Semantics-Preserving Data Layout Transformations for Improved Vectorisation

Artjoms Šinkarovs
School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh
a.sinkarovs@macs.hw.ac.uk

Sven-Bodo Scholz
School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh
s.scholz@hw.ac.uk

Abstract

Data-Layouts that are favourable from an algorithmic perspective often are less suitable for vectorisation, i.e., for an effective use of modern processor's vector instructions. This paper presents work on a compiler driven approach towards automatically transforming data layouts into a form that is suitable for vectorisation. In particular, we present a program transformation for a first-order functional array programming language that systematically modifies they layouts of all data structures. At the same time, the transformation also adjusts the code that operates on these structures so that the overall computation remains unchanged. We define a correctness criterion for layout modifying program transformations and we show that our transformation abides to this criterion.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers, Optimization; D.1.1 [*Functional Programming*]; C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Single-instruction-stream, multiple-data-stream processors (SIMD)

Keywords Vectorisation, Type systems, Correctness, Program transformation

1. Introduction

Most programming languages directly relate type definitions and type declarations to one particular layout of the data in memory. For example, FORTRAN maps arrays in a column major order into memory, whereas C uses a row major scheme. The fields of records or objects are typically adjacent in memory and algebraic data types such as those in many functional languages typically are mapped to pointer connected graphs in memory.

In languages that support an explicit notion of memory and pointers there is very little that can be done about this tight coupling between types and their corresponding mapping into memory. In contrast, language that completely abstract away from the notion of memory, such as purely functional languages, allow for almost arbitrary mappings of data into memory. While this opportunity may be less relevant in the context of algebraic data types, it definitely can have a huge impact on the performance of programs that operate on arrays.

It is well known from the optimisation of compute intensive applications in FORTRAN or C, that a reorganisation of data accesses can vastly improve the overall runtime [1, 8, 12] Improvements typically do not only stem from better cache locality but also from improved chances of the utilisation of vector instructions, i.e., auto-vectorisation. However, data dependencies in programs and a fixed data-layout often constrain what can be achieved. The high-performance computing literature provides many cases where manual re-writes for enforcing different memory layouts are crucial for achieving a sufficient level of performance [7, 9].

Here, a purely functional setting offers a unique opportunity. The implicitness of the memory management allows the compiler to adjust a programmer-specified layout in a way that improves locality and that enables better vectorisation opportunities. In [13] we have demonstrated for a small functional core language how a type system can be used to ensure consistent layout adjustments, we sketched an inference algorithm and we demonstrated the potential of the approach by looking at the N-body problem and an implementation in SAC. In this paper, we present a formal layout-transformation scheme that takes a program annotated with layout-types and transforms the program on the source level so that a straight-forward mapping into a pre-defined memory mapping achieves the desired layout transformation. This approach allows layout transformations to be implemented as a high-level AST to AST translation without requiring any adjustments on the low-level code generation. Besides the type-driven code translation scheme we also provide a correctness proof of our transformation. It is based on a correctness notion that captures the essence of data-layout independence by looking at equality as being factored by index-space transformations of function arguments and function results.

The rest of the paper is organised as follows: the next section formalises the programming language we use to describe our transformation, briefly introduces the layout type system and formalises correctness criteria. Section 3 introduces the transformation itself providing informal explanation across the lines. Section 4 gives a formal proof of the correctness of our transformation according to the criteria formulated earlier. Then follows a brief example of application of the described technique using matrix multiplication as a case study. Section 6 discusses related work and we conclude with Section 7.

2. Setting

We use a first order functional core language as basis for our formalisms. The language is a stripped-down version of SAC [2] very similar to the core language introduced in [3]. In essence, the language constitutes an applied λ-calculus with a C flavour when it

FHPC '13, September 23, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2381-9/13/09…$15.00.
http://dx.doi.org/10.1145/2502323.2502332

comes to the syntax for function definitions and function applications. Furthermore, it contains the core constructs for defining arrays and data-parallel operations on them.

$$
\begin{array}{lcl}
\mathit{Program} & \Rightarrow & \textbf{letrec} \left[\mathit{FunDef} \right]^* \textbf{ in } E \\
\mathit{FunDef} & \Rightarrow & \mathit{FunId} \; (\left[\mathit{Id} \left[\, , \mathit{Id} \right]^* \right]) \; = E \\
E & \Rightarrow & \mathit{Const} \mid \mathit{Id} \mid \mathit{FunId} \; (\left[E \left[\, , E \right]^* \right]) \\
 & | & \mathit{Prf} \; (\left[E \left[\, , E \right]^* \right]) \\
 & | & \textbf{if } E \textbf{ then } E \textbf{ else } E \\
 & | & \textbf{let } \mathit{Id} = E \textbf{ in } E \\
 & | & \textbf{map } \mathit{Id} < E \; E \\
 & | & \textbf{reduce } \mathit{Id} < E \; (\; \mathit{FunId} \;) \; \; E \\
\mathit{Const} & \Rightarrow & c \mid \; [\left[E \left[\, , E \right]^* \right]] \\
\mathit{Prf} & \Rightarrow & \textbf{sel} \mid + \mid - \mid ...
\end{array}
$$

Figure 1. The syntax of SAC-λ

Fig. 1 shows the syntax of SAC-λ. The overall goal expression is preceded by a set of mutually recursive function definitions. Constants can either be scalars or vectors of expressions, each of which should evaluate to arrays of identical shape. Function applications are written in a C style for both, user defined functions and for built-in functions (*P*rf).

The built-in functions are mainly responsible for arithmetic operations and for element-selections. For convenience we use $e[iv]$ interchangeably with sel (iv, e). Central to the data-parallelism support in SAC-λ are the language constructs `map` and `reduce`, which are array versions of their well-known counterparts on lists. While `map` defines all elements of an array `reduce` defines a fold operation over an index range. Index ranges, for both these operations, are specified directly after the keywords `map` and `reduce`, respectively. They are denoted by an identifier which will range over the index space, followed by the symbol `<`, followed by the upper bound of the index range.

2.1 Semantics of SAC-λ

Every value in SAC-λ, conceptually, is a multi-dimensional array, which is being represented as a pair: $\langle s, d \rangle$, where s is a shape and d is data. Both shape and data are vectors, which are denoted in square brackets, e.g. $[1, 2, 3]$. Scalar constants are represented as arrays with empty shapes. Further down we are going to define each term formally using big-step semantics. Scalar constants are defined as:

$$\text{SCALAR}: \frac{}{n \Downarrow \langle \, [], [n] \, \rangle}$$

For non-scalar constants the following semantics holds:

$$\text{VECT}: \frac{\forall i \in \{1, \ldots, n\} : e_i \Downarrow \langle \, [s_1, \ldots, s_m], [d_1^i, \ldots, d_p^i] \, \rangle}{\begin{array}{c} v_d \equiv [d_1^1, \ldots, d_p^1, \ldots, d_1^n, \ldots, d_p^n] \\ [e_1, \ldots, e_n] \Downarrow \langle [n, s_1, \ldots, s_m], v_d \rangle \end{array}}$$

For non-scalar values, there has to be a mapping between a multi-dimensional array and its flat representation. As a default mapping we use standard row-major order. This mapping is fixed for any SAC-λ program. In order to define the semantics of selections, we first formalise the mapping of index vectors within n-dimensional index spaces into offsets within the row-major representation of n-dimensional arrays:

$\mathcal{R}m :: \langle [n], [idx_1, \ldots, idx_n] \rangle \times \langle [n], [shape_1, \ldots, shape_n] \rangle \rightarrow \mathbb{N}$. with

$$\mathcal{R}m(\langle [n], [i_1, \ldots, i_n] \rangle, \langle [n], [s_1, \ldots, s_n] \rangle) = \sum_{k=1}^{n} \left(\prod_{j=k+1}^{n} s_j \right) i_k$$

For all legal indices, i.e., $0 \leq idx_i < shape_i$ for all $i \in \{1, ..., n\}$, $\mathcal{R}m$ is a bijective function for which we can define $\mathcal{R}m^{-1}$ using integer division operations denoted with `div` and `mod`, where `div` is a quotient and `mod` is a modulo. We have

$$\mathcal{R}m^{-1}(\langle [], [a] \rangle, \langle [n], [s_1, \ldots, s_n] \rangle) = \langle [n], [a_1, \ldots, a_n] \rangle$$

where

$$\forall k \in \{1, \ldots, n\} : a_k = (a \; \texttt{mod} \prod_{i=k}^{n} s_i) \; \texttt{div} \prod_{i=k+1}^{n} s_i.$$

With these definitions we obtain as a semantics for selections:

$$\text{SEL}: \frac{\begin{array}{c} iv \Downarrow \langle \, [n], [i_1, \ldots, i_n] \, \rangle \\ e \Downarrow \langle \, [s_1, \ldots, s_n], [d_1, \ldots, d_m] \, \rangle \end{array}}{\begin{array}{c} \texttt{sel} \; (iv, \; e) \Downarrow \langle \, [], [d_l] \, \rangle \\ \text{where } \langle [], [l] \rangle = \mathcal{R}m(iv, \langle [n], [s_1, \ldots, s_n] \rangle) + 1 \end{array}}$$

Please note that we add one to the row-major order as our enumeration of array elements in tuples starts with one while our row-major mapping assumes the C convention for indices, i.e., 0 is considered the lowest legal index.

Binary scalar arithmetic functions like addition, multiplication, etc. are defined on all the numerical types for scalar values only. Further down we demonstrate semantics of plus, which can be generalised to any primitive binary function by trivial substitution.

$$+: \frac{e_1 \Downarrow \langle \, [], [d_1] \, \rangle \quad e_2 \Downarrow \langle \, [], [d_2] \, \rangle}{e_1 \; \texttt{+} \; e_2 \Downarrow \langle \, [], [d_1 + d_2] \, \rangle}$$

Any scalar primitive function f has a built-in vector variant (denoted $\vec{f}$) with the following semantics:

$$\vec{f}(\vec{x}, \vec{y}) = [f(\vec{x}[0], \vec{y}[0]), \ldots, f(\vec{x}[V-1], \vec{y}[V-1])]$$

V is an architecture-specific constant which denotes a number of elements in a SIMD vector allowing the code generation to map these operations into assembly instructions directly. For the purpose of this paper we simply assume one fixed given constant V.

We omit the semantics of LET, APP and IF as they are box-standard rules as they can be found in text books such as in [10].

The semantics for the data parallel operations MAP and REDUCE can be found at Fig 2. These two operations represent a simplified version of the with-loop construct of SAC. Both operators compute expressions over multidimensional index spaces; `map` combines the resulting expressions into an array and `reduce` folds them using a binary function. It is within the obligation of the programmer to ensure that the folding function is associative and commutative and that the expression provided as the last part of the `reduce` operation constitutes the neutral element of the folding function.

2.2 Layout inference

The basis of the code transformation presented in this paper is a type inference which identifies how each individual data item should be laid out in memory. Using a type system to encode data layouts in memory serves a dual purpose: it ensures coherence of the layouts of all data structures and it enables a local code transformation scheme. The details of the type system and a type

$$\text{MAP}: \frac{\begin{array}{c} e_u \Downarrow \langle\, [n], [u_1, \ldots, u_n]\, \rangle \\ \forall i_1 \in \{0, \ldots, u_1 - 1\} \ldots \forall i_n \in \{0, \ldots, u_n - 1\} : (\texttt{let}\ iv =\ [i_1, \ldots, i_n]\ \texttt{in}\ e_{op}) \\ \Downarrow \langle\, [s_1, \ldots, s_m], [d_1^{[i_1,\ldots,i_n]}, \ldots, d_p^{[i_1,\ldots,i_n]}]\, \rangle \end{array}}{\begin{array}{c} \texttt{map}\ iv < e_u\ e_{op} \\ \Downarrow \langle\, [u_1, \ldots, u_n, s_1, \ldots, s_m], [d_1^{[0,\ldots,0]}, \ldots, d_p^{[0,\ldots,0]}, \ldots, d_1^{[u_1-1,\ldots,u_n-1]}, \ldots, d_p^{[u_1-1,\ldots,u_n-1]}]\, \rangle \end{array}}$$

$$\text{REDUCE}: \frac{\begin{array}{c} e_u \Downarrow \langle\, [n], [u_1, \ldots, u_n]\, \rangle \qquad e_{neut} \Downarrow \langle\, [s_1, \ldots, s_m], [d_1, \ldots, d_p]\, \rangle \\ \forall i_1 \in \{0, \ldots, u_1 - 1\} \ldots \forall i_n \in \{0, \ldots, u_n - 1\} : (\texttt{let}\ iv =\ [i_1, \ldots, i_n]\ \texttt{in}\ e_{op}) \\ \Downarrow \mathfrak{D}^{[i_1,\ldots,i_n]} \equiv \langle\, [s_1, \ldots, s_m], [d_1^{[i_1,\ldots,i_n]}, \ldots, d_p^{[i_1,\ldots,i_n]}]\, \rangle \\ f(f(\ldots f(f(e_{neut}, e_{neut}), \mathfrak{D}^{[0,\ldots,0]}), \ldots), \mathfrak{D}^{[u_1-1,\ldots,u_n-1]}) \\ \Downarrow res \equiv \langle\, [s_1, \ldots, s_m], [r_1, \ldots, r_p]\, \rangle \end{array}}{\texttt{reduce}\ iv < e_u\ (f)\ e_{op} \Downarrow res}$$

Figure 2. Semantics of map and reduce operations in SAC-λ

inference scheme are presented in [13]. Here, we present only the bare essentials to the extent needed for this paper.

For n-dimensional arrays we potentially consider n different layout transformations (across each axis) denoted by layout types $1, \ldots, n$. A layout type k indicates that we intend to vectorise across axis k and therefore expect V elements of that axis to be positioned adjacently in memory. In SAC-λ, we can express this as a reshaping transformation that changes the shape of an array from $[s_1, \ldots, s_n]$ to $[s_1, \ldots, \lceil {}^{s_k}/_V \rceil, \ldots, V]$.

In addition to the layout types $1, \ldots, n$, the layout type 0 refers to an unmodified row-major layout.

Furthermore, the layout type $\triangle$ denotes a stacking of V arrays of the original shape. In this case, the shape $[s_1, \ldots, s_n]$ transforms into $[s_1, \ldots, s_n, V]$. This layout type plays a key role in the vectorisation of map/reduce operations as it denotes those program areas that will be executed in a fully vectorised form. The dual to $\triangle$ are layout types denoted by $idx(k), k \in \mathbb{Z}_+$. They identify index vectors of data parallel operations that induce a vectorisation across axis k. Note here, that the layouts of data of $idx(k)$ type are unchanged and, thus, identical to data of layout type 0.

We introduce layout types for functions as $(l_1, \ldots, l_n) \to l$ for n-argument functions where all l_i and l are layout types as described above.

The essence of the layout transformation lies in the identification of a suitable data parallel construct, i.e. a `map` or a `reduce` construct, as well as an axis k within that construct over which vectorisation is meant to happen. As soon as that has happened, the index vector that represents the iteration space of that data parallel construct will get layout type $idx(k)$ the expression in the body of that construct gets type $\triangle$ and the result of that construct gets type k. For example, consider element-wise addition of two matrices `a` and `b`:

```
map i < [n,m] a[i] + b[i]
```

we can choose a vectorisation for the overall `map` construct by giving $idx(1)$ or $idx(2)$ layout type to `i`, which would have an impact on a and b vectorisation (over first dimension in case of $idx(1)$ and over second dimension in case of $idx(2)$). We would also have to replace $+$ with $\vec{+}$, which is of layout type $(\triangle, \triangle) \to \triangle$. In general, instead of $+$ we can have an arbitrary expression, so the $idx(k)$ layout types have to be propagated inside the inner expression of `map`, as these layout types introduce constraints on selections from arrays.

In order to make sure that the layout types are sound across the overall program, we use the layout type system, introduced in [13]. In the same work we sketch an inference algorithm to assign layout types to non-vectorised programs. We are not going to repeat all the rules here, however, we are going to use the fact that a certain expression can have a limited number of valid layout-types in the proof of correctness of the transformation.

2.3 Correctness criteria

The key questions here are: When is a program transformation that manipulates the array layouts correct and how can we inductively prove that?

What is needed here is a factorisation of the semantics that considers different arrays as identical modulo their layout types, i.e., a program F with layout type k is identical to a program F' with layout type 0 if the corresponding elements in the results are identical and if their shapes match accordingly. While this appears to be a rather straight-forward criterion, an inductive proof requires a bit more formalism-armory as we have to deal with expressions of arbitrary layout types including our special types $\triangle$ and $idx(k)$.

First, we define element correspondence between layout type 0 and layout-types $k \in \mathbb{Z}_+$. We denote this mapping with I_k and define it as follows:

$$\begin{aligned} &I_k(\langle [n], [i_1, \ldots, i_n] \rangle) \\ &\quad = \langle [n+1], [i_1, \ldots, i_k \ \texttt{div}\ V, \ldots, i_n, i_k \ \texttt{mod}\ V] \rangle \end{aligned}$$

and the inverse variant of the same function:

$$\begin{aligned} &I_k^{-1}(\langle [n], [i_1, \ldots, i_n] \rangle) \\ &\quad = \langle [n-1], [i_1, \ldots, V i_k + i_n, \ldots i_{n-1}] \rangle \end{aligned}$$

Note here, that for our proofs we only require this mapping to be bijective. As a consequence, our results could be rather straight-forwardly generalised beyond the particular mappings that our layout type system considers.

With these definitions, we can now capture the trivial case. Assume an expression e with layout type k and a transformed expression $e' = \mathcal{T}(e)$. Here, correctness holds iff $e[j] = \mathcal{T}(e)[I_k(j)]$ for all legal indices j in e.

In case we are dealing with an expression of type $idx(k)$ we can be sure that we are dealing with a vector as the type system only attributes this type to index vectors. We use $e\{e_k \to^{e_k} /_V\}$ to denote that the k^{th} component of e has been divided by V. With this definition, it suffices to show that $e\{e_k \to^{e_k} /_V\} = e'$.

The final case we need to consider is the case of expressions of layout type $\triangle$. In that case, a correctness criterion requires slightly different contexts between the original expression and the transformed one. We first define two forms of concatenation and a slicing operation that allow us to expand contexts. The standard concatenation is defined as a binary operation and is denoted with a symbol $+\!\!+$. Shape-wise $+\!\!+$ concatenates two arrays extending the first component of the shape. The following semantic rule defines the operation:

$$+\!\!+ : \frac{\begin{array}{c} e_1 \Downarrow \langle\, [s_1, s_2, \dots, s_n], [d_1, \dots, d_p] \,\rangle \\ e_2 \Downarrow \langle\, [s'_1, s_2, \dots s_n], [d'_1, \dots, d'_q] \,\rangle \end{array}}{\begin{array}{c} s \equiv [s_1 + s'_1, s_2, \dots, s_n] \\ e_1 +\!\!+\, e_2 \Downarrow \langle\, \text{s}, [d_1, \dots, d_p, d'_1, \dots, d'_q] \,\rangle \end{array}}$$

The $+\!\!+$ would work as follows: $[1, 2, 3] +\!\!+ [3, 4] = [1, 2, 3, 3, 4]$.

Another operation is called array stacking, denoted with $\oplus$. This is an n-ary operation defined using the following rule:

$$\oplus : \frac{\begin{array}{c} e_1 \Downarrow \langle\, [s_1, s_2, \dots, s_m], [d^1_1, \dots, d^1_p] \,\rangle \\ \dots \\ e_n \Downarrow \langle\, [s'_1, s_2, \dots s_m], [d^n_1, \dots, d^n_p] \,\rangle \end{array}}{\begin{array}{c} s \equiv [s_1, \dots, s_m, n] \\ \oplus(e_1, \dots, e_n) \Downarrow \langle\, \text{s}, [d^1_1, \dots d^n_1 \dots, d^1_p, \dots, d^m_p] \,\rangle \end{array}}$$

The application of $\oplus$ works as follows: $\oplus([1, 2, 3], [4, 5, 6]) = [[1, 4], [2, 5], [3, 6]]$. It works similarly to the concatenation, but adds an extra dimension and does concatenation on the last shape component rather than on the first one. In the rest of the paper we are going to use the following notation: $\oplus^n_{i=0} f(i)$ which is a shortcut for $\oplus(f(0), \dots, f(n-1))$.

The slicing of an array e with last index component fixed at j is denoted as $e[*, j]$. In essence this is a reverse operation for $\oplus$ which allows to grab j-th component. Semantically it looks like this:

$$\text{SLICE} : \frac{\begin{array}{c} e \Downarrow \langle\, [s_1, \dots, s_{n-1}, s_n], [d_1, \dots, d_{ps_n}] \,\rangle \\ iv \Downarrow \langle\, [], [j] \,\rangle \wedge j < s_n \end{array}}{\begin{array}{c} s \equiv [s_1, \dots, s_{n-1}] \\ e[*, iv] \Downarrow \langle\, \text{s}, [d_{j+1}, d_{s_n+j+1}, \dots, d_{(p-1)s_n+j+1}] \,\rangle \end{array}}$$

For example, $([[1, 2, 3], [4, 5, 6]])[*, 1] = [2, 5]$.

Furthermore, we need a form of inverse transformation that allows us to identify the relevant context. We define

$$\mathcal{T}^{-1}(x, j) = \begin{cases} x & x :: 0 \\ x' | x'[I^{-1}_k(i)] = x[i] & x :: k \\ x\{x_k \to V x_k + j\} & x :: idx(k) \\ x[*, j] & x :: \triangle \end{cases}$$

The $x' | x'[I^{-1}_k(i)] = x[i]$ expression used in $\mathcal{T}^{-1}$ definition means reconstruction of the array x before the transformation was applied. In order to do so, we use I^{-1} index mapping, and as we know the shape relation of x and x' we can do the reconstruction.

With these definitions we can now define correctness for $\triangle$ types as the equivalence between the transformed expression e' and a V-fold stacking of the original expression e where all free variables a_i in e are successively being replaced by $\mathcal{T}^{-1}(\mathcal{T}(a_i), j)$.

We formalise this notion of correctness by means of a predicate $\mathcal{C}(\mathcal{T}, e)$, which has a co-domain $\{T, F\}$ (true or false). The predicate is defined separately for expressions (E in SAC-λ grammar), function definitions ($FunDef$) and the whole programs ($LetRec$).

Every expression e in a program can be considered as an application of some function f which has arguments identical to $FV(e)$ (free variables of e) and the body identical to e. We are going to use such a normal form in the subsequent definitions and proofs interchangeably with e.

Expressions of layout-type 0 are correct: $\mathcal{C}(\mathcal{T}, e) \equiv T$ iff $\mathcal{T}(e) \equiv e$.
Expressions of layout-type $\triangle$ are correct iff

$$\begin{aligned} &\oplus^V_{j=0} f(\mathcal{T}^{-1}(\mathcal{T}(a_1), j), \dots, \mathcal{T}^{-1}(\mathcal{T}(a_n), j)) \\ &\quad \equiv \mathcal{T}(f)(\mathcal{T}(a_1), \dots, \mathcal{T}(a_n)) \end{aligned}$$

Expressions of layout-type $k, k \in \mathbb{Z}_+$ are correct iff

$$\forall i < s_f : f(a_1, \dots, a_n)[i] \equiv \mathcal{T}(f)(\mathcal{T}(a_1), \dots, \mathcal{T}(a_n))[I_k(i)]$$

Expressions of layout-type $idx(k), k \in \mathbb{Z}_+$ are correct iff

$$f(a_1, \dots, a_n)\{f_k \to f_k/V\} \equiv \mathcal{T}(f)(\mathcal{T}(a_1), \dots, \mathcal{T}(a_n))$$

In the case of k layout type we use $\forall_{i<s_e}$ quantification which reads as: for all indexes starting from $[0, \dots, 0]$ and ending with the shape of expression e. We are going to use this notation through the rest of the paper.

Finally, a function definition is correct: $\mathcal{C}(\mathcal{T}, f(a_1, \dots, a_n) = e) \equiv T$ if the function body is correct: $\mathcal{C}(\mathcal{T}, e) \equiv T$. The overall program is correct if all the function-definitions are correct and the goal expression is correct. Please note that we do not need to require lack of assumptions about the correctness of the functions used in the goal expression of `letrec` as the original program guarantees lack of external references in the goal expression and function definition expressions. The `letrec` programs are closed with respect to function calls.

3. Program transformation

It is easy to see that if the layout inference succeeded then LET term can be used as is. APP term has to make sure that it picks the right function from the environment, which is a matter of layout-type signature matching. IF term has to be transformed in case its condition is of layout type $\triangle$. In that case *true* branch and *false* branch have to be evaluated and the results have to be masked using the evaluated condition. This is a standard control-flow to data-flow transformation widely discussed in [6]. One important aspect which is only relevant to our setting is potential hardware exceptions by evaluating both condition branches in isolation. For example consider the following code:

```
map i < N if a[i] == 0 then 0 else 1/a[i]
```

When the condition is vectorised, the else branch evaluation may result in a hardware exception may due to division by zero. Please note, that this sort of exception happens not because the original program were generating it but due to the way SIMD instructions work. In order to deal with this problem properly we need to rewrite the expression in the branches propagating the mask from the condition. Whenever a special operation like division of integers, or square root is found, we need to mask the arguments with a neutral element for a given operation. Implementation of such an analysis can be done straightforwardly, but for the sake of simplicity of this paper we would consider that the behaviour of transformed and non-transformed code is the same with respect to hardware exceptions and termination. In case it is not statically decidable we reject the vectorisation.

So the essential parts regarding the transformation are CONST, SEL, PRF, MAP and REDUCE.

Informally speaking, array vectorisation in terms of the given work is a cutting of an array over axis τ using tiles of size $1 \times V$ and putting V-sized chunks in such a way that the elements become adjacent in flattened representation (in our cases under $\mathcal{R}m$). Please note, that by doing so we potentially increase the number of elements in the vectorised array, as the size of τ axis might be not divisible by V. This would be a problem if we would ever make a reverse array vectorisation with further selection at newly introduced indexes. We never do that, and we never shuffle elements of the vector, so the new “phantom” elements of the array never participate in the original computations. The only thing that we should show is that index translations within the program stay sound.

Array vectorisation presented by I_τ can be done differently, which would have an impact on the further transformation. For example one could split arrays into two parts, using floor operation instead of ceiling, or permuting the vectorised axis, however, in this paper we will fix it to the one we find most practically relevant.

3.1 CONST

A constant c can get the following layout types: 0, in which case the constant stays the same as in the original program; $1, \dots, \mathcal{D}(C)$, in which case the data is being vectorised and $\triangle$, in which case the constant is being stacked V times.

Constant transformation at $\tau \in \mathbb{Z}_+$ is cutting an array at axis τ and puts V adjacent elements of the τ axes in a such a way that they are identical in the flattened row-major representation.

$$\begin{aligned}
s_c &\equiv [s_1, \dots, s_n] \\
s_{c'} &\equiv [s_1, \dots, \lceil {}^{s_\tau}/_V \rceil, \dots, s_n, V] \\
c &\equiv \langle s_c, [d_1, \dots, d_p] \rangle \\
c' &\equiv \langle s_{c'}, [d'_1, \dots, d'_{p'}] \rangle \\
\mathcal{T}(c) &= c'
\end{aligned}$$

where

$$\begin{aligned}
&\forall k \in \{1, \dots, \prod_{j=1}^{n+1} s_{c'}[j]\} : \\
&\quad cidx' \equiv \mathcal{R}m^{-1}(k-1, s_{c'}) \\
&\quad zero' \equiv [1, \dots, 1, 0] \\
&\quad d'_k = \begin{cases} d_{\mathcal{R}m(I_\tau^{-1}(cidx'), s_c)+1} & I_\tau^{-1}(cidx') < s_c \\ d_{\mathcal{R}m(I_\tau^{-1}(cidx' \cdot zero'), s_c)+1} & \text{otherwise} \end{cases}
\end{aligned}$$

when $\tau \in 1, \dots, \mathcal{D}(c)$

In this case multiplication and less than operators are used per vector component. Please note the way we treat "phantom" elements of the vectorised array – we fill the empty cells with the first element of each vector, which should be always present, as zero axes sizes are not supported. As a result, any vectorised operation on such a vector succeeds, if according scalar operation succeeds in the original program. However, if the original program terminates with hardware exception due to invalid arguments of a certain operation, this behaviour is going to be carefully preserved.

Finally we consider the case when c is of layout-type $\triangle$ which means that the array has to be stacked V times.

$$\mathcal{T}(c) = \oplus_{i=0}^{V} c$$

3.2 SEL

Selections can manifest themselves in three different ways: scalar selection, vector selection and scalar selection on a vectorised array. The scalar case is applicable when the expression we are selecting from is of layout-type 0, for example in $\texttt{sel}(iv, a), a :: 0$.

Vector selections can happen in two cases:

1. $a :: k \wedge iv :: idx(k)$, which means that this selection is invoked within a map/reduce operation, and index space is vectorised over the k-th axis. As an example consider a map over 1-d array where the inner operation uses selection:

   ```
   map i < [N] f (a[i])
   ```

   in which case if $i :: idx(1) \wedge a :: 1$, the selection has to return an expression of vector layout type rather than of scalar;

2. $a :: \triangle \wedge iv :: 0$ – this case can be found in nested maps, when selection happens on the axis which was not vectorised. For example, assume we have 2-d array a of shape $[M, N]$:

   ```
   map i < [M]
       let
           line = map j < [N] a[i ++ j]
       in
           f (line[iv])
   ```

   in this case, assuming that $i :: idx(1)$, j can be only of type 0, and the $line$ has to become a 1-d array of vectors rather than of scalars, which means that $line$ gets a layout type $\triangle$. Any further selections on $line$ implies vector selections with index of layout type 0.

Semantically, vector selection is a concatenation of V scalar selections. Keep in mind, that the length of an index vector in this cases is one element shorter than the dimensionality of the vectorised array.

$$\text{VSEL}: \frac{\begin{array}{c} iv \Downarrow \langle\, [n-1], [i_1, \dots, i_{n-1}] \,\rangle \\ e \Downarrow \langle\, \langle [n], [s_1, \dots, s_{n-1}, V] \rangle, [d_1, \dots, d_m] \,\rangle \end{array}}{\begin{array}{c} \texttt{vsel (}iv\texttt{, }e\texttt{)} \Downarrow \langle\, [\text{V}], [d'_1, \dots, d'_i] \,\rangle \\ \text{where } \forall i \in \{1, \dots, V\} : \\ \langle [], [d'_i] \rangle = \texttt{sel}(iv +\!\!+ [i-1], e) \end{array}}$$

Finally, when selections on a vectorised array are performed outside the map/reduce operation, the result of the selection has to be scalar. To illustrate that assume a data-parallel operation on the array a after which one scalar element is being accessed:

```
let
    r = map i < u f(a[i])
in
    g (r[e])
```

Assuming that $a :: k \wedge r :: k \wedge e :: 0$, we adopt index e by applying I_k index transformation.

Summarising transformations together, we get the following expression:

$$\mathcal{T}(\texttt{sel}(iv, a)) = \begin{cases} \texttt{sel}(iv, a) & iv :: 0 \wedge a :: 0 \\ \texttt{vsel}(iv, a) & iv :: idx(k) \wedge a :: k \\ \texttt{vsel}(iv, a) & iv :: 0 \wedge a :: \triangle \\ \texttt{sel}(I_k(iv), a) & iv :: 0 \wedge a :: k \end{cases}$$

3.3 Primitive functions PRF

All the primitive scalar operations have built-in vector variants, so the only thing the transformation should take care of is vectorisation of the scalar component, which is allowed by the layout-type system. Again, without lose of generality we consider plus being a representative of all primitive functions.

$$\mathcal{T}(a + b) = \begin{cases} a + b & a :: 0 \wedge b :: 0 \\ \oplus_{i=0}^{V} a \vec{+} b & a :: 0 \wedge b :: \triangle \\ a \vec{+} \oplus_{i=0}^{V} b & a :: \triangle \wedge b :: 0 \\ a \vec{+} b & a :: \triangle \wedge b :: \triangle \end{cases}$$

3.4 MAP

The only case when a map construct requires code transformation is when its index is of layout-type $idx(k)$ and its expression is of layout-type $\triangle$. In this case the upper bound will be transformed as follows:

$$\mathcal{T}(\mathtt{map}\, j < \langle[n], [u_1, \ldots, u_k, \ldots, u_n]\rangle\; e)$$
$$= \mathtt{map}\, j < \langle[n], [u_1, \ldots, \lceil u_k/V\rceil, \ldots, u_n]\rangle\; \mathcal{T}(e)$$

We apply I_k to u, but we get rid of the last vector component, which makes sure that selection at the index variable j would require an expression we are selecting from to be of type k.

In all the other cases:

$$\mathcal{T}(\mathtt{map}\, j < u\; e) = \mathtt{map}\, j < \mathcal{T}(u)\; \mathcal{T}(e)$$

3.5 REDUCE

The only case when `reduce` requires a transformation is, similarly to `map`, when the layout type of the index vector is $idx(k)$. The inner expression of `reduce` would be of layout type $\triangle$. The upper bound expression has to be transformed in the same way as we do in `map`, however, in contrast to `map`, the resulting layout type of `reduce` has to be 0. It does not happen automatically as a return layout type of a reduction function is $\triangle$, which implies an extra step of reduction. First we consider the case when the axis of vectorisation is divisible by V i.e. there are no "phantom" elements:

```
$\mathcal{T}$(reduce $i$ < $\langle[n], [u_1, \ldots, u_k, \ldots, u_n]\rangle$ ($f$) $e$) $=$
let
  $\vec{r}$ = reduce
          $i$ < $\langle[n], [u_1, \ldots, {}^{u_k}/_V, \ldots, u_n]\rangle$ ($\mathcal{T}(f)$) $\mathcal{T}(e)$
in
  reduce $i$ < $[V]$ ($f$) r[*,$i$]
```

In order to see this transformation in action, consider a summation (with +) of a 1-d array a of shape $[NV]$:

```
reduce $i$ < [$N \cdot V$] ($+$) a[$i$]
```

Assuming that a has layout type 1, the following vectorisation

```
$\vec{r} \equiv$ reduce $i$ < [$N$] ($\vec{+}$) a[$i$]
```

would deliver a result of shape $[V]$ which has to be reduced with a scalar plus:

```
reduce $i$ < [$V$] ($+$) $\vec{r}$[$i$]
```

Now let us consider the case when the size of the vectorisation axis is not divisible by V. Two things to notice here: first, we cannot perform a vector operation on the last vector (an element of vectorised array which has $\lceil s_k/V\rceil$ at its k-th position in the index), as it would contain "phantom" elements. For example, if $V \equiv 4$ then vectorised vector $[1, 2, 3, 4, 5, 6, 7, 8, 9]$ will look like $[[1, 2, 3, 4], [5, 6, 7, 8], [9, x, x, x]], x = 9$, and we cannot add the last vector, but we can still add first two. Using this observation we split the index space of vectorised reduce into two parts: $[0, \ldots, 0] \le i < [u_1, \ldots, \lfloor u_k/V\rfloor, \ldots, u_n]$ and $[0, \ldots, \lfloor u_k/V\rfloor, 0] \le i < [u_1, \ldots, \lceil u_k/V\rceil, \ldots, u_n]$. Formally we denote that as follows:

```
$\mathcal{T}$(reduce $i$ < $\langle[n], [u_1, \ldots, u_k, \ldots, u_n]\rangle$ $(f)$ $e$) $=$
let
  $\vec{r}$ = reduce
          $i$ < $\langle[n], [u_1, \ldots, \lceil({}^{u_k}/_V - \lfloor{}^{u_k}/V\rfloor)/V\rceil, \ldots, u_n]\rangle$
          $(\mathcal{T}(f))$ $\mathcal{T}(e)$;
  $\vec{r'}$ = reduce
          $i'$ < $u' \equiv \langle[n], [u_1, \ldots, \lceil u_k/V\rceil, \ldots, u_n]\rangle$
          $(\mathcal{T}(f))$
          let
            $i$ = $\langle[n], [i'_1, \ldots, i'_k + \lfloor u_k/V\rfloor, \ldots, i'_n]\rangle$
          in
            $\mathcal{T}(e)$;
in
  let
    $r_1$ = reduce $i$ < $[V]$ ($f$) $\vec{r}$[*, $i$];
    $r_2$ = reduce $i$ < $[u_k\ \ mod\ V]$ ($f$) $\vec{r'}$[*,$i$]
  in
    $f(r_1, r_2)$
```

In order to illustrate the final formula, consider a reduction with plus over an array of shape $[10, N]$ being of layout-type 1. In this case $\vec{r}$ would hold a result of vectorised addition of vectors $[0, *, *]$ and $[1, *, *]$. Then $\vec{r'}$ would sum-up vectors $[2, *, *]$, using vector plus. Please note that we can do this, as operation on "phantom" elements would not affect the result. Finally, r_1 would sum-up the elements inside the vector and r_2 would sum-up non-phantom elements of $\vec{r'}$.

Finally, in case when reduce is of layout type $\triangle$ or k the transformation is defined as:

```
$\mathcal{T}$(reduce $i$ < $u$ ($f$) $e$) $=$ reduce $i$ < $u$ ($\mathcal{T}(f)$) $\mathcal{T}(e)$
```

3.6 IF

Transformation of a conditional requires control-flow to data-flow transformation with potential stacking of a non-vectorised branch. Formally we denote it as follows:

```
$\mathcal{T}$(if $e_1$ then $e_2$ else $e_3$) =
$e'_2$ = $\oplus_{i=0}^{V} e_2\ \ if\ \ e_2 :: 0 \land e_3 :: \triangle$ or $\mathcal{T}(e_2)$ otherwise
$e'_3$ = $\oplus_{i=0}^{V} e_3\ \ if\ \ e_3 :: 0 \land e_2 :: \triangle$ or $\mathcal{T}(e_3)$ otherwise
let
  $b$ = $\mathcal{T}$($e_1$);
in
  mask ($t$, $e'_2$, $e'_3$)
```

for the cases when $e_1 :: \triangle$.

The `mask` operation chooses either from the second or from the third argument depending on the boolean value in the first argument:

```
mask ($m$, $t$, $f$)
  = $\oplus_{i=0}^{V}$ (if $m[i]$ then $t$[*,$i$] else $f$[*,i])
```

Real implementation of `mask` might differ from the code from above, as some of the architectures have either built-in masking facilities for 1-d vectors, or can be implemented efficiently using vector binary operations like bitwise and, bitwise or and bitwise not. Semantically it does exactly the same thing.

3.7 APP, LET **and** LETREC

The transformation of these three expression types is pretty much straight forward, we just propagate $\mathcal{T}$ to the arguments, the only important moment here is the way function definitions are being stored.

Vectorisation of a function definition boils down to vectorisation of its body:

```
$\mathcal{T}$(f($a_1, \ldots, a_n$) = $e$) = f($a_1, \ldots, a_n$) = $\mathcal{T}(e)$
```

However, the vectorised function does not replace the original one, but it rather creates a new instance with the same name but different layout-type signature. It means that we have a family of functions, which can be differentiated by SAC-λ using overloading mechanisms with respect to layout-types.

The application of a function boils down to matching the right function overloading according to its layout-type signature. The matching is trivial so we would assume that the compiler does it under the hood. The transformation for the application is defined as follows:

```
$\mathcal{T}$($f$($e$)) = $f$($\mathcal{T}$($e$))
```

The same reasoning is applicable to let constructs:

$\mathcal{T}(\texttt{let x = } e_1 \texttt{ in } e_2\texttt{) = let } x \texttt{ = } \mathcal{T}(e_1) \texttt{ in } \mathcal{T}(e2)$

Finally the letrec construct is a vectorisation of its goal expression and all the fundefs according to the new layout-type signatures.

$\mathcal{T}(\texttt{letrec } f_1(\ldots) = e_1, \ldots f_n(\ldots) = e_n \texttt{ in } e)$
$\quad = \texttt{letrec } \mathcal{T}(f_1(\ldots) = e_1), \ldots \mathcal{T}(f_n(\ldots) = e_n) \texttt{ in } \mathcal{T}(e)$

4. Semantical correctness

In this section we are going to construct a proof that any program in SAC-λ transformed by $\mathcal{T}$ is semantically correct in terms of $\mathcal{C}$. We are going to use structural induction over terms of the language.

First of all let us start with simple general observations.

Theorem 1. *For a given function $f(a_1, \ldots, a_n)$ and transformation $\mathcal{T}(f) :: (0, \ldots, 0) \to 0$, $\mathcal{C}(\mathcal{T}, f) \equiv T$.*

Proof. Obvious due to identity. □

The second observation concerns the $idx(k)$ expressions.

Theorem 2. *For a given expression e, where $\mathcal{T}(e) :: idx(k)$, $\mathcal{C}(\mathcal{T}, e) \equiv T$.*

Proof. By definition of $\mathcal{C}$ we assume that there is a function with arguments $FV(e)$, body e and return type $idx(k)$. We need to show that the application of $\mathcal{T}(f)$ to $\mathcal{T}(FV(e))$ results in the same value as application of f to $FV(e)$. Now, according to the layout-type system, we conclude that if a function returns a value of type $idx(k)$ then there is an argument in $iv \in \mathcal{T}(FV(e))$ of type $idx(m)$ and only two cases are possible:

1. *propagation*, $k = m$ which means that the function returns iv, in which case the correctness is obvious;
2. *concatenation* which means that iv is a part of $+\!\!+$ operation. If the operation is $iv +\!\!+ e$, then assuming that iv is correct, the k-th component of the expression would be identical to the k-th component of iv, which makes resulting expression correct. Alternatively, the expression could be $e +\!\!+ iv$, where $e :: 0$ and by construction of $+\!\!+$ the k-th component of the resulting expression is mapped to m-th component of iv. That makes resulting expression correct.

□

We deliberately restrict the range of operations on index vectors for the time being, so the proof simply runs structural induction for this types.

The base case of the structural induction is a constant, as we do not allow external parameters, and all the values are conceptually defined by means of constants.

Theorem 3. *Constant transformation is semantically correct.*

$$\frac{c \textit{ is const}}{\mathcal{C}(\mathcal{T}, c) \overset{?}{\equiv} T}$$

Proof. By the construction of CONST transformation. Let us consider three possible cases of the layout-type of a transformed constant:

$\mathcal{T}(c) :: 0$ non-vectorised case, obvious.

$\mathcal{T}(c) :: \triangle$ in which case we show that

$$\oplus_{i=0}^{V} \mathcal{T}(c)[*, i] \overset{?}{\equiv} \mathcal{T}(c)$$

which is obvious as $\mathcal{T}(c) \equiv \oplus_{i=0}^{V} c$. By the way, this is the only case for expressions of layout type $\triangle$ when the scalar elements of the transformed one can be mapped to the scalar elements of the original one.

$\mathcal{T}(c) :: k \in \mathbb{Z}_+$ in which case we show that

$$\forall_{i<s_c} : c[i] \overset{?}{\equiv} \mathcal{T}(c)[I_k(i)]$$

This follows from the construction of $\mathcal{T}(c)$, and the phantom elements are not in the range of i.

□

Theorem 4. *Vectorisation of selection is semantically correct.*

$$\frac{e \equiv a[iv] \quad \mathcal{C}(\mathcal{T}, a) \equiv T \quad \mathcal{C}(\mathcal{T}, iv) \equiv T}{\mathcal{C}(\mathcal{T}, e) \overset{?}{\equiv} T}$$

Proof. Let us consider all the legal layout types for transformed e, iv and a:

$\mathcal{T}(e) :: 0 \wedge \mathcal{T}(a) :: 0 \wedge \mathcal{T}(iv) :: 0$ non-vectorised, obvious.

$\mathcal{T}(e) :: \triangle \wedge \mathcal{T}(a) :: k \wedge \mathcal{T}(iv) :: idx(k)$ in which case we need to show that:

$$\oplus_{j=0}^{V} \texttt{sel}(\mathcal{T}(iv)\{\mathcal{T}(iv)_k \to V\mathcal{T}(iv)_k + j\}, \mathcal{T}^{-1}(\mathcal{T}(a)) \equiv \texttt{vsel}(\mathcal{T}(iv), \mathcal{T}(a))$$

From the construction of `vsel` we construct the following equality:

$$\texttt{vsel}(\mathcal{T}(iv), \mathcal{T}(a)) \equiv \oplus_{j=0}^{V} \texttt{sel}(\mathcal{T}(iv) +\!\!+ [j], \mathcal{T}(a))$$

Now, from $\mathcal{T}^{-1}$ construction for layout type k, we get the following mapping between $\mathcal{T}(a)$ and $\mathcal{T}^{-1}\mathcal{T}(a)$:

$$\forall_{i<s_{\mathcal{T}(a)}} \mathcal{T}^{-1}(\mathcal{T}(a))[I_k^{-1}(i)] \equiv \mathcal{T}(a)[i]$$

From $\mathcal{C}(\mathcal{T}, a)$ the shape of $\mathcal{T}(a)$ is $[s_1, \ldots, s_n, V]$. The shape of $\mathcal{T}(iv)$ is $[s_1, \ldots, s_n]$, which allows to rewrite the original expression to:

$$\oplus_{j=0}^{V} \texttt{sel}(\mathcal{T}(iv)\{\mathcal{T}(iv)_k \to V\mathcal{T}(iv)_k + j\}, \mathcal{T}^{-1}(\mathcal{T}(a)) \equiv \oplus_{j=0}^{V} \texttt{sel}(\mathcal{T}(iv) +\!\!+ [j], \mathcal{T}(a))$$

which is obvious due to construction of I_k^{-1}.

$\mathcal{T}(e) :: \triangle \wedge \mathcal{T}(a) :: \triangle \wedge \mathcal{T}(iv) :: 0$ in which case we need to show that

$$\oplus_{j=0}^{V} \texttt{sel}(iv, \mathcal{T}(a)[*, j]) \equiv \texttt{vsel}(iv, \mathcal{T}(a))$$

From $\mathcal{C}(\mathcal{T}, a)$ we deduce that $\mathcal{T}(a) = \oplus_{i=0}^{v} \mathcal{T}(a)[*, i]$ and from the construction of `vsel`, correctness becomes obvious.

$\mathcal{T}(e) :: 0 \wedge \mathcal{T}(a) :: k \in \mathbb{Z}_+ \wedge \mathcal{T}(iv) :: 0$ in which case we need to show that

$$\texttt{sel}(iv, a) \equiv \texttt{sel}(I_k(iv), \mathcal{T}(a))$$

which directly follows from $\mathcal{C}(\mathcal{T}, a)$.

□

Theorem 5. *Vectorisation of a primitive function is semantically correct.*

$$\frac{e \equiv x + y \quad \mathcal{C}(\mathcal{T}, x) \equiv T \quad \mathcal{C}(\mathcal{T}, y) \equiv T}{\mathcal{C}(\mathcal{T}, e) \overset{?}{\equiv} T}$$

Proof. There are four possible layout-type combinations for a given expression.

$\mathcal{T}(e) :: 0 \wedge \mathcal{T}(x) :: 0 \wedge \mathcal{T}(y) :: 0$ which is obvious.

$\mathcal{T}(e) :: \triangle \wedge \mathcal{T}(x) :: \triangle \wedge \mathcal{T}(y) :: 0$ in which case we need to show that:

$$\oplus_{j=0}^{V} (\mathcal{T}(x)[j] + (\oplus_{i=0}^{V} y)[j]) \equiv \mathcal{T}(x) \vec{+} \oplus_{i=0}^{V} y$$

Please note that $\mathcal{T}(x)$ has a shape $[V]$ so we can replace slice operation with a simple selection. Now, from $\mathcal{C}(\mathcal{T}, x)$ we deduce that $\mathcal{T}(x) = \oplus_{i=0}^{V} \mathcal{T}(x)[i]$. Finally, as

$$\begin{aligned} a \vec{+} b &\overset{def}{\equiv} [a[0] + b[0], \ldots, a[V-1] + b[V-1]] \\ &\equiv \oplus_{i=0}^{V} a[i] + b[i] \end{aligned}$$

as a and b are of shape $[V]$. This transformation makes original equality obvious.

$\mathcal{T}(e) :: \triangle \wedge \mathcal{T}(x) :: 0 \wedge \mathcal{T}(y) :: \triangle$ same reasoning as in the previous case.

$\mathcal{T}(e) :: \triangle \wedge \mathcal{T}(x) :: \triangle \wedge \mathcal{T}(y) :: \triangle$ in which case $\mathcal{C}(\mathcal{T}, a) \implies \mathcal{T}(a) = \oplus_{i=0}^{V} \mathcal{T}(a)[i]$ and $\mathcal{C}(\mathcal{T}, b) \implies \mathcal{T}(b) = \oplus_{i=0}^{V} \mathcal{T}(b)[i]$ in which case we can use the same reasoning as in previous cases.

□

Theorem 6. *Vectorisation of a map is semantically correct:*

$$\frac{e \equiv \texttt{map}\ i < u\ e_1 \quad \mathcal{C}(\mathcal{T}, e_1) \equiv T}{\mathcal{C}(\mathcal{T}, e) \overset{?}{\equiv} T}$$

Proof. As e_1 is correct there is a function f with body e_1. Let us assume, without loss of generality, that f arguments have i at the first position. In case it is not a free variable of e_1 this argument will be skipped, but we can still pass it. Let us consider legal layout-type combinations of transformed e, i and f.

$\mathcal{T}(e) :: 0 \wedge \mathcal{T}(i) :: 0 \wedge \mathcal{T}(f) :: (0) \to 0$ non-vectorised case, obvious;

$\mathcal{T}(e) :: k \wedge \mathcal{T}(i) :: idx(k) \wedge \mathcal{T}(f) :: (idx(k)) \to \triangle$ Let us assume that the shape of non-transformed e_1 is scalar. Later we will demonstrate that the reasoning works for non-scalars as well. We need to show that:

$$\begin{aligned} &\forall_{j<s_e} (\texttt{map}\ i < u\ f(i, \ldots))[j] \\ &\quad \overset{?}{\equiv} (\texttt{map}\ i' < u\{u_k \to \lceil u_k / V \rceil\}\ \mathcal{T}(f)(i', \ldots))[I_k(j)] \end{aligned}$$

From the correctness of f we have

$$\begin{aligned} &\mathcal{T}(f)(\mathcal{T}(i), \ldots) \\ &\quad \equiv \oplus_{j=1}^{V} f(\mathcal{T}(i)\{\mathcal{T}(i)_k \to V\mathcal{T}(i)_k + j\}, \ldots) \end{aligned}$$

From the construction of I_k follows that

$$I_k(j) = j\{j_k \to j_k\ \texttt{div}\ V\} +\!\!+ [j_k\ \texttt{mod}\ V]$$

As we assumed that shape of f is scalar, let us note that the index space of j under the quantifier and i in the first map are the same. The index space of $j\{j_k \to j_k\ \texttt{div}\ V\}$ and the index space of i' us the same as well. The shape of $\mathcal{T}(f)$ is $[V]$, and we can rewrite the original expression as:

$$\forall_{j<s_e} f(j, \ldots)) \overset{?}{\equiv} \mathcal{T}(f)(j\{j_k \to j_k\ \texttt{div}\ V\}, \ldots))[j_k\ \texttt{mod}\ V]$$

which is obvious due to correctness of f.

$\mathcal{T}(e) :: \triangle \wedge \mathcal{T}(i) :: 0 \wedge \mathcal{T}(f) :: (0) \to \triangle$ in which case we need to show that

$$\oplus_{j=0}^{V} \mathcal{T}(\texttt{map}\ i < u\ \mathcal{T}(e_1)[*, j]) \equiv \texttt{map}\ i < u\ \mathcal{T}(e_1)$$

As $\mathcal{T}(e_1) :: \triangle$ and $\mathcal{C}(\mathcal{T}, e_1) \implies \oplus_{j=0}^{V} \mathcal{T}(e_1)[*, j] \equiv \mathcal{T}(e_1)$ the correctness of expression is obvious due to semantics of `map`.

$\mathcal{T}(e) :: \mathcal{D}(i) + k \wedge \mathcal{T}(i) :: 0 \wedge \mathcal{T}(f) :: (0) \to k \in \mathbb{Z}_+$ In this case we need to show that

$$\begin{aligned} &\forall_{j<s_e} (\texttt{map}\ i < u\ e_1)[j] \\ &\quad \overset{?}{\equiv} (\texttt{map}\ i < u\ \mathcal{T}(e_1))[I_{\mathcal{D}(i)+k}(j)] \end{aligned}$$

From $\mathcal{C}(\mathcal{T}, e_1)$ follows that $\forall i : \forall j < s_{e_1} e_1[j] = \mathcal{T}(e_1)[I_k(j)]$ and the fact we prove reduces to this condition if for every index we drop $\mathcal{D}(i)$ first elements.

Finally we have to make sure that if e_1 is not scalar, the correctness still holds. Assuming that the shape of e_1 is s_{e_1} we would apply the same reasoning considering that all the values in our program are defined as $\langle [s_1, \ldots, s_n], [d_1, \ldots, d_p] \rangle$, where d_i is of shape s_{e_1}. That reduces non-scalar map to the scalar one, and makes the same reasoning applicable.

□

Theorem 7. *Vectorisation of reduce is semantically correct:*

$$\frac{\begin{array}{c}\mathcal{C}(\mathcal{T}, f_{bin}) \equiv T \qquad \mathcal{C}(\mathcal{T}, e_1) \equiv T \\ e \equiv \texttt{reduce}\ i\ \texttt{<}\ u\ (f_{bin})\ e_1\end{array}}{\mathcal{C}(\mathcal{T}, e) \overset{?}{\equiv} T}$$

Similarly to map, due to $\mathcal{C}(\mathcal{T}, e_1)$ there exists a function f for which without lose of generality we introduce an argument i at the first position.

First of all let's consider the transformation informally, proving the properties we will use later. First observation is that a reduce operation can be considered as the following expression:

$$r = w_{i_1} \times w_{i_2} \times \cdots \times w_{i_n}$$

where w_k is a result of application of function $f(i, \ldots)$ to $(k, \ldots)$ and $\times$ is f_{bin} written in the infix notation.

First of all, as $\times$ is associative $(a \times (b \times c) \equiv (a \times b) \times c$, the original expression can be divided into two subgroups at index k where each subgroup can be computed concurrently:

$$\begin{aligned} r_1 &= (w_{i_1} \times \cdots \times w_k) \\ r_2 &= (w_{k+1} \times \cdots \times w_{i_n}) \\ r &= r_1 \times r_2 \end{aligned}$$

The fact that $\times$ is commutative $(a \times b \equiv b \times a)$ allows us to permute elements e_k in the r expression:

$$r = w_{\psi(i_1)} \times w_{\psi(i_2)} \times \cdots \times w_{\psi(i_n)}$$

where $\psi(k)$ is a valid permutation in the range of $\{i_1, \ldots, i_n\}$.

Now let us consider what happens when vectorising r with $\vec{\times}$:

$$\begin{aligned} &[w_1' \ldots, w_{V-1}'] \vec{\times} [w_1'', \ldots, w_{V-1}''] \\ &\quad \equiv [w_1' \times w_1'', \ldots, w_{V-1}' \times w_{V-1}''] \end{aligned}$$

as follows:

$$\begin{aligned} \vec{r} &= [w_1^{(1)}, \ldots, w_{V-1}^{(1)}] \vec{\times} \cdots \vec{\times} [w_1^{(n/V)}, \ldots, w_{V-1}^{(n/V)}] \\ r &= \vec{r}[0] \times \cdots \times \vec{r}[V-1] \end{aligned}$$

We assume here that $n \equiv 0 \mod V$. The only thing we have to demonstrate for operations to be identical is that for $e_k^{(j)}$ and e_i there exists a bijective mapping between (j, k) and i.

Finally, if every vector in the vectorisation contains "phantom" elements starting from the k-th to $V-1$ position, then the operation has to be adjusted to:

$$\begin{aligned}\vec{r} &= [w_1^{(1)}, \dots, w_k^{(1)}, x, \dots, x] \\ &\quad \vec{\times} \cdots \vec{\times} [w_1^{(n/V)}, \dots, w_k^{(n/V)}, x, \dots, x] \\ r &= \vec{r}[0] \times \cdots \times \vec{r}[k]\end{aligned}$$

Proof. Let us consider a combination of all valid types for $\mathcal{T}(e)$, $\mathcal{T}(i)$ and $\mathcal{T}(f)$.

$\mathcal{T}(e) :: 0 \wedge \mathcal{T}(i) :: 0 \wedge \mathcal{T}(f) :: (0) \to 0$ non-vectorised case;

$\mathcal{T}(e) :: \tau \wedge \mathcal{T}(i) :: 0 \wedge \mathcal{T}(f) :: (\tau) \to \tau, \tau \in \mathbb{Z}_+ \cup \{\triangle\}$ In this cases, as f_{bin} is semantically correct and $f_{bin} :: (\tau, \tau) \to \tau$, this only difference that the previous case is non-scalar shape of f application. Using the same argument for assuming that the array components of e_1 and e are of shape s_{e_1} we reduce the proof to the previous case.

$\mathcal{T}(e) :: 0 \wedge \mathcal{T}(i) :: idx(k) \wedge \mathcal{T}(f) :: (idx(k)) \to \triangle$ As we know, we can permute the elements that we are reducing with f_{bin}. We need to show that the mapping between e_1 and $\mathcal{T}(e_1)$ elements is bijective for all e_1; that the number of scalar reductions is the same in transformed and original cases and that "phantom" elements do not affect the result.

First of all, the mapping between e_1 and $\mathcal{T}(e_1)$ is I_k, which is bijective by definition, and as e_1 is evaluated per every element in the index set, all the original e_1 are mapped. Secondly, "phantom" elements do not affect the result as every vector in $\vec{r'}$ contains exactly $u_k \bmod V$ elements starting from the first position. As f_{bin} is correct, the operation is happening component wise, so the resulting vector is not affected and contains $u_k \bmod V$ "real" values. Finally, the construction of r_2 guarantees that only "real" values are reduced.

Now, let us count the number of elements in a non-vectorised case, which would be $N = \prod_{j=0}^{\mathcal{D}(u)} u[j]$. The number of f_{bin} applications is $N-1$. Now, consider the $\vec{r}$ part of the transformation. Assuming that $u_k = p_k V + q_k$, where $0 < q_k < V$, the number of scalar elements reduced in $\vec{r}$ is $\frac{N}{u_k} p_k$. Each vector operation is the same as V scalar operations, so we have $\left(\frac{N}{u_k} p_k - 1\right) V$ scalar operations for this part.

For the $\vec{r'}$ part we have $\frac{N}{u_k} \cdot 1$ vector elements, where only q_k of them are non-phantom. It means that totally we have $(\frac{N}{u_k} - 1) q_k$ operations.

Finally, r_1 adds $V-1$ operations to reduce $\vec{r}$, r_2 adds $q_k - 1$ operations to reduce $\vec{r'}$, and final call of f_{bin} adds one more operation. If we sum-up all together we will get:

$$\left(\frac{N}{u_k} p_k - 1\right) V + \left(\frac{N}{u_k} - 1\right) q_k + V - 1 + q_k - 1 + 1$$

which adds up to $N-1$.

□

Another thing to notice regarding reduce is the fact that $\vec{r}$ might result in an empty range if $u_k < V$ in which case the neutral element is being returned. Also, the $\vec{r'}$ might result in empty range, if $u_k \bmod V = 0$ in which case, again, the neutral element would be used. One of $\vec{r}$ or $\vec{r'}$ is not empty as $u_k \geq 1$.

Finally we are going to consider IF, APP, LET, FUNDEF and LETREC constructions. The correctness of IF follows directly from the correctness of `mask` operation. Correctness of APP follows directly from the definition of $\mathcal{C}$ under assumption that the function itself and its arguments are correct. The same reasoning holds for LET.

For FUNDEF, as $\mathcal{T}(f(a_1, \dots, a_n) = e) = f(a_1, \dots, a_n) = \mathcal{T}(e)$, we can see that correctness trivially holds under the assumption that e is correct.

The same reasoning holds for LETREC. However, please note that, as we might have recursive functions in the body of function definitions, for which we need to do a fixed point iteration, but as the co-domain of $\mathcal{C}$ the fixed point becomes trivial, we assume that an expression is correct and continue the induction.

5. Application

Now we are going to consider a small example of matrix multiplication to demonstrate how the transformation is applied on the real code. We are going to consider a single function, assuming that it is being called in some LETREC construct.

```
matmul(a,b) =
  map i < [N]
    map j < [N]
      reduce k < [N] (+) a[i++k] * b[k++j]
```

Both arguments a and b are of shape $[N, N]$. Now let us consider that the layout type inference is done, and we have the following types for the transformed expressions:

```
matmul(a, b):: $(2,1) \to 0$ =
  map $i :: 0$ < [N]
    map $j :: 0$ < [N]
      reduce $k :: idx(1)$ < [N]
            (+) a[$(i ++ k) :: idx(2)$]:: $\triangle$
               * b[$(k ++ j) :: idx(1)$]:: $\triangle$
```

Now we can apply $\mathcal{T}$ on `matmul` which result in the following program:

```
matmul(a,b) =
  map i < [N]
    map j < [N]
      let
        rv1 = reduce
                k < [N/V] (vplus)
                vmul (vsel (i++k, a),
                      vsel (k++j, b));
        rv2 = reduce
              k < [if N mod V == 0 then 0 else 1]
              (vplus) vmul (vsel (i++[N/V], a),
                            vsel ([N/V]++j, b));
      in
        (reduce i < [V] (+) rv1[i])
        + (reduce i < [N mod V] (+) rv2[i])
```

Built-in vector operations are denoted as `vplus` and `vmul` for addition and multiplication accordingly. In the final step of the `reduce` transformation we use the fact that both `rv1` and `rv2` return 1-d arrays of shape $[V]$. That allows us to replace slicing `[*,i]` with a standard selection `[i]`.

6. Related Work

The idea to modify data layouts by means of compiler transformations is not new. There was a number of works trying to improve a cache behaviour or streaming through GPUs. Some of the works attempt to solve a very similar problem of transforming data layouts for better SIMD usage. However, despite the fact that the goals are very similar, to our knowledge none of the existing works attempted to consider layout modifications as a whole program transformation and to demonstrate why it might be correct.

Tiling optimisations like [1] changes the order of the iterations in a given loop nest. Such an optimisation technique might be not sufficient for effective SIMD code generation. We have shown this in [13] at the example of N-Body – a solution with the best spatial locality grants less vectorisation opportunities than tiling across the first axis of a 2-d array, with subsequent reorganisation of the data structure.

A number of works use a polyhedra model to optimise data layouts for a given loop nest. On the one hand, polyhedra transformations is a very powerful tool that allows to make a number of things under the hood, in which case most of the considerations regarding correctness disappear due to the known correctness of polyhedra itself. On the other hand, it stays unclear how to apply such a transformation on the whole program, where different loop nests potentially grant different layouts. The information has to be propagated outside the loop-nests and the overall program has to be adjusted. Alternatively, layout changes are possible at the beginning of the loop nest and then restoring original layouts on exit, or even employing some of just-in-time compilation techniques. In that case it is not clear how to justify performance penalty for memory copying operations.

As an example of such works we consider [8] where layout transformations are used to reduce non-local accesses for localizable computations, aiming better performance on CMPs (chip multi-processors); or [5] where the overall goal is to optimise stencil computations for SIMD capable architectures.

Peter Hawkins et al. in [4] propose to synthesize data representations for concurrent programs written using concurrent relations. The main stress of this work lies in optimising locks and synchronisations in concurrent environments, making sure that the generated code stays correct. The overall technique they propose could be used to solve the problem of finding optimal data layouts for vectorised array operations. However, the proposed setup makes it very difficult to use existing languages and codes. Our setting is more restrictive, as we do not allow tree-like or graph-like data structures, we support multi-dimensional arrays only. Because of that decisions about synchronisation or locking placement are much simpler, as data structures are uniform. At the same time it allows us to maintain a close gap between existing languages like C and Fortran and the proposed transformation, making it applicable to these languages. In order to make it happen programs have to be analysed in order to reveal data-parallel operations. Large portion of such an analysis can be done using polyhedra model.

Finally, we give a special attention to [11] where authors use data layout transformation to optimise grid applications for efficient execution on GPUs. The overall transformations they are studying are very similar to our work, however the application domain is bounded to one particular class of applications. The underlying programming language used in the study is C with custom extensions regarding propagation of the layout information. The last facts makes it hard to judge how hard would it be to generalise this technique on a wider class of applications.

7. Conclusions and future work

In this paper we have demonstrated how layout types can be used to transform data placement in memory alongside with the operations on these data structures. We have also demonstrated that the transformed program stays correct with respect to the scalar elements it computes. Finally we illustrated the proposed technique at the example of matrix multiplication.

These contributions constitute a significant stepping stone towards automating the layout transformation process. In particular our formulation of the correctness criterion plays a key role here. It enables formal reasoning about the correctness of our transformation. Besides creating the formal basis of this transformation it also is key for enabling a description of the layout transformation as a high-level program transformation. This, in turn, does not only benefit a description of the transformation in a high-level fashion, but it also can serve as a blue-print for an effective implementation.

Although we describe our technique in a functional setting, in principle, our technique can be applied in non-functional contexts as well. However, it would require quite some additional effort such as a program analysis to reveal data-parallel operations, several contextual restrictions on the legitimate statements and operations such as restrictions on pointer arithmetics, etc. Finally, the most challenging part would be to propagate the layout information across modules, not to mention potential semantical conflicts in languages where data descriptions and their layout in memory are tightly coupled.

When combining this work with our previous work on data layout inference (see [13]) a full automation of the entire process seems within reach. All that is missing is a cost model to estimate an expected performance of transformed programs. Even for simple examples the number of transformations can be quite big. For example, even for matrix multiplication two layout types are possible. And this number, at least potentially, grows with the dimensionality of the function arguments. We are confident that it will be possible to apply existing approaches towards cost models for functional languages to close this gap and, thus, to achieve a fully automated layout adjustment process.

References

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. . URL `http://doi.acm.org/10.1145/1375581.1375595`.

[2] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006. .

[3] C. Grelck, S.-B. Scholz, and A. Shafarenko. A binding scope analysis for generic programs on arrays. In *Proceedings of the 17th international conference on Implementation and Application of Functional Languages*, IFL'05, pages 212–230, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-69174-X, 978-3-540-69174-7. . URL `http://dx.doi.org/10.1007/11964681_13`.

[4] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Concurrent data representation synthesis. *SIGPLAN Not.*, 47(6):417–428, June 2012. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/2345156.2254114`.

[5] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19860-1. URL `http://dl.acm.org/citation.cfm?id=1987237.1987255`.

[6] R. Karrenberg and S. Hack. Whole Function Vectorization. In *Code Generation and Optimization*, 2011. . URL `http://www.cdl.uni-saarland.de/papers/karrenberg_wfv.pdf`.

[7] C. Kulkarni, F. Catthoor, and H. D. Man. Advanced data layout optimization for multimedia applications. In *Proc. Workshop on Parallel and Distributed Computing in Image, Video and Multimedia Processing (PDIVM'00), IPDPS'2000*, pages 186–193, 2000.

[8] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T. fook Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:348–357, 2009. ISSN 1089-795X. .

[9] L. Peng, R. Seymour, K. ichi Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong. High-order stencil computations on multicore clusters. In *IPDPS*, pages 1–11. IEEE, 2009.

[10] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1 edition, Feb. 2002. ISBN 0262162091.

[11] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 513–522, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. . URL `http://doi.acm.org/10.1145/1854273.1854336`.

[12] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9. . URL `http://dx.doi.org/10.1109/PACT.2009.18`.

[13] A. Šinkarovs and S.-B. Scholz. Data layout inference for code vectorisation. HPCS'13. Accepted for publication in HPCS'13 in 2013, to appear. `http://ashinkarov.github.io/publications/data-layouts.pdf`.