Semi-automatic Composition of Data Layout Transformations for Loop Vectorization*

Shixiong Xu[1,2] and David Gregg[1,2]

[1] Lero, The Irish Software Engineering Research Centre,
[2] Software Tools Group, Department of Computer Science,
University of Dublin, Trinity College,
Dublin, Ireland
{xush,dgregg}@scss.tcd.ie

Abstract. In this paper we put forward an annotation system for specifying a sequence of data layout transformations for loop vectorization. We propose four basic primitives for data layout transformations that programmers can compose to achieve complex data layout transformations. Our system automatically modifies all loops and other code operating on the transformed arrays. In addition, we propose data layout aware loop transformations to reduce the overhead of address computation and help vectorization. Taking the Scalar Penta-diagonal (SP) solver, from the NAS Parallel Benchmarks as a case study, we show that the programmer can achieve significant speedups using our annotations.

1 Introduction

Single instruction multiple data (SIMD) vector computational units are widely available in processors from large supercomputers to energy-efficient embedded systems. Programmers often depend on compilers to auto-vectorize key loops. However, some program features can hinder the compilers from fully unleashing the power of SIMD. One important feature is interleaved data access coming from the data organized in the manner of an array of structures (AoS). In order to efficiently deal with interleaved data access, vectorizing compilers generate a sequence of data shuffling instructions (e.g. *pshuffle*, *pblend* in Intel SSE) for data reorganization. As long as data is accessed in a non-linear pattern, there will always be a cost of shuffling or gathering data for vectorization.

We observe that for many scientific computing applications with data in AoS, different loops in the program often repeat the same patterns of data permutation. These patterns usually first do data permutations on a small portion of the whole data needed before the computation in each loop iteration, and apply data permutations on the results after the computation is done. One way of getting rid of these repeated data permutation operations is to transform the layout

* This work was supported, in part, by Science Foundation Ireland grant 10/CE/I185 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

of the data throughout the program. There are two main approaches to transforming array layouts in programs: automatic transformation by the compiler, or manual changes by the programmer.

Compilers face two major challenges when performing automatic data layout transformations for vectorization. First, the compiler needs a very sophisticated **whole-program** data dependency and pointer aliasing analysis to make sure that the transformation is safe. Secondly, it is difficult for the compiler to choose the best layout. It is perhaps easier for the programmer to determine whether modifying the data layout is safe. But it is tedious and error-prone for programmers to change their code by hand. They may have to change the type declarations and any code that operates on the array. This may involve modifications to many parts of the program, and may result in changes to array indexing, and even the introduction of new statements and loops.

To allow compositions of data layout transformations and evaluate the performance impact of data layout transformations on vectorization, in this paper we put forward a new program annotation (using C language pragma) to enable programmers to specify a sequence of data layout transformations. This data layout transformation pragma is implemented in the Cetus source-to-source compiler framework [1]. Our prototype implementation currently supports static arrays but can be easily extended to support dynamically allocated arrays using Sung et al.'s approach [2]. Our compiler changes data type declarations for all modified arrays, rewrites all functions that operate on modified arrays to change array indexing, and introduces additional loops and other code. Similar to other pragma annotation systems, such as OpenMP, we assume that where the programmer requests a transformation, that transformation is safe.

In this paper, we make the following contributions:

1. We put forward a new C language pragma to allow programmers to specify a sequence of data layout transformations. This language annotation serves as a script to control data layout transformations and thus can be integrated into a performance auto-tuning framework as an extra tuning dimension.
2. We implemented our proposed data layout transformation pragma in the Cetus source-to-source compiler. To reduce the overhead of address computation and help vectorization, we introduce data layout aware loop transformations along with the data layout transformations.
3. Manual tuning of data layout transformations on the SP in the NAS Parallel Benchmarks shows that with proper data layout transformations, significant speedups are possible from better vectorization.

2 Language Support for Data Layout Transformations

2.1 Motivating Examples

In this section, we take the kernel of `tezar()` in the SP (**S**calar **P**enta-diagonal), one of the benchmarks in the NAS Parallel Benchmarks (NPB) to demonstrate

```
double us     [KMAX][JMAXP][IMAXP];
double vs    [KMAX][JMAXP][IMAXP];
double ws    [KMAX][JMAXP][IMAXP];
double speed    [KMAX][JMAXP][IMAXP];
double qs     [KMAX][JMAXP][IMAXP];
double rhs     [KMAX][JMAXP][IMAXP][5];
double u     [KMAX][JMAXP][IMAXP][5];

 for (k = 1; k <= nz2; k++) {
    for (j = 1; j <= ny2; j++) {
      for (i = 1; i <= nx2; i++) {
        xvel = us[k][j][i];
        yvel = vs[k][j][i];
        zvel = ws[k][j][i];
        ac   = speed[k][j][i];
        ac2u = ac*ac;
        r1 = rhs[k][j][i][0];
        r2 = rhs[k][j][i][1];
        r3 = rhs[k][j][i][2];
        r4 = rhs[k][j][i][3];
        r5 = rhs[k][j][i][4];
        uzik1 = u[k][j][i][0];
        btuz  = bt * uzik1;
        t1 = btuz/ac * (r4 + r5);
        t2 = r3 + t1;
        t3 = btuz * (r4 - r5);
        rhs[k][j][i][0] = t2;
        rhs[k][j][i][1] = -uzik1*r2 +
              xvel*t2;
        rhs[k][j][i][2] = uzik1*r1 +
              yvel*t2;
        rhs[k][j][i][3] = zvel*t2 + t3;
        rhs[k][j][i][4] =
              uzik1*(-xvel*r2 + yvel*r1)
              + qs[k][j][i]*t2 +
              c2iv*ac2u*t1 + zvel*t3;
      } } }
```

Fig. 1. The kernel of function tzetar() in the SP of NPB

the advantage of data layout transformations for efficient loop vectorization. This kernel conducts block-diagonal matrix-vector multiplication on the data.

There is a loop nest of depth three enclosing the main computations and all these loops are parallel, shown in Fig. 1. When vectorizing the innermost parallel loop i, compilers directly generate vector loads and stores for the data references to array `us, vs, ws`. On the contrary, the inter-leaved data access exposed by the references to array u and rhs may require compilers to apply suitable data reorganization. Compilers can treat these inter-leaved loads as gather operations. But the support for these gather operations in modern commodity processors is still not good enough [3]. Instead, the compiler may utilize available data permutation instructions to transform the inter-leaved data access into consecutive data access. On the other hand, the cost of data permutation instructions introduced by the data reorganization may not be well offset by the performance benefits gained by vectorization on the computations.

Table 1. Data layout schemes and vectorization strategies

Description	Declaration	Vectorization Strategy
Pure AoS	double u [KMAX][JMAXP][IMAXP][5];	Data permutation with stride 5

Split AoS (1:4)	double u1 [KMAX][JMAXP][IMAXP];
	double u2 [KMAX][JMAXP][IMAXP][4];
		Consecutive data accesses
		Data permutation with stride 4

Split AoS (4:1)	double u1 [KMAX][JMAXP][IMAXP][4];
	double u2 [KMAX][JMAXP][IMAXP];
		Data permutation with stride 4
		Consecutive data accesses

Split AoS (1:2:2)	double u1 [KMAX][JMAXP][IMAXP];
	double u2 [KMAX][JMAXP][IMAXP][2];
	double u3 [KMAX][JMAXP][IMAXP][2];
		Consecutive data accesses
		Data permutation with stride 2
		Data permutation with stride 2

Split AoS (2:2:1)	double u1 [KMAX][JMAXP][IMAXP][2];
	double u2 [KMAX][JMAXP][IMAXP][2];
	double u3 [KMAX][JMAXP][IMAXP];
		Data permutation with stride 2
		Consecutive data accesses
		Consecutive data accesses

Pure SoA	double u [5][KMAX][JMAXP][IMAXP];	Consecutive data accesses
Hybrid SoA	double u [KMAX][JMAXP][IMAXP/4][5][4];	Consecutive data accesses

Instead of compilers generating data permutation instructions to reorganize data, programmers can change the data layout into a form amenable to vectorization. Table 1 gives several possible data layout schemes of array u and their related vectorizing strategies compilers may take. The vectorizing strategies shown in Table 1 illustrate that some data layout transformations may simplify the vectorization of interleaved data access. For instance, compilers deal with the inter-leaved data access with stride 2 in `Split AoS` instead of stride 5 in `Pure AoS`, demonstrated in Section 4. Similarly, since the data references to the array *rhs* are inter-leaved with stride 5, the array *rhs* could also have same data layout transformation schemes as the array u.

2.2 Data Layout Transformation Pragmas

In this paper, we put forward a program annotation, *array transform*, a C language pragma to express data layout transformations on the static arrays. The syntax of this new pragma is shown in Fig. 2.

⟨*pragma*⟩	::=	#pragma array_transform ⟨*array_name*⟩ ⟨*descriptor*⟩ ⟨*actions*⟩
⟨*descriptor*⟩	::=	[⟨*identifier*⟩] ⟨*descriptor_list*⟩
⟨*descriptor_list*⟩	::=	[⟨*identifier*⟩] ⟨*descriptor_list*⟩ | ⟨*empty*⟩
⟨*actions*⟩	::=	-> ⟨*pre_actions*⟩ ⟨*post_actions*⟩
⟨*pre_actions*⟩	::=	⟨*strip_mine*⟩ | ⟨*interchange*⟩ | ⟨*pad*⟩ | ⟨*pre_actions*⟩ | ⟨*empty*⟩
⟨*post_actions*⟩	::=	⟨*peel*⟩ | ⟨*empty*⟩ | ⟨*post_actions*⟩
⟨*strip_mine*⟩	::=	STRIP_MINE (⟨*identifier*⟩ , ⟨*stride_size*⟩ , ⟨*identifier*⟩)
⟨*interchange*⟩	::=	INTERCHANGE (⟨*identifier*⟩, ⟨*identifier*⟩)
⟨*pad*⟩	::=	PAD (⟨*identifier*⟩, ⟨*pad_size*⟩)
⟨*peel*⟩	::=	PEEL (⟨*identifier*⟩, ⟨*peel_size*⟩)

Fig. 2. Syntax of the data layout transformation pragma

The *array transform* pragma consists of *array descriptor* and *transform actions*. The *array descriptor* gives a name to each array dimension, and these names are used in the *transform actions* to record the related data layout transformations. The *transform actions* present the basic data layout transformations. In this paper, we define four basic data layout transformations, *strip-mining*, *interchange*, *pad*, and *peel*. These terms for data layout transformations are borrowed from the classic loop transformations [4].

The data storage of an array A can be viewed as a rectangular polyhedron. In [5], formal indices $\boldsymbol{\mathcal{I}}$ is introduced to describe the array index space

$$\boldsymbol{\mathcal{I}} = [i_1, i_2, \ldots, i_n]^T \tag{1}$$

where n is the dimension of the array A. The range of the formal indices $\boldsymbol{\mathcal{I}}$ describes the size of the array, or index space, as follows:

$$\boldsymbol{\lambda} \leq \boldsymbol{\mathcal{I}} < \boldsymbol{\mu} \tag{2}$$

where the lower bound vector $\boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_n]^T$ and the upper bound vector $\boldsymbol{\mu} = [\mu_1, \ldots, \mu_n]^T$ are $n \times 1$ vectors. The array index in C language can only start from 0, therefore, the lower bound vector $\boldsymbol{\lambda}$ in this paper is $\mathbf{0}$. As each array dimension is given a name by the *array descriptor*, these names can be treated as the formal indices to the arrays.

In contrast to the loop transformations which transform the loop iteration space formed by the loop indices, data layout transformations change the array index space. Since the array index space is changed, the subscripts in references to the array also have to be transformed accordingly.

The subscripts in a reference to an array in loops represent a function that maps the values of the loop iteration space to the array index space and this function is often expressed in the form of a memory access matrix [6]. Consider a data reference to an M dimensional array in the loop nest of depth D, where D and M do not need to match. The memory access pattern of the array in the loop is represented as a memory access vector, $\boldsymbol{m}$, which is a column vector of size M starting from the index of the first dimension. The memory access vector is then decomposed as an affine form:

$$\boldsymbol{m} = \mathbf{M}\boldsymbol{i} + \boldsymbol{o} \tag{3}$$

where $\mathbf{M}$ is a memory access matrix whose size is $M \times D$, $\boldsymbol{i}$ is an iteration vector of size D traversing from the outermost to the innermost loop, and $\boldsymbol{o}$ is an offset vector that is a column vector of size M and determines the starting access point in an array.

The semantics of the four data layout transformations are defined as follows:

Strip-mining: STRIP_MINE (*id*$_1$, *stride_size*, *id*$_2$)
This transformation splits the array dimension i indicated by the id_1 into tiles of size *stride_size* and creates a new formal indices vector $\boldsymbol{\mathcal{I}'}$ and two new dimension range vectors $\boldsymbol{\lambda'}$ which is $\mathbf{0}$ and $\boldsymbol{\mu'}$. Intuitively, the strip-mining splits the array dimension into two adjacent dimensions with dimension name id_1 and id_2, respectively. The new dimension id_1 takes the position of i and the new dimension id_2 takes the position of $i+1$ in the $\boldsymbol{\mathcal{I}'}$. $\boldsymbol{\mu'}$ is created by dividing $\boldsymbol{\mu}_i$ into $\boldsymbol{\mu}_h$ and $\boldsymbol{\mu}_l$, where $\boldsymbol{\mu}_h = \lceil \boldsymbol{\mu}_i / stride\_size \rceil$ and $\boldsymbol{\mu}_l = stride\_size$. For each reference with subscripts $\boldsymbol{s}$ to the target array in the corresponding scope, new subscripts $\boldsymbol{s'}$ for each reference are created by dividing $\boldsymbol{s}_i$ into $\boldsymbol{s}_h$ and $\boldsymbol{s}_l$, where $\boldsymbol{s}_h = \lfloor \boldsymbol{s}_i / stride\_size \rfloor$ and $\boldsymbol{s}_l = \boldsymbol{s}_i \bmod stride\_size$. Note that, when the original dimension size is not a multiple of block size *stride_size*, padding is introduced automatically at dimension i.

Interchange: INTERCHANGE (*id*$_1$, *id*$_2$)
This transformation interchanges the array dimensions i, j indicated by id_1

and id_2 and creates a new formal indices vector $\boldsymbol{\mathcal{I}'}$ and two new dimension range vectors $\boldsymbol{\lambda'}$ which is $\mathbf{0}$ and $\boldsymbol{\mu'}$. The upper bound vector $\boldsymbol{\mu'}$ is created by interchanging $\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}_j$. For each reference with subscripts $\boldsymbol{s}$ to the target array in the corresponding scope, new subscripts $\boldsymbol{s'}$ for each reference are created by interchange $\boldsymbol{s}_i$ and $\boldsymbol{s}_j$.

Pad: PAD (*id*, *pad_size*)

This transformation pads the array dimension i indicated by *id* by the size of $|pad\_size|$ either from the beginning if the integer *pad_size* is negative or from the end if the integer *pad_size* is positive. Two new dimension range vectors $\boldsymbol{\lambda'}$ and $\boldsymbol{\mu'}$ are created, where $\boldsymbol{\lambda'}$ is $\mathbf{0}$ and $\boldsymbol{\mu'}$ is formed by increasing $\boldsymbol{\mu}_i$ by $|pad\_size|$. If the pad_size is negative, for each reference with subscripts $\boldsymbol{s}$ to the target array in the corresponding scope, new subscripts $\boldsymbol{s'}$ for each reference are created, where $\boldsymbol{s'}_i = \boldsymbol{s}_i + |pad\_size|$.

Peel: PEEL (*id*, *peel_size*)

This transformation peels the dimension i of an array $\mathcal{A}$ indicated by *id* by reducing the dimension size by $|peel\_size|$ and creates two arrays $\mathcal{A}_1, \mathcal{A}_2$. Two pairs of range vectors $(\boldsymbol{\lambda'}_h, \boldsymbol{\mu'}_h)$,$(\boldsymbol{\lambda'}_l, \boldsymbol{\mu'}_l)$ are created for resulting arrays $\mathcal{A}_1, \mathcal{A}_2$, respectively, where $\boldsymbol{\lambda'}_h, \boldsymbol{\lambda'}_l$ are $\mathbf{0}$, and $\boldsymbol{\mu'}_h, \boldsymbol{\mu'}_l$ are as follows:

$$\boldsymbol{\mu'}_h = \begin{cases} |peel\_size| & \text{if } peel\_size > 0 \\ \boldsymbol{\mu}_i - |peel\_size| & \text{otherwise} \end{cases}$$

$$\boldsymbol{\mu'}_l = \begin{cases} |peel\_size| & \text{if } peel\_size < 0 \\ \boldsymbol{\mu}_i - |peel\_size| & \text{otherwise} \end{cases}$$

For each reference with subscripts $\boldsymbol{s}$ to the target array $\mathcal{A}$ in the corresponding scope, new subscripts $\boldsymbol{s'}$ are created by first choosing the right array, $\mathcal{A}_1$ if $\boldsymbol{s}_i$ is less than μ_i of array $\mathcal{A}_1$ or $\mathcal{A}_2$ otherwise; then new subscripts are calculated as follows:

$$\boldsymbol{s'}_i = \begin{cases} \boldsymbol{s}_i & \text{if refers to } \mathcal{A}_1 \\ \boldsymbol{s}_i - \boldsymbol{\mu'}_{hi} & \text{otherwise} \end{cases}$$

Note that, according to the semantics of array peeling, the subscripts in the dimension i of all the references to the array $\mathcal{A}$ should be compile-time constants. As the array peeling transformations can be chained together, in this case, all these chained array peeling actions should apply on the same array dimension. The input to the next array peeling transformation is decided by the current peeling size. If the current peeling size is positive, which means the target array dimension is peeled off from the beginning, the remaining array $\mathcal{A}_2$ will be the input for the next array peeling action. Otherwise, the target array dimension is peeled off from the end and thus the remaining array $\mathcal{A}_1$ will be the input for the next array peeling action, demonstrated by the `Split AoS` in Table 2.

The four data layout transformations are classified into two classes, *pre-action* and *post-action*. The *post-action* means all actions of this class can only be added after all the actions in the class of *pre-action*. We define *array peeling* as a member of the class *post-action* because we observe that for vectorization, array peeling is mainly used to split one array dimension for the data alignment or making the size of the array dimension power-of-two.

2.3 Composition of Data Layout Transformations

Our proposed *array transform* supports four primitive data layout transformations on static arrays. More complex data layout transformations can be achieved by composing these primitive transformations.

Array permutation permutes several array dimensions according to a given permutation command. It is more general than array interchange, which only swaps two array dimensions indicated by the dimension names. It is intuitive that array permutation can be decomposed as a sequence of array interchange actions. For example, given an array: `float A[SIZE_I][SIZE_J][SIZE_K]`, where i, j, k are the dimension names for each array dimension from the first to the last dimension, the permutation command (k, i, j), which rearranges the array dimensions indicated by i, j, k into a new order k, i, j, can be decomposed into a sequence of array interchange transformations, $(k, j) - > (i, k)$. Therefore, programmers can put the array transform pragma as `#pragma array_transform A[i][j][k] -> INTERCHANGE(k, j) -> INTERCHANGE(i, k)`

Rectangular array tiling blocks array dimensions into tiles, and thus decomposes the whole array into blocks which may help improve data locality. Array tiling is a process of choosing suitable hyperplanes according to certain conditions (e.g. data reuse distance) and partitioning the array data space with these hyperplanes. Here, *rectangular array tiling* means the determined tiling hyperplane for each array dimension is perpendicular to the axis of the array dimension to be tiled. Similar to the loop tiling which is a combination of loop strip-mining and loop interchange, *rectangular array tiling* can be decomposed into a sequence of array strip-mining, and array interchange, which are the primitive transformations defined in the *array transform* pragma.

As listed in Table 1 in Section 2.1, there are seven possible data layout transformation schemes for the motivating example. With our proposed *array transform* pragma, programmers can easily specify these data layout schemes by giving varying sequences of valid transformation actions, as shown in Table 2.

3 Data Layout Aware Loop Transformations

Array strip-mining introduces modulus operations to get offsets in the resulting tiles, illustrated in line 8 of Fig. 3. This kind of operation is not friendly to vectorization, because it might hinder the native compiler from detecting possible consecutive data access. Both the Intel C compiler and GCC are not able to identify that the data references to the transformed array are consecutive. We introduce data layout aware loop transformations to address this problem.

The modulus operations in the data references to the transformed arrays are from the array strip-mining. Therefore, if the data references to the target

Table 2. Data layout transformations assuming the array `u` is originally in the `Pure AoS`

Description	Declaration	Data Layout Transformation
Pure AoS	double u [KMAX][JMAXP][IMAXP][5];	NA

Split AoS (1:4)	double u1 [KMAX][JMAXP][IMAXP];
	double u2 [KMAX][JMAXP][IMAXP][4];
		#pragma array_transform u[i][j][k][m]->
		PEEL(m, 1)

Split AoS (4:1)	double u1 [KMAX][JMAXP][IMAXP][4];
	double u2 [KMAX][JMAXP][IMAXP];
		#pragma array_transform u[i][j][k][m]->
		PEEL(m, -1)

Split AoS (1:2:2)	double u1 [KMAX][JMAXP][IMAXP];
	double u2 [KMAX][JMAXP][IMAXP][2];
	double u3 [KMAX][JMAXP][IMAXP][2];
		#pragma array_transform u[i][j][k][m]->
		PEEL(m, 1) -> PEEL(m, 2)

Split AoS (2:2:1)	double u1 [KMAX][JMAXP][IMAXP][2];
	double u2 [KMAX][JMAXP][IMAXP][2];
	double u3 [KMAX][JMAXP][IMAXP];
		#pragma array_transform u[i][j][k][m]->
		PEEL(m, 2) -> PEEL(m, 2)

Pure SoA	double u [5][KMAX][JMAXP][IMAXP];	#pragma array_transform u[i][j][k][m]-> INTERCHANGE(m, k) -> INTERCHANGE(m, j) -> INTERCHANGE(m, i)
Hybrid AoS	double u [KMAX][JMAXP][IMAXP/4][5][4];	#pragma u[i][j][k][m]-> STRIP_MINE(k, 4, kk) -> INTERCHANGE(m, kk)

array to be transformed are enclosed in loops, one easy way to get rid of the modulus operations is to strip-mine the corresponding loops. In this paper we only consider the case where all the references to the arrays to be transformed have uniform effects to the surrounding loops. By which it means, if a loop is strip-mined with stride δ according to one data reference, there should be no other data references which require the same loop to be strip-mined with stride other than δ.

Data layout aware loop strip-mining according to the array strip-mining may include pre-loop peeling and post-loop peeling depending on whether the loop iteration space and the data index space are aligned, as shown in line 12-14, 20-22 of Fig. 3. If a loop starts from 0 and ends at `SIZE-1` and the corresponding array dimension has a range from 0 to `SIZE-1`, in this case, the loop iteration space and the data index space are aligned, otherwise they are unaligned. Regarding the legality of these data layout aware loop peeling and loop strip-mining, they are always legal because these loop transformations inherently will not change the data dependencies across loop iterations.

In addition to the elimination of the modulus operations, the data layout ware loop strip-mining helps solve the alignment issue in vectorization. If the loop iteration space and the data index space are not aligned, pre-loop peeling and post-loop peeling are applied according to the boundaries of tiles from the array strip-mining. If the array starting address is aligned to 32 bytes and the tile size is 32 bytes, for instance, all the boundaries of tiles will be aligned to 32 bytes as well. As a result, all the loads from these boundaries are aligned to 32 bytes.

```
#pragma ary[i] -> STRIP_MINING(i, 4)
  float ary[32];
  /* before transformation: */
  for (i = 1; i < 31; i++)
    ... = ary[i];
  /* after transformation: */
  for (i = 1; i < 31; i++)
    ... = ary[i/4][i%4];

  /* data layout aware
        transformation:*/
  /* from pre-loop peeling */
  for (i = 0; i < 1; i++)
    for (ii = 1; ii < 4; ii++)
        ... = ary[i][ii];
  /* from loop strip-mining */
  for (i = 1; i < 7; i++)
    for (ii = 0; ii < 4; ii++)
      ... = ary[i][ii];
  /* from post-loop peeling*/
  for (i = 7; i < 8; i++)
    for (ii = 0; ii < 3; ii++)
      ... = ary[i][ii];
```

Fig. 3. Data layout aware loop transformation.

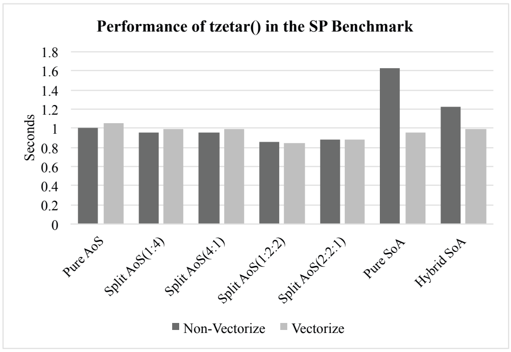

Fig. 4. Performance of `tzetar()` with different data layout transformations

4 Experimental Evaluation

4.1 Implementation

Our proposed array transform pragma is implemented in the Cetus source-to-source C compiler. All the *transform actions* are processed and collected in the pragma parsing phase. The actual data layout transformations and the data layout aware loop optimizations are done as transform passes in the Cetus compiler. The high-level internal presentation in the Cetus compiler keeps the array access close to the source code and thus simplifies the array transformation and the substitution of subscripts in array references.

4.2 A Case Study: Data Layout Tuning for Loop Vectorization

In this section, we use the SP in the NAS Parallel Benchmarks [7] as a case study to show the performance impact of data layout transformations upon loop vectorization. **SP** is one of the simulated CFD applications that solve the discretized compressible Navier-Stokes equations. We choose the data set of Class A in NPB, which has the size of $64 \times 64 \times 64$ with 400 iterations. All the experiments are conducted on an Intel Haswell platform (Intel Core i7-4770) running the Ubuntu Linux 13.04. We choose the Intel C compiler 13.1.3 to compile both the original and transformed code with the compiler option `-march=core-avx2 -O3 -fno-alias` for vectorization.

Performance of the Motivating Example. Fig. 4 gives the performance of the motivating example in different data layouts shown in Table 2. The results show that the best vectorization performance is given by the data layout

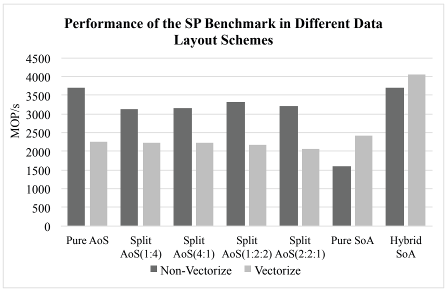

Fig. 5. Performance of the SP in different data layouts

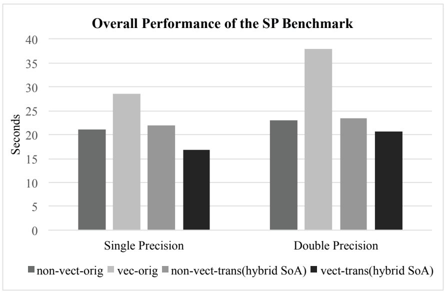

Fig. 6. Performance of the SP of the NAS Parallel Benchmarks

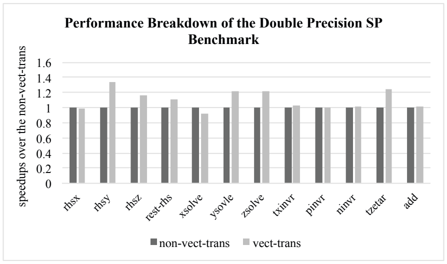

Fig. 7. Performance breakdown of the double precision SP of the NAS Parallel Benchmarks

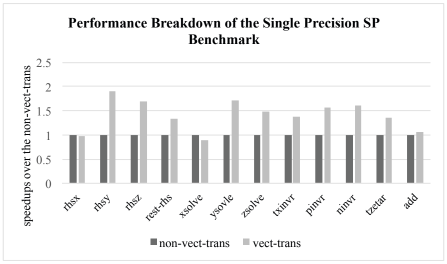

Fig. 8. Performance breakdown of the single precision SP of the NAS Parallel Benchmarks

transformation `Split 1:2:2`. Splitting the last dimension of the array `u` (line 22 in Fig. 1) into three parts with sizes of 1, 2 and 2 helps the native compiler vectorize the load of array `u` with a contiguous vector load. In the mean time, data permutation instructions (e.g. `vperm2f128, vunpacklpd`) are used for the data reorganization of the array `rhs` (line 17 - 21 in Fig. 1) instead of gather instructions.

Overall Performance. We manually tune the data layout transformations for the SP and constrain the search space of data layout transformations to the ones mentioned in Table 2. Fig. 5 presents the overall performance of the SP in different data layouts. Among the seven data layouts, the `Hybrid SoA` gives the best overall performance. We also evaluated the performance of the single precision SP with the data layout `Hybrid SoA`, where the strip-mining size is 8. Compared to the double precision SP, the performance boost from vectorization for the single precision SP is more significant, as depicted in Fig. 6.

Fig. 7 and Fig. 8 give the performance breakdown of the single precision and double precision SP, respectively. With naive manual tuning of data layouts, for the SP, vectorization on the transformed data can outperform the vectorization on the untransformed data by a factor of 1.8. The experimental results demonstrate that it is necessary to introduce data layout tuning into existing performance auto-tuning systems, in particular, for the better performance of vectorization.

5 Related Work

Data layout transformations have primarily been applied to improving cache locality and localizing memory accesses in nonuniform memory architectures and clusters [8]. Maleki et al. [9] evaluated the vectorizing compilers and found that manually changing the data layout is a valuable way to help compilers to efficiently vectorize loops with non-unit stride accesses. However, compilers rarely automatically perform the memory layout transformations.

Our work is mainly inspired by the the work on semi-automatic composition of loop transformations for deep parallelism and memory hierarchies [10]. The main approach of previous work is introducing a script language to control the loop transformations upon the target loops. As far as we know, there are no such script languages available to control the data layout transformations. Similar language support for data layout transformations is designed mainly for optimizing data locality, such as the align and distribute directives in HPF [11].

Henretty et al. [12] propose a novel data layout transformation, dimension-lifted transposition, for stencil computations. This domain-specific technique solves the memory stream alignment issue. On the contrary, our work is a general solution to manual data layout transformations. Our work is greatly close to the work by Sung [2], which presents a framework that enables automatic data layout transformations for the structured grid codes in CUDA. Our work not only supports more data layout transformations but also presents data layout aware loop transformations for loop vectorization.

Jang et al.[6] optimize memory access into DRAM bursts (i.e. coalescing) by gaining unit-stride accesses with data layout transformations in the case of GPGPUs. Mey et al. [13] put forward a meta-data framework that allows both programmers and tuning experts to specify architecture specific and domain specific information for parallel-for loops of programs. The data layout transformations considered in this work are only AoS-to-SoA and SoA-to-AoS. Sinkarovs et al. [14] also present a compiler driven approach towards automatically transforming data layouts into a form that is suitable for vectorization. Their work is studied in the case of a first-order functional array programming language while our work focuses on the imperative C language.

6 Conclusion

In this paper, we put forward a new program annotation (using C language pragma) to enable programmers to specify data layout transformations and implemented it in the Cetus source-to-source compiler. In terms of loop vectorization, we introduce data layout ware loop transformations to help the native compilers to do better vectorization as well. The four primitive data layout transformations presented are suitable to be composed into more complex data layout transformations. The experimental results indicate that it is necessary to introduce semi- or fully automatic tuning of data layout transformations in order to help compilers to achieve better performance on vectorization.

References

1. Bae, H., Mustafa, D., et al.: The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation. Int. J. Parallel Program. 41, 753–767 (2013)
2. Sung, I.-J., Stratton, J.A., Hwu, W.-M.W.: Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010 (2010)
3. Ramachandran, A., Vienne, J., et al.: Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi. In: 2013 42nd International Conference onParallel Processing (ICPP), pp. 736–743 (2013)
4. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler Transformations for High-performance Computing. ACM Comput. Surv. 26, 345–420 (1994)
5. O'Boyle, M.F.P., Knijnenburg, P.M.W.: Non-singular Data Transformations: Definition, Validity and Applications. In: Proceedings of the 11th International Conference on Supercomputing, ICS 1997 (1997)
6. Jang, B., Mistry, P., et al.: Data Transformations Enabling Loop Vectorization on Multithreaded Data Parallel Architectures. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010 (2010)
7. Bailey, D.H., Barszcz, E., et al.: The NAS Parallel Benchmarks. Technical report, The International Journal of Supercomputer Applications (1991)
8. Kennedy, K., Kremer, U.: Automatic Data Layout for Distributed-memory Machines. ACM Trans. Program. Lang. Syst. 20, 869–916 (1998)
9. Maleki, S., Gao, Y., et al.: An Evaluation of Vectorizing Compilers. In: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011 (2011)
10. Girbal, S., Vasilache, N., et al.: Semi-automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. Int. J. Parallel Program. 34, 261–317 (2006)
11. Rice University, CORPORATE:High Performance Fortran Language Specification. SIGPLAN Fortran Forum 12 (1993)
12. Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J., Sadayappan, P.: Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 225–245. Springer, Heidelberg (2011)
13. Majeti, D., Barik, R., Zhao, J., Grossman, M., Sarkar, V.: Compiler-Driven Data Layout Transformation for Heterogeneous Platforms. In: an Mey, D., et al. (eds.) Euro-Par 2013. LNCS, vol. 8374, pp. 188–197. Springer, Heidelberg (2014)
14. Sinkarovs, A., Scholz, S.B.: Semantics-Preserving Data Layout Transformations for Improved Vectorisation. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC 2013 (2013)