Data Layout Inference for Code Vectorisation

Artjoms Šinkarovs, Sven-Bodo Scholz
School of Mathematica and Computer Sciences
Heriot-Watt University
Edinburgh, United Kingdom
a.sinkarovs@macs.hw.ac.uk

***Abstract*—SIMD instructions of modern CPUs are crucially important for the performance of compute-intensive algorithms. Auto-vectorisation often fails due to an unfortunate choice of data layout by the programmer. This paper proposes a data layout inference for auto-vectorisation which identifies layout transformations that convert SIMD-unfavorable layouts of data structures into favorable 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.**

I. Introduction

In the last decade vectorisation became an important research topic again, as most of the modern CPUs grant vectorisation capabilities by means of SIMD instructions [10]. Classical research into auto-vectorisation focuses on the optimisation of loop nestings [8]. Data-independent operations within such loop nestings are identified and 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 which we need to 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 all those 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 and, consequently, they also impact the choice of data layouts.

In this paper, we focus on the challenge of identifying suitable layout combinations that enable auto-vectorisation. 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 which we use as a vehicle for our experiments. In SaC, all the data structures are n-dimensional arrays, data parallel loop-nests are expressed using an explicit syntactical construction, and memory management is fully implicit.

The key idea of our approach is a layout inference that identifies ideal layouts (wrt SIMD vectorisation) for each individual loop nesting and then employs representational changes whenever necessary. 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 III introduces our running example in that language and it discusses the key ideas of our approach at it. Section IV provides a formal presentation of our type system and Section V sketches a possible algorithm for inferring the layout types and applies it to the N-body example before Section VI shows the effect of these transformations on the execution times of our running examples. Related work is presented in Section VII before Section VIII concludes.

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

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

Fig. 1. The semantics of MAP and REDUCE operations

II. CORE PROGRAMMING LANGUAGE

The inference and program transformation described in this paper are based on a stripped down version of SaC, as described in [5]. It contains only the bare essentials of the language adjusted to a λ-calculus style in order to facilitate a more concise description of our techniques.

Fig. 2 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, i.e. lambda abstractions, are not supported. Function applications are written in C style, i.e. arguments is 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. Additionally, our core language contains two combinators, *map* and *reduce*. They serve as vehicle for expressing data parallel operations.

$$\begin{array}{lcl} \mathit{Program} & \Rightarrow & \textbf{letrec}\ \lceil\, \mathit{FunDef}\, \rfloor^{*}\ \textbf{in}\ E \\ \mathit{FunDef} & \Rightarrow & \mathit{FunId}\ (\ \lceil\, \mathit{Id} \lfloor\ ,\ \mathit{Id}\, \rfloor^{*}\ \rfloor\ )\ =\ E \\ E & \Rightarrow & \mathit{Const} \mid \mathit{Id} \mid \mathit{FunId}\ (\ \lceil\, E \lfloor\ ,\ E\, \rfloor^{*}\ \rfloor\ ) \\ & \mid & \mathit{Prf}\ (\ \lceil\, E \lfloor\ ,\ E\, \rfloor^{*}\ \rfloor\ ) \\ & \mid & \textbf{if}\ E\ \textbf{then}\ E\ \textbf{else}\ E \\ & \mid & \textbf{let}\ \mathit{Id} = E\ \textbf{in}\ E \\ & \mid & \textbf{map}\ \mathit{Id} < E\ E \\ & \mid & \textbf{reduce}\ \mathit{Id} < E\ (\ \mathit{FunId}\ )\ \ E \\ \mathit{Prf} & \Rightarrow & +\ \mid\ -\ \mid\ \textbf{sel}\ \mid\ \ldots \end{array}$$

Fig. 2. The syntax of SaC

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 represent the meaning of a program 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, i.e. 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.

We use a standard big-step operational semantics as defined in detail in [5]. 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.

A. Map/Reduce

These two operators constitute simplified versions of the with-loop-constructs in full-fledged SaC[a]. They are array versions of the well-known combinators. Both operators compute expressions over an N-dimensional index space and then either combine the results in an array (map variant) or fold them using a binary operator. For example, the expression `map i<[2,3] 3` evaluates to `[[3,3,3],[3,3,3]]`, i.e., both these expressions have the formal semantics $\langle [2,3], [3,3,3,3,3,3] \rangle$; while the expression `reduce i<[2,3] (+) 3` results in `18`. Note here, that the indexing variable can be referred to within the expression: the expression `map i<[3] i` yields `[0,1,2]`.

A formalisation of the semantics of `map` and `reduce` based on the deduction system from [5] is presented in Fig. 1.

The map-rule shows that the upper limit e_u has to reduce to an n-element vector which 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 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.

[a]For details on with-loops in SaC see [4].

B. Vector operations

For the context of this paper, we consider all primitive functions to have SIMD counterparts. The only exception is the selection operation `sel (iv, a)` which selects the element at the index position `iv` of the array `a`. Furthermore, we assume that all SIMD operations have a built-in fixed vector length $[V]$ where V is a given target architecture specific constant.

III. Running Example

The main motivating example that we are going to use across the rest of the paper is an implementation of the N-body problem. The problem is defined as an iterative approximation, where on 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 position of i-th planet is updated using the newly computed acceleration. For more details, please refer to [13] which discusses the N-body problem in more detail. Further down we provide a core implementation of the benchmark using our stripped-down version of SaC. It has been slightly adjusted from the version discussed in [13] to be better suited for demonstrating our inference technique.

```
# (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)
```

Please note that we use 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 Function `acceleration` which computes the acceleration between two planets `posx` and `posy`.

It is remarkable that 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 which 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 increased memory footprint of the array may slow down the overall performance. This is something that is very easy to miss, by just looking at the code, most importantly it is easy to miss an alternative solution.

One might consider a vectorisation of the array of accelerations over the components of triplets rather than over triplets themselves. 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 programs; ii) the transformation might not be beneficial.

A. 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 $[s_1, ..., s_n]$ we consider vectorisations in all possible axes $1, ..., n$. A vectorisation in axis k will lead to a layout remapping into an array A^k of shape $[s_1, ..., s_{k-1}, s_k/V, s_{k+1}, ..., s_n, V]$ 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 axis 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, axis with 2 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.

IV. A type system for data-layouts

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 $1,\ldots,n$, where i refers to the transformation of the shape $[s_1,\ldots,s_n]$ into $[s_1,\ldots,s_{i-1}/V,\ldots,s_n,V]$. In addition, we use 0 to denote the shape identity and we use $\triangle$ to denote a shape extension from $[s_1,\ldots,s_n]$ into $[s_1,\ldots,s_n,V]$. 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 $idx(m), m \in \mathbb{Z}_+$ which denote that the m-th component of the index vector is considered for vectorisation. Now, we can define the set or layout types as

$$L = \mathbb{N} \cup \{\triangle\} \cup idx(m), m \in \mathbb{N} \qquad [b]$$

We also introduce layout-type-signatures to denote the different possible layout transformations an individual function can be applied to. Formally, these are described by $(l^1,\ldots,l^t) \to l$ where all l^i and l are layout types as defined above.

We denote the union of L with all layout-type-signatures over L by LT.

A. 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, i.e., 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 $\mathcal{E} \oplus (v, \langle l_1,\ldots,l_n\rangle)$ denotes an environment that returns the vector type $\langle l_1,\ldots,l_n\rangle$ for the variable v.

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 $\mathcal{F} \subset Id \times \mathcal{E}$. Lookup of a function identifier f and presence of function identifiers are denoted in the same way as its done for environments, i.e., we use $\mathcal{F}(f)$ and $\mathcal{F} \oplus \mathcal{E}$, respectively.

As we require all entries of one environment to have the same length, an environment $\mathcal{E}$ of a function f takes the general form

$$\mathcal{E} = \{v_1 : \langle l_1^1,\ldots,l_n^1\rangle, \ldots v_m : \langle l_1^m,\ldots,l_n^m\rangle, f : \langle l_1^f,\ldots,l_n^f\rangle\}$$

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 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 $\mathcal{E}^{[i]}$.

B. 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 are summarised below. Please note that in the rest of the paper we use a $\mathcal{D}(e)$ notation to denote a number of axes in e.

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

$$\textsc{Var}: \frac{}{\mathcal{F}, \mathcal{E} \vdash x : \mathcal{E}(x)}$$

$$\textsc{App}: \frac{\begin{array}{c}\forall 1 \le i \le t \quad \mathcal{F}, \mathcal{E} \vdash e_i : \langle \tau_1^i,\ldots,\tau_n^i\rangle \\ \mathcal{F}, \mathcal{F}(f) \vdash f : \langle (l_1^1,\ldots,l_1^t) \to l_1,\ldots,(l_m^1,\ldots,l_m^t) \to l_m\rangle \\ \forall 1 \le i \le n \quad \exists 1 \le j \le m \\ \forall 1 \le k \le t \quad (l_j^k = \tau_i^k) \wedge (l_j = \tau_i)\end{array}}{\mathcal{F}, \mathcal{E} \vdash f(e_1,\ldots,e_t) : \langle \tau_1,\ldots,\tau_n\rangle}$$

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

[b] 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.

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

Fig. 3. LETREC layout rule.

Finally the rule that allows application of all the functions defined in the program can be found in Fig. 3. 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 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 which 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 below.

$$\textsc{Prf}[\triangle]: \frac{\begin{array}{c}\mathcal{F}, \mathcal{E} \vdash e_1 : \langle \tau_1, \ldots, \tau_n \rangle \\ \mathcal{F}, \mathcal{E} \vdash e_2 : \langle \tau_1', \ldots, \tau_n' \rangle \\ \forall 1 \le i \le n \\ \sigma_i = \begin{cases} 0 & (\tau_i = 0) \wedge (\tau_i' = 0) \\ \triangle & (\tau_i = 0) \wedge (\tau_i' = \triangle) \\ \triangle & (\tau_i = \triangle) \wedge (\tau_i' = 0) \\ \triangle & (\tau_i = \triangle) \wedge (\tau_i' = \triangle) \end{cases}\end{array}}{\mathcal{F}, \mathcal{E} \vdash +(e_1, e_2) : \langle \sigma_1, \ldots, \sigma_n \rangle}$$

$$\textsc{Map}[\triangle]: \frac{\begin{array}{c}\mathcal{F}, \mathcal{E} \oplus (j, \langle \tau_1, \ldots, \tau_n \rangle) \vdash e : \langle \sigma_1, \ldots, \sigma_n \rangle \\ \forall 1 \le i \le n \\ \rho_i = \begin{cases} k & \tau_i = idx(k) \wedge \sigma_i = \triangle \\ \triangle & \tau_i = 0 \wedge \sigma_i = \triangle \\ 0 & \tau_i = 0 \wedge \sigma_i = 0 \\ \mathcal{D}(j) + k & \tau_i = 0 \wedge \sigma_i = k \in \mathbb{Z}_+ \end{cases}\end{array}}{\mathcal{F}, \mathcal{E} \vdash \text{map } j < u \; e : \langle \rho_1, \ldots, \rho_n \rangle}$$

$$\textsc{Red}[\triangle]: \frac{\begin{array}{c}\sigma_k^{bin} \equiv (\sigma_k, \sigma_k) \to \sigma_k \\ \mathcal{F}, \mathcal{F}(f) \vdash f : \langle \sigma_1^{bin}, \ldots, \sigma_n^{bin} \rangle \\ \mathcal{F}, \mathcal{E} \oplus (j, \langle \tau_1, \ldots, \tau_n \rangle) \vdash e : \langle \sigma_1, \ldots, \sigma_n \rangle \\ \forall 1 \le i \le n \\ \rho_i = \begin{cases} 0 & \tau_i = idx(k) \wedge \sigma_i = \triangle \\ \triangle & \tau_i = 0 \wedge \sigma_i = \triangle \\ \sigma_i & \tau_i = 0 \wedge \sigma_i \in \mathbb{N} \end{cases}\end{array}}{\mathcal{F}, \mathcal{E} \vdash \text{reduce } j < u \; f \; e : \langle \rho_1, \ldots, \rho_n \rangle}$$

$$\textsc{Idx}[\triangle]: \frac{\begin{array}{c}\mathcal{F}, \mathcal{E} \vdash j : \langle \tau_1, \ldots, \tau_n \rangle \\ \mathcal{F}, \mathcal{E} \vdash h : \langle \tau_1', \ldots, \tau_n' \rangle \\ \forall 1 \le i \le n \\ \sigma_i = \begin{cases} idx(k) & \tau_i = idx(k) \wedge \tau_i' = 0 \\ idx(\mathcal{D}(j) + k) & \tau_i = 0 \wedge \tau_i' = idx(k) \\ 0 & \tau_i = 0 \wedge \tau_i' = 0 \end{cases}\end{array}}{\mathcal{F}, \mathcal{E} \vdash j +\!\!+ \; h : \langle \sigma_1, \ldots, \sigma_n \rangle}$$

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

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

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

1) The map construct may control a layout transformation, i.e., 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 ($\triangle$).
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.

V. LAYOUT INFERENCE

The layout inference can be directly deduced from the layout rules similarly to monomorphic type systems. 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 empty $\mathcal{F}$ which is extended whenever a function is being processed. For an individual function we also start with empty environment $\mathcal{E} = \{\}$ and extend it whenever a rule uses the $\oplus$ operation.

There are two important things to consider: i) how to extend an environment with a new type; ii) how the initial types get into an environment.

The first question is about the meaning of $\mathcal{E} \oplus \{(x, \vec{\tau})\}$. In order to approach it in a bit more general setup we are going to consider the right hand side as a new environment $\mathcal{E}' = \{x : \vec{\tau}\}$ and investigate an intersection of two environments. That would allow us to add not only a constraint for a single variable but a set of constraints for multiple variants.

A. Environment intersection

For two environments $\mathcal{E}_1$ and $\mathcal{E}_2$ environment intersection is built as a Cartesian products of columns of $\mathcal{E}_1$ with columns of $\mathcal{E}_2$. Let us assume that $\mathcal{E} = \langle \mathcal{E}^{[1]}, \ldots, \mathcal{E}^{[n]} \rangle$.

1) Create a Cartesian product of columns of $\mathcal{E}_1$ and $\mathcal{E}_2$.
2) For each element of the pair $(\mathcal{E}^{[i]}, \mathcal{E}^{[j]})$ build a new column $\mathcal{E}^{[(i,j)]}$ using the following rules. For every variable v that can be found in both columns ensure that the type is the same or fail.
   Remaining variables that are not found in both columns are added in the result with a unique layout-type variable.
3) Join non-empty columns $\mathcal{E}^{[k]}, 1 \leq k \leq m$ back into environment.

As for the second question, constants and variables extend the existing environment by adding all the possible layouts that the given constant or variable can take. Map/reduce operations extend the environment by adding all possible layouts for the index variable iv which are $\langle 0, idx(1), \ldots, idx(\mathcal{D}(iv)) \rangle$. Finally function applications extend the environment by constraints on the arguments. For example, for a function application `f (a, b)`, where f have layout types $f : \langle (1,0) \to 0, (1,2) \to 2 \rangle$, the constraints are $\mathcal{E}' = \{a : \langle 1,1 \rangle, b : \langle 0,2 \rangle\}$.

For mutually recursive functions we can always generate all the potential layouts for a given function application, and eliminate contradicting ones. While the size of environments theoretically can grow exponentially with the number of variables, in all examples we have investigated so far the existing constraints have limited the growth to very few eligible entries. As we intend to create these entries on demand, we do not expect this to constitute a practically relevant issue.

B. Layout inference for the N-body code

Due to space limitations we restrict ourselves to looking at two key functions: `vplus` and `planet_acc`.

In `vplus` the primitive function + (according to PRF[$\triangle$]) has three possible layout-signatures that result in the layout-type $\triangle$. The index of `map` can be either of type 0 or $idx(1)$; combining these facts with the SEL[$\triangle$] rule we get $(1,1) \to 1$ layout signature for the `vprod` when the index of `map` is of layout-type $idx(1)$ and the arguments of layout-type 1. Alternatively we get $(\triangle, \triangle) \to \triangle$ as signature when the arguments are of type $\triangle$ and the index of `map` is of layout-type 0. The default $(0,0) \to 0$ is of course also possible, but all the other variants are cancelled out.

As for the `planet_acc` function, we know layout-type signatures of the `vprod`, the layout-types of the vectorised `acceleration` are $\{(1,1,0) \to 1, (\triangle, \triangle, 0) \to \triangle, (\triangle, \triangle, \triangle) \to \triangle\}$. So in order to infer the types, we have to consider layout-types for index variables i and j. If both are of layout-type 0, then we are back to the standard non-vectorised case. If both are $idx(1)$ – it would be canceled out by SEL[$\triangle$] rule. If $j : idx(1) \wedge i : 0$ then in order to make selection from `pos` of type $\triangle$, `pos` has to be of layout-type 2, layout type of `p` must be 1 and the first acceleration is taken. In that case the return type of the `planet_acc` would be 1. If $j : 0 \wedge i : idx(1)$, then `pos` has to be of layout-type 1, `masses[i]` is of type $\triangle$ and the third variant of acceleration is chosen. The return type of the `planet_acc` would be $\triangle$.

VI. INITIAL EVALUATION

In Fig. 4 we summarise experimental results regarding the layout inference algorithm described in the previous section. The main goal of the presented measurements is to demonstrate that first of all, the application of the transformation guided by the inference system brings expected performance, and secondly, that we can observe substantial difference in performance depending on a program vectorisation we chose.

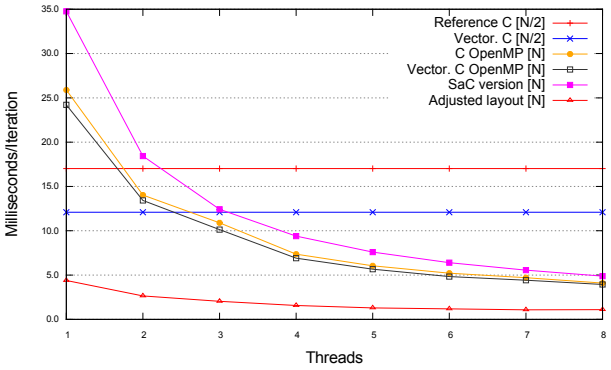

Fig. 4. Runtimes of SaC and C implementations of N-body. All the measurement were taken on Dell PowerEdge 2950 with Intel Xeon 4 core machine and SSE-4.2 vector instruction set.

The starting point of the measurements is the reference C implementation of the N-body benchmark. It is important to mention that single-threaded version uses the fact that the absolute value of the acceleration for (i, j) planets is the same as acceleration for (j, i) planets. We are going to mark such a solution with $[N/2]$ postfix and the program that computes accelerations for all the pairs with $[N]$ postfix. The version which 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 it prevents from parallelising on the outer level. The vectorisation of the reference C implementation across the inner axis is shown by "Vector. C [N/2]" line.

The transformation of the program is happening on a very high-level language which eventually has to be compiled down to some target language. The target language that we are dealing with is the C language and in order to express vector instructions we use a GCC-based portable framework we have developed earlier in [12]. 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, assuming that the inference would be implemented. In order to mimic multi-threaded execution we are using OpenMP annotations.

The key observation regarding hand-coded C versions is that vectorisation on the outer axis ("Adjusted layout [N]") performs significantly better than the version vectorised across the inner axis ("Vector. C OpenMP [N]") – even on a single core it manages to outperform the other version on eight cores. Another thing to notice is that inner axis vectorisation is negligible when comparing with non-vectorised version ("C OpenMP [N]").

Additionally, we correlate the runtimes of non-vectorised C versions with non-vectorised SaC version ("SaC version [N]") in order to judge on a potential impact of the vectorisation in SaC. As we can see, the overheads in SaC setting are larger than in C/OpenMP setting, however the scaling across multiple threds is similar. As the inference system produces both vectorisations of the N-body problem, as we have demonstrated earlier, it suggests that the vectorised code that could be generated by the SaC compiler would relate to "Vector. C OpenMP [N]" and "Adjusted layout [N]" in a similar way as "SaC version [N]" relates to "C OpenMP [N]".

VII. 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 and, more recently, for improved streaming through GPUs. 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 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.

G. Chen et al. in [2] 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 defiend 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 [1] 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 which 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 [11] present a polyhedra based transformation for automatic vectorisation. Similar to [1], this work assumes that layouts are fixed and the transformation is substituting identical arithmetic operations with vector ones.

T. Hanretty *et al.* in [6] 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 [6] 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 [9] demonstrate a language which is an extension of C, that allows to annotate data types which 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 [3] 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 [7] propose to change memory layouts dynamically to achieve better locality in the loop nest(s). This is an interesting approach. However, their dynamic adaptation is experimentally based rather than being analysis driven.

VIII. 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 due to 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 inference issues in detail and 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 problem. They also show that these improvements are orthogonal to non-vector-based parallelisations that stem from the use of multi-core CPUs.

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. As a first step we intend to implement our inference for the SaC compiler. Secondly, 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. Finally, the program transformation needs to be implemented to reflect the newly inferred data layouts.

Since most of these techniques are well-known we expect this system to deliver the performance we have seen in our hand-coded examples, including the N-body code presented in the paper.

References

[1] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[2] G. Chen. A constraint network based approach to memory layout optimization. In *In Proc. of the Conference on Design, Automation and Test in Europe*, pages 1156–1161, 2005.

[3] Philippe Clauss and Benoît Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests, 2000.

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

[5] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A binding scope analysis for generic programs on arrays. In *Proceedings of the 17th international conference on Implementation and Application of Functional Languages*, IFL'05, pages 212–230, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Proceedings of the 20th international conference on Compiler construction: part of the joint European conferences on theory and practice of software*, CC'11/ETAPS'11, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] M. Kandemir and I. Kadayif. Compiler-directed selection of dynamic memory layouts. In *Hardware/Software Codesign, 2001. CODES 2001. Proceedings of the Ninth International Symposium on*, pages 219 –224, 2001.

[8] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[9] Roland Leißa, Sebastian Hack, and Ingo Wald. Extending a c-like language for portable simd programming. In *Principles and Practice of Parallel Programming*, 2012.

[10] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Artjoms Šinkarovs and Sven-Bodo Scholz. Portable support for explicit vectorisation in c. In *16th Workshop on Compilers for Parallel Computing (CPC'12)*, 2012.

[13] Artjoms Šinkarovs, Sven-Bodo Scholz, Robert Bernecky, Roeland Douma, and Clemens Grelck. Sac/c formulations of the all-pairs n-body problem and their performance on smps and gpgpus. Accepted for publication in "Concurrency and Computation: Practice and Experience" journal in 2012, to appear. http://ashinkarov.github.io/publications/sexynbody.pdf.