CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **28**:2092–2119
Published online 18 May 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3501

SPECIAL ISSUE PAPER

Type-driven data layouts for improved vectorisation

Artjoms Šinkarovs[*,†] and Sven-Bodo Scholz

School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS, UK

SUMMARY

Vector instructions of modern CPUs are crucially important for the performance of compute-intensive algorithms. Auto-vectorisation often fails because of an unfortunate choice of data layout by the programmer. This paper proposes a data layout inference for auto-vectorisation that identifies layout transformations that convert single instruction, multiple data-unfavourable layouts of data structures into favourable ones. We present a type system for layout transformations, and we sketch an inference algorithm for it. Finally, we present some initial performance figures for the impact of the inferred layout transformations. They show that non-intuitive layouts that are inferred through our system can have a vast performance impact on compute intensive programs. Copyright © 2015 John Wiley & Sons, Ltd.

Received 23 December 2013; Revised 13 January 2015; Accepted 10 March 2015

KEY WORDS: vectorisation; data parallelism; hpc

1. INTRODUCTION

In the last decade, vectorisation became an important research topic again, as most of the modern CPUs grant vectorisation capabilities by means of single instruction, multiple data (SIMD) extensions [1]. Classical research into auto-vectorisation focuses on the optimisation of loop nestings [2]. Data-independent operations within such loop nestings are identified; the loop-nestings as well as the order of operations within the loop nestings are reorganised to match pre-defined vectorisation patterns, typically sequences of identical arithmetic operations within loops. For vectorisation to be effective, such subsequent operations need to work on data that are adjacent in memory. Otherwise, loading/storing overheads in most cases outweigh any possible performance gains from using SIMD operations. Furthermore, vectorisation only yields a substantial benefit if it can be applied within loop nestings, preferably within the innermost loops. As a consequence, classical auto-vectorisation fails to deliver substantial performance improvements whenever loop nestings cannot be re-arranged to match the layout of the data structures that are being computed on.

In this paper, we propose a novel approach towards program vectorisation: rather than focusing on a reorganisation of loop nestings, we suggest a reorganisation of data layouts to enable vectorisations. Key to this approach is a program analysis for inferring suitable data layouts. Based on this inference, a subsequent program transformation followed by classical auto-vectorisation achieve the overall goal.

Data layout inference comes with several challenges: most importantly, we have to make sure that any new layout can be accommodated by means of a semantics preserving program transformation. Languages such as C give guarantees on how data are being stored in memory that we need to

*Correspondence to: Artjoms Šinkarovs, School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS, UK.
†E-mail: A.Sinkarovs@macs.hw.ac.uk

observe. Interfaces of modules need to be preserved, as well as potential effects that result from the sharing of data or code.

Another core challenge lies in the identification of suitable data layouts. The theoretically very large number of possible layouts for any given array needs to be narrowed down to a manageable size. At the same time, we need to ensure that layouts that enable very good vectorised performance stay in this set. Further challenges arise from the differences in the executing hardware. They impact directly the way code can or should be vectorised; consequently, they also impact the choice of data layouts.

Putting all the mentioned challenges together, we get a rather complex vectorisation framework. We decompose it into smaller interrelated parts and study them separately. In [3] we have identified the necessity to have a way to express vectorisation explicitly in a portable way. In this paper, we focus on the challenge of identifying suitable layout combinations that enable auto-vectorisation. In [4], we demonstrate transformations that have to follow after the inference is performed and prove that semantics of transformed programs is being preserved. We study first experimental results at the end of this paper; however, extensive experimental evaluation is future work.

We propose a type inference that identifies data layouts suitable for vectorisation. A functional core language serves as the basis for our formalisation. It constitutes a strip-down version of the programming language SAC [5], which we use as a vehicle for our experiments and implement the inference. Essential features of a functional language that our system depends on are the following: pure functions, n-dimensional arrays as first class citizens, data parallel loop nests expressed using an explicit syntactical construction and memory management being fully implicit to allow adjustments of data layouts.

The layout inference identifies ideal layouts (with respect to SIMD vectorisation) for each individual loop nesting, then employs representational changes whenever necessary and transforming the overall program. We provide a solution to the separate compilation problem by utilising the overloading capabilities of SAC. This enables code adaptations at the calling site without making representational changes inevitable.

We use the N-body problem as a case study throughout the paper. It nicely demonstrates the difficulties when attempting the classical approach to vectorisation, and it also shows the effectiveness of our proposed approach. Finally, we present some initial performance measurements that show substantial speedups even in the presence of multi-threaded executions.

The paper is structured as follows: In the next section, we introduce our core language, a stripped-down version of SAC. Section 3 introduces our running example in that language, and it discusses the key ideas of our approach at it. Section 4 provides a formal presentation of our type system, and Section 5 describes an algorithm for inferring the layout types and applies it to one of the functions of the N-body example before Section 6 shows the effect of these transformations on the execution times of our running examples. Related work is presented in Section 7 before Section 8 concludes.

2. CORE PROGRAMMING LANGUAGE

The inference described in this paper is based on a functional language called SAC-λ, which is a stripped down version of SAC, as described in [6]. The main reason we are not using the full version is to make our presentation and reasoning simpler; however, the implementation of the inference uses full-fledged SAC.

The choice of a functional language as input language might require some clarification, as most of the existing high-performance codes are written in imperative languages like C or Fortran, which at the first glance, makes our work inapplicable. Using an imperative language as input comes with a large number of challenges, all of which are orthogonal to the core problem we are trying to solve. One would have to provide a rather complex program analysis to identify iteration-independent loop nests, prove that inner functions are pure (otherwise we will not be able to transform them) and deal with duality of arrays and pointers in C. Furthermore, data layouts of arrays in C are fixed to row-major representation; in case of separate compilation, newly inferred data layouts will have to be propagated outside the modules. Finally, the lack of type safety and the presence of explicit

$$\begin{array}{lcl}
\textit{Program} & \Rightarrow & \textbf{letrec}\ \big[\,\textit{FunDef}\,\big]^{*}\ \textbf{in}\ E \\[1em]
\textit{FunDef} & \Rightarrow & \textit{FunId}\ (\ \big[\,\textit{Id}\big[\ ,\textit{Id}\,\big]^{*}\ \big]\ )\ =E \\[1em]
E & \Rightarrow & \textit{Const}\mid \textit{Id}\mid \textit{FunId}\ (\ \big[\,E\ \big[\ ,E\,\big]^{*}\ \big]\ ) \\
 & \mid & \textit{Prf}\ (\ \big[\,E\ \big[\ ,E\,\big]^{*}\ \big]\ ) \\
 & \mid & \textbf{if}\ E\ \textbf{then}\ E\ \textbf{else}\ E \\
 & \mid & \textbf{let}\ \textit{Id}=E\ \textbf{in}\ E \\
 & \mid & \textbf{map}\ \textit{Id}<E\ E \\
 & \mid & \textbf{reduce}\ \textit{Id}<E\ (\ \textit{FunId}\ )\ E \\[1em]
\textit{Prf} & \Rightarrow & +\mid -\mid \textbf{sel}\mid \ldots
\end{array}$$

Figure 1. The syntax of SAC-λ.

flow-control and low-level operations potentially limit the applicability of our approach. A lot of research effort is being spent to overcome many of those hardships, but we still do not have a unified solution. However, most of the modern C/Fortran compilers in their optimisation cycle perform a number of passes like transformation to single-assignment form, building a control flow graph, and so on, which transform the original program into the form that is very similar to the formalism we introduce. If we add polyhedra analysis [7] to identify loop-independent iterations and link-time optimisations (to get hold of full program), our technique should be applicable in the imperative setting as well. SAC-λ can be seen as internal representation of imperative programs.

The language contains only the bare essentials of the SAC language adjusted to a λ-calculus style in order to facilitate a more concise description of our techniques. Figure 1 shows the syntax of our core language.

As in full-fledged SAC, our stripped down version consists of a set of potentially, mutually recursive function definitions and a dedicated goal expression (main function). Expressions are either constants, variables or function applications. Anonymous functions, that is, lambda abstractions, are not supported. Function applications are written in C style, that is, arguments are in a comma-separated list of expressions wrapped in parentheses. Local variable definitions are expressed as let-constructs, and conditionals exist in the form of if-then-else expressions. Primitive operations contain set of standard arithmetic operations, for example, `+`, `-`, `sel`, and so on, comparisons and mathematical functions like *sqrt*, *sin*, and so on. Additionally, our core language contains two combinators, *map* and *reduce*. They serve as vehicles for expressing data parallel operations.

All constants in the language are n-dimensional arrays. Scalars are represented by numbers, and higher-dimensional arrays are represented by nested lists of numbers in square brackets.

All SAC programs compute n-dimensional arrays as results. We denote the n-dimensional array by a pair $\langle[s_1,\ldots,s_n],[d_1,\ldots,d_m]\rangle$, where $[s_1,\ldots,s_n]$ denotes a shape of the array, that is, its extent with respect to n individual axes and the vector $[d_1,\ldots,d_m]$ containing all elements of the array in a linearised row-major format. Please note that we use the term *data layout* to denote a mapping of the indexes of the array to its linearised form. If $A[i_1,\ldots,i_n]=d_k$, then the data layout is a function relating $[i_1,\ldots,i_n]$ and k.

We use a standard big-step operational semantics as defined in detail in [6]. The only two constructs that require special attention here are the operators *map* and *reduce* as well as the primitive operations for which vectorised versions exist. Jointly, these language constructs play key roles in our inference.

2.1. Map/reduce

These two operators constitute simplified versions of the with-loop-constructs in full-fledged SAC[‡]. They are array versions of the well-known combinators. Both operators compute expressions over

[‡]For details on with-loops in SAC see [5].

MAP

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

REDUCE

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

Figure 2. The semantics of MAP and REDUCE operations.

an N-dimensional index space and then either combine the results in an array (map variant) or fold them using a binary operator.

A formalisation of the semantics of `map` and `reduce` based on the deduction system from [6] is presented in Figure 2. The map-rule shows that the upper limit e_u has to reduce to an n-element vector that determines the outermost n dimensions of the overall result. For each index vector within the n-dimensional index range, the expression e_{op} needs to evaluate to an m-dimensional result of one fixed shape. These m-dimensional results are then composed to the overall result by concatenating their element values.

Similar to the map-rule, the reduce rule computes identically shaped values e_{op} for all indices within the index space defined by the upper limit vector e_u. However, here, the result is obtained by consecutive folding using the binary function f. As one can see, the semantics definition prescribes left-to-right folding with respect to a row-major unrolling in the index space. Despite this definition, we demand f to be associative and commutative in order to enable arbitrary folding orders. For example:

$$\mathrm{map}\, i < [2,3]\; 3 \xrightarrow{\text{evaluates to}} [[3,3,3],[3,3,3]] \qquad \mathrm{reduce}\, i < [2,3]\; (+) 3 \xrightarrow{\text{evaluates to}} 18.$$

The iteration space always starts with zero vector and ends with the upper bound of the operator. For the aforementioned example, the iterations space would be

$$\{[0,0],[0,1],[0,2],[1,0],[1,1],[1,2]\}.$$

Please note, that an iteration space forms an array of the shape identical to the upper bound ($[2,3]$ in our example). In both cases, the goal expression is being evaluated at every iteration:

$$\{[0,0] \rightarrow 3, [0,1] \rightarrow 3, [0,2] \rightarrow 3, [1,0] \rightarrow 3, [1,1] \rightarrow 3, [1,2] \rightarrow 3\}.$$

The order of the evaluation is non-deterministic. Finally, the map operator produces an array $[[3,3,3],[3,3,3]]$ as a result. The reduce operator takes an extra step to combine evaluated expressions using the binary function:

$$[0,0] \rightarrow 3 + [0,1] \rightarrow 3 + [0,2] \rightarrow 3 + [1,0] \rightarrow 3 + [1,1] \rightarrow 3 + [1,2] \rightarrow 3.$$

Evaluation order is again non-deterministic. A binary function of the reduce must be associative and commutative, and the shape of the reduced result would be identical to the shape of the evaluated goal expression.

Note here that the indexing variable can be referred to within the expression

$$\text{map}\, i < [3]\, i \xrightarrow{\text{evaluates to}} [0, 1, 2].$$

2.2. *Vector operations*

We assume that all SIMD operations have a built-in fixed vector length $[V]$, where V is a given target architecture-specific constant. For the context of this paper, we consider all primitive functions to have SIMD counterparts with the following semantics:

$$\vec{f}([a_1, \ldots, a_V]) = [f(a_1), \ldots, f(a_V)].$$

The only exception is the selection operation `sel (iv, a)` which selects the element at the index position `iv` of the array `a`. It also has a vectorised counterpart, but its semantics is not as straight-forward. For example, related work is presented in Section to [4].

3. RUNNING EXAMPLE

The main motivating example that we are going to use throughout the paper is an implementation of the N-body problem. The problem is defined as an iterative approximation of the movement of N planets. During each step, accelerations, velocities and positions of all the planets are being recomputed. Acceleration of the i-th planet is computed from the relative positions of all the other planets. Then the velocity and in turn, the position of the i-th planet is updated using the newly computed acceleration. For more details, please refer to [8], which discusses the N-body problem in more detail. We provide a core implementation of the benchmark using SAC-λ in Figure 3. It has been slightly adjusted from the version discussed in [8] to be better suited for demonstrating our inference technique. We use the symbol `#` for inserting comments and the symbol `;` as a shortcut for nested *let* expressions.

The function `advance` is updating arrays of positions and velocities on each time step. To do so, it first computes the mutual accelerations between all planets. This computation is achieved by mapping the function `planet_acc` over all planets. `planet_acc` computes the summed up forces that all planets have on the given position `pos`. This reduction in turn makes use of the function `acceleration`, which computes the acceleration between two planets `posx` and `posy`.

It is remarkable that the N-body implementation grants a number of vectorisation opportunities, most of which are not valid under classical auto-vectorisers. The main reason is the way acceleration is computed between two planets. Acceleration, velocities and positions have shape $[N, 3]$. The most compute-intensive operation happens at the inner dimension of the position array. Theoretically, that would be an ideal scenario for vectorisation; however, the problem is that the size of this dimension is too small. For most of the architectures, the length of float vector would be four, which means that loading/storing within a given layout would require masking, and has an implication on the alignment of load/store that might introduce noticeable overheads on some targets. As a consequence, at that point, classical auto-vectorisers would typically give up.

As a programmer, one might predict such a behaviour and extend the innermost dimension to match the vector length. That might bring some performance gains, but the danger is that the increased memory footprint of the array will slow down the overall performance, an aspect that is very easy to overlook. More importantly, an alternative solution is typically being missed out.

Rather than considering a vectorisation over the triplets, one might consider a vectorisation of the array of accelerations over the components of triplets. In that case, memory overhead would be substantially lower, and the number of elements processed per vector operation would be higher. The drawback is that such a transformed data layout has an impact on the whole program. It might be (i) arbitrary difficult to rewrite large program, and (ii) the transformation might not be beneficial.

```
letrec
    # (float[3], float[3]) -> float[3]
    vplus (x,y) = map i < [3]  (x[i] + y[i])

    # (float[3]) -> float
    l2norm (x) =
        sqrt ((reduce i < [3]  (+) x[i]*x[i]) + 0.01)

    # (float[3], float[3], float) -> float[3]
    acceleration (posx, posy, mass) =
        let diff = map i < [3] (posx[i] - posy[i]);
            norm = l2norm (diff);
            norm3 = norm * norm * norm
        in
            map i < [3]  (diff[i] * mass / norm3)

    # (float[3], float[N, 3], float[N]) -> float[3]
    planet_acc (pos, positions, masses) =
        reduce i < [N]  (vplus)
               let
                   p = map j < [3] positions[i ++ j];
               in
                   acceleration (pos, p, masses[i])

    # (float[N,3], float[N,3], float[N], float)
    # -> (float[N,3], float[N,3])
    advance (positions, velocities, masses, dt) =
        let
            dt_vec = [dt, dt, dt];
            acc = map i < [N]
                  let
                      p = map j < [3] positions[i ++ j]
                  in
                      planet_acc (p,positions,masses);
            vel = map iv < [N,3]
                  velocities[iv] + acc[iv] * dt_vec;
            pos = map iv < [N,3]
                  positions[iv] + vel[iv] * dt_vec
        in
            (pos, vel)
in
    ...
```

Figure 3. Implementation of the N-body.

3.1. *The key ideas in a nutshell*

We generalise the idea of vectorisation across non-innermost dimensions as follows: For any given array A with shape $[s_1, \ldots, s_n]$, we consider vectorisations in all possible axes $1, \ldots, n$. A vectorisation in axis k will lead to a layout remapping into an array A^k of shape $[s_1, \ldots, s_{k-1}, s_k/V, s_{k+1}, \ldots, s_n, V]$, where the V elements that in A are adjacent in axis k are now adjacent in the innermost axis $n+1$ of A^k.

The key problem then lies in the necessity to find out which layouts can result in vectorisation. Vectorisation potential in general stems from applications of primitive operations like $+$ to elements of an array within the context of an independent loop. In such a setting, any of the array's axes can be chosen for vectorisation whose corresponding index is traversed by an independent loop. However, in practice, the choices in most cases are more limited due to other selections that are present within such a nesting of independent loops. If selections into more than one array exist, we get correlations between the layout choices of the arrays involved; examples for this situation can be observed in the function `vplus` of our N-body application, where we receive correlations between the arrays `x` and `y`. If several elements within the same array are selected, axes with two or more different accesses require more involved code transformations and are therefore less favourable. Furthermore, we have to take into account that the same array can be used in several loop nests, all of which may provide vectorisation opportunities; an example for this situation is the use of the array `positions` within the body of the function `advance` in our running example. Finally, the nests of independent loops may not exist within a single function body. Instead, we can have situations where layout demands for vectorisations may need to be propagated through function calls; an example for this situation is the function `acceleration` of the N-body application which exposes the layout demands on its first two arguments to the calling context in `planet_acc`.

The main contribution of this paper is a type system to describe the layout inference. It allows us to control all the aforementioned aspects: We can propagate layouts through function calls, and we can control tightly what happens to all loop indices involved. Furthermore, it takes into account multiple uses of the same data structure within an entire application and ensures consistent layout transformations throughout.

4. A TYPE SYSTEM FOR DATA-LAYOUTS

Before we describe the layout types, let us clarify the term *type* and put into perspective of the language. We use layout types to denote transformations of expressions. Additionally to that, we have standard element types like: *int*, *float*, and so on; we have shapes as a part of the types that form a subtyping hierarchy and participate in function overloading. For more details, please refer to [5]. That means that every expression of the language is annotated with a type that consists of three orthogonal components: element type, shape and layout type. In further discussions we are going to consider only layout types assuming that the element types and shapes have been inferred and are sound.

As explained informally in the previous section, for a given n-dimensional array, we consider n different layout transformations. We denote these by the natural numbers $i \in \{1, \dots, n\}$, where i refers to the layout transformation when the shape of an array changes from $[s_1, \dots, s_n]$ into $[s_1, \dots, s_i/V, \dots, s_n, V]$; V neighbour elements at axis i are placed at the newly introduced $n+1$-th axis. In addition, we use 0 to denote the shape identity and we use $\triangle$ to denote a shape extension from $[s_1, \dots, s_n]$ into $[s_1, \dots, s_n, V]$. The transformed values are either replicated in case of constants or a represent V-fold selection. The latter is needed for the actual vectorisation of expressions that happen inside our map or reduce constructs. Finally, we add a layout type for index vectors. They play a crucial role in the layout inference as they introduce constraints between layouts of different arrays whenever they are used for selections from more than one array. Index vectors can have types $idx(m)$ and $m \in \mathbb{Z}_+$ that denote that the m-th component of the index vector is considered for vectorisation. Now, we can define the set or layout types as

$$L = \mathbb{N} \cup \{\triangle\} \cup idx(m), m \in \mathbb{Z}_+ \quad ^{\S}.$$

We also introduce layout-type-signatures to denote the different possible layout transformations an individual function can be applied to. Formally, an n-argument function is described by a $(\tau_1, \dots, \tau_n) \to \tau_{n+1}$ type, where all τ_i are layout types as defined earlier. We denote the union of L with all layout-type-signatures over L by LT.

Here is a simple example to develop an intuition for the layout types and corresponding transformations. Let us consider a function that adds two matrices presented in Figure 4.

Both input matrices as well as the result are two dimensional arrays of shape $[N, N]$ with the element type *float*. Our goal is to find a valid layout transformation for the arrays in the program that would lead to replacement of scalar primitive operations with vector ones. In this work, we restrict

§Note here that despite of the infinite nature of the definition of L, for any given program L is finite as the natural numbers are bound by the maximum number of array axes present.

```
# (float[N,N], float[N,N]) -> float[N,N]
matplus (a, b) =
    map i < [N, N]
        a[i] + b[i]
```

Figure 4. Addition of two $N \times N$ matrices.

layout transformations of the arrays and only consider tiling with tiles of size $1 \times V$ across one of the array's axes and moving those tiles into newly created dimension. For our example, we have the following cases (assuming that $N = 4$ and $V = 2$):

$$M^0 = \begin{pmatrix} [01,02,03,04] \\ [05,06,07,08] \\ [09,10,11,12] \\ [13,14,15,16] \end{pmatrix} \quad M^1 = \begin{pmatrix} [[01,05],[02,06],[03,07],[04,08]] \\ [[09,13],[10,14],[11,15],[12,16]] \end{pmatrix} \quad M^2 = \begin{pmatrix} [[01,02],[03,04]] \\ [[05,06],[07,08]] \\ [[09,10],[11,12]] \\ [[13,14],[15,16]] \end{pmatrix}.$$

Here, M^i denotes a transformed array M with respect to the layout type i. We expect from our inference to identify two vectorisation possibilities for *matplus*: when both of the arguments are of layout type 1, and both of the arguments are of layout type 2. Formally, we would expect to see the following typings:

```
# Corresponds to M1
matplus (a, b):: $(1,1) \to 1$ =
  map i:: $idx(1)$ < [N, N]
    a[i]:: $\triangle$ + b[i]:: $\triangle$
```

```
# Corresponds to M2
matplus (a, b):: $(2,2) \to 2$ =
  map $i$ :: $idx(2)$ < [N, N]
    a[i]:: $\triangle$ + b[i]:: $\triangle$
```

We use here $e :: \tau$ notation to denote that e is of layout type τ to avoid confusion with standard typing for which we are using $e : float[N]$ notation. These typings will correspond to the following transformations:

```
# (float[N/V,N,V], float[N/V,N,V])
# -> float[N/V,N,V]
matplus (a1, b1) =
  map i < [N/V, N]
    vplus (vsel (i, a1),
           vsel (i, b1))
```

```
# (float[N,N/V,V], float[N,N/V,V])
# -> float[N,N/V,V]
matplus (a2, b2) =
  map i < [N, N/V]
    vplus (vsel (i, a2),
           vsel (i, b2)),
```

where `vplus` and `vsel` correspond to vector variants of + and `sel`.

4.1. Environments

The purpose of the layout type system is to infer which combinations of layout choices for all data structures will enable some code vectorisation. To describe n, such combinations within a single environment, we formalise environments as mappings from identifiers to n-element vectors of types, that is, we have environments $\mathcal{E} \subset Id \times LT^n$. We denote the lookup of a variable v in $\mathcal{E}$ by $\mathcal{E}(v)$, and $\mathcal{E} \oplus (v, \langle \tau_1, \ldots, \tau_n \rangle)$ denotes an environment that returns the vector type $\langle \tau_1, \ldots, \tau_n \rangle$ for the variable v.

Unless specified otherwise, we are going to use small Greek letters to denote vectors of layout types and indexes to get individual components. For example, $\tau \equiv \langle \tau_1, \ldots, \tau_n \rangle, \tau_i \in LT$. We use $|\tau|$ to denote the number of components of a vector type τ, that is, $|\langle \tau_1, \ldots, \tau_n \rangle| = n$.

For a more succinct presentation of the type system, we use separate environments for all functions. We denote the collection of all these environments by a 'function environment' $\mathcal{F} \subset Id \times \mathcal{E}$. Look-up of a function identifier f and presence of function identifiers are denoted in the same way as its done for standard environments, that is, we use $\mathcal{F}(f)$ and $\mathcal{F} \oplus \mathcal{E}$, respectively.

In order to access the resulting type of the function f, we are going to introduce a meta-variable f in the relevant environment. We can look-up a function type using $\mathcal{F}(f)(f)$ notation. As we require all entries of one variable environment to have the same length, an environment $\mathcal{E}$ of a function f takes the general form:

$$\mathcal{E} = \left\{ v_1 : \left\langle \tau_1^1, \ldots, \tau_n^1 \right\rangle, \ldots v_m : \left\langle \tau_1^m, \ldots, \tau_n^m \right\rangle, f : \left\langle \tau_1^f, \ldots, \tau_n^f \right\rangle \right\}.$$

It can be seen as a matrix of size $(m + 1) \times n$, where m is the number of local variables and arguments in the function this environment captures; $+1$ comes from the meta variable denoting the type of the function, and n is the number of valid layout combinations for that function.

Each column in the matrix represents a layout combination for all variables and arguments of the function including its signature. We refer to the i-th column of an environment $\mathcal{E}$ by $\mathcal{E}^{[i]}$.

4.2. *Type rules*

With these definitions at hand, we can define a deduction system in order to characterise the validity of layout-transformations. The judgements of this deduction system are of the form $\mathcal{F}, \mathcal{E} \vdash expr : \langle \tau_1, \ldots, \tau_m \rangle$ where

$\mathcal{F}$	is a function environment; it contains separate environments for all functions;
$\mathcal{E}$	is an environment containing valid layout transformations for the identifiers in the current context;
$expr$	is an expression;
m	is the number of valid layout transformations for the function under consideration; and
τ_i	are the m layout-transformations that $expr$ within the current function can undergo.

The type rules for the non-array-specific core of the language can be seen on Figure 5.

Please note that, in the rest of the paper, we use $\mathcal{D}(e)$ to denote a number of axes in e and the length of the vector $\mathcal{S}_0(e)$ to denote the length of the first dimension.

$$\frac{e \Downarrow \langle [s_1, \ldots, s_n], [d_1, \ldots, d_p] \rangle}{\mathcal{D}(e) = n} \qquad \frac{e \Downarrow \langle [n], [d_1, \ldots, d_p] \rangle}{\mathcal{S}(e) = n}.$$

Most of these rules are vectorised versions of the standard rules for typing a first order applied λ-calculus: they only differ from their standard counterparts by dealing with vectors of n types for

$$\textsc{Const}\quad \frac{\forall 1 \leqslant i \leqslant n \quad \tau_i \in \{0, \ldots, \mathcal{D}(c)\} \cup \{\triangle\}}{\mathcal{F}, \mathcal{E} \vdash c : \langle \tau_1, \ldots, \tau_n \rangle} \qquad \textsc{Var}\quad \frac{}{\mathcal{F}, \mathcal{E} \vdash x : \mathcal{E}(x)}$$

$$\textsc{App}\quad \frac{\begin{array}{c} \forall 1 \leqslant i \leqslant t \quad \mathcal{F}, \mathcal{E} \vdash e_i : \langle \tau_1^i, \ldots, \tau_n^i \rangle \\ \mathcal{F}, \mathcal{F}(f) \vdash f : \langle (\sigma_1^1, \ldots, \sigma_1^t) \to \sigma_1, \ldots, (\sigma_m^1, \ldots, \sigma_m^t) \to \sigma_m \rangle \\ \forall 1 \leqslant i \leqslant n \quad \exists 1 \leqslant j \leqslant m \quad \forall 1 \leqslant k \leqslant t \quad \sigma_j^k = \tau_i^k \land \sigma_j = \tau_i \end{array}}{\mathcal{F}, \mathcal{E} \vdash f(e_1, \ldots, e_t) : \langle \tau_1, \ldots, \tau_n \rangle}$$

$$\textsc{Let}\quad \frac{\mathcal{F}, \mathcal{E} \vdash e_1 : \tau \qquad \mathcal{F}, \mathcal{E} \oplus (x, \tau) \vdash e_2 : \sigma}{\mathcal{F}, \mathcal{E} \vdash \text{let } x = e_1 \text{ in } e_2 : \sigma}$$

$$\textsc{Letrec}\quad \frac{\begin{array}{c} \mathcal{F}' = \mathcal{F} \bigoplus_{i=1}^{n} \left(f_i, \mathcal{F}(f_i) \oplus \left(f_i, \langle (\tau_1^1, \ldots, \tau_{A_i}^1) \to \sigma_1, \ldots, (\tau_1^{V_i}, \ldots, \tau_{A_i}^{V_i}) \to \sigma_{V_i} \rangle \right) \right) \\ \forall 1 \leqslant i \leqslant n \quad \mathcal{F}', \mathcal{F}'(f_i) \bigoplus_{j=1}^{A_i} \left(a_j^i, \langle \tau_j^1, \ldots, \tau_j^{V_i} \rangle \right) \vdash e_i : \langle \sigma_1, \ldots, \sigma_{V_i} \rangle \\ \mathcal{F}', \mathcal{E} \vdash e : \rho \end{array}}{\mathcal{F}, \mathcal{E} \vdash \text{ letrec } f_1(a_1^1, \ldots, a_{A_1}^1) = e_1, \ldots, f_n(a_1^n, \ldots, a_{A_n}^n) = e_n \text{ in } e : \rho}$$

Figure 5. Non array-specific layout rules.

each identifier rather than a single type. Two rules are of special interest here: the CONST rule for typing constants and the APP rule for typing function applications.

The CONST rule allows us to attribute any layout transformation type as long as we stay within the dimensionality of the constant, or we choose to extend the shape. As a consequence of this liberty, any possible type inference will have to imply type constraints from the context in order to constrain the types for constants.

The APP rule correlates *n* potential layout combinations within the calling context with *m* potential layout combinations of the called context. This ensures, that only those layout combinations are present for which suitable function layout transformations exist that effectively ensures consistency throughout the entire program.

The rules that give rise to layout transformations are those for primitive operations and those for the map and reduce constructs are shown in Figure 6.

As explained informally in the previous section, we look for patterns where a primitive operation for which a vector counterpart exists (PRF[△] rule) is applied to element selections (SEL[△] rule) into arrays that are located within a data parallel context (MAP[△] rule or RED[△] rule). Like for example in Figure 4, we have a vector version of plus that is applied on selections to the arrays inside the map. Depending on the nesting of map constructs, the MAP[△] rule propagates type relations in a different way. We have to distinguish four different cases:

(1) The map construct may control a layout transformation, that is, it may be responsible for the data-parallel loop that is due to be vectorised. In this case, the corresponding axis k is attributed as type $idx(k)$ for the index variable and the expression *e* needs to be of expansion type (△).

PRF[△]

$$\frac{\begin{array}{c}\mathcal{F},\mathcal{E}\vdash e_1:\langle\tau_1,\ldots,\tau_n\rangle\\ \mathcal{F},\mathcal{E}\vdash e_2:\langle\sigma_1,\ldots,\sigma_n\rangle\\ \underset{1\leqslant i\leqslant n}{\rho_i}=\begin{cases}0 & \tau_i=0\wedge\sigma_i=0\\ \triangle & \tau_i=0\wedge\sigma_i=\triangle\\ \triangle & \tau_i=\triangle\wedge\sigma_i=0\\ \triangle & \tau_i=\triangle\wedge\sigma_i=\triangle\end{cases}\end{array}}{\mathcal{F},\mathcal{E}\vdash +(e_1,e_2):\langle\rho_1,\ldots,\rho_n\rangle}$$

MAP[△]

$$\frac{\begin{array}{c}\mathcal{F},\mathcal{E}\oplus(j,\langle\tau_1,\ldots,\tau_n\rangle)\vdash e:\langle\sigma_1,\ldots,\sigma_n\rangle\\ \mathcal{F},\mathcal{E}\vdash u:\langle\phi_1,\ldots,\phi_n\rangle\wedge\forall 1\leqslant i\leqslant n\quad\phi_i=0\\ \underset{1\leqslant i\leqslant n}{\rho_i}=\begin{cases}k & \tau_i=idx(k)\wedge\sigma_i=\triangle\\ \triangle & \tau_i=0\wedge\sigma_i=\triangle\\ 0 & \tau_i=0\wedge\sigma_i=0\\ \mathcal{S}_0(j)+k & \tau_i=0\wedge\sigma_i=k\in\mathbb{Z}_+\end{cases}\end{array}}{\mathcal{F},\mathcal{E}\vdash \text{map } j<u\ e:\langle\rho_1,\ldots,\rho_n\rangle}$$

RED[△]

$$\frac{\begin{array}{c}\mathcal{F},\mathcal{F}(f)\vdash f:\langle(\upsilon_1,\upsilon_1)\rightarrow\upsilon_1,\ldots,(\upsilon_m,\upsilon_m)\rightarrow\upsilon_m\rangle\\ \mathcal{F},\mathcal{E}\vdash u:\langle\phi_1,\ldots,\phi_n\rangle\wedge\forall 1\leqslant i\leqslant n\quad\phi_i=0\\ \mathcal{F},\mathcal{E}\oplus(j,\langle\tau_1,\ldots,\tau_n\rangle)\vdash e:\langle\sigma_1,\ldots,\sigma_n\rangle\\ \forall 1\leqslant i\leqslant n\quad\exists 1\leqslant j\leqslant m\quad\sigma_i=\upsilon_j\\ \underset{1\leqslant i\leqslant n}{\rho_i}=\begin{cases}0 & \tau_i=idx(k)\wedge\sigma_i=\triangle\wedge\phi_i=0\\ \triangle & \tau_i=0\wedge\sigma_i=\triangle\wedge\phi_i=0\\ \sigma_i & \tau_i=0\wedge\sigma_i\in\mathbb{N}\wedge\phi_i=0\end{cases}\end{array}}{\mathcal{F},\mathcal{E}\vdash \text{reduce } j<u\ f\ e:\langle\rho_1,\ldots,\rho_n\rangle}$$

IF[△]

$$\frac{\begin{array}{c}\mathcal{F},\mathcal{E}\vdash p:\langle\tau_1,\ldots,\tau_n\rangle\\ \mathcal{F},\mathcal{E}\vdash t:\langle\sigma_1,\ldots,\sigma_n\rangle\\ \mathcal{F},\mathcal{E}\vdash f:\langle\phi_1,\ldots,\phi_n\rangle\\ \underset{1\leqslant i\leqslant n}{\rho_i}=\begin{cases}\phi_i & \tau_i=0\wedge\sigma_i=\phi_i\\ \triangle & \tau_i=\triangle\wedge\sigma_i=0\wedge\phi_i=\triangle\\ \triangle & \tau_i=\triangle\wedge\sigma_i=\triangle\wedge\phi_i=0\\ \triangle & \tau_i=\triangle\wedge\sigma_i=\triangle\wedge\phi_i=\triangle\end{cases}\end{array}}{\mathcal{F},\mathcal{E}\vdash \text{if }(p)\text{ then } t \text{ else } f:\langle\rho_1,\ldots,\rho_n\rangle}$$

IDX[△]

$$\frac{\begin{array}{c}\mathcal{F},\mathcal{E}\vdash j:\langle\tau_1,\ldots,\tau_n\rangle\\ \mathcal{F},\mathcal{E}\vdash h:\langle\sigma_1,\ldots,\sigma_n\rangle\\ \underset{1\leqslant i\leqslant n}{\rho_i}=\begin{cases}idx(k) & \tau_i=idx(k)\wedge\sigma_i=0\\ idx(\mathcal{S}_0(j)+k) & \tau_i=0\wedge\sigma_i=idx(k)\\ 0 & \tau_i=0\wedge\sigma_i=0\end{cases}\end{array}}{\mathcal{F},\mathcal{E}\vdash j +\!\!\!+\, h:\langle\rho_1,\ldots,\rho_n\rangle}$$

SEL[△]

$$\frac{\begin{array}{c}\mathcal{F},\mathcal{E}\vdash j:\langle\tau_1,\ldots,\tau_n\rangle\\ \mathcal{F},\mathcal{E}\vdash a:\langle\sigma_1,\ldots,\sigma_n\rangle\\ \underset{1\leqslant i\leqslant n}{\rho_i}=\begin{cases}\triangle & \tau_i=idx(k)\wedge\sigma_i=k\wedge k\in\mathbb{Z}_+\\ \triangle & \tau_i=0\wedge\sigma_i=\triangle\\ 0 & \tau_i=0\wedge\sigma_i\in\mathbb{N}\end{cases}\end{array}}{\mathcal{F},\mathcal{E}\vdash \text{sel }(j,a):\langle\rho_1,\ldots,\rho_n\rangle}$$

Figure 6. Layout rules enabling layout transformations.

(2) The map-construct can be syntactically located between the controlling map construct and the expression that is to be vectorised. In this case, the type for the index j has to be of type 0, and the expression is of expansion type.
(3) If we do not have a vectorisation at all, the types for the index, expression and the entire map construct are all 0.
(4) Finally, we can have a situation, where the map construct surrounds a map construct that controls a vectorisation. In that case, the expression is of some type k already, and the result type of the map-construct has to reflect that we have a layout transformation on an inner dimensionality. This is done by adding the number of axes of the surrounding map to k.

The PRF[$\triangle$] rule captures all possible vectorisation cases: vectorisation is possible (indicated by the expansion type $\triangle$), whenever at least one argument has expansion type. Finally, the only rule that gives rise to such an expansion type is the SEL[$\triangle$] rule for array selections. Similar to the MAP[$\triangle$] rule, the SEL[$\triangle$] rule has to deal with potential nestings of array selections:

(1) The case that gives rise to vectorisation is the case where the index has type $idx(k)$ and the array to select from has a matching layout transformation k.
(2) If a selection is applied to an array that has given rise to vectorisation already (it is of type $\triangle$) but the selection is still located inside the controlling map construct, the index needs to be of type 0 and the expansion type is propagated on.
(3) Finally, the selection can be located outside of a controlling map construct, in which case the array is of type k and the result type as well as the index type are both of type 0.

The IDX[$\triangle$] rule allows for nested map/reduce constructs to be typable. The main use case for that is a function application on non-scalar selections from an array.

5. LAYOUT INFERENCE

The layout inference can be directly deduced from the layout rules similarly to monomorphic type systems. However, the main challenge comes from the facts as follows:

(1) The length of vector types in the environment is not known at the time we start the inference;
(2) CONST defines the valid layout-types for components of the vector type, but we do not know which variant we should use for a certain component;
(3) Recursive functions require a fixed point iteration.

The overall intuition behind the algorithm is that we start with adding all the valid type combinations and we cross out those combinations that have proven to be untypable at every step of the inference. Every time we see a constant, we expand existing environment by assuming that every column in the environment is compatible with any valid layout type for the given constant. Finally, as for recursive functions, we introduce a $\perp$ type when the type of the function is not yet known; we use a fixed-point iteration to eliminate $\perp$ types.

The algorithm can be seen as a top-down traversal over the program, where for every term, the layout rule corresponding to the type of the term is applied. We start the inference with an empty function $\mathcal{F}$, which is extended whenever a function is being processed.

5.1. The inference algorithm

We formulate the inference using $\mathcal{T}_{\text{inf}}$ schemata. The algorithm is a top-down traversal, and the type of the overall program can be inferred by applying $\mathcal{T}_{\text{inf}}$ to the `letrec` expression. Further, in this section, we define $\mathcal{T}_{\text{inf}}$ application for all the expression kinds according to Figure 1 and explain the details. That allows us to infer types for all the programs in our language.

Formally, $\mathcal{T}_{\text{inf}}$ application has the following form:

$$\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, e),$$

where $\mathcal{F}$ is a function environment, and $\mathcal{E}$ is an environment related to a `term` we are inferring a type for. The inference step evaluates to a triplet:

$$(\mathcal{F}', \mathcal{E}', \tau) = \mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, e),$$

with potentially modified function environment $\mathcal{F}'$, potentially modified environment $\mathcal{E}'$ and a type (in the sense of environments, i.e. a tuple of layout types) τ. The meaning of this application is the following:

$$\frac{(\mathcal{F}', \mathcal{E}', \tau) = \mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, \text{term})}{\mathcal{F}', \mathcal{E}' \vdash \text{term} : \tau}.$$

Before we formalise the algorithm, we have to introduce a couple of new meta-operators and meta-types. First of all, we introduce a bottom type, denoted with $\perp$. This type is used only in terms of the inference algorithm to drive a fixed point iteration and does not represent any valid layout. Intuitively, it denotes the lowest type in a subtyping hierarchy: $\forall \tau \in L \quad \perp <: \tau$, if we have had sub-typing. The second meta-type is called *nil* and is being denoted with $\square$. Semantically, it means that, in a given environment at the position where $\square$ appears, an expression is proven to be untypable. The main purpose of $\square$ is to mark a column in the environment that has to be deleted.

The variable typing rule VAR says that the type of a variable is just a lookup in the environment. However, the variables have to get there somehow. If we look at the inference rules, an environment is being extended every time we use the $\oplus$ operator. For let expressions that directly translates into the algorithm step, however, for function arguments, map and reduce index variables, and constants, the type has to be guessed. To solve this, we are going to consider all the valid typings for and eliminate those that are not sound during the inference. We introduce the T meta-operator to generate all the valid typings for an object x:

$$T(x) = \begin{cases} \langle 0, 1, \ldots, \mathcal{D}(x), \triangle \rangle & x \text{ is constant} \\ \langle 0, 1, \ldots, \mathcal{D}(x), idx(1), \ldots, idx(\mathcal{S}_0(x)), \triangle \rangle & x \text{ is a variable or argument.} \end{cases}$$

At every step, we are going to reconstruct the environment by either expanding it with a new layout or shrinking it in case a certain layout combination is proven to be untypable. In order to express this process formally, we introduce three more meta-operators on types: n-times type replication $R(\tau, n)$ and n-times type component replication $C(\tau, n)$, and a helper meta-operator for tuple concatenation $+\!\!+$. We defined those as follows:

$$\begin{aligned} \tau +\!\!+ \sigma &= \langle \tau_1, \ldots, \tau_{|\tau|}, \sigma_1, \ldots, \sigma_{|\sigma|} \rangle \\ \tau +\!\!+ \langle\rangle &= \langle\rangle +\!\!+ \tau = \tau \\ C\left(\langle \tau_1, \ldots, \tau_{|\tau|} \rangle, n\right) &= +\!\!+_{i=1}^{|\tau|} \left(+\!\!+_{j=1}^{n} \langle \tau_i \rangle\right) \\ R\left(\langle \tau_1, \ldots, \tau_{|\tau|} \rangle, n\right) &= +\!\!+_{i=1}^{n} \tau. \end{aligned}$$

For example, $R(\langle 1, 2, 3 \rangle, 2) = \langle 1, 2, 3, 1, 2, 3 \rangle$ and $C(\langle 1, 2, 3 \rangle, 2) = \langle 1, 1, 2, 2, 3, 3 \rangle$.

Now, we define two operations on environments: environment extension $\oplus$ and environment shrinking $\ominus$. Please note that, here, $\oplus$ has a different semantics than in the inference rules. However, they are related in the following sense: before we can add a new variable, which relates to the inference rules $\oplus$, an environment has to be extended using the $\oplus$ we are defining as

$$\mathcal{E} \oplus \tau = \{(v, R(\sigma, |\tau|)) \quad | \quad (v, \sigma) \in \mathcal{E}\}.$$

For example, assuming that $\mathcal{E} = \{(v, \langle 0, 1, \triangle \rangle)\}$, then $\mathcal{E} \oplus \langle 2, 3 \rangle = \{(v, \langle 0, 1, \triangle, 0, 1, \triangle \rangle)\}$. Please note that the number of components of every type in the environment is an invariant denoted with $l(\mathcal{E})$.

Environment shrinking is defined using a helper rm meta-operator as follows:

$$\begin{aligned} \mathcal{E} \ominus \tau = \mathcal{E}' \quad & \texttt{where} \\ \mathcal{E}' = \{(v, rm(\sigma, \tau) \quad | \quad (v, \sigma) \in \mathcal{E}\} \quad & \texttt{where} \\ rm(\sigma, \tau) = +\!\!+_{i=1}^{|\tau|} \text{ if } \tau_i \neq \square \text{ then } \langle \sigma_i \rangle \text{ else } \langle\rangle & \end{aligned}$$

For example, assuming that $\mathcal{E} = \{(v_1, \langle 1, 0\rangle), (v_2, \langle \triangle, 1\rangle)\}$, we delete the columns of $\mathcal{E}$ at the position where we have $\square$ in the right-hand side operator. $\mathcal{E} \ominus \langle \square, 1\rangle = \{(v_1, \langle 0\rangle), (v_2, \langle 1\rangle)\}$. This is being used to remove columns of the environment that are proven to be untypable.

With these definitions at hand, we can now formally describe individual cases of the $\mathcal{T}_{\text{inf}}$ application.

5.1.1. Letrec expression. Letrec inference consists of two steps—inferring types for the functions and inferring type for the goal expression. However, as functions might call each other in their bodies and functions can be recursive, we populate $\mathcal{F}^0$ with function types that return $\perp$ for any possible layout combination that arguments can take. That ensures that a look-up in the function environment $\mathcal{F}^0$ always is successful. After that, we infer types for all the functions again, considering their goal expressions. Formally, we say that the inference of `letrec` is an inference of its goal expression assuming that functional environment $\mathcal{F}^n$ contains function types.

$$\mathcal{T}_{\text{inf}}\left(\{\}, \{\}, \text{letrec } f_1\left(a^1_1, \ldots, a^1_{m_1}\right) = e_1, \ldots, f_n\left(a^n_1, \ldots, a^n_{m_n}\right) = e_n \text{ in } e\right) = \mathcal{T}_{\text{inf}}\left(\mathcal{F}^n, \{\}, e\right).$$

To construct the environment $\mathcal{F}^n$, we start with creating environments $\mathcal{E}_i$ for every function f_i. The environment $\mathcal{E}_i$ contains a Cartesian product of all the possible argument types plus the type of the function. Components of this function type will have a form $(\tau_1, \ldots, \tau_n) \rightarrow \perp$ for every possible layout combination of all the arguments. A Cartesian product of the argument types is being created by consequently adding every argument x to $\mathcal{E}_i$ using the following procedure: expand all the existing types of $\mathcal{E}_i$ by applying $\mathcal{E}_i \oplus T(x)$ and add the entry $x : C(T(x), l(\mathcal{E}))$. This can be formalised as

$$\begin{aligned}
\mathcal{E}^1_i &= \left\{\left(a^i_1, T\left(a^i_1\right)\right)\right\} \\
\mathcal{E}^2_i &= \mathcal{E}^1_i \oplus T\left(a^i_2\right) \cup \left\{\left(a^i_2, C\left(T\left(a^1_2\right), l\left(\mathcal{E}^1_i\right)\right)\right)\right\} \\
&\cdots \\
\mathcal{E}^{m_i}_i &= \mathcal{E}^{m_i-1}_i \oplus T\left(a^i_{m_i}\right) \cup \left\{\left(a^i_{m_i}, C\left(T\left(a^i_{m_i}\right), l\left(\mathcal{E}^{m_i-1}_i\right)\right)\right)\right\} \\
\forall^{l\left(\mathcal{E}^{m_i}_i\right)}_{j=1}\ \rho_j &= \left(\mathcal{E}^{m_i}_i(a_1)_j, \ldots, \mathcal{E}^{m_i}_i(a_n)_j\right) \rightarrow \perp \\
\mathcal{E}^{m_i+1}_i &= \mathcal{E}^{m_i}_i \cup \{(f_i, \rho)\}.
\end{aligned}$$

Please note that the ρ_j is a function type for the layout combination of the arguments in the j-th column of $\mathcal{E}^{m_i}_i$. The notation $\mathcal{E}^{m_i}_i(a_k)_j$ denotes j-th component of the type that the argument a_k has in the environment $\mathcal{E}^{m_i}_i$. For example, for the function

```
# (float[N], int) -> int
f (a, b) = ...,
```

we expect the following environment:

$$\begin{array}{rcccccc}
a: & 0 & 1 & \triangle & 0 & 1 & \triangle \\
b: & 0 & 0 & 0 & \triangle & \triangle & \triangle \\
f: & (0,0) \rightarrow \perp & (1,0) \rightarrow \perp & (\triangle,0) \rightarrow \perp & (0,\triangle) \rightarrow \perp & (1,\triangle) \rightarrow \perp & (\triangle,\triangle) \rightarrow \perp.
\end{array}$$

We can construct a functional environment $\mathcal{F}^0$ that captures all the functions returning $\perp$ for any valid layout combination of the arguments as follows:

$$\mathcal{F}^0 = \bigcup_{i=1}^{n} \left\{\left(f_i, \mathcal{E}^{m_i+1}_i\right)\right\}.$$

Now, using $\mathcal{F}^0$ we can precise function types by inferring the types of the goal expressions. Formally, we denote it as follows:

$$(\mathcal{F}^1, \_, \_) = \mathcal{T}_{\text{inf}}(\mathcal{F}^0, \mathcal{F}^0(f_1), f_1(a^1_1, \dots, a^1_{m_1}) = e_1)$$
$$\dots$$
$$(\mathcal{F}^n, \_, \_) = \mathcal{T}_{\text{inf}}(\mathcal{F}^{n-1}, \mathcal{F}^{n-1}(f_n), f_n(a^n_1, \dots, a^n_{m_n}) = e_n).$$

Fixed point iterator. We use fixed point iterator as we can have recursive functions. The main principle is based on the fact that we introduce a bottom type if function application hits yet unfinished function. The bottom types can be absorbed by the condition, which gives a raise to the fixed point. The step of a fixed point is an application of $\mathcal{T}_{\text{inf}}$ to the letrec expression.

The fixed point iteration stops when none of the types got changed. The fixed point terminates because the size of any environment is bound by the product of dimensionalities of the arguments, constants and map/reduce index variables. That is because environments can grow only on application of $\oplus$ operation, which happens in case of constants, arguments and map/reduce index variables. The let case does not count as environment expansion happens via the expression we substitute with. It means that, even if an environment would grow after elimination of a bottom type, it would not grow bigger than the bound. And as bottom types never replace non-bottom types, we can either precise a type or leave it unchanged.

5.1.2. Function definition. When we start the inference of a function definition, the functional environment will contain entries for all the functions from the letrec. Environments of individual functions will at least contain the arguments and the function type. The only thing that we need to do at this point is to infer the type for the body of a function and replace the functional type. Formally, we denote it as follows:

$$\begin{aligned}
\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{F}(f), f(a_1, \dots, a_n) = e) &= (\mathcal{F}'', \mathcal{E}', \mathcal{E}'(f)) \quad \texttt{where} \\
(\mathcal{F}', \mathcal{E}, \sigma) &= \mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{F}(f), e) \\
\forall_{i=1}^{l(\mathcal{E})} \rho_i &= (\mathcal{E}(a_1)_i, \dots, \mathcal{E}(a_n)_i) \to \sigma_i \\
\mathcal{E}' &= \{(v, \tau) \quad | \quad (v, \tau) \in \mathcal{E} \wedge v \neq f\} \cup \{(f, \rho)\} \\
\mathcal{F}'' &= \{(g, \mathcal{E}_g) \quad | \quad (g, \mathcal{E}_g) \in \mathcal{F}' \wedge g \neq f\} \cup \{(f, \mathcal{E}')\}.
\end{aligned}$$

In the last two steps, we reconstruct environment $\mathcal{E}$ by removing potentially imprecise type for f and adding a newly inferred one and we have updated functional environment $\mathcal{F}$ replacing environment of f with $\mathcal{E}'$.

5.1.3. Variables and constants. As the VAR rule suggests, the type of a variable can be obtained by looking-up the environment:

$$\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, v) = (\mathcal{F}, \mathcal{E}, \mathcal{E}(v)).$$

As we said earlier, for constants, we need to guess the type, as they might have a number of typings; we cannot say which of them are sound. We extend the environment with $T(c)$:

$$\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, c) = (\mathcal{F}, \mathcal{E} \oplus T(c), C(T(c), l(\mathcal{E}))).$$

Assuming that we have an environment $\mathcal{E}$,

$$a : 1\ 2,$$

and we apply $\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, 42)$ we expect an environment to become

$$a : 1\ 2\ 1\ 2$$

as $T(42) = \langle 0, \triangle \rangle$.

5.1.4. Function application. Function applications require a bit more work. First, we acquire layout types for all the arguments; after that, we generate valid types for the results; finally, we shrink the environment and add the resulting type.

The problem we potentially have is that environment changes on every $\mathcal{T}_{\text{inf}}$ reduction step. It means that, if we have an application of f to $e_1, \ldots, e_n$ and we infer a type for e_1 with $(\mathcal{F}, \mathcal{E}^1, \tau) = \mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, e_1)$, it might be invalidated in environment $\mathcal{E}^1$. For example, we infer the type for the second argument: $(\mathcal{F}, \mathcal{E}^2, \sigma) = \mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}^1, e_2)$, then the type of e_1 in $\mathcal{E}^2$ might differ from τ. In other words, for a function application, we need to find types of $e_1, \ldots, e_n$ in environment $\mathcal{E}^n$. The easiest way to achieve that would be to keep expressions in the environment; in which case, it would be automatically updated on every reconstruction. For technical reasons, we would assume that all the argument expressions of a function application are variables. That would allow us to add the type of the argument expressions in the environment making sure that it is being updated properly, as adding e_2 has a potential effect on the type of e_1. The transformation itself is very straight-forward: instead of $f(e_1, \ldots, e_n)$, we consider expression let $v_1 = e_1$ in let $v_2 = e_2 \ldots$ in $f(v_1, \ldots, v_n)$. We start with acquiring types for the arguments.

$$\begin{gathered}\mathcal{T}_{\text{inf}}\left(\mathcal{F}, \mathcal{E}, f(v_1, \ldots, v_n)\right) = \left(\mathcal{F}, \mathcal{E}^2, rm(\psi', \psi')\right) \\ \forall_{i=1}^{l(\mathcal{E}^n)} \phi^i = \mathcal{E}(v_i).\end{gathered}$$

Every argument v_i has type ϕ^i in environment $\mathcal{E}$. The next step would be to obtain a type of function f. We do that by enquiring functional environment $\mathcal{F}$ and then the environment that is bound to f.

$$\left\langle \left(\tau_1^1, \ldots, \tau_n^1\right) \to \sigma_1, \ldots, \left(\tau_1^m, \ldots, \tau_n^m\right) \to \sigma_m \right\rangle = \mathcal{F}(f)(f).$$

After that, we have to match $\phi_k^1, \ldots, \phi_k^n$ argument types with the function arguments. Please note that function type might be not unique, that is, for a chosen $\phi_k^1, \ldots, \phi_k^n$, we might get several valid return types. To deal with this fact, the result of the match would be a $l(\mathcal{E})$-element tuple of tuples, where every inner tuple consists of valid return types.

Because of fixed point, some of the arguments might have $\perp$ as a part of its types. Obviously, they would not match any function type, as the arguments are always consist of non-$\perp$ types. However, it does not mean that we have to consider this layout combination untypable. In order to deal with this situation, we say that if any of $\phi_k^1, \ldots, \phi_k^n$ is $\perp$, then the result of the application is $\perp$. Here is how we express it:

$$\forall_{i=1}^{l(\mathcal{E})} \psi_i = \begin{cases} \langle \sigma_1', \ldots, \sigma_x' \rangle & \forall_{j=1}^{x} \sigma_j' = \sigma_k \implies \exists k \leqslant m \left(\forall_{w=1}^{n} \phi_w^i = \tau_w^k \wedge \phi_w^i \neq \perp\right) \\ \langle \perp \rangle & \exists k \leqslant n \left(\phi_k^i = \perp\right) \\ \langle \square \rangle & \text{otherwise.} \end{cases}$$

In case $|\psi_i| > 1$, we need to replicate the i-th column of $\mathcal{E}$ $|\psi_i|$ times. Then we flatten ψ by concatenating its components; finally, we remove environment columns where ψ has $\square$, and we add updated ψ type for the function application.

$$\begin{aligned}&\mathcal{E}^1 = \{(e, R(\tau_i, |\psi_i|)) \mid (e, \tau) \in \mathcal{E}\} \\ &\psi' = +\!\!+_{i=1}^{m} \psi_i \\ &\mathcal{E}^2 = \mathcal{E}^1 \ominus \psi'.\end{aligned}$$

ψ' is a flattened ψ, and $rm(\psi', \psi')$ is a ψ' type with $\square$ components being removed.

5.1.5. Let expression. The let expression is processed by inferring a type for the expression we are substituting with, adding the variable of this type to the environment and inferring a type for the goal expression within the new environment.

$$\begin{aligned}
\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, \text{let}\, x = e_1 \,\text{in}\, e_2) &= \left(\mathcal{F}^2, \mathcal{E}^3, \tau''\right) \quad \texttt{where} \\
(\mathcal{F}^1, \mathcal{E}^1, \tau) &= \mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, e_1) \\
\mathcal{E}^2 &= \mathcal{E}^1 \cup \{(x, \tau)\} \\
(\mathcal{F}^2, \mathcal{E}^3, \tau') &= \mathcal{T}_{\text{inf}}(\mathcal{F}^1, \mathcal{E}^2, e_2).
\end{aligned}$$

5.1.6. Primitive functions. Primitive functions can be handled via constructing a function type for the primitive function and adding it into the function environment:

$$\begin{aligned}
\tau_+ &= \langle (0,0) \to 0, (0, \triangle) \to \triangle, (\triangle, 0) \to \triangle, (\triangle, \triangle) \to \triangle \rangle \\
\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, a + b) &= \mathcal{T}_{\text{inf}}(\mathcal{F} \cup \{(+, \{(+, \tau_+)\})\}, \mathcal{E}, +(a, b)).
\end{aligned}$$

Constraints in the other rules use polymorphic types, so we cannot reconstruct them easily in the form of function types; however, we can still reuse it at ψ type generation. Let us consider a MAP[$\triangle$] rule:

5.1.7. Map and reduce expressions. Similarly to the application, we abstract the upper bound expression u into a variable v_u; we expand the environment with all the valid types for the index variable j and infer a type for the goal expression e.

$$\begin{aligned}
\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, \text{map}\, j < v_u\, e) &= \left(\mathcal{F}, \mathcal{E}^3, rm(\psi, \psi)\right) \quad \texttt{where} \\
\tau &= \langle 0, idx(1), \ldots, idx(\mathcal{S}_0(j)) \rangle \\
\mathcal{E}^1 &= \mathcal{E} \oplus \tau \cup \{(j, C(\tau, l(\mathcal{E}))\} \\
(\mathcal{F}, \mathcal{E}^2, \sigma) &= \mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}^1, e) \\
\upsilon &= \mathcal{E}^2(v_u), \tau = \mathcal{E}^2(j) \\
\forall_{i=1}^{l(\mathcal{E}^2)} \psi_i &= \begin{cases} k & \tau_i = idx(k) \wedge \sigma_i = \triangle \wedge \upsilon_i = 0 \\ \triangle & \tau_i = 0 \wedge \sigma_i = \triangle \wedge \upsilon_i = 0 \\ 0 & \tau_i = 0 \wedge \sigma_i = 0 \wedge \upsilon_i = 0 \\ \mathcal{S}_0(j) + k & \tau_i = 0 \wedge \sigma_i = k \in \mathbb{Z}_+ \wedge \upsilon_i = 0 \\ \perp & \tau_i = \perp \vee \sigma_i = \perp \vee \upsilon_i = \perp \\ \square & \text{otherwise} \end{cases} \\
\mathcal{E}^3 &= \mathcal{E}^2 \ominus \psi.
\end{aligned}$$

The inference for reduce expression is performed in a similar fashion up to the generation of ψ type that directly follows from the conditions on type ρ in the RED[$\triangle$] typing rule.

5.1.8. Selection and index concatenation. Similarly to the application, both arguments of the selection are abstracted into variables v_i and v_e, and the resulting type is generated using conditions on ρ type in the SEL[$\triangle$] rule.

$$\begin{aligned}
\mathcal{T}_{\text{inf}}(\mathcal{F}, \mathcal{E}, \text{sel}(v_i, v_e)) &= \left(\mathcal{F}, \mathcal{E}', rm(\psi, \psi)\right) \quad \texttt{where} \\
\tau &= \mathcal{E}(v_i), \sigma = \mathcal{E}(v_e) \\
\forall_{i=1}^{l(\mathcal{E})} \psi_i &= \begin{cases} \triangle & \tau_i = idx(k) \wedge \sigma_i = k \wedge k \in \mathbb{Z}_+ \\ \triangle & \tau_i = 0 \wedge \sigma_i = \triangle \\ 0 & \tau_i = 0 \wedge \sigma_i \in \mathbb{N} \\ \perp & \tau_i = \perp \vee \sigma_i = \perp \\ \square & \text{otherwise} \end{cases} \\
\mathcal{E}' &= \mathcal{E} \ominus \psi.
\end{aligned}$$

The inference for the index concatenation is constructed similarly.

5.1.9. Conditions. Conditions deserve a special attention here as it is the only kind of expressions that is able to absorb $\perp$ types in a branch, and propagate non-$\perp$ layout-types. This is a basic mechanism of the fix point iterator—in case there is a recursive call to a not yet inferred function in one of the branches, the return type of this function call would be $\perp$, and it would be propagated up to the branch expression. That would allow us to infer the type of the function and on the next iteration of the fixed point to precise it in case some of the layout combinations in the branch are untypable. Similarly to the case of application, we abstract predicate then branch and else branch expressions into the variables v_p, v_t and v_f accordingly. Here is the rule:

$$\begin{aligned}
&\mathcal{T}_{\text{inf}}\left(\mathcal{F}, \mathcal{E}, \text{if } v_p \text{ then } v_t \text{ else } v_f\right) = \left(\mathcal{F}, \mathcal{E}', rm(\psi, \psi)\right) \quad \texttt{where} \\
&\qquad \tau = \mathcal{E}(v_p), \sigma = \mathcal{E}(v_t), \phi = \mathcal{E}(v_f) \\
&\qquad \forall_{i=1}^{l(\mathcal{E})} \psi_i = \begin{cases}
\phi_i & \tau_i = 0 \wedge \phi_i = \sigma_i \\
\triangle & \tau_i = \triangle \wedge (\tau_i, \sigma_i) \in \{(0, \triangle), (\triangle, 0), (\triangle, \triangle)\} \\
\perp & \tau_i = \perp \vee (\phi_i = \perp \wedge \sigma_i = \perp) \\
\phi_i & \tau_i = 0 \wedge \phi_i \neq \perp \wedge \sigma_i = \perp \\
\sigma_i & \tau_i = 0 \wedge \phi_i = \perp \wedge \sigma_i \neq \perp \\
\triangle & \tau_i = \triangle \wedge ((\sigma_i \in \{0, \triangle\} \wedge \phi_i = \perp) \\
 & \qquad \vee (\phi_i \in \{0, \triangle\} \wedge \sigma_i = \perp)) \\
\square & \text{otherwise}
\end{cases} \\
&\qquad \mathcal{E}' = \mathcal{E} \ominus \psi.
\end{aligned}$$

5.2. *Sample layout inference*

We are going to consider an application of the inference algorithm to the `vplus` function, which is a part of the N-body code. The function is defined as

```
# (float[3], float[3]) -> float[3]
vplus (x,y) = map i < [3] (x[i] + y[i]).
```

It adds two three-element vectors component-wise. When the inference of the function definition starts, the letrec rule has created an environment that consist of all the possible argument layout type combinations returning bottom type. Here is how the environment $\mathcal{F}(\texttt{vplus})$ looks like:

$$\begin{array}{rcccccccccccccccc}
x: & 0 & 1 & idx(1) & \triangle & 0 & 1 & idx(1) & \triangle & 0 & 1 & idx(1) & \triangle & 0 & 1 & idx(1) & \triangle \\
y: & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & idx(1) & idx(1) & idx(1) & idx(1) & \triangle & \triangle & \triangle & \triangle \\
*: & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp.
\end{array}$$

We use $*$ to denote the return type of the function body to save some space on the page.

Now, according to the map inference rule, we expand $\mathcal{E}$ further by adding a type for i and a type for the upper expression. The valid types for i would be $\sigma = \langle 0, idx(1)\rangle$. For presentation purposes, we are going to add σ without expanding the environment. As for the upper expression, we could have expanded the environment with all the valid types for [3]; but as we are inferring the type inside the *map* expression, we know that all the types other than 0 would be cancelled out, so we might just add a vector of zeroes. Here is an updated environment:

$$\begin{array}{rcccccccccccccccc}
x: & 0 & 1 & idx(1) & \triangle & 0 & 1 & idx(1) & \triangle & 0 & 1 & idx(1) & \triangle & 0 & 1 & idx(1) & \triangle \\
y: & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & idx(1) & idx(1) & idx(1) & idx(1) & \triangle & \triangle & \triangle & \triangle \\
i: & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma & \sigma \\
[3]: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
*: & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp.
\end{array}$$

Please note that, when we use [3] in the environment, we assume the matching variable for the upper expression in the map. The body of the map is a primitive operation. Following the primitive function rule, we assume that we have surrounding variables for subexpression so we infer types for those first. The left-hand side expression is a selection sel(i, x), which in this particular case, would produce a type for combinations: $(0, 0), (0, 1), (idx(1), 1), (0, \triangle)$ and cross out all the rest columns. After this operation is performed, the environment would look as follows:

$$\begin{array}{rcccccccccccccccc}
i: & 0 & 0 & idx(1) & 0 & 0 & 0 & idx(1) & 0 & 0 & 0 & idx(1) & 0 & 0 & 0 & idx(1) & 0 \\
x: & 0 & 1 & 1 & \triangle & 0 & 1 & 1 & \triangle & 0 & 1 & 1 & \triangle & 0 & 1 & 1 & \triangle \\
y: & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & idx(1) & idx(1) & idx(1) & idx(1) & \triangle & \triangle & \triangle & \triangle \\
x[i]: & 0 & 0 & \triangle & \triangle & 0 & 0 & \triangle & \triangle & 0 & 0 & \triangle & \triangle & 0 & 0 & \triangle & \triangle \\
[3]: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
*: & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot.
\end{array}$$

Please note that we got rid of all the σ-s at this step. The right-hand side of the plus operation is also selection: `sel`(i, y), so we perform a similar inference step, and the new environment looks as follows:

$$\begin{array}{rcccccccccc}
i: & 0 & 0 & 0 & 0 & 0 & idx(1) & 0 & 0 & 0 & 0 \\
x: & 0 & 1 & \triangle & 0 & 1 & 1 & \triangle & 0 & 1 & \triangle \\
y: & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \triangle & \triangle & \triangle \\
x[i]: & 0 & 0 & \triangle & 0 & 0 & \triangle & \triangle & 0 & 0 & \triangle \\
y[i]: & 0 & 0 & 0 & 0 & 0 & \triangle & 0 & \triangle & \triangle & \triangle \\
[3]: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
*: & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot.
\end{array}$$

Finally, we apply the plus rule on $x[i]$ and $y[i]$ that is an inner body of the map; it allows us to infer the type for map that would be also a type for the body of the function (denoted with _ in the environment). After the application the environment would look as follows:

$$\begin{array}{rcccccccccc}
i: & 0 & 0 & 0 & 0 & 0 & idx(1) & 0 & 0 & 0 & 0 \\
x[i]: & 0 & 0 & \triangle & 0 & 0 & \triangle & \triangle & 0 & 0 & \triangle \\
y[i]: & 0 & 0 & 0 & 0 & 0 & \triangle & 0 & \triangle & \triangle & \triangle \\
x[i]+y[i]: & 0 & 0 & \triangle & 0 & 0 & \triangle & \triangle & \triangle & \triangle & \triangle \\
[3]: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
x: & 0 & 1 & \triangle & 0 & 1 & 1 & \triangle & 0 & 1 & \triangle \\
y: & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \triangle & \triangle & \triangle \\
*: & 0 & 0 & \triangle & 0 & 0 & 1 & \triangle & \triangle & \triangle & \triangle.
\end{array}$$

That finalises the inference for the `vplus`. The function type can be seen from the last three lines of the environment.

The resulting type of such a simple function like vector addition might be quite surprising or counterintuitive, but let us try to develop some intuition regarding this matter. The variety of possibilities comes from two facts:

(1) We are allowed perform a primitive operation on mixed scalar/vector arguments. Despite the fact that it is not necessary for constants, as we can always promote a scalar to a vector by assigning a $\triangle$ type to it; we cannot do that in case of expressions with dependencies. For example, in the expression $a[i] :: \triangle + b[j] :: 0$, there is no way to promote $b[j]$ to vector; however, we can still apply a vector plus on it.

(2) Scalar selection can be performed on arrays of layout-type $k \in \mathbb{Z}_+$. That is easy to understand if you think about the underlying transformation. In case an array has a type k, then we grouped its elements across dimension k into vectors. But we can still get a scalar component from the vectorised array by selecting a vector and then selecting the component from the vector.

It is easy to see that `vprod` types $(0,0) \rightarrow 0, (1,1) \rightarrow 1$, and $(\triangle, \triangle) \rightarrow \triangle$ should be valid for vector additions. Now, in $(0,0) \rightarrow 0$ type, we can replace first or second, or both arguments with 1; we should still get a 0 type as a result. Finally, we can have $(0, \triangle) \rightarrow \triangle$ as we can promote the scalar selection via vector plus. One can swap the arguments and obtain $(\triangle, 0)$ in the arguments, because plus is commutative. Finally, if we can make a scalar selection from 0 argument, we should be able to select from the argument of type 1, which gives us $(1, \triangle)$ and $(\triangle, 1)$ argument types.

6. INITIAL EVALUATION

In this section, we are going to present experimental results achieved so far. Please note that this is mostly a proof of concept showing that our framework can deliver excellent performance. Extensive measurements are left for future work.

The measurements we are going to present consist of two benchmarks: the N-body problem described in Figure 3 and the Mandelbrot problem. The properties of those benchmarks are rather different. The N-body is both memory intensive and compute intensive application involving an iterative update of a reasonably sized multi-dimensional array. Such a pattern can be found in many scientific applications such as solving partial differential equations numerically or other approximation problems. The Mandelbrot benchmark represents a class of applications where most of the execution time is spent on computations while the number of memory operations being very small. Also, the Mandelbrot benchmark requires a more complex pattern of vectorisation as a computation of an individual element is expressed as a tail-recursive function that has to be vectorised in order to match the new layout. This requires an additional effort when it comes to masking elements of individual vectors, and this pattern is not present in the N-body benchmark.

In both cases, the best vectorisation requires layout modifications none of which is achievable using existing compilers even with the highest level of optimisations, when the proposed layout inference delivers required vectorisations in both cases.

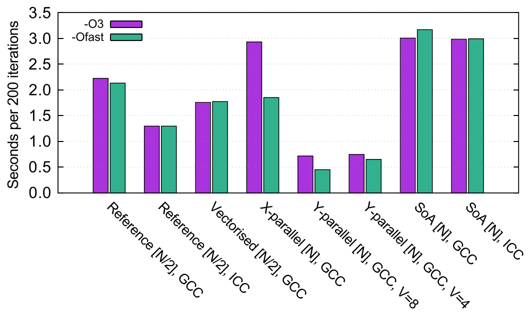
In the current set of experiments, we want to verify two things:

(1) The proposed inference does improve vectorisation.
(2) Effects of vectorisation are orthogonal to multi-threaded execution.

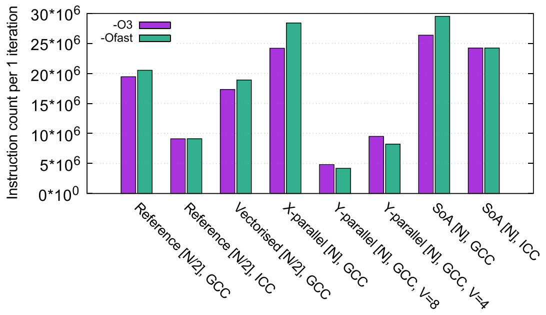
To verify the first statement, the only thing that matters is availability of SIMD instructions set on a CPU. For the second statement we require a CPU to have multiple cores. We use three different machines whose descriptions are presented in Figure 7. The 'amaterasu' and 'jove' machines can be seen as typical nodes of a cluster that have four and 12 cores with hyperthreading accordingly. The 'laptop' is a low-profile machine but with a strong vectorisation capabilities. As a consequence, our testbed allows to observe behaviour on a server-type hardware and verify both the vectorisa-

Name	**Description**
laptop	"Intel(R) core(TM) i3-2310m CPU @ 2.10Ghz" processor, 2 cores with hyperthreading, equipped with AVX instruction set (8 floats per SIMD register), running Gentoo Linux with kernel version "3.14.4-gentoo" with GCC compiler version "Gentoo 4.8.2 p1.3r1, pie-0.5.8r1" and ICC version "14.0.3 20140422".
amaterasu	"Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz" processor, 4 cores with hyperthreading, equipped with AVX2 instruction set (8 floats per SIMD register), running Linux with kernel version "2.6.32-431.17.1.el6.x86_64" and GCC version 4.9.0.
jove	Intel box with "Intel(R) Xeon(R) CPU X5650 @2.67Ghz", 12 cores with hyperthreading, equipped with SSE4.2 instruction set (4 floats per SIMD vector), running Linux with kernel version "2.6.32-431.17.1.el6.x86_64" and GCC version 4.9.0.

Figure 7. Machines used to produce the measurements.



(a) Runtime relationship.



(b) Instruction count relationship.

Figure 8. Runtime and instruction count relations for the N-body 1024 planets, 200 iterations on the “laptop”.

tion and scalability, while experiments on the ‘laptop’ tests put our vectorisation system in a more restrictive setup. This restrictiveness is very important for embedded or low-profile devices like smartphones, where vectorisation plays a crucial role in image or video processing applications. The SIMD instruction sets are different for all the three machines: AVX, AVX2 and SSE4.2.

All the machines are Intel-based, we use GNU Compiler Collection (GCC, GNU Project, Massachusetts Institute of Technology, MA, USA), Intel ICC (Intel, Santa Clara, CA, USA) and perf tool to count a number of instructions via performance counters of a CPU. The runtime is being measured by putting a timer around the core regions that excludes initialisation and output times. For every data point, the minimum of five runs is being taken. We compile all of the benchmarks with the following compilation flags:

```
GCC -Ofast -Wall -Wextra -mtune=native -march=native -std=gnu99
    -fomit-frame-pointer -fopenmp -lm -lrt
ICC -gcc -Ofast -Wall -Wextra -mtune=native -march=native
    -std=gnu99 -fomit-frame-pointer -openmp -lrt.
```

The transformation of the program is happening on a very high-level language that eventually has to be compiled down to some target language. The target language that we are dealing with is the C language; in order to express vector instructions, we use a GCC-based portable framework we have developed earlier in [3]. By hand-coding C programs, we mimic programs that we expect to be generated automatically by the SAC compiler—that should give us an idea of what runtimes we can expect. In order to mimic multi-threaded execution, we are using OpenMP annotations.

6.1. N-body

We start with observing runtime and instruction count relations of different implementations of the N-body on a single core in Figure 8.

On the left graph, we can see runtime figures of the N-body with 1024 planets and 200 iterations altering the level of optimisation between `-O3` and `-Ofast`. On the right graph, we relate the runtimes from the left graph with the number of instructions obtained by the perf tool. The unit of measure is instruction count per one N-body iteration that we obtain by: $\frac{Inst_{300}-Inst_0}{300}$, where $Inst_i$ is a number of instructions per i iterations of the N-body. The names of the benchmarks in Figure 8 mean the following:

Reference [N/2] is a reference C implementation of the N-body benchmark as it can be found at the Debian Shootout[¶]. One important thing to note is that this benchmark uses the fact that the absolute value of the acceleration for (i, j) planets is the same as acceleration for (j, i) planets, but has a different sign. We are going to mark such a solution with [N/2] postfix and the program that computes accelerations

[¶]The Computer Language Benchmarks Game, see http://benchmarksgame.alioth.debian.org/ for more details.

for all the pairs with [N] postfix. The version that is doing half of the computations makes a lot of sense in a single-threaded environment but makes it less favourable in a multi-threaded context, as outer level parallelisation introduces large overheads because of scheduling complexity. Please refer to [8] for more details.

Vectorised [N/2]	is the reference implementation vectorised across the inner axis.
X-parallel [N]	is vectorised across the inner (x) axis, with SIMD vectors of length 4, and it pads an array by adding dummy elements to the velocity triplet and position triplet.
Y-parallel [N]	is a vectorised implementation across the outer (y) axis. It does not add dummy elements, so the amount of memory that planets take is the same as in the reference implementation. One important property about this benchmark is that it can efficiently use long SIMD vectors, as every vector stores individual components of different planets, where the X-parallel version requires to pack individual triplets in the vector, which is more expensive. That is why we prefix this benchmark with the length of the vector we use: either four elements ($V = 4$) or eight elements ($V = 8$).
SoA [N]	is a reference implementation that computes all the pairs but transforms arrays of structures into structures of arrays.

The key observation from Figure 8 is that vectorisation across the outer axis performs the best, and the main reason for that is substantially smaller instruction count. Please note that instruction count is not the most precise metrics, as it does not directly correlate with runtimes. As you can see, in case of Reference and X-parallel with -Ofast, we have more instructions but better runtime. However, for the Y-parallel, the difference is too large to be ignored. As for the effects from turning -Ofast, we can see that GCC can do a better job, applying some of the vectorisation techniques and being able to change the order of reductions. As in case of Y-parallel, we change the order of the reduction anyway, we consider that an honest comparison would be in case of -Ofast. We would assume that main effects of the -Ofast are in the application of vector variant of the square root (we do that explicitly in case of Y-parallel) and from doing vector reduction in a more efficient way, as we express it as just as a sum of components. Finally, we can see that transforming an array of structures into a structure of arrays (SoA beanchmark) does not have the same effect as our layout transformations (X-parallel and Y-parallel), either none of the compilers recognised a potential for vectorisation or the locality effect resulted in disappointing performance. We are not going to consider SoA in further measurements.

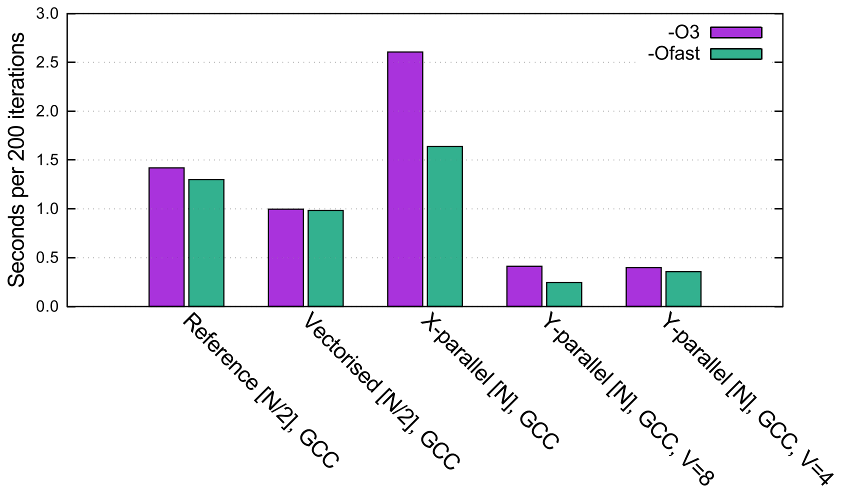
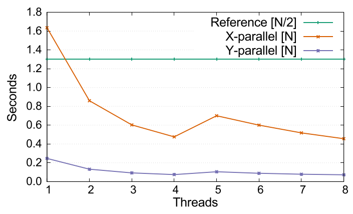
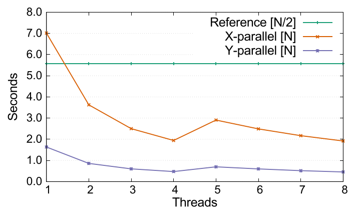


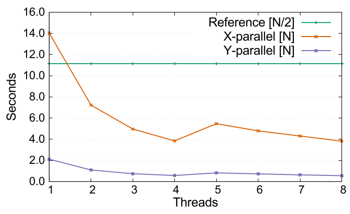
Figure 9. Runtime relations for the N-body 1024 planets, 200 iterations on the “amaterasu”.



(a) 1024 planets, 200 iterations.

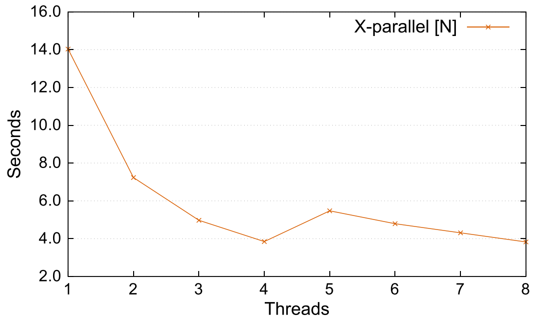


(b) 3000 planets, 100 iterations.

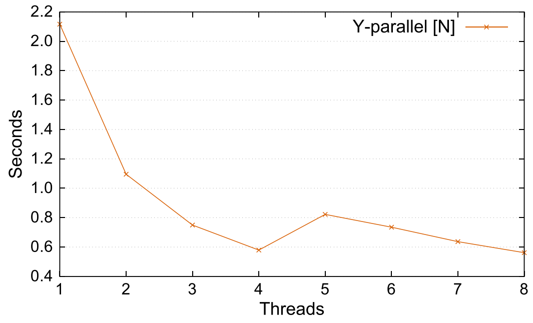


(c) 6000 planets, 50 iterations.

Figure 10. Scaling of the N-body on the “amaterasu”.



(a) 6000 planets, 50 iteration, X-parallel.



(b) 6000 planets, 50 iteration, Y-parallel.

Figure 11. Individual scaling of X-parallel and Y-parallel N-body versions.

Now, we would like to investigate how the implementations are going to behave in the presence of multithreading. First, we are going to run the experiment on the Intel-based CPU with four hyper-threaded cores. We do not have Intel compiler installed as well as ‘perf’ software on this machine, so we are going to start with single-core runtime relationships, which you can find on Figure 9.

As you can see, we have a similar relations as in Figure 8; however, the difference between X-parallel and Y-parallel is higher: 4.5 on the laptop and 6.5 here. The reason is not obvious; it could be either newer instruction set or newer compiler or faster memory or a combination of those.

To observe the effects of multi-core presence, we are running three series of experiments; increasing the size of the input data to observe if larger memory footprints influences the performance. The results are presented in Figure 10.

As we can see, the scaling across all the three figures is similar. The difference between X-parallel and Y-parallel decreases when the structure does not fit in the cache, but then it increases again in case of 6000 planets. From the graphs, it might seem that the Y-parallel scales worsen; in case of larger amount of cores, the runtimes of the X-parallel and Y-parallel can merge. That is not actually true; it is just the scale of the graph. In order to demonstrate that, we are going to look at individual graphs of the N-body with 6000 planets for X-parallel and Y-parallel versions. The runtimes are presented at Figure 11.

As you can see, the scaling graphs are very similar; actually, the Y-parallel version scales better. Both benchmarks have a ‘jump’ after four threads, which happens as there are only four physical cores, which means that in case of five and more threads two threads will share one cache.

Finally, we are measuring the N-body scaling at the example of Intel machine with 12 hyperthreaded cores and see how it scales there. Please find this experiment in Figure 12.

This experiment confirms the scaling claim—having more CPUs does not merge the runtimes of X-parallel and Y-parallel. The Y-parallel is four times faster than the X-parallel within a single thread and on 24 threads.

6.1.1. Mandelbrot. As a second example we look at a computation of Mandelbrot sets. The formulation of the Mandelbrot algorithm in SAC-λ, assuming that complex numbers are built-in and `<a, b>` denotes a complex constant $a + bi$, can be found in Figure 13 on the left. We assume that depth, height, width, X1, Y1, DX and DY are compile-time constants.

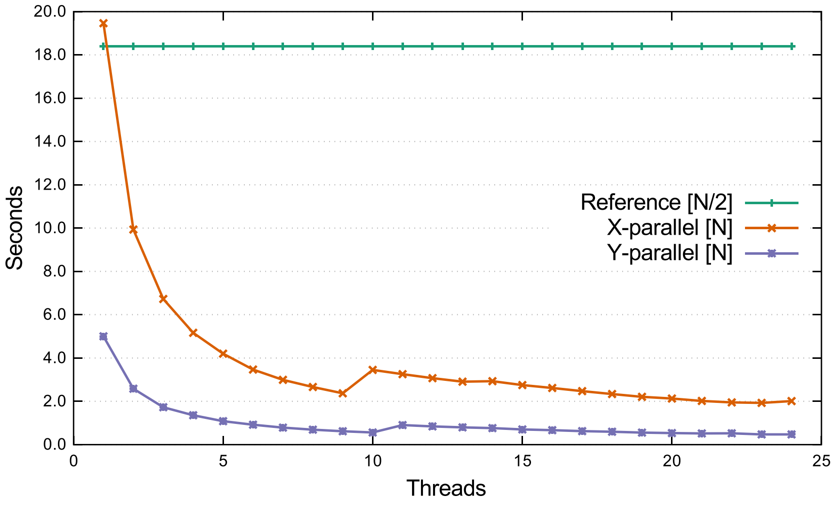
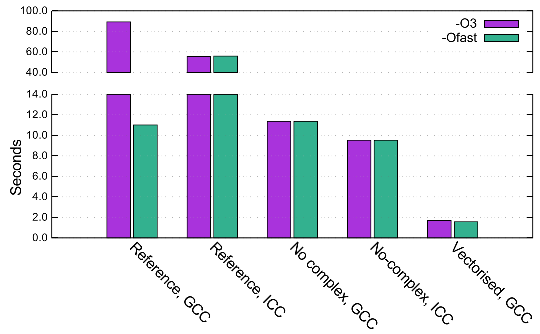


Figure 12. Scaling of the N-body implementations on the "jove".

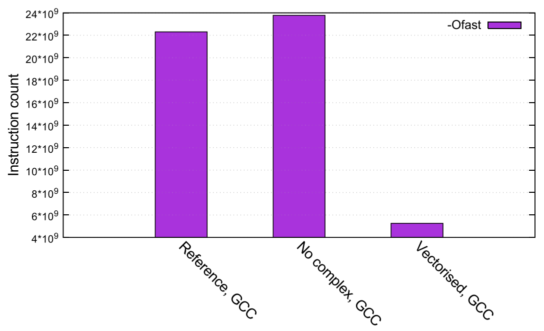
```
letrec
  iter (i, z, a) =
    if i < DEPTH
       and cabsf (a) < 2
    then
      iter (i + 1, z * z + a, a)
    else
      i
in
  map i < HEIGHT
    map j < WIDTH
      iter (0,
            <0, 0>,
            <X1 + DX*j, Y1 + DY*i>)
```

```
letrec
  iter (i, z, a) =
    if i < DEPTH and
       sqrt (z[0]*z[0]
             + z[1]*z[1]) < 2
    then
      iter (i + 1,
            [z[0]*z[0] - z[1]*z[1] + a[0],
             z[0]*z[1] + z[1]*z[0] + a[1]],
            a)
    else
      i
in
  map i < HEIGHT
    map j < WIDTH
      iter (0, [0, 0],
            [X1 + DX*j, Y1 + DY*i])
```

Figure 13. Formulation of the Mandelbrot problem in SAC-λ using built-in complex numbers on the lef and using array-based representation of complex numbers on the right.



(a) Runtime relationship — *please mind the break in the graph.*



(b) Instruction count relationship.

Figure 14. Runtime relations for the Mandelbrot 2048 × 2048 floats with depth=4096 on the "laptop".

No matter if complex numbers are built-in or not, at some point, they are going to be represented as two-element structures; scalar operations on complex numbers will be expressed via normal scalar operations. We demonstrate this in Figure 13 on the right.

As in the case of the N-body, we are going to start with sequential runtime and instruction count relations obtained on the 'laptop'. Please find it in Figure 14. The names of the benchmarks on Figure 14 mean the following:

Reference is a literal translation of the high-level formulation found on the left hand side of Figure 13 in C using float-based complex numbers.

No-complex is a literal translation of the high-level formulation found on the right hand side of Figure 13 in C using floats. We also get avoid square root computation by translating $\sqrt{z_0^2 + z_1^2} < 2$ into $z_0^2 + z_1^2 < 4$.

Vectorised is a vectorisation of the No-complex function using $(\triangle, \triangle, \triangle) \to \triangle$ layout type for the function `iter`, which means that we compute V iterations simultaneously. The function itself is quite simple; however, the required code transformation are non-trivial—it requires creating data-flow masks with further predication, and it all happens in the recursive context. The transformed code will look like this:

```
# (float[V], float[2,V], float[2,V]) -> float[V]
iter ($\vec{i}$, $\vec{z}$, $\vec{a}$) =
    let
        $\vec{m}$ = ($\vec{i}$ < $\vec{d}$) and ($\vec{z}$ * $\vec{\vec{z}}$ < $\vec{4}$);
    in
        if $\vec{m}$ == $\overrightarrow{false}$ then
            $\vec{i}$;
        else
            let
                $\vec{t_1}$ = iter ($\vec{i} + \vec{1}$, $\vec{z}$ * $\vec{z}$ + $\vec{a}$, $\vec{a}$);
            in
                select ($\vec{m}$, $\vec{t_1}$, $\vec{i}$),
```

where $\vec{x}$ corresponds to the variable x with its shape being replicated V times; $\vec{z}$ * $\vec{\vec{z}}$ corresponds to vectorised version of $z_0^2 + z_1^2$; boolean operations are computed component-wise; multiplication and addition are overloaded for complex numbers and `select(m,a,b)` corresponds to `if m[i] then a[i] else b[i]` applied to all V vector components.

As we can see, we have a similar to N-body situation with respect to runtime and instruction relations—vectorised version runs much faster and has significantly less instructions. Please note that here we have measured instruction counts in the overall program; it does include I/O operations, but their contribution to the overall instruction count is negligible. As for the Reference version, we can see that GCC compiler can bring it down to No-complex at -Ofast, where ICC fails to do so. We do exclude this version from the further measurements.

The scaling runtimes on the 'amaterasu' can be found in Figure 15. Please note that we are using default *dynamic* scheduler in OpenMP, as computations are non-uniform over the grid.

As we can see, the scaling is much smoother than in the case of the N-body, and we do not have a 'jump' after four threads. That is because the Mandelbrot problem is clearly more compute-bound than memory bound.

Finally, we present scaling figures on 'jove' to ensure that adding more cores does not make the runtimes to merge. The results are presented at Figure 16

7. RELATED WORK

The idea to modify data layouts by means of compiler transformations is not new. There has been quite some work in the context of optimisations for improved cache behaviour [9, 10] and, more recently, for improved streaming through GPUs [11, 12]. In that work, improvements of spatial and temporal locality are the key goals. While this may seem to be a goal very similar to what we

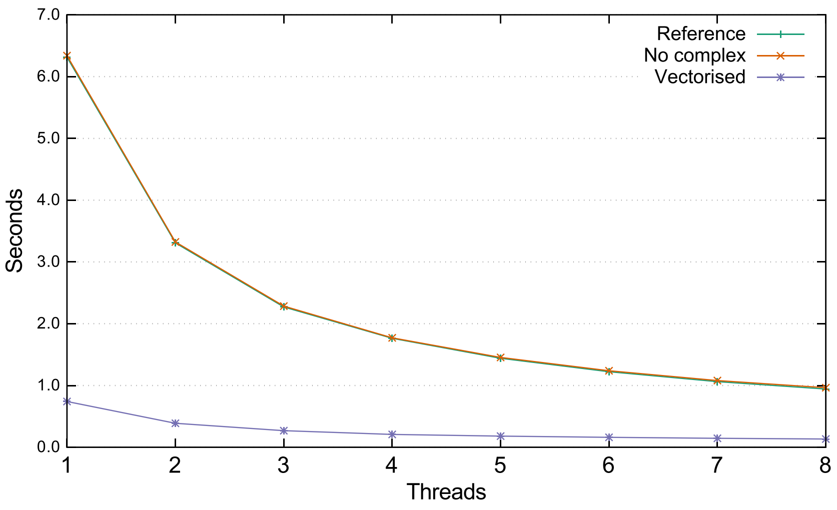


Figure 15. Scaling for the Mandelbrot implementations with 2048 × 2048 floats and depth=4096 on the "amaterasu".

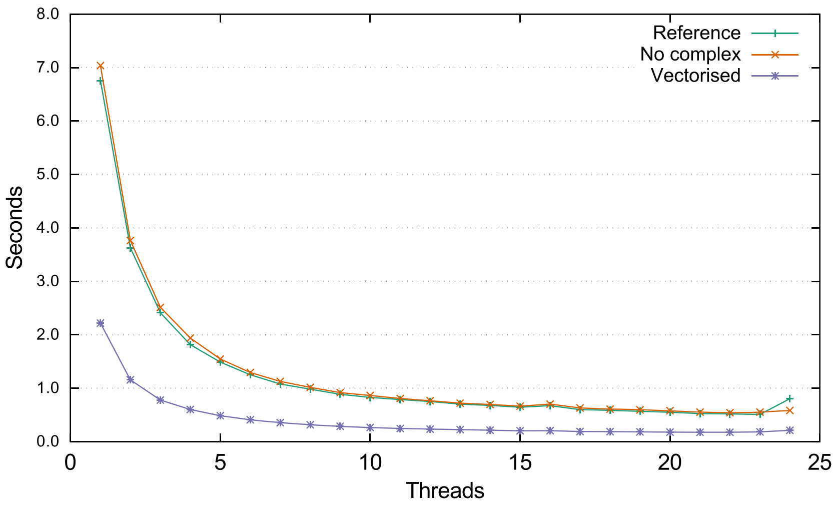


Figure 16. Scaling for the Mandelbrot implementations with 2048 × 2048 floats and depth = 4096 on the "jove".

propose here, spatial locality is not sufficient for an efficient vectorisation, as we experienced at the example of a N-body—vectorisation across the inner dimension has better spatial locality than the variant we have inferred.

A lot of related work is based on polyherda model, so we will start with a couple of comments about it. In principle, the work that we describe in this paper can be formulated in terms of polyhedra. Our maps and reduces have similarities to the loop-nests and by trying to vectorise every loop in a loop nest using 'direct outer loop vectorisation' technique from [13] with further checks if the vectorisation succeeded we can generate layout types for the arrays referenced in this loop nest. Generated constraints have to be resolved and polyhedra model would not help here. After that, the code have to be transformed that can be solved by the polyhedra. The main weakness of this technique in our set-up is the lack of support for function applications inside loop nests; obviously, not all the functions can be inlined. That would have a serious impact when it comes to inferring the layout types for the overall program. Second, we plan to use this work as a basis for more complex vectorisations like grid and stencil computations by introducing new kinds of layout types, in which case, we believe that pure functional setup would allow more aggressive optimisations.

Now, we would like to pay attention to the individual works. G. Chen *et al.* in [14] describe a similar approach. They as well propose to infer data layouts of the arrays in the whole program. The main focus of their work is to formulate potential layouts for arrays as a constraint network and solve it. The layouts are defined as vectors in an N-dimensional space. The work is more on the theoretical side of things, and the application is not described. There is no discussion about selection of the layout-setting for the whole program assuming that constraint resolution returned a number of alternatives.

U. Bondhungula *et al.* in [15] present a polyhedra model based transformation for tiling loop nests for further parallelisation by means of OpenMP. The polyhedral framework described in their paper is able to handle sophisticated loop nests; however, the transformation changes only the order of the iterations that might not be sufficient for efficient vectorisation. Transition from the inferred iteration order to the new layout is non-trivial, as the layouts have to match for the variables that are being reused.

K. Trifunovic *et al.* in [16] present a polyhedra based transformation for automatic vectorisation. Similar to [15], this work assumes that layouts are fixed, and the transformation is substituting identical arithmetic operations with vector ones.

T. Hanretty *et al.* in [17] present a framework to optimise alignment conflicts caused by stencil computations. The key idea of the transformation is very similar to what we do—interchanging dimensions with further transposition. The main difference of the approaches is that in our work we are concerned by the overall program performance, as layout transformations for the sake of optimisation of a certain operation may have a negative effect on the overall performance. So we are concerned with generating all the potential program vectorisations and choosing the best performing. On the other hand, the transformation described in [17] would not currently be applicable in our setup as it uses operations in selection functions that are not allowed. However, by applying a preprocessing step on the stencil-like computations, we can express it in the acceptable form for our inference system.

Roland Leißa *et al.* in [18] demonstrate a language that is an extension of C, which allows to annotate data types that later are used by the type inference to infer and propagate vector operations using scalar code. The main use-case demonstrated in the paper is very similar to the N-body case where vectors of triplets are being vectorised over the individual component axis rather than over the whole structure. The main difference of the approaches is that we concentrate on an automatic inference of the layout without providing any annotations. Another difference is that we use multidimensional arrays instead of vectors of records.

P. Clauss and B. Meister in [19] present a framework to optimise a data locality of the loop-nest by rearranging data layouts of arrays. The transformation proposed in the paper, for a given loop-nest generates new indexing functions for the dependent arrays such that iterations would access arrays sequentially in terms of the loop nest. It looks like an ideal solution from the theoretical point of view; however, it is not clear how to solve the same problem for multiple loop-nests.

M. Kandemir and I. Kadayif in [10] propose to change memory layouts dynamically to achieve better locality in loop nests. This is an interesting approach that can be considered as a next step for our framework, as currently we explicitly avoid layout changes of any array at runtime. On the other hand, one can easily construct a situation where two expressions require one and the same array to be have contradicting layouts. The main idea the technique proposes is to estimate while execution if changing a layout improves the cost of the loop-nest we are about to execute, and if it does perform a dynamic adjustment. In our case, we would have to adjust the cost function, as we are not concerned with locality, we are rather concerned with a runtime, which is a bit harder to estimate. Also, the approach assumes that a program is a series of loop-nests joined by control-flow, which is not directly applicable in our setup.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we advocate a novel systematic approach towards data layout transformation that enables vectorisation. This approach is motivated by the observation that many scientific codes have vectorisation potential that cannot be utilised because of an algorithm driven choice of data-layouts that is at odds with an effective vectorisation. The paper presents one such example, namely, the naive N-body code and discusses why the straight-forward formulation leads to an unfavourable data layout.

Building on the N-body example, the paper develops our approach towards a systematic inference of layout transformations. By means of a type system, we abstract from all program detail that is not relevant for a choice of data layouts. Using this abstraction facilitates not only the inference of layouts themselves it also guarantees the consistency of all inferred layouts.

We describe the type system as well as an inference algorithm in detail; we show how this identifies a few possible layout variations for our running example, the N-body code.

The paper also provides some initial performance figures. Manually modified codes that reflect the inferred layout transformations show that substantial runtime improvements close to the vector-width of the architecture used are achieved over competitive C implementations of the N-body and Mandelbrot problems. They also show that these improvements are orthogonal to non-vector-based parallelisations that stem from the use of multi-core CPUs and that effects of our techniques can be achieved on low-profile and high-end architectures.

The orthogonality between vectorisation and multi-threaded parallel execution renders this work particularly powerful in the context of code generation for high-performance executions. The complexity of the program transformations that might be necessary to achieve the inferred layouts suggests that the full potential of this approach would be ideally realised through a fully automated, compiler driven process.

This vision guides our future work. We have implemented the layout inference in SAC, and we are working on code generation; specifically, we would like to make sure that we can at least match performance of the hand-written code presented in the evaluation section. Second, a cost model similar to that of existing auto-vectorisers should be used to decide which legal layout variant to choose for a given target platform.

Further, we intend to use this platform as a basis for more complicated vectorisation patterns. For example, the stencil-like computations are not properly supported, that is, if we have an expression similar as follows:

```
map i < N
    a[i-1] + a[i] + a[i+1].
```

Our system can vectorise it, but the update is not going to be in-place. Currently, there is an optimisation in SAC that converts such a code into in-place update; but then for efficient vectorisation, we would have to merge this techniques with new array layouts described in [17], infer boundary conditions and fit it together in the compiler.

Another area of research is to allow for dynamic layout transformation, as currently, it is easy to construct an example where none of the data layouts would be changed, as two or more expressions have contradicting requirements. One could consider adjusting layout on the fly but, in that case, has to make sure that this transformation is not harmful.

Finally, cost of the vectorised functions can be arbitrary hard or even impossible to compare as they might depend on the statically unknown parameters. With that respect, we may apply a symbolic evaluation for the comparison; however, it does not cover all the cases. One might consider asking a programmer to provide annotations for a certain variables. In that case, we can apply interval analysis techniques, or the annotations could be in some different form with a clear algebra.

ACKNOWLEDGEMENTS

The authors are very grateful to the anonymous reviewers who spotted a number of inconsistencies and who proposed improvements for the paper to be better.

REFERENCES

1. Nuzman D, Henderson R. Multi-platform auto-vectorization. *Proceedings of the International Symposium on Code Generation and Optimization,* CGO '06, IEEE Computer Society: Washington, DC, USA, 2006; 281–294.
2. Kennedy K, Allen JR. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2002.
3. Šinkarovs A, Scholz S-B. Portable support for explicit vectorisation in C. *16th Workshop on Compilers for Parallel Computing (CPC'12)*, 2012.
4. Šinkarovs A, Scholz S-B. Semantics-preserving data layout transformations for improved vectorisation. *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing,* FHPC '13, Boston, Massachusetts, USA, ACM: New York, NY, USA, 2013; 59–70.
5. Grelck C, Scholz S-B. sac: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming* 2006; **34**(4):383–427.
6. Grelck C, Scholz S-B, Shafarenko A. A binding scope analysis for generic programs on arrays. *Proceedings of the 17th International Conference on Implementation and Application of Functional Languages,* IFL'05, Springer-Verlag: Berlin, Heidelberg, 2006; 212–230.
7. Feautrier P. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 1991; **20**(1):23–53.
8. Šinkarovs A, Scholz S-B, Bernecky R, Douma R, Grelck C. SAC/C formulations of the all-pairs N-body problem and their performance on SMPS and GPGPUS. *Concurrency and Computation: Practice and Experience* 2014; **26**(4):952–971.
9. Lu Q, Alias C, Bondhugula U, Henretty T, Krishnamoorthy S, Ramanujam J, Rountev A, Sadayappan P, Chen Y, Lin H, Ngai T-F. Data layout transformation for enhancing data locality on NUCA chip multiprocessors. *18th International Conference on Parallel Architectures and Compilation Techniques, 2009. PACT '09*, Raleigh, NC, September 2009; 348–357.
10. Kandemir M, Kadayif I. Compiler-directed selection of dynamic memory layouts. *Proceedings of the Ninth International Symposium on Hardware/Software Codesign, 2001. CODES 2001*, Copenhagen, 2001; 219–224.
11. Liu GD, mei W. Hwu Wen. Dl: A data layout transformation system for heterogeneous computing. *Proceedings IEEE Conference Innovative Parallel Computing (INPAR 12), IEEE*, San Jose, CA, 2012; 1–11.
12. Sung I-J, Stratton JA, Hwu W-MW. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques,* PACT '10, ACM: New York, NY, USA, 2010; 513–522.
13. Nuzman D, Zaks A. Outer-loop vectorization: revisited for short SIMD architectures. *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques,* PACT '08, ACM: New York, NY, USA, 2008; 2–11.
14. Chen G. A constraint network based approach to memory layout optimization. *In Proceedings of the Conference on Design, Automation and Test in Europe*, Munich, Germany, 2005; 1156–1161.
15. Bondhugula U, Hartono A, Ramanujam J, Sadayappan P. A practical automatic polyhedral parallelizer and locality optimizer. *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation,* PLDI '08, ACM: New York, NY, USA, 2008; 101–113.
16. Trifunovic K, Nuzman D, Cohen A, Zaks A, Rosen I. Polyhedral-model guided loop-nest auto-vectorization. *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques,* PACT '09, IEEE Computer Society: Washington, DC, USA, 2009; 327–337.
17. Henretty T, Stock K, Pouchet L-N, Franchetti F, Ramanujam J, Sadayappan P. Data layout transformation for stencil computations on short-vector SIMD architectures. *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, Springer-Verlag: Berlin, Heidelberg, 2011; 225–245.
18. Leißa R, Hack S, Wald I. Extending a C-like language for portable SIMD programming. *Principles and Practice of Parallel Programming*, New Orleans, Louisiana, USA, 2012; 65–74.
19. Clauss P, Meister B. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests, 2000.