Pizzuti, F., Steuwer, M. and Dubach, C. (2020) Generating Fast Sparse Matrix Vector Multiplication from a High Level Generic Functional IR. In: ACM SIGPLAN 2020 International Conference on Compiler Construction (CC 2020), San Diego, CA, USA, 22-23 Feb 2020, pp. 85-95. ISBN 9781450371209 (doi:10.1145/3377555.3377896).

There may be differences between this version and the published version. You are advised to consult the publisher's version if you wish to cite from it.

© The Authors 2020. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of the ACM SIGPLAN 2020 International Conference on Compiler Construction (CC 2020), San Diego, CA, USA, 22-23 Feb 2020, pp. 85-95. ISBN 9781450371209.

http://eprints.gla.ac.uk/206621/

Deposited on: 13 February 2020

Enlighten – Research publications by members of the University of Glasgow
http://eprints.gla.ac.uk

Generating Fast Sparse Matrix Vector Multiplication from a High Level Generic Functional IR

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

Usage of high-level intermediate representations promises the generation of fast code from a high-level description, improving the productivity of developers while achieving the performance traditionally only reached with low-level programming approaches.

High-level IRs come in two flavors: 1) domain-specific IRs designed only for a specific application area; or 2) generic high-level IRs that can be used to generate high-performance code across many domains. Developing generic IRs is more challenging but offers the advantage of reusing a common compiler infrastructure across various applications.

In this paper, we extend a generic high-level IR to enable efficient computation with sparse data structures. Crucially, we encode sparse representation using reusable dense building blocks already present in the high-level IR. We use a form of dependent types to model sparse matrices in CSR format by expressing the relationship between multiple dense arrays explicitly separately storing the length of rows, the column indices, and the non-zero values of the matrix.

We achieve high-performance compared to sparse low-level library code using our extended generic high-level code generator. On an Nvidia GPU, we outperform the highly tuned Nvidia cuSparse implementation of SpMV (Sparse-matrix vector multiplication) multiplication across 28 sparse matrices of varying sparsity on average by 1.7×.

Keywords. Sparse Matrix, Code Generation, Dependent Types

1 Introduction

Achieving high performance on modern parallel hardware is a challenging task even for experienced programmers. The trend towards specialized hardware fulled by the end of Moore's law makes this even more challenging: Programmers in low-level languages are now required to develop specially optimized solutions for each new hardware target. It is costly and not always feasible to manually develop optimized solutions for new hardware targets. High-level IRs attempt to address this challenge. They allow the generation of fast code from a high-level platform-independent program description. They can target a wide variety of hardware, such as multi-core CPUs, GPUs and FPGAs, This approach has been pioneered by projects such as Delite [21], Halide [17], Lift [19, 20], AnyDSL [13], or more recently in the domain of machine learning XLA [10] and TWM [5].

Many high-level IRs are domain specific focusing on a single application domain and exploit domain knowledge embedded in the high-level programming abstractions generating optimized code. Examples of these are Halide [17] for image processing and the tensor algebra compiler TACO [12] for sparse tensor and linear algebra applications. These code generators develop their own intermediate representation geared towards the specific domain, such as *iteration graphs* used as an intermediate representation by TACO to express sparse tensor computations.

Developing specialized tools and infrastructure requires significant effort for each new domain as reuse is severely limited. Therefore, several projects attempt to simplify the development of domain-specific compilers themselves by providing a reusable high-level IR that is reused across many domains. Delite [21] pioneered this approach together with more recent projects such as Lift [19, 20] and AnyDSL [13]. Delite and Lift provide universal parallel patterns as building blocks to describe computations.

High-level IRs have to date mostly been used for fast computations over dense data structures, such as higher dimensional arrays known as *tensors.* While many essential applications operate on dense data, there are many others – such as graph algorithms – that naturally operate on sparse data. Furthermore, in some domains such as deep learning, there is a push towards greater efficiency by investigating sparse data representations. There is clearly a need to develop a high-level IR for efficient computations on sparse data. While TACO has demonstrated that this is possible using a custom-designed domain-specific IR, we are interested in exploring the extension of an existing generic high-level IR to for the generation of efficient code for computations on sparse data.

In this paper, we show that it is possible to extend a Lift-like pattern-based high-level IR for the generation of efficient code computing on sparse data structures. We reuse the existing design and implementation to keep the extension small, allowing us to take advantage of the existing infrastructure as much as possible. Our approach follows the observation that in low-level programming, programmers are explicitly encoding sparse data structures in memory buffers storing dense data. Crucially, sparse matrix formats such as compressed sparse row (CSR) represent a single sparse matrix in

$$
\begin{array}{lll}
\textbf{\textit{map}} & : & (T \to U) \to [T]_N \to [U]_N \\
\textbf{\textit{reduce}} & : & (T \to T \to T) \to T \to [T]_N \to T \\
\textbf{\textit{zip}} & : & [T]_N \to [U]_N \to [(T,U)]_N \\
\textbf{\textit{split}} & : & N \to [T]_{N \cdot M} \to [[T]_N]_M \\
\textbf{\textit{join}} & : & [[T]_N]_M \to [T]_{N \cdot M}
\end{array}
$$

Figure 1. Data parallel patterns

multiple memory buffers that relate to each other: the length of rows and column indices stored in two dense arrays enables to index the values that are stored in a third array. We express these relationships in the types of our pattern-based functional intermediate representation. For this, we use a limited form of dependent typing, a typing discipline where types are allowed to depend on runtime values – in our case the length of arrays is allowed to be represented by elements of a different array. This work builds upon prior work [16] that has used dependent typing in a high-level IR to representing irregularly shaped data structures such as triangular matrices. This papers significantly extends the use of dependent types to express relationships between multiple data structures, enabling the representation of sparse data structures such as sparse matrices in CSR format.

Our experimental results demonstrate that the proposed minor extension of the generic high-level IR enables the generation of efficient code for sparse matrix vector multiplication on an Nvidia GPU across 28 sparse matrices of varying sparsity.

In summary, this paper makes the following contributions:

- We describe sparse matrix data types with a limited form of dependent types allowing to express computations with existing parallel patterns;
- we describe our compiler implementation generating efficient OpenCL code for sparse matrix computations;
- we present a detailed performance evaluation demonstrating the competitive performance of the code generated form our high-level IR when compared to optimized low-level library code.

2 Pattern-Based High-Level Code Generation

Parallel patterns, aka algorithmic skeletons [8], capture computational patterns for which an efficient parallel implementation over container data types exists. A small set of generic parallel patterns has proven to be sufficient as a flexible and powerful way to represent data parallel computations. Patterns such as *map* and *reduce* are nowadays common vocabulary for describing applications at a high level. These patterns have been used as the foundation for pattern-based high-level IRs such as Delite [21] and Lift [19, 20].

$$
\begin{array}{lll}
\langle expr \rangle ::= x & & \textit{variables} \\
\quad \mid \underline{0} & & \textit{literals} \\
\quad \mid x \ @ \ \langle expr \rangle & & \textit{indexing into arrays} \\
\quad \mid (\langle expr \rangle, \langle expr \rangle) & & \textit{pair construction} \\
\quad \mid \langle expr \rangle.1 \mid \langle expr \rangle.2 & & \textit{pair projection} \\
\quad \mid \langle function \rangle(\langle expr \rangle) & & \textit{function application} \\
\quad \mid \langle expr \rangle \ {:}{>>} \ \langle function \rangle & & \textit{reverse fun. application} \\
\quad \mid \langle expr \rangle : \langle datatype \rangle & & \textit{type annotations} \\
\\
\langle function \rangle ::= \textbf{\textit{map}} \mid \textbf{\textit{reduce}} & & \\
\quad \mid \textbf{\textit{zip}} \mid \textbf{\textit{split}} \mid \textbf{\textit{join}} & & \textit{data parallel patterns} \\
\quad \mid \mathtt{fun}(x \Rightarrow \langle expr \rangle) & & \textit{lambda expression} \\
\quad \mid \mathtt{nfun}(n \Rightarrow \langle expr \rangle) & & \textit{dependent lambda expr.} \\
\\
\langle datatype \rangle ::= \mathtt{float} \mid \mathtt{int} & & \textit{scalar types} \\
\quad \mid (\langle datatype \rangle, \langle datatype \rangle) & & \textit{pair type} \\
\quad \mid [\langle datatype \rangle]_{\langle nat \rangle} & & \textit{array type} \\
\quad \mid [i \mapsto \langle datatype \rangle]_{\langle nat \rangle} & & \textit{position dep. array type} \\
\\
\langle nat \rangle ::= n \mid i & & \textit{variables} \\
\quad \mid \underline{0} & & \textit{literals} \\
\quad \mid \langle nat \rangle + \langle nat \rangle \mid \langle nat \rangle \cdot \langle nat \rangle & & \textit{arithmetic ops.} \\
\quad \mid \sum_{i=\langle nat \rangle}^{\langle nat \rangle} \langle nat \rangle & & \textit{summation} \\
\quad \mid \langle bool \rangle \ ? \ \langle nat \rangle : \langle nat \rangle & & \textit{ternary operator} \\
\quad \mid \mathtt{toNat}(\langle expr \rangle) & & \textit{expressions at the type level}
\end{array}
$$

Figure 2. Grammar for a data parallel function language

Figure 1 shows the small set of data parallel patterns that is sufficient to express a large class of algorithms across multiple domains, as demonstrated by Lift.

Figure 2 shows the grammar of a small functional language that is suitable as a compiler intermediate representation and is close to the one used by Lift and described here [20]. The two crucial items for representing sparse matrices are highlighted in the figure.

Expressions in this language are variables, literals, capability for indexing in arrays and handling pairs. Function application is either written conventionally with parentheses or in reverse order using the pipe operator (:>>), finally expressions can be annotated with their data type.

Functions in this language can be a data parallel pattern, a function definition (also called a *lambda expression*), or a dependent function definition. A dependent function definition is written as **nFun**(*n* => body) where the parameter n is a variable over natural numbers that can appear in the *type* of body or one of its sub expressions.

Data types are either scalar, pair or array types. Array types are written as $[T]_N$ where T is an arbitrary data type

and N is a natural number representing the length of the array. Note that arrays can be nested to represent arbitrarily deep nested data structures. For supporting computations over sparse matrices we add *position dependent array types* that allow the data type of elements to depend on the position in the array. These will be described in more detail in section 3.1.

Natural numbers describe the length of arrays and are algebraic formulas consisting of variables, literals and computations over them, including common arithmetic operations such as addition and multiplication. Notable is the summation operator, that allows to express formulas such as $\Sigma_{i=0}^{n} i$. The ternary operator ? allows to express if-then-else style conditional arithmetic expressions. Boolean expressions are standard and not shown for brevity. In this paper, we add the **toNat** construct allowing to embed expressions – such as indexing into an array – at the type level. These will be described in more detail in section 3.3.

Using this language we represent computations such as the dense matrix vector multiplication as shown in Listing 1. Here the ***map*** primitive is used to compute the dot product between each row of the matrix mat with the input vector xs. The dot product itself is expressed in line 4 using the ***zip***, ***map***, and ***reduce*** primitives.

```
1 nFun(n => nFun(m =>
2  fun(matrix: [[float]n]m => fun(xs: [float]n =>
3   matrix :>> map(fun(row =>
4    zip(xs, row) :>> map(*) :>> reduce(+, 0)
5   )) )) ))
```

Listing 1. Dense matrix vector multiplication

The Lift project has shown how this high-level program can be transformed into high-performance code on a series of hardware architectures by exploring optimization and implementation choices expressed as rewrite rules. We are keen to make us of these existing ideas and infrastructure and, therefore, aim for reusing the same computational patterns when expression computations over sparse matrices. In this paper, we propose to use two extensions to the standard pattern-based functional language for representing sparse matrices. The two additional constructs are: position dependent array types and **toNat** for embedding expressions at the type level. We discuss these extensions in the next section.

3 Representing Sparse Data

There are many possible approaches for representing sparse data structures at a high-level. TACO [6] proposed the use of a specific abstraction for specifying sparse array formats. This domain-specific representation allows the representation of many different formats, but requires a matching domain-specific code generator. Mixing and integrating such specialized representations with generic representations is not obvious hindering the development of holistic code generators that allow to optimize entire applications or that enable the smooth transition from dense to sparse code.

Generic high-level code generators have to stick to generic data representations. In some high performance code generators, such as Accelerate [15], runtime information is injected in the data structure directly to model irregularity [7]. While this approach is very flexible, it introduces runtime computation that could be avoided, and it limits somewhat the range of possible sparse formats expressed.

In this paper, we advocate an approach to represent sparse matrix formats using position dependent array types [16] and by using certain runtime values at the type level using the ***toNat*** construct. We will show how these two additions are sufficient to express popular sparse matrix formats.

3.1 Position Dependent Array Types

Traditionally, array type requires all elements to be homogeneous — they are all of the same type. This important property guarantees that it is always possible to find the address in memory of any element using the index of the element and the element size, which is constant. When the length of the array is part of the type, as presented in section 2, this restricts nested arrays to regularly shaped tensors. Often this is overly restrictive, but high-level code generators, like Lift, rely on the encoding of the length of arrays in the type for generating efficient code.

Prior work [16] introduced the notion of *position dependent array types* that are a generalization of traditional array types allowing a limited degree of heterogeneity. More precisely, the type of the elements is allowed to depend on their position within the array itself. We write these arrays as:

$$[i \mapsto T]_N$$

where N is a natural number denoting the length of the array, i is a natural number variable ranging from 0 up to $N-1$, representing the indices. Finally, T is a data type that might contain i. As the only type containing natural numbers is the array type representing the length of arrays, a form of homogeneity is still maintained – one can not store elements of different scalar types, such as ints and floats, in the same array. This means that a *position dependent array* is still implemented efficiently with a flat memory representation.

3.2 Statically-Shaped Matrix Representation

*Position dependent array*s encode arbitrary statically shaped data structures. Figure 3-a, shows an example of two statically shaped matrices. The first is a 3×3 matrix with a flat representation in memory and its type on the right.

As shown in prior work [16], it is possible to define more interesting matrix shape using position-dependent array types. A lower triangular matrix can be represented using this abstraction easily as shown in the example, where the lengths of the inner array is a function of its position (*e.g.*, first row has 1 element, the second has 2, and so on).

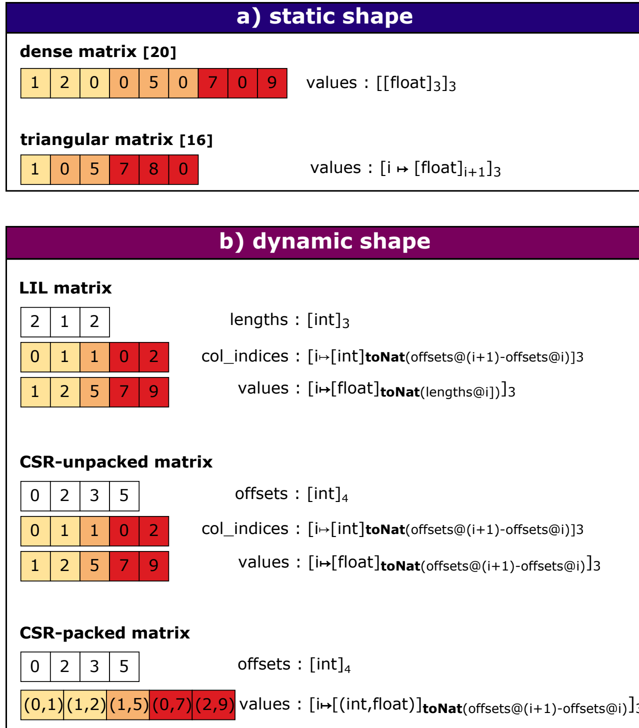

Figure 3. Layout and data types for representing matrices.

3.3 Expressions as Types: `toNat`

Position dependent arrays are not sufficient to implement sparse data structures since their shape depends on a runtime values. To express this dependence we extend the natural number part of our language with a new construct we call `toNat`. This constructs wraps an expression of type `int`, and represents the results of evaluating that expression at runtime. Using `toNat`, it is now possible to express an array whose length is not statically determined at compile time, but instead is read from another array by indexing into it.

3.4 Sparse Matrix Formats

Sparse matrix formats are encoded explicitly by explaining how a sparse matrix is stored in usually multiple flat memory buffers. Often one array contains the non-zero elements of the matrix, supported by a number of additional arrays of metadata, specifying the data structure's logical layout and access patterns. In a lower-level language, these details are manually specified by the programmer explicitly in the code, impacting every computation involving the sparse matrix.

With the introduction of `toNat`, we have all the necessary components to describe the relationships between these multiple low-level arrays representing the sparse matrix in the types of our high-level programming language. By doing so, we define the sparse matrix concisely and locally and use the regular patterns and abstractions transparently through the rest of the code. We will now explore how the LIL (List of Lists) and CSR (Compressed sparse row) formats for sparse matrices are encoded at the type level, shown in Figure 3-b.

LIL Matrix Format: The LIL format uses 3 arrays: one for each row's length, one for column indices of non-zero elements and one for values.

As shown in Figure 3-b, the type of the `values` array is position-dependent and dynamic. The length of each inner array depends not only on the position, but also on the data in the `length` array storing the length of each row.

CSR-Unpacked Matrix Format: Matrices in the CSR-unpacked format use 3 arrays [18]: an `offsets` array pointing to the start of each row, one containing the column index of each non-zero element and one for the values. The type of `values` contains the computation of the length of each row by subtracting two successive offsets. This format is usually preferred over LIL, as the start of each row in memory is immediately accessible, which is useful when parallelism is introduced since each thread knows its rows offset. For LIL, inferring the start of a row involves computing the sums of all the previous rows lengths. Section 5.2 introduces an optimisation for reducing this overhead.

CSR-Packed Matrix Format: The last format we are discussing is CSR-packed, a variant of the previous one. Instead of using two distinct arrays to store the indices and values, these are fused into a single array. The rational for this format is that the values and indices are often accessed together, therefore, storing them contiguously improves locality. As will see in the next section, packing column indices with values simplify the high-level implementation of many programs. Secondly, it has a positive impact on performance, as demonstrated in section 6.4

3.5 Summary

We have seen in this section how different matrix layouts can be represented using our type system and abstractions. We allow nested array element types to depend on their position in the outer array. We combine this with the capability to allow expressions computing integer values to represent the length of arrays at the type level. These two additions enable the representation of sparse matrix formats. The next section shows how sparse-matrix multiplication is implemented using these different data layouts.

4 Sparse Matrix Vector Product

In section 3, we showed how we could leverage the type system to represent matrix formats. In this section, we show how to express computations over these matrices.

We consider the case of matrix vector product, one of the most common operations for sparse matrices. We shall first see how to implement this for dense matrices, and then explore what changes are needed to use sparse matrices.

Dense Matrix Vector Product: Listing 2 contains the code for dense matrix-vector multiplication. The ***map*** primitive is used to compute the dot product between each row of `matrix` with the vector `xs`. The dot product itself is expressed in line 4 using the ***zip***, ***map***, and ***reduce*** primitives.

```
1 nFun(n => nFun(m =>
2 fun(matrix:[[float]n]m => fun(xs:[float]n =>
3  matrix :>> map(fun(row =>
4   zip(xs,row) :>> map(*) :>> reduce(+,0)))))))
```

Listing 2. Dense matrix vector multiplication

LIL Matrix Vector Product: Listing 3 shows the LIL sparse matrix vector product. The `matrix` parameter of the dense case has been split out in three parameters: `row_length` is the array with the length of each row, and `col_indices` is the array of column indices.

```
1 nFun(n => nFun(m =>
2 fun(row_length:[int]n =>
3 fun(col_indices:[i ↦ [int]toNat(lengths@i)]n =>
4 fun(values:[i ↦[float]toNat(lengths@i)]n =>
5 fun(xs:[float]m =>
6  zip(col_indices, values) :>>
7  map(fun(rowPair =>
8   zip(rowPair.1, rowPair.2) :>>
9   map(fun(x=>x.2*(xs@x.1))) :>> reduce(+, 0) )) ))))))
```

Listing 3. SpMV implementation for LIL matrix

The core of the program is similar to the dense case of listing 2, with two critical changes. First, and additional top-level ***zip***, used to iterate in lockstep over the rows of both the values and the index array. Secondly, we no longer use a ***zip*** to pair matrix and vector elements for the multiplication. Instead, we rely on pairing together row index and value and use the row index to access the correct vector elements.

CSR-Unpacked Matrix Vector Product: The code for the unpacked CSR format is identical to that of the LIL format, except for the definition of the `values` array, which uses a different size formula that is shown in listing 4. We also renamed `row_length` to `offset` for clarity.

```
values: [i ↦[float]toNat(offset@(i+1)-offset@i]n
```

Listing 4. SpMV implementation for CSR-unpacked matrix

CSR-Packed Matrix Vector Product: In section 3.4 we mentioned an alternative to storing column indices in a separate array, using a *structure of arrays to array of structures* transformation to pack the column index and the value together. We mentioned that this transformation has benefits both for performance and program clarity.

Listing 5 shows the code for the CSR format with this transformation applied. As we can see, the code is much simpler, not only due to the removal of the additional `col_indices` array, but also of the applications to ***zip***, as column index and value are already associated.

```
1 nFun(n => nFun(m =>
2 fun(offset:[int]n+1 =>
3 fun(values:[i ↦[(int,float)]toNat(offset@(i+1)-offset@i)]n =>
4 fun(xs:[float]m =>
5  values :>> map(fun(row =>
6  row :>> map(fun(x=>x.2*(xs@x.1)))
7      :>> reduce(+, 0) )) ))))))
```

Listing 5. SpMV implementation for CSR-packed matrix

5 Code Generation

In a high-level code generator, operations over data structures are expressed with algorithmic patterns such as ***map*** and ***reduce***. These operations abstract away aspects of a low-level implementation, including iteration, parallelism and data structure access, and use the type information of the data they operate upon to generate the necessary implementation. In section 3, we showed how to represent the format of sparse matrices by encoding the relationship between multiple arrays in their types. In this section, we present the transformation from a functional expression to C code.

5.1 Compilation Process

To illustrate the compilation steps necessary to implement our proposed method, we use a simple example: summing up every row of a sparse matrix encoded in the CSR format:

```
1 nFun(n=>
2 fun(offsets:[int]n+1 =>
3 fun(values:[i ↦[float]toNat(offsets@(i+1)-offsets@i)]n =>
4  values :>> map(reduce(+, 0.0f)) ) ))
```

Listing 6. Sum of rows for CSR matrix

Figure 4 shows a diagram of the compilation steps. For simplicity, here we illustrate the compilation process using sequential C as our target. In reality, our implementation produces parallel OpenCL code.

Parameter Collection: The first construct the compiler sees is the dependent function defining the arithmetic variable N

```
nFun(n=>...)
```

No code is generated: the compiler simply records that the following expression is parametric in this variable. If this expression is at the top-level of the program, then this information will also be used to generate an input parameter `int n` for the resulting kernel. Next, the compiler sees:

```
fun(offset:[int]n+1 => ...)
```

Then, another lambda definition is encountered

```
fun(values:[[float]toNat(dict@(i+1)-dict@i)]n =>...)
```

Similarly to the prior lambda, the information is recorded in the scope's environment. The array `values` contains a ***toNat*** node in its size, indicating it is a sparse matrix. As

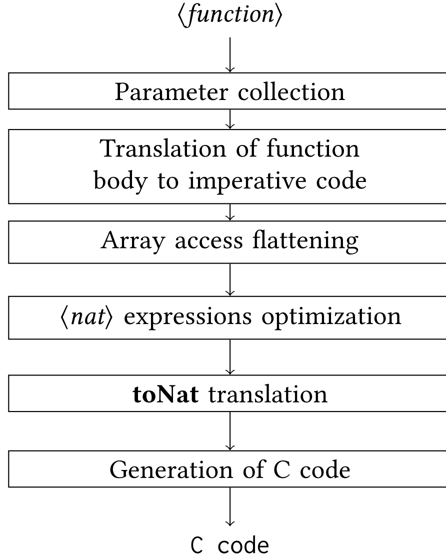

Figure 4. Steps for compiling programs with computations over sparse matrices.

this is considered as an ordinary symbolic expression, the compiler treats it no differently that any other array.

Translation of Function Body to Imperative Code: Finally, we arrive at the function's body:

```
values :>> map(reduce(+,0))
```

The first transformation performed is to translate this functional expression into an abstract imperative language.

```
1  for i in 0...n
2   float accum = 0.0f
3   for j in 0...toNat(offset@(i+1)-offset@i)
4     accum += (matrix@i)@j
5   output@i = accum;
```

We replace the ***map*** and ***reduce*** patterns with a **for** loop and explicit array accesses. Additional memory buffers implied in the functional expressions are generated. Multiple methodologies exist for this transformation, e. g. [2, 20].

Array Access Flattening: At this point, the abstract language still supports the notion of nested arrays. In the next step, all array accesses are flattened. Using the typing information, the compiler can generate the appropriate symbolic formula for flattening chains of accesses. In our case, we are accessing the two dimensional dependent array `values` at indices (i, j). The resulting flattened access formula is thus

$$(\sum_{i=0}^{N-1} \texttt{toNat(offset@(}i\texttt{+1)-offset@}i\texttt{)}) + j$$

Notice that the summation is necessary due to the usage of a dependent array: as every element has different size, we cannot simply multiply the index by the inner dimension as is the case for homogeneous arrays.

⟨nat⟩ Expressions Optimization: We now have an imperative program that is not too far from our C target. The last step necessary is to translate the various algebraic formulas, that currently contain constructs such as $toNat$ and the $\sum$, into an imperative implementation.

Translate $\sum$: For $\sum$ we adopt one of three strategies:

- Naively implement it as a sequential for loop. This solution has very poor performance: it would imply computing a prefix sum for every array access.
- If possible, use algebraic properties of $\sum$ to express it in an equivalent closed form formula.
- Offload the responsibility to compute the prefix sum to the host code, memorizing the result into a temporary array, and replacing $\sum$ with lookups.

For our running example, we can rely on properties of the lookup function to eliminate the $\sum$. We will explore the case of memorization on the host in section 5.2.

Closed Form of $\sum$: In order to simplify the index access formula, we notice that the `toNat(offset@(`i`+1)-offset@`i`)`, being a lookup into a array of offsets as specified by the CSR format, can never be a negative number. We notice that for any such never-negative function f, we have that

$$\sum_{i=c}^{j} f(i+1) - f(i) = f(j+1) - f(c)$$

The compiler uses this rule, matching $f(i{+}1)$ to `offset@(i+1)` and $f(i)$ to `offset@i`, to replace the summation with the closed form formula `toNat(offset@(i+1)-offset@0)`.

Further ⟨nat⟩ Expression Optimizations: Using knowledge about the CSR algorithm, we could further optimize the generated code removing the access to offset@0, as it is always known to be 0. To enable this additional optimization, we do not rely on an ad-hoc method. Instead, we can further refine the expression of the `values` array to be

```
values:[[float]toNat(offset@(i+1)) - i == 0 ? 0 : toNat(offset@i)]n
```

In this more advanced version, we split the index computation in two different calls to ***toNat***, and guard the second call with the conditional operation `if i == 0 ? 0 : toNat(offset@i)`. As our symbolic algebra engine can reasoning with conditional branches when this expression is matched to $f(0)$, constant propagation evaluates the whole branch to a constant 0, thus reducing the whole offset computation to a single lookup.

`toNat` ***Translation:*** Having removed $\sum$ terms, the last step is the translation of `toNat` nodes into imperative code. This process is relatively straightforward: because of scoping rules, we know by construction that all information necessary is already in scope. We then recursively start the code generation process for the expression embedded in the `toNat` node and inline the results.

Generation of C Code: Having removed all remaining level abstractions, the resulting C code can be trivially generated

```
1 for(int i=0;i<n;i++) {
2   float accum = 0.0f
3   for(int j=0;j<offset[i+1]-offset[i];j++) {
4     accum += values[offset[i+1]+j] }
5   output[i] = accum; }
```

5.2 Precomputation of $\sum$ Formulas

In the previous section we have shown one possible strategy for removing $\sum$ from arithmetic formulas by symbolic simplification. In general, however, this is not possible. Consider for example the case of a sparse matrix in LIL format:

```
1 values: [i=>[float]_toNat(lengths@i)]_n
```

Accessing this array at index (i, j) is expressed as:

$$(\sum_{0}^{i-1} \texttt{toNat(lengths@i)}) + j$$

Unlike for the CSR case, there is no closed form formula for this summation, the $\sum$ cannot be simplified away . We previously mentioned the possibility of implementing $\sum$ with a sequential for loop. Such an implementation is very inefficient when implemented naively, leading to much redundant computation in every array access computation.

An alternative implementation is to compute all values of the $\sum^{i}$ in a single pass and cache the results. Then, every instance of $\sum_{0}^{j} e_j$ is replaced by a lookup into the generated data structure at index j. As a further optimization, as our work is in the context of the generation of GPU kernels, we move the computation from the GPU to the host side, where it can be efficiently computed once, and then is passed onto the GPU as an additional parameter.

Host Side Computation: The computation of the concrete values for each memorized expression happens on the host runtime, before the GPU kernel is launched. At that time, the host runtime has access to all kernel arguments, and thus can derive the exact iteration spaces and values of every index in the program. In our implementation, these arrays are computed by interpreting the algebraic formula, including the programs used to compute runtime arithmetic values.

Automatic Derivation of CSR from LIL format: The precomputation of $\sum$ formulas is a generic mechanism, that is not specific to sparsity. We apply this generic technique in the context of sparsity to automatically derive the CSR formats from the LIL format. Generally, it is easier to work in the LIL format, that directly captures the intuition of sparse matrix formats that the length of rows is computed by a function l: `[i=>[T]`$_{l(i)}$`]`$_N$. Where l looks up the length in some helper data structure. Accesses to *matrix* at index (i, j) would then yield the arithmetic formula

$$(\sum_{k=0}^{i-1} l(k)) + j$$

There is no simplification rule for this general formula. Instead, we precompute the prefix sum of $l(i)$ for $i \in [0; N]$, that is, for all possible values that the sum could take in our program. As it turns out, the resulting metadata is the same as the array of offsets that the CSR format expects.

5.3 Summary

This section has described the compilation from a functional expression to C code. We discussed optimizations for generating efficient code by simplifying or precomputing algebraic formulas. Next, we will evaluate our approach on 28 matrices using the sparse matrix vector product as our case study.

6 Evaluation

In this section we evaluate our approach using SpMV (Sparse-matrix vector multiplication). We first investigate the effect of different optimizations discussed in section 5 and different matrix formats. Then, we compare the performance of the generated code against Nvidia's cuSparse library.

6.1 Experimental Setup

We conducted an experimental evaluation using single precision floats on a GeForce GTX TITAN X with CUDA version 10.0, driver version 375.66. All runtimes are the median of 100 executions. Measurements are reported using the OpenCL and CUDA profiling APIs. We ignore data transfer times since we focus on measuring the quality of the generated kernel code. We used 28 sparse matrices obtained from the SuiteSparse Matrix Collection [1] that cover a range of density and sizes, as detailed in table 1.

6.2 Closed Form of $\sum$

Section 5.1 discussed the challenges faced while generating code for symbolic expressions containing $\sum$ terms. We noticed that the naive implementation of generating a sequential for-loop for each such term had prohibitive performance costs, and have highlighted the need for optimising the calculation of $\sum$ terms. Figure 5 shows the speedup for using algebraic simplification to replace the $\sum$ with a closed form expression for CSR SpMV. The speedup correlates with matrix size, as the more rows, the more $\sum$ iterations need be computed. The high speedup obtained demonstrates that the for-loop strategy is not viable in practice.

6.3 LIL vs CSR

Section 3 demonstrated how the LIL and CSR formats can be represented using our technique. Figure figure 7 shows the relative performance of the version compared with LIL.

Table 1. Matrix benchmarks

Matrix Name	Group	Rows	Non-zero entries
soc-sign-bitcoin-alpha	SNAP	3.7K	24M
Goodwin_030	Goodwin	10K	312K
TSOPF_FS_b300_c1	TSOPF	29K	4.4M
GaAsH6	PARSEC	61K	3M
gupta2	Gupta	62K	4.2M
mip1	Andrianov	66K	10M
soc-Slashdot0902	SNAP	5K	948K
pkustk12	Chen	94K	7.5M
torso1	Norris	116K	8.5M
kron_g500-logn17	DIMACS10	131K	10M
engine	TKK	143K	4.7M
Dubcova3	UTEP	146K	3.6M
gearbox	Rothberg	153K	9M
SiO2	PARSEC	155K	11M
Goodwin_127	Goodwin	178K	5.7M
shipsec5	DNVS	179K	4.5M
kron_g50-logn18	DIMACS10	262K	21M
web-Stanford	SNAP	281K	2.3M
amazon0505	SNAP	410K	3.3 M
kim2	Kim	456K	11M
rajat30	Rajat	643K	6.1M
web-Google	SNAP	916K	5.1M
NACA0015	DIMACS10	1M	6.2M
com-Youtube	SNAP	1.1M	2.9M
wikipedia-20051105	Gleich	1.6M	19M
wiki-Talk	SNAP	2.3M	5M
FullChip	Freescale	2.9M	26M
Freescale2	Freescale	2.9M	14M
Freescale1	Freescale	3.4M	17M

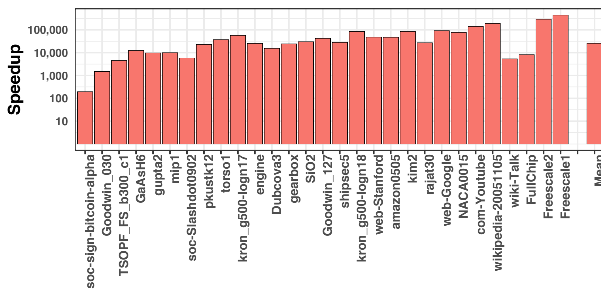

Figure 5. Speedup of CSR SpMV with closed form of $\sum$ optimisation vs a version where the for loops are generated

Without the $\sum$ precomputation from section 5.2, the LIL version has abysmal performance. Using this optimisation the performance characteristics of LIL becomes much closer to CSR. While CSR uses the array of offsets for both loop iteration and indexing, the optimised LIL version uses the array of lengths for the former and the precomputed metadata for the latter. This differences in implementation are responsible for the still measurable differences in runtime.

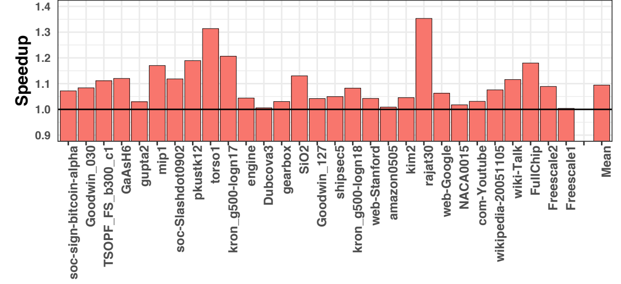

Figure 6. Effect of packing column indices in CSR matrix

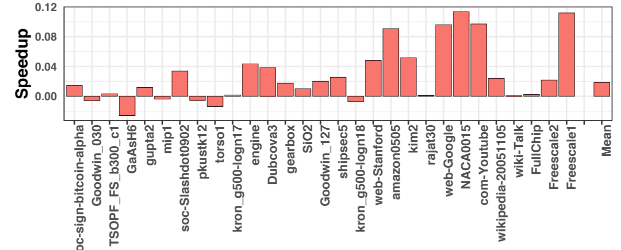

Figure 7. Speedup of CSR compared to LIL

6.4 Packing Column Indices

Section 3.4 we discussed two approaches to dealing with column indices in CSR formats: to explicitly handling of them in terms of an additional metadata array as commonly done in most low-level implementations, or to pack them together with their referent values in the element array. This latter approach may result in simpler code.

Figure 6 show the speedup of CSR-packed formats against an CSR-unpacked baseline. The packed representation leads to a speedup of up to 1.35×. By storing the index and the value next to each other in memory, we increase cache utilization. Moreover, by accessing a single array, we reduce the number of index computations generated, which in programs may impose a measurable runtime cost.

6.5 Comparison with cuSparse

Finally, we are interested in the quality of the generated kernels. We compare our SpMV implementation to the one provided by cuSparse (CUDA sparse matrix library), NVidia's proprietary library for sparse computation on GPUs (Graphics processing units). To have a fair comparison, our version will an implementation of CSR-unpacked, rather the higher performing CSR-packed, as that is the format cuSparse uses.

Listing 7 shows the code of our implementation. This version is somewhat more complex than the code shown in section 4, as it is an optimized version for targeting GPUs.The

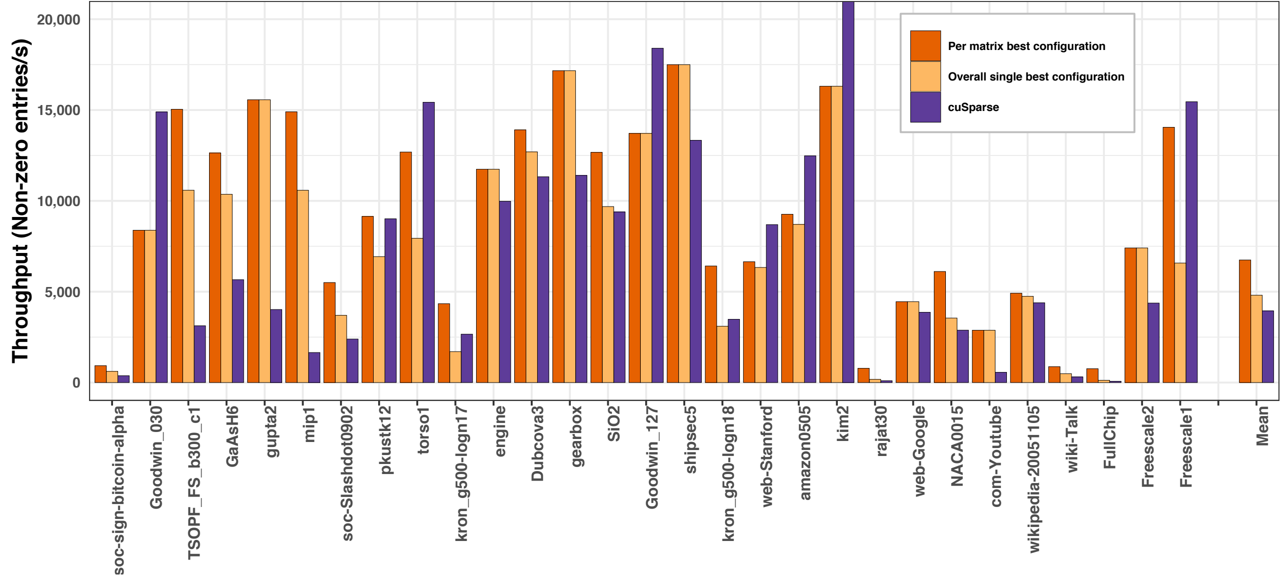

Figure 8. Comparison of the throughput of various configuration vs cuSparse, in input matrix elements read per second

```
1  nFun(n => nFun(m => fun(offsets:[int]_{n+1} =>
2  fun(col_indices:[i ↦[int]_{toNat(offsets@(i+1)-offsets(i))}]_n =>
3  fun(values:[i ↦[float]_{toNat(offsets@(i+1)-offsets(i))}]_n =>
4  fun(xs:[float; m] =>
5   zip(col_indices,values) :>> split(n) :>> mapWorkgroup(
6    split(ROWS_PER_WKGP) :>> mapWorkgroup(fun(rows =>
7      zip(rows.1, rows.2) :>>
8      split(NUM_SPLITS) :>> transpose :>>
9      mapLocal(map(fun(x=>x.2*(xs@x.1))) :>> reduce(+,0))
10   :>> reduce(x => x.1 + x.2))) :>> join)
11  :>> join
12  )))
13 )))
```

Listing 7. Functional code for parallel CSR Spmv

primitives ***mapWorkgroup*** and ***mapLocal*** are used to explicitly parallelize the computation for OpenCL, each corresponding to levels of the OpenCL parallel hierarchy [20]. ***split*** and ***join*** distribute the work among work-items, and a two-stage reduction aggregates results.

Two parameters are configurable: the *number of rows per workgroup* and the *number of work-items per row*. Each determines slight variations in the generated code and, and the computation's scheduling. Since the benchmark matrices vary significantly in size and density, no single configuration is best performing across all of them. In the results below, we will refer to two notable configurations for each benchmark:

- *Per matrix best configuration* - the configuration yielding the best speedup for a given benchmark. This configuration yields a 1.7× average speedup
- *Overall single best configuration* - the configuration with the maximum average speedup across all benchmarks. This configuration yields a 1.21× average speedup

Figure 8 shows a comparison of the throughput for each benchmark matrix of both the *specific* and *general* configurations, together with the cuSparse implementation, together with the geometric mean over all benchmarks. As we can see from the throughput and speedup results, the generated kernels are competitive with cuSparse, often surpassing them, even while using the *general* configuration.

7 Future Work

7.1 Additional Matrix Formats

In this work, we have shown how to express LIL and CSR matrices. An important benefit of our approach however, is that it is relatively easy to describe more sparse matrix formats. For example, a BSR (Block Compressed Sparse Row) matrix with block size (b, k) can be expressed as:

```
[i ↦[[[float]_k]_b]_{toNat(blk_offsets@(i+1)-blk_offsets@i)}]_{blk_n}
```

In the future, we hope to explore these and other formats, to fully take advantage of the versatility of the approach.

7.2 Automatic Generation of Sparse Implementation form Dense Source

Another possibility is the automatic generation of sparse matrix implementation from a dense matrix program. There are a number of schemes for the derivation of a sparse implementation from a high-level description, such as described in [9]. The high-level representation used throughout this paper could be a good fit for these techniques.

8 Related Work

8.1 General Purpose High Level IRs

Languages such as Futhark [11], Halide [17], Accelerate [15] and Lift [19, 20] use higher-level IRs modeling abstract transformations, and rely on type system to generate fast implementations. While these have been shown to be competitive for algorithms over dense arrays, many of them lack a systematic way to deal with sparse data structures.

8.2 Streaming Irregular Arrays

Streaming irregular arrays [7] represent a solution for expressing sparse data structure in the high-level code generator Accelerate [15]. Similarly to our approach, it use typing information to model the shape of the sparse data structure, but is more reliant on runtime support.

8.3 Domains Specific High Level IRs

Domain specific compilers such as Halide [17] for image processing applications and the tensor algebra compiler TACO [12] have specialised IRs for representing sparse data. For instance, the TACO compiler has introduced a *format abstraction* for expressing sparse data [6] in a principled away. Their domain specific nature however limits applicability: by presenting a general solution, our work may be useful in a wider number of contexts.

8.4 Dependent Types

Dependently typed languages such as Epigram [14] and Idris [4], rely on their complex type systems to express sophisticated properties. Some of them, like Agda [3], are used for theorem proving. Dependent typing adds a significant complexity to a language, and compiling such languages into efficient code is still an open area of research. Our solution takes inspiration from the techniques pioneered in this area in a controlled way that allows efficient code generation.

9 Conclusion

This paper presented a technique for the implementation of sparse matrix formats via the use of dependent types in a general purpose high-level IR. Generic IRs have focused on computation over dense arrays. Domain specific IRs have offered some solutions for expressing sparse matrix computation, but these are not reusable across application areas.

We have shown how to express sparse matrices in a generic way using the the concept of *position dependent arrays* together with *dependent typing*. We have demonstrated how to represent several commonly used sparse matrix formats using this system, and how this extension composes well with the existing algorithmic patterns model of computation.

We have detailed the compilation steps necessary to support our proposed additions, including a number of optimisations reliant on a symbolic algebra system.

We have evaluated our approach using SpMV and compared with NVidia's highly optimized cuSparse implementation. We demonstrate the performance is comparable, with an 1.7× average speedup over cuSparse. We also explore the effects of a number of optimizations proposed in this paper, such as measuring the effects of symbolic simplification and measuring the effect of index packing.

In the future, we would like to extend our exploration to more complex sparse matrix formats, and investigate the automatic derivation of sparse matrix programs from a higher-level algorithmic description.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics.

References

[1] [n.d.]. SuiteSparse Matrix Collection. https://sparse.tamu.edu. Accessed: 1st Nov 2019.

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

[3] Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

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

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*. USENIX Association, 578–594.

[6] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 123.

[7] 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.

[8] Murray Cole. 1988. *Algorithmic skeletons : a structured approach to the management of parallel computation.* Ph.D. Dissertation. University of Edinburgh, UK.

[9] Stephen Fitzpatrick, M Clint, and P Kilpatrick. 1995. *The automated derivation of sparse implementations of numerical algorithms through program transformation.* Technical Report. Tech. Rep. 1995/Apr-SF. MC. PLK, Dept. Comput. Sci., The Queen's University ....

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

[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] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. *PACMPL* 1, OOPSLA (2017), 77:1–77:29.

[13] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt.

2018. AnyDSL: a partial evaluation framework for programming high-performance libraries. *PACMPL* 2, OOPSLA (2018), 119:1–119:30.

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

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

[16] Federico Pizzuti, Michel Steuwer, and Christophe Dubach. 2019. Position-dependent arrays and their application for high performance code generation. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing, FHPNC@ICFP 2019, Berlin, Germany, August 18, 2019*. 14–26.

[17] 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, 519–530.

[18] Ivan Simecek, Daniel Langr, and Pavel Tvrdík. 2012. Space-efficient Sparse Matrix Storage Formats for Massively Parallel Systems. In *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012, Liverpool, United Kingdom, June 25-27, 2012*. 54–60.

[19] 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.

[20] 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.

[21] 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. *ACM Trans. Embedded Comput. Syst.* 13, 4s (2014), 134:1–134:25.