While these facilities are adequate to deal with regularly shaped arrays, they are lacking when dealing with irregular data structures, such as triangular matrices.

Streaming Irregular Arrays Streaming irregular arrays [5] is an addition to the Accelerate [13] language which provides support for irregular data structures backed by a flattened representation, and provides type-system support for reasoning with nested irregular arrays. Unlike the work presented here however, it relies on run-time support for tracking the sizes of arrays, as opposed to attempting to resolve index computations fully at compile time. The main reason for the difference is it's focus on sparse data structures who are accessed as data streams, as opposed to our work, which has to date focused on dense data representations backed by arrays.

Dependent Types Dependently typed languages such as the earlier Epigram [12] or the more modern Idris [4] possess type systems capable of encoding and enforcing complex properties over values in the type. This might include properties concerning the shape of data structures, which enable dependently typed programs to naturally express irregular data structures.

This great expressive power comes however at a cost, and efficient compilation of dependent type languages is an active area of research. We use a limited form of dependent types in this work for a domain-specific purpose: to describe parallel computations and to generate efficient parallel code.

Irregular structures in linear algebra applications The linear algebra community has seen the development of a number of approaches to produce efficient implementations for complex linear algebra problems, such as the FLAME methodology [9], a systematic way for deriving parallel algorithms for linear algebra operations, and Linnea [2], a rule based rewrite engine capable of generating efficient implementations of complex linear algebra expressions from a high-level mathmetical expression.

The work presented in this paper would allow Lift to serve as a high-performance code generator for high level programs derived using such tools.

8 Conclusions

This paper has presented an extension to classical array types. While existing functional code generators already track the size of arrays in the type, this is overly restrictive and prevents useful data structures such as triangular matrices or trees to be represented precisely as types. In this paper we have shown how to design an extended array type with a limited form of dependent typing that allows for nested array sizes to depend on their position in the surrounding array.

We have shown our practical implementation as an extension of the Lift compiler and presented the implementation challenges mostly related to index simplification. We have demonstrated that this approach enables the efficient code generation of triangular matrix vector multiplication, with performance improvements over cuBLAS on an Nvidia GPU by up to 2×. We have discussed a use case for representing and implementing a low level optimization for avoiding out-of-bound checks, leading to up to 3× performance improvement over already optimized stencil codes

In the future, we would like to further extend our array type, exploring the practical implications of representing tree data structures in a packed memory representation, as common in computer graphics applications.

Acknowledgments

We thank Larisa Stoltzfus for her help with plotting results and Bastian Hagedorn for advice with the Lift stencil codes. We thank the entire Lift team for their development efforts.

References

[1] Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach. 2017. Strategy Preserving Compilation for Parallel Functional Code. *CoRR* abs/1710.08332 (2017).

[2] Henrik Barthels and Paolo Bientinesi. 2017. Linnea: Compiling Linear Algebra Expressions to High-Performance Code. In *Proceedings of the International Workshop on Parallel Symbolic Computation, PASCO@ISSAC 2017, Kaiserslautern, Germany, July 23-24, 2017*. 1:1–1:3. https://doi.org/10.1145/3115936.3115937

[3] Paolo Bientinesi, Brian Gunter, and Robert A. van de Geijn. 2008. Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix. *ACM Trans. Math. Softw.* 35, 1, Article 3 (July 2008), 22 pages. https://doi.org/10.1145/1377603.1377606

[4] Edwin Brady. 2013. Idris: general purpose programming with dependent types. In *PLPV*. ACM, 1–2.

[5] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. 2017. Streaming irregular arrays. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*. 174–185. https://doi.org/10.1145/3122955.3122971

[6] Thiago de Castro Martins, Jacqueline de Miranda Kian, André Kubagawa Sato, and Marcos de Sales Guerra Tsuzuki. 2012. Matrix-vector multiplication and triangular linear solver using GPGPU for symmetric positive definite matrices derived from elliptic equations. In *SCIS&ISIS*. IEEE, 1286–1291.

[7] Yin Ding and Ivan W Selesnick. 2016. Sparsity-based correction of exponential artifacts. *Signal Processing* 120 (2016), 236–248.

[8] Google et al. 2017. XLA (Accelerated Linear Algebra): domain-specific compiler for linear algebra that optimizes TensorFlow computations.

[9] John A Gunnels, Fred G Gustavson, Greg M Henry, and Robert A Van De Geijn. 2001. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software (TOMS)* 27, 4 (2001), 422–455.

[10] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with Lift. In *CGO*. ACM.

[11] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *PLDI*. ACM.

[12] Conor McBride. 2004. Epigram: Practical Programming with Dependent Types. In *Advanced Functional Programming (Lecture Notes in Computer Science)*, Vol. 3622. Springer, 130–170.

[13] Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising purely functional GPU programs. In *ICFP*. ACM.

[14] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*. ACM.

[15] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. 2016. Performance portable GPU code generation for matrix multiplication. In *GPGPU@PPoPP*. ACM, 22–31.

[16] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018).

[17] 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.

[18] 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.

[19] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018).